BURRA SEC

Centrifuge Security Review

Reviewed by: Goran Vladika, Pranav Garg

22nd April - 25th April, 2025

Centrifuge Security Review Report

Burra Security

April 29, 2025

Introduction

A time-boxed security review of the **Centrifuge** 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

Centrifuge V3 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* - efcf3bcdbbeaca913d4240aade6e99c8a9bff7c6**

Scope

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

- src/common/Gateway.sol
- src/misc/libraries/TransientBytesLib.sol
- src/misc/libraries/TransientArrayLib.sol

Findings Summary

ID	Title	Severity	Status
C-01	Attacker can use re-entrancy to drain the Gateway	Critical	Resolved
M-01	Underpaid messages can't be submitted	Medium	Resolved
L-01	Only one duplicated message can be in recovery process at a time	Low	Ack
I-01	Mark more event fields as `indexed`	Info	Resolved
I-02	Skip pulling funds from escrow if it is empty	Info	Resolved

Detailed Findings

[C-01] Attacker can use re-entrancy to drain the Gateway

Target

- Gateway.sol

Severity:

- Impact: High
- Likelihood: High

Description:

There is a critical re-entrancy vulnerability in Gateway's implementation of TX refund:

```
    function _refundTransaction() internal {
        if (transactionRefund == address(0)) return;

        if (fuel > 0) {
            (bool success,) = payable(transactionRefund).call{value:
                fuel}(new bytes(0));

            if (!success) {
                // If refund fails, move remaining fuel to global pot
                _subsidizePool(GLOBAL_POT, transactionRefund, fuel);
            }

            fuel = 0;
        }

        transactionRefund = address(0);
    }
```

The core of the issue is that the external call to the untrusted party is performed **before** transient state is updated. So contract set as `transactionRefund` can re-enter the batch processing process and get the refund again, and again, until the Gateway is drained. Gateway holds the subsidy funds, so all of those would be lost. Additionally, attacker can craft the messages to pull in all the funds from the escrows and then steal those funds as well.

Let's say Gateway holds 100 ETH. Here's how attack could look like:

- attacker crafts exploit contract which can trigger the batch process by calling ie. `VaultRouter`'s `multicall`
- batch costs 0.1 ETH but attacker provides 20.1 ETH
- flow goes `startBatching` -> `payTransaction` -> multiple `sends` -> `endBatching` -> `_refundTransaction`
- when `_refundTransaction` is started, `isBatching` = false, `fuel` = 20ETH
- attacker's exploit contract is called to refund 20 ETH
- exploit contract receives 20 ETH and triggers the batch process again by calling `multicall`, this time providing 0 value
- batch is processed, refund is triggered while `fuel` is still set at 20 ETH. Attacker receives 20 ETH again
- attacker's exploit contract repeats this process 5 times
- at the end, attacker holds 120 ETH (20 ETH from the initial overpayment and 100 ETH drained from Gateway)

This attack is also valid when sending a single message (not necessarily a batch).

Recommendation:

Update transient state (`fuel` and `transactionRefund`) before doing the external call.

Client:

Fixed in PR`#300`

BurraSec:

Fix looks good

[M-02] Underpaid messages can't be submitted

Target

- Gateway.sol

Severity:

- Impact: Medium
- Likelihood: Medium

Description:

Centrifuge lets users and external protocols to submit batches without paying for them. The goal is to let pool issuer to pay for submitted underpaid batches separately when needed. This approach works with Axelar adapter. Axelar supports requesting a cross-chain message and then paying for it separately. But Wormhole does not support it. Wormhole adapter calls `sendPayloadToEvm` on the relayer. Relayer will check if provided value matches the actual cost for delivering, and if that's not the case TX will revert. Thus message cannot be submitted without being paid for, breaking the use-case which is supposed to be supported.

Recommendation:

Add logic on the gateway side to store underpaid messages and to enable separate payments for transferring them.

Client:

Fixed in PR#300

BurraSec:

Fix looks good, submission of underpaid msgs is now supported

[L-01] Only one duplicated message can be in recovery process at a time

Target

- Gateway.sol

Severity:

- Impact: Medium
- Likelihood: Low

Description:

Centrifuge supports processing of identical messages. There are no nonces, so counters are used to keep track of duplicated messages, ie:

```
    bytes32 messageHash = keccak256(message);
    failedMessages[centrifugeId][messageHash]++;
    emit FailMessage(centrifugeId, message, err);
```

There is also a recovery mechanism which can be used by admin to recover lost messages:

```
    function initiateRecovery(uint16 centrifugeId, IAdapter adapter,
       bytes32 payloadHash) external auth {
        require(_activeAdapters[centrifugeId][adapter].id != 0,
           InvalidAdapter());
        recoveries[centrifugeId][adapter][payloadHash] = block.
           timestamp + RECOVERY_CHALLENGE_PERIOD;
        emit InitiateRecovery(centrifugeId, payloadHash, adapter);
    }
```

One issue with this implementation is that it lacks support to handle recovery of multiple identical messages at a time. If `initiateRecovery` is called for a message, when recovery of identical message is already ongoing, result will be delaying the recovery of a single message. Second identical message can be recovered only when 1st one is executed.

Recommendation:

If this edge case seems realistic enough, consider using counter-based approach for recoveries in addition to timestamps.

Client:

Acknowledged, this is by design.

BurraSec:

Acknowledged

[I-01] Mark more event fields as `indexed`

Target

- `IGateway.sol`

Severity:

INFO

Description:

Having certain event fields marked as `indexed` enables straight-forward way for searching or filtering events by that field. Ie. it could be useful to mark `payloadId` as `indexed` in all events, so that a single `eth_getLogs` can fetch all the event data for a specific batch. Similarly, marking `underpaid` field as `indexed` enables having a simple call to track all batches which need manual intervention.

Client:

Fixed in `PR#300`

BurraSec:

Fix looks good

[I-02] Skip pulling funds from escrow if it is empty

Target

- Gateway.sol

Severity:

INFO

Description:

Gateway has logic to fetch funds from escrow if current subsidy amount is not big enough:

```
    function _requestPoolFunding(PoolId poolId) internal {
        IRecoverable refund = subsidy[poolId].refund;
        if (!poolId.isNull() && address(refund) != address(0)) {
            uint256 refundBalance = address(refund).balance;

            // Send to the gateway GLOBAL_POT
            refund.recoverTokens(ETH_ADDRESS, address(this),
                refundBalance);

            // Extract from the GLOBAL_POT
            subsidy[GLOBAL_POT].value -= uint96(refundBalance);
            _subsidizePool(poolId, address(refund), refundBalance);
        }
    }
```

It's reasonably likely that escrow might be empty, so in case `refundBalance` is 0 early return would save gas on unnecessary external call and multiple storage operations.

Client:

Fixed in PR#300

BurraSec:

Fix looks good