Centrifuge: Protocol v3 - 2nd Review

security review

reviewed by:

xmxanuel
cantina.xyz/u/xmxanuel

9th May 2025

Revision v3.0.1
July 2025

Centrifuge: Protocol v3 - 2nd Review

1. Executive Summary

The Centrifuge team has asked xmxanuel to review their Solidity codebase for `protocol v3` as a security reviewer to give feedback on overall design and to identify security vulnerabilities.

The following report only includes findings related to security and not the overall design feedback.

2. Disclaimer

This security review report is provided "as is" and for informational purposes only. The purpose of this report is to assist the client in identifying potential security vulnerabilities in the reviewed code, based on the scope and methodology agreed upon. It does not constitute legal, investment, financial, or any other form of professional advice.

No warranties or guarantees are made regarding the completeness, accuracy, or security of the reviewed code. The author and any affiliated parties make no representations or warranties, express or implied, and expressly disclaim any liability or responsibility for any loss, damage, or other consequence arising from or related to the use of, reliance on, or inability to use this report or the reviewed code. This includes, without limitation, direct, indirect, incidental, consequential, special, exemplary, or punitive damages, even if advised of the possibility of such damages.

This report reflects the results of a best-effort security review conducted within the scope and time constraints agreed upon with the client. Although an attempt was made to identify potential security issues, there is no assurance that all vulnerabilities have been discovered. A smart contract security review cannot be considered a guarantee or certification of security. Users and project teams remain solely responsible for the use, deployment, and management of any smart contracts.

This report does not constitute an endorsement or recommendation of the reviewed project. Third parties should not rely on this report and are strongly encouraged to perform their own independent assessments.

3. Repository

https://github.com/centrifuge/protocol-v3/

Version	Commit Hash	Date	Note
Review Version	f2ef1edefaa2d22e8c00ab70b647dbce3bf859a6	3rd May 2025	Version of the main protocol v3 review
v3.0.1	7ca5819788f6d28d8481932b237951c1ffb7ff3b	July 2025	Reviewed change PRs for the new version. Not included are the deploy scripts in `/script`.

4. Findings

The following findings were identified during the security review of the codebase. Each finding is categorized by severity level based on its potential impact and includes a detailed description and recommended remediation.

Summary

Severity	ID	Title	Status
MEDIUM	M1	`executeRecovery` lacks of `pausable` modifier which can lead to undisputed invalid recoveries	Acknowledged
MEDIUM	M2	`BalanceSheet.deposit` allows depositing funds from other pool managers' `owner` addresses if approval exists	Fixed
LOW	L1	ETH can be stuck in the Gateway when `_isManagerAndPaid` is called with a local centrifugeId	Acknowledged.
LOW	L2	Missing `centrifugeId` validation in `MessageProcessor` allows adapter to initiate recovery for unrelated chains	Fixed
LOW	L3	Excessive privileges in `ShareToken` for `hooks`	Fixed
LOW	L4	`BalanceSheet.setQueue` to `false` doesn't submit existing queue and would break the correct hub transactions ordering	Fixed
INFO	I1	TransientArrayLib implementation could be optimized to match Solidity's storage pattern	Fixed
INFO	I2	Missing validation allows multiple axelarIds to be mapped to the same centrifugeId	Acknowledged
INFO	I3	`Gateway.retry` not possible for `messageRecovery`	Acknowledged
INFO	I4	Not calling `Gateway.repay` enables multiple state edge cases	Acknowledged
INFO	I5	Potential reentrancy in `endBatching` without the `auth` modifier	Fixed
INFO	I6	Any failed `handle` call can be executed in the future again with `Gateway.retry`	Acknowledged
INFO	I7	`refund.recoverTokens` call in `Gateway._requestPoolFunding` assumes tokens are always transferred	Acknowledged

INFO	I8	Gateway Underpaid gasLimit always overwriting for each underpaid transaction	Acknowledged

MEDIUM Findings

M1. `executeRecovery` lacks of `pausable` modifier which can lead to undisputed invalid recoveries

Gateway.sol#L248

The `executeRecovery` function lacks a `pausable` modifier, creating a vulnerability where recovery messages can be executed even when the protocol is paused.

