PUBLIC

Code Assessment

of the Grove xchain-helpers Smart Contracts

December 23, 2025

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help GroveLabs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Grove xchain-helpers according to Scope to support you in forming an opinion on their security risks.

GroveLabs offers a library (a fork of Spark xchain-helpers) for cross-chain messaging between Ethereum Mainnet and L2s.

This review focused on the additions to Grove xchain-helpers compared to Spark xchain-helpers commit 913ae38, introducing support for Arbitrum chains with ERC-20 gas tokens, Circle's CCTP v2, and LayerZero V2.

The most critical subjects covered in our audit are integration with the supported bridges, access control and functional correctness.

The general subjects covered are documentation and integrity.

Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Grove xchain-helpers repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	20 Aug 2025	670964a31e11ef3994f79fff4701baee25b70c9c	Initial Version
2	01 Sep 2025	a718a8cb3f9cb42f4ddedac8b78e37a09bb2d1c3	After Intermediate Report
3	27 Nov 2025	03390e2146b88ca36785f24b7a4555ccd51f82d0	LZ and CCTPv2 Integration
4	19 Dec 2025	53f6ced11ae4828e7bd55ba7a6a713c97f2b19c4	LZ and CCTPv2 Fixes

For the Solidity library contracts under review, the code is compatible with compiler version `0.8.x`.

Grove xchain-helpers is a fork of Spark xchain-helpers. The review scope covered the differences between Grove xchain-helpers's initial commit and Spark xchain-helpers commit 913ae38. The following files are in scope:

```
src/forwarders/ArbitrumERC20Forwarder.sol
src/forwarders/CCTPv2Forwarder.sol
src/forwarders/LZForwarder.sol
src/receivers/CCTPv2Receiver.sol
src/receivers/LZReceiver.sol
```

2.1.1 Excluded from scope

Generally, all contracts not mentioned above are out of scope. The correctness of the underlying bridging mechanisms is out of scope.

2.2 System Overview

This system overview describes the latest version (Version 4) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

GroveLabs offers a library (a fork of Spark xchain-helpers) for cross-chain messaging between Ethereum Mainnet and L2s. Note that these are helper contracts.

The contracts can be categorized into two types:

- **Forwarders**: Internal library contracts that act as a wrapper for underlying message-passing mechanism.
- **Receivers**: Contracts intended to be deployed on the respective chain to allow for receiving messages on L2 so that arbitrary calls can be made to a fixed target contract.

2.2.1 Forwarders

The `ArbitrumForwarder` implements:

- `sendMessageL1toL2`: Sends a message to the respective Arbitrum chain through its corresponding Inbox (invokes `createRetryableTicket`). Note that both Arbitrum One and Nova are supported (selected by passing inbox argument accordingly). Note that excess gas is burned and no L2 call value is supported.
- `sendMessageL2toL1`: Sends a message from an Arbitrum chain to Ethereum through the `ArbSys` system contract (invokes `sendTxToL1`).

The `ArbitrumERC20Forwarder` was added to enable cross-chain messaging between Ethereum and Arbitrum stack chains with non-ETH as the native token. It implements:

- `sendMessageL1toL2`: Sends a message to L2 via its ERC20Inbox by invoking `createRetryableTicket()`. Gas token (an ERC-20) is paid on Ethereum. Excess gas is burned, and L2 call value is not supported. For sending messages to Plume Network, PLUME token is used.
- `sendMessageL2toL1`: Sends a message from L2 to Ethereum through the `ArbSys` system contract (invokes `sendTxToL1`).

The `OptimismForwarder` implements:

- `sendMessageL1toL2`: Sends a message to the respective Optimism chain through its corresponding cross-chain messenger (invokes `sendMessage`). Note that multiple Optimism chains are supported (selected by passing the cross-chain messenger argument accordingly).
- `sendMessageL2toL1`: Sends a message from an Optimism chain to Ethereum through the L2-side of the cross-chain messenger (invokes `sendMessage`).

The `AMBForwarder` implements:

- `sendMessage`: Generic function that sends a message to the chain on the other side of the bridge through the arbitrary-message passing bridge (invokes `requireToPassMessage`). Note that this function supports multiple AMB bridges (selected by passing destination argument accordingly).
- `sendMessageEthereumToGnosisChain`: Specialized function routing from Ethereum to Gnosis through the Ethereum-to-Gnosis AMB.
- `sendMessageGnosisChainToEthereum`: Specialized function routing from Gnosis to Ethereum through the Gnosis-to-Ethereum AMB.

The `CCTPForwarder` implements:

- `sendMessage` (two variants: address as `address` or as `bytes32`): Sends a message to the chain with the given domain ID through Circle's CCTP bridge (invokes `sendMessage`). Note that this function supports multiple CCTP bridges (selected by passing the destination ID accordingly).