However, in the `paused` state, it would not be possible to initiate a new `recovery` or dispute an existing one. Both `initiateRecovery` and `disputeRecovery` require the protocol to be unpaused since they are called via the `MessageProcessor` through `Gateway.handle` .

All messages require multiple votes by different adapters before the protocol `handles` them.

When an adapter fails, the message recovery system allows other adapters to recover that adapter's message to still reach the required quorum.

An invalid message by an adapter must be `disputed` by any of the other adapters.

This design creates an exploitable attack vector. If an attacker compromises one adapter and governance attempts to pause the protocol in response, the attacker could front-run the pause transaction with multiple malicious `initiateRecovery` calls.

Once the protocol is paused, legitimate adapters would be unable to call `disputeRecovery` to challenge these invalid recoveries although it would be possible to execute the invalid messages later.

The attacker could use a `scheduleRely` message to gain access to the vault contracts and withdraw all funds.

Recommendation: Don't allow `messages` to be executed if the protocol is `paused` .

Centrifuge: Agreed, but in that case the Guardian can still intervene! The Guardian gets `auth` access on the Gateway. For now the system only works with the assumption that the guardian is there and not only other adapters. The Guardian has the permission to call the `Gateway` contract.

xmxanuel: Acknowledged.

M2. `BalanceSheet.deposit` allows depositing funds from other pool managers' `owner` addresses if approval exists

BalanceSheet.sol#L96

The `deposit` function allows a balance sheet manager to deposit funds back into the pool. The `owner` parameter is the address from which the funds are taken.

```
function deposit(PoolId poolId, ShareClassId scId, address asset, uint256 tokenId, address owner,
uint128 amount)
    external
    authOrManager(poolId)
{
    AssetId assetId = poolManager.assetToId(asset, tokenId);
    _noteDeposit(poolId, scId, assetId, asset, tokenId, owner, amount);
    _executeDeposit(poolId, asset, tokenId, owner, amount);
}
```

There is no `msg.sender` check for the `owner` parameter in `_executeDeposit`. This means if another pool manager still has an existing or unlimited token approval, it would be possible to deposit the funds from their `owner` address.

The balance sheet manager could afterward `withdraw` the funds of the unrelated balance sheet manager again.

Currently, it requires a permissioned call to set the balance sheet manager, and there are some trust assumptions. Nevertheless, since the protocol's goal is to be more permissionless in the long term, this should not be possible.

Recommendation: Remove the `owner` parameter for `deposit` and use `msg.sender` instead.

Centrifuge: Fixed in 9c45f4

LOW Findings

L1. ETH can be stuck in the Gateway when `_isManagerAndPaid` is called with a local centrifugeId

Hub.sol#L649

The `_isManagerAndPaid` function in the Hub contract forwards any `msg.value` to the Gateway for transaction payment:

```
function _isManagerAndPaid(PoolId poolId) internal {
    _isManager(poolId);
    _pay();
}

function _pay() internal {
    if (!gateway.isBatching()) {
        gateway.payTransaction{value: msg.value}(msg.sender);
    }
}
```

Yet, this function does not consider the case where the `centrifugeId` of the `poolId` is on the same chain. (`centrifugeId == localCentrifugeId`). In such cases, the Gateway won't be called afterward to send a cross-chain message, as the operation is performed locally.

When a transaction contains `msg.value` and the operation is local, the ETH is still forwarded to the Gateway but will never be used, causing it to be stuck in the Gateway contract.