The `CCTPv2Forwarder` was added to support Circle's CCTP v2 protocol. It implements:

- `sendMessage` (two variants: address as `address` or as `bytes32`): Sends a message to the chain with the given domain ID through Circle's CCTP v2 message transmitter. The forwarder uses standard finality (`minFinalityThreshold = 2000`) and allows any caller to relay (`destinationCaller = 0x0`). Supports numerous chains including Ethereum, Avalanche, Optimism, Arbitrum One, Base, Polygon PoS, Unichain, Linea, Sonic, Worldchain, Sei, BSC, HyperEVM, Ink, and Plume.

The `LZForwarder` was added to support LayerZero V2 cross-chain messaging. It implements:

- `sendMessage`: Sends a message to a destination endpoint through the LayerZero V2 protocol. The function quotes the fee, optionally pays in LZ token, and sends the message via the endpoint. Supports Ethereum, Avalanche, Base, BNB Chain, Monad, and Plasma.
- `configureSender`: Configures the contract as a LayerZero sender for a specific remote endpoint, setting up the ULN configuration (DVNs, confirmations) and executor configuration (max message size, executor address). Two variants exist: a full version with all parameters and a simplified version with defaults.

2.2.2 Receivers

Receivers can receive cross-chain messages. Typically, they receive a call from the integrated bridge, validate the origin (access control) and perform a call on a given target contract. The following receivers are implemented:

- `ArbitrumReceiver`: Implements a `fallback` function. The `msg.sender` with undone alias is validated to be the expected sender's address on L1. The call is then simply forwarded to the target contract. Supports Arbitrum chains (e.g. Arbitrum Nova and One). This receiver is also used for `ArbitrumERC20Forwarder`.
- `OptimismReceiver`: Implements a `fallback` function. The `msg.sender` is expected to be the L2 side of the cross-domain messenger while the `xDomainMessageSender` is expected to be the expected sender's address on L1. Supports Optimism chains (e.g. Optimism, Base).
- `AMBReceiver`: Implements a `fallback` function. The `msg.sender` is expected to be the L2 side of the arbitrary-message passing bridge while the `messageSourceChainId` and `messageSender` are expected to match the expected origin chain and sender's address. Supports chains with AMB bridges (e.g. Gnosis).
- `CCTPReceiver`: Implements the CCTP recipient's `handleReceiveMessage` function. The `msg.sender` is expected to be the destination CCTP messenger while the source domain and sender are expected to match the expected source domain and authority. Supports chains where CCTP is deployed.
- `CCTPv2Receiver`: Implements CCTP v2's recipient interface. The `handleReceiveFinalizedMessage` function validates that the caller is the destination messenger, the source domain matches, and the sender matches the expected source authority. It then forwards the message body to the target contract. Unfinalized (fast) messages are rejected via `handleReceiveUnfinalizedMessage`.
- `LZReceiver`: Extends LayerZero's `OAppReceiver` to receive cross-chain messages. Upon deployment, it configures itself as a receiver by setting the peer and ULN configuration. The `_lzReceive` function validates the source endpoint ID and sender, then forwards the message to the target contract with any attached value. The `allowInitializePath` function restricts initialization to the expected source.

2.2.3 Roles and Trust Model

Bridges are fully trusted. The usage of the Forwarder libraries is expected to be correct. The receivers are expected to be deployed only on supported chains. The deployment of receivers is expected to be set up correctly (e.g. targets are correct). Further, connecting forwarders with receivers requires a careful setup of messages sent.

The following additional privileged roles exist for Arbitrum stack chains with ERC-20 gas tokens:

- **Bridge admin**: Fully trusted. Can upgrade the bridge contract to DoS cross-chain messaging or drain the gas tokens approved.
- **Rollup contract and its owner**: Semi-trusted. Can pause the bridge or turn on the allowlist to block certain cross-chain messaging in the worst case.

The L2 bridge is trusted to work honestly and correctly as documented.

For LayerZero:

- **DVNs (Decentralized Verifier Networks)**: The configured DVNs are trusted to honestly verify cross-chain messages. The `LZReceiver` requires specific DVNs to be configured at deployment.
- **LayerZero Endpoint**: Fully trusted to correctly route and deliver messages.
- **Delegate/Owner**: The `LZReceiver` has an owner and delegate that can modify OApp configuration.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	2

- Forwarder and Receiver Mismatch in Configurability of LZ Settings Code Corrected
- Dangling Approval Code Corrected

6.1 Forwarder and Receiver Mismatch in Configurability of LZ Settings

Informational Version 3 Code Corrected

CS-GRVXCH-004

On the Forwarder side (`LZForwarder`) the `confirmations`, `optionalDVNs` and `optionalDVNTreshold` can be configured through function arguments, whereas on the Receiver side (`LZReceiver`) they are hardcoded.

There are two `configureSender` functions on the Forwarder side, one where the `confirmations`, `optionalDVNs` and `optionalDVNTreshold` are hardcoded, and one version where these values can be manually set through function arguments.

On the Receiver side there is only one `_configureReceiver` function, in which these values are hardcoded.

This creates an inconsistency where the Forwarder supports configurations that the hardcoded Receiver cannot match without manual modifications.