Recommendation: Modify the `_pay` function to check if the operation is local before forwarding ETH to the Gateway:

```
function _pay(PoolId poolId) internal {
    // Only forward ETH to gateway if not local operation
    if (!gateway.isBatching() && poolId.centrifugeId() != sender.localCentrifugeId()) {
        gateway.payTransaction{value: msg.value}(msg.sender);
    }
}
```

Centrifuge: Acknowledged. In the current design, the user should not send ETH for a local chain.

xmxanuel: Acknowledged.

L2. Missing `centrifugeId` validation in `MessageProcessor` allows adapter to initiate recovery for unrelated chains

MessageProcessor.sol#L75

The `handle` function in the `MessageProcessor` processes the messages from adapters, including `InitiateRecovery` messages. The function extracts the `centrifugeId` from the message and passes it directly to the `Gateway` to initiate the recovery process.

The `recovery` features should allow adapters from one chain to recover a message from a bridge which is not working.

```
if (kind == MessageType.InitiateRecovery) {
    MessageLib.InitiateRecovery memory m = message.deserializeInitiateRecovery();
    gateway.initiateRecovery(m.centrifugeId, IAdapter(m.adapter.toAddress()), m.hash);
}
```

However, there is no validation to ensure that the `centrifugeId` in the message matches the `centrifugeId` of the chain that sent the message.

This allows an adapter for chain A to initiate recovery for chain B, which could potentially be abused in cross-chain scenarios.

The `handle` function already receives the source `centrifugeId` as its first parameter, but doesn't use it to validate the message's `centrifugeId` .

Recommendation: Add validation to ensure that the `centrifugeId` in recovery messages matches the source chain's `centrifugeId` :

The same validation should be applied to the `DisputeRecovery` message handler as well.

Centrifuge: Fixed in PR#358

xmxanuel: Fixed.

L3. Excessive privileges in `ShareToken` for `hooks`

ShareToken.sol#L80

The `shareToken` ERC20 implementation allows to set a hook contract which is called by each `_onTransfer` call.

```
/// @inheritdoc IShareToken
function file(bytes32 what, address data) external authOrHook {
    if (what == "hook") hook = data;
    else revert FileUnrecognizedParam();
    emit File(what, data);
}
```

The `file` method for changing a `hook` is `authOrHook` . This means a `hook` can change itself to another implementation or deactivate itself.

We assume each hook implementation will be carefully reviewed, however this seems like too much power for the hook implementation itself.

If the hook changes itself, it would also bypass the governance voting.

Recommendation: Use only the `auth` modifier to allow changing the `hook` implementation.

Centrifuge: Fixed in 9c45f4e

L4. `BalanceSheet.setQueue` to `false` doesn't submit existing queue and would break the correct hub transactions ordering

BalanceSheet.sol#L206

The `BalanceSheet` has a `queue` feature to avoid always triggering a bridge transaction when a balance is increased or decreased. Instead of sending the message each time, it is stored in a queue.

The functions `submitQueuedAssets` and `submitQueuedShares` send the accumulated `increase` or `decrease` to the `Hub` via the bridges.

When the queue is not activated, the messages are sent as part of the function call.

A call to the `setQueue` function can enable the queue.

```
function setQueue(PoolId poolId, ShareClassId scId, bool enabled) external auth {
    queueEnabled[poolId][scId] = enabled;
}
```

However, when the `setQueue` is set back to `false`, there is no check if `submitQueuedAssets` or `submitQueuedShares` has been called.

This would break the ordering of the hub transactions, as newer messages would be sent immediately, but older queue messages are not submitted.

Currently, the `Hub` logic can handle the missing or out-of-order messages, but the state would be inconsistent, and the hub manager needs to be aware.

Recommendation: Revert if the queue for assets or shares still has pending transactions before allowing it to be set to `false` again.

Centrifuge: Fixed in 57820a1

xmxanuel: Fixed.

Informational

I1. TransientArrayLib implementation could be optimized to match Solidity's storage pattern

TransientArrayLib.sol

The `TransientArrayLib` library implements a transient storage array using a custom storage pattern. While the implementation is functionally correct, it doesn't follow the same pattern that Solidity uses for dynamic arrays in storage, which could lead to confusion and has higher gas costs.

Solidity's native implementation for dynamic arrays computes the slot for array elements differently. For a dynamic array at storage slot `p`, Solidity:

1. Stores the array length at slot `p`
2. Stores the array elements starting at slot `keccak256(p) + i`, where `i` is the index

This is more gas-efficient than the current implementation which computes a new hash for each element access.

The current TransientArrayLib implementation:

- Stores length at `keccak256(abi.encodePacked(key, type(uint256).max))`
- Stores each element at `keccak256(abi.encodePacked(key, i))`

Recommendation: Consider refactoring the library to match Solidity's storage pattern for better gas efficiency and consistency with standard patterns:

- Store length at `keccak256(abi.encode(key))` (similar to how namespaced storage slots work)
- Store elements at `keccak256(keccak256(abi.encode(key))) + i`

References:

- Solidity Storage Layout Documentation

Centrifuge: Fixed in PR#268

xmxanuel: Fixed.

I2. Missing validation allows multiple axelarIds to be mapped to the same centrifugeId

AxelarAdapter.sol#L42-L52

The `file` function in the AxelarAdapter contract allows wards to configure mappings between Axelar chain IDs (axelarId) and Centrifuge chain IDs (centrifugeId). However, the implementation lacks validation to ensure a 1:1 relationship between these IDs.

```
function file(bytes32 what, string calldata axelarId, uint16 centrifugeId, string calldata source)
external auth {
    if (what == "sources") sources[axelarId] = AxelarSource(centrifugeId,
keccak256(bytes(source)));
    // ...
}
```

This lack of validation allows multiple Axelar chains (different axelarIds) to be mapped to the same centrifugeId.

Recommendation: Implement validation checks to ensure a 1:1 relationship between axelarIds and centrifugeIds, or maintain a reverse mapping from centrifugeId to axelarId to easily check for existing mappings.

Centrifuge: Acknowledged! This would require adding additional storage maps. Since it is fully auth and only set up once per chain, we prefer to keep it simple.

xmxanuel: Acknowledged.

I3. `Gateway.retry` not possible for `messageRecovery`

Gateway.sol#L147

If the `MessageProcessor.handle` function reverts the `Gateway.retry` allows to try to execute the message again.

However, this feature is not available for recovery messages, since the `processor_.handle` function is not called within a try/catch block. Therefore, a revert can't be cached to enable a retry later.

```
if (processor_.isMessageRecovery(payload)) {
    require(!isRecovery, RecoveryPayloadRecovered());
    return processor_.handle(centrifugeId, payload);
}
```

Recommendation: Consider if the reply functionality is needed for the `messageRecovery` .

Centrifuge: These can only trigger initiate/disputeRecovery which practically should never fail.

xmxanuel: Acknowledged.

I4. Not calling `Gateway.repay` enables multiple state edge cases

Gateway.sol#L372

The Gateway has a `repay` feature for underpaid transactions. When a transaction doesn't have enough gas, the protocol applies the state changes locally but skips the bridge transaction. Anyone can later call `repay` to execute the missing bridge transaction.

It is only required to provide the same calldata and enough gas to the `repay` function.

This opens multiple new edge cases the protocol needs to correctly handle:

- The actual bridge transaction is never executed
  - Example: Tokens are locked in the vault after a `depositRequest`, but hub contract is not aware.
- The actual bridge transaction is executed at a future point in time
  - Example: Vault is paused, `repay` could still trigger a `depositRequest`.

Not paying enough gas to produce a transaction where a `repay` call is required can be performed by any user.

Recommendation: Consider all possible edge cases carefully. Ensure that `underpaid` Gateway transactions are monitored off-chain and the `repay` call is triggered.

Another option could be to introduce a time limit to execute the `repay`.