Code corrected:

In the updated code, the LZReceiver's UlnConfig settings are no longer hardcoded. The constructor now accepts a `UlConfigParams` struct, making `confirmations`, `optionalDVNs` and `optionalDVNThreshold` configurable.

6.2 Dangling Approval

Informational Version 1 Code Corrected

CS-GRVXCH-001

When creating a retryable ticket with the ERC20Inbox contract, it will only pull extra tokens required from the `msg.sender`. In case the gas token balance of ERC20Inbox is positive, the allowance approved

during `sendMessageL1toL2()` will not be used up, leading to dangling approval from the `msg.sender` to ERC20Inbox.

Code corrected:

At the end of `sendMessageL1toL2()`, the gas token allowance is now explicitly reset to 0, ensuring no dangling approval remains.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 `configureSender` Makes Two Separate External Calls to `setConfig` CODECORRECTED

Informational **Version 3**

CS-GRVXCH-002

The `configureSender` function invokes `ILayerZeroEndpointV2.setConfig` twice, once for the ULN configuration and once for the Executor configuration:

```
ILayerZeroEndpointV2(endpoint).setConfig(address(this), sendLib, ulnConfigParams);
// ...
ILayerZeroEndpointV2(endpoint).setConfig(address(this), sendLib, executorConfigParams);
```

The `setConfig` function accepts an array of `SetConfigParam` structs, allowing both configurations to be applied in a single external call.

Code corrected:

The ULN and Executor configurations are now applied in a single call by passing both `SetConfigParam` entries in one array.

7.2 `confirmations` Parameter Type Mismatch With LayerZero `UlnConfig` Struct

Informational **Version 3**

CS-GRVXCH-003

The `configureSender` function declares the `confirmations` parameter as `uint32`, however, LayerZero's `UlnConfig` struct defines `confirmations` as `uint64`. The code functions correctly due to implicit upcasting from `uint32` to `uint64`, but the type inconsistency deviates from the LayerZero interface. In practice, this has no effect since realistic confirmation counts are orders of magnitude smaller than the `uint32` maximum.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Fee Calculation Trusts Bridge Endpoints

Note Version 3

The LZForwarder and both Arbitrum forwarders trust their respective bridge endpoints for the fee amount returned (`ILayerZeroEndpoint.quote()` and `ICrossDomainArbitrum.calculateRetryableSubmissionFee()`). These amounts are sent directly to the bridge without any validation, using funds from the executing contract's balance.

Since the xchain-helpers already require full trust in these bridges for cross-chain message passing, this is only a note to highlight and raise awareness to integrators of this library.

8.2 LayerZero Configuration Considerations

Note Version 3

LayerZero integrations require proper configuration to work correctly. In general, this includes:

- Correctly setting peers (and their configurations) on each chain. Thus, for `LZReceiver`, the origin must be configured correctly. For users of `LZForwarder`, the setting is specific to the library's usage.
- Correctly setting a DVN configuration, including optional settings such as block confirmations, security threshold, the Executor, max message size, and send/receive libraries. If no send and receive libraries are explicitly set, the Endpoint will fall back to the default settings set by LayerZero Labs. In the event that LayerZero Labs changes the default settings, the integration will be impacted and use the new default settings, which implies trust in LayerZero Labs. For `LZReceiver`, the configuration for receiving messages should be considered. For users of `LZForwarder`, the configuration for sending messages (and potentially more) should be considered.

8.3 LayerZero Initial Trust Model

Note Version 3

The trust in LayerZero can be minimized by setting custom configurations. The LZReceiver contract implements `_configureReceiver()` to atomically configure custom DVNs during deployment, eliminating the initial trust window.

However, one trust assumption remains due to the default receiver library being queried from the LayerZero-controlled endpoint.

In `_configureReceiver()`, the receive library is obtained by querying the endpoint:

```
( address receiveLib, ) = endpoint.getReceiveLibrary(address(0), srcEid);
endpoint.setConfig(address(this), receiveLib, configParams);
```

This creates a trust assumption because:

1. The `LZReceiver` queries the DEFAULT receive library from the LayerZero-controlled endpoint during deployment.
2. LayerZero governance controls what `endpoint.getReceiveLibrary(address(0), srcEid)` returns and could provide a malicious library address.
3. Even though custom DVNs are configured for this library, a malicious library can ignore the DVN configuration and forge verifications in `EndpointV2.verify`.
4. The forged message can then be executed with `lzReceive`.

Note that similarly, this issue exists in the `LZForwarder` library where `configureSender()` queries the default send library from the Endpoint.

8.4 LzToken Can Be Changed in LayerZero EndpointV2

Note **Version 3**

When paying fees in `lzToken`, the LZForwarder dynamically queries the token address from EndpointV2. Since the EndpointV2 owner can change `lzToken` to any address, users calling `sendMessage()` should be aware that the token could potentially be switched to a more valuable asset right before execution, resulting in unexpected losses.

In contrast, other LayerZero bridges in the Sky ecosystem implement `lzToken` as a configurable parameter that can be updated by privileged roles.