Centrifuge: Acknowledged.

xmxanuel: Acknowledged.

I5. Potential reentrancy in `endBatching` without the `auth` modifier

Gateway.sol#L449

The `endBatching` function changes the batching state after calling untrusted adapter code, which could allow reentrancy attacks if the `auth` modifier were removed in the future.

While the function is currently protected by the `auth` modifier as good practice the `isBatching` flag could be set to `false` before calling the adapter to prevent executing the same batch again.

Recommendation: Reset the `isBatching` flag immediately after the require statement to follow the checks-effects-interactions pattern and prevent potential reentrancy issues:

```
function endBatching() external auth {
    require(isBatching, "Gateway/not-batching");
    isBatching = false; // Reset flag immediately after check
    // ... rest of function
}
```

The same checks-effects-interaction pattern could be applied to `repay` and reducing the `counter--` before calling `_send` as well. See Gateway.sol#L380.

Centrifuge: Fixed in commit bf6b8d3

xmxanuel: Fixed.

I6. Any failed `handle` call can be executed in the future again with `Gateway.retry`

Gateway.sol#L224

Any `handle` message call in Gateway that reverts can be executed again using the `Gateway.retry` feature. For example, if the initial `Gateway.handle` call didn't have enough gas, anyone could call `retry` with sufficient gas for the transaction to succeed.

However, there is also the case where a `handle` transaction correctly fails, such as with unknown message types or incorrect messages.

All these messages could be executed again with a retry. Since there is no time constraint, it could mean that after a version upgrade, a previously reverted message could theoretically pass in a new version.

Recommendation: Consider introducing a time window for the `Gateway.retry` call.

Centrifuge: We did but ultimately decided that could create more issues. It is rather up to the pool manager/sender to ensure messages don't stay unexecuted for too long.

xmxanuel: Acknowledged.

I7. `refund.recoverTokens` call in `Gateway._requestPoolFunding` assumes tokens are always transferred

Gateway.sol#L412

The `Gateway._requestPoolFunding` function calls `refund.recoverTokens` to request the needed ETH to pay for the gas. The assumption is that the `refund.recoverTokens` exactly deposited the requested `refundBalance` into the Gateway contract. This is not checked by `_requestPoolFunding` function.

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

This could lead to an incorrect subsidy tracking and a revert of bridge transaction payments since the tracking is incorrect.

Recommendation

The `refund` contract for every `poolId` needs to be a trusted contract which always transfers the requested amount or revert otherwise. In addition the `_requestPoolFunding` could check the pre and post ETH balance difference to verify it.

Centrifuge: Acknowledged. The contract implementing the `recoverTokens` is always a trusted source.

xmxanuel: Acknowledged.

I8. Gateway Underpaid gasLimit always overwriting for each underpaid transaction

Gateway.sol#L332

In case a `Gateway._send` transaction doesn't have enough `gas`, the underpaid counter is increased for the `batchHash`.

```
underpaid[centrifugeId][data.batchHash].counter++;
underpaid[centrifugeId][data.batchHash].gasLimit = batchGasLimit;
```

This allows executing the same `batchHash` again with `repay`. Therefore, the `batchGasLimit` is stored as well, which has been previously calculated by the `gasService` before calling `_send`.

```
_send(centrifugeId, poolId, message, gasService.gasLimit(centrifugeId, message));
```

However, theoretically multiple transactions could have the same `batchHash` but all would use the same most recent `gasService.gasLimit` return since `underpaid[centrifugeId][data.batchHash].gasLimit` gets overwritten.

This means if the `gasService` has changed because of an upgrade to a newer chain version this is not reflected in the stored `gasLimit`.

Recommendation: Consider this limitation in future versions of the gas service. Alternatively, a specific gasLimit could be added per underpaid transaction although it would increase the complexity.

Centrifuge: Acknowledged. This fact will be considered to return a high enough `gasService.gasLimit`.

xmxanuel: Acknowledged.