PUBLIC

Code Assessment

of the xchain-helpers Smart Contracts

October 08, 2025

Produced for

Spark

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

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

SparkDAO implements a library for cross-chain message passing along with contracts able to receive cross-chain messages. The latest version reviewed adds support for LayerZero.

The most critical subjects covered in our audit are integration with the supported bridges, access control and functional correctness. The general subjects covered are unit testing, documentation and trustworthiness. 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	1
• Code Corrected	1

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 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	13 July 2024	e1f9443cfa7c2b13771c32cd4c89f21576210bb6	Initial Version
2	27 August 2025	610cede5a97fc07bdfc77fcfa95d8c486d194046	v1.1.0-beta.0
3	08 September 2025	80e689e4f716d03a5fdb45755bc0976a7b90f8a2	v1.1.0-rc.0
4	08 October 2025	33baf1e96833cb0d1de0ca57c115459b76565fd4	v1.1.0

For the Solidity code under review, the implementation is compatible with compiler versions `0.8.x`.

The contracts below are in scope:

```
forwarders:
    AMBForwarder.sol
    ArbitrumForwarder.sol
    CCTPForwarder.sol
    OptimismForwarder.sol
receivers:
    AMBReceiver.sol
    ArbitrumReceiver.sol
    CCTPReceiver.sol
    OptimismReceiver.sol
```

The following files were added in Version 2:

```
forwarders:
    LZForwarder.sol
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 received 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.

SparkDAO offers a library 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 `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).

In Version 2, the `LZForwarder` was added which implements:

- `sendMessage`: Generic function that sends a message to the chain with the given `dstEid` through LayerZero's V2 messaging protocol (invokes `send` on the LayerZero endpoint passed as parameter). Since Version 4, the fee can be paid on both; native and lzToken.

2.2.2 Receivers

Receivers can receive cross-chain messages. Typically, they receive a call from the integrated with 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).
- `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 recipients `handleReceiveMessage` function. The `msg.sender` is expected to be the deployed on chains side of the messenger while the source domain and sender are expected to match the expected values. Supports chains where CCTP is deployed.

In Version 2, the `LZReceiver` was added which implements a `lzReceive` function. The `msg.sender` is validated to be the configured LayerZero endpoint while the `origin` param is checked to match the configured `srcEid` and `sender`. The call is then forwarded to the target contract. In Version 3, the `LZReceiver` contract was updated to inherit from `OApp`. Note that this exposes functionality for delegatee and peer configuration, however the accepted peer configuration is hardcoded in `LZReceiver`. Any other configured peers will lead to reverts. This inheritance introduces an owner through OpenZeppelin's `Ownable`. In Version 4, the `LZReceiver` was updated to inherit from OAppSender to expose the correct `oAppVersion`.

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.

Note that for LayerZero the trust model depends on the configuration. LayerZero in general is considered fully trusted as long as not all configuration parameters are set. After that the DVN is trusted not to misbehave. Further, the off-chain infrastructure as well as the on-chain components are expected to work correctly. Additionally, see LayerZero Initial Trust Model. Note that the owner and delegate of the `LZReceiver`, the roles introduced as part of Version 3, are fully trusted. Additionally, the peer configuration is expected to be set correctly. Note that `LZReceiver` hardcodes the accepted peer sending the message; a misconfiguration of the peer blocks the receiver contract.

Since Version 4, the `LZForwarder` supports paying lzToken as fees. The lzToken address is queried from the LayerZero v2 Endpoint, which is fully trusted. In the worst case, a malicious endpoint could substitute a different token address, causing call to pay fees in an another unexpected token.

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:

- Correctness: Mismatches between specification and implementation

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	1

- Incomplete ILayerZeroReceiver Interface Implementation Code Corrected

6.1 Incomplete ILayerZeroReceiver Interface Implementation

Correctness Low Version 2 Code Corrected

CS-SPRK-XCH-001

`LZReceiver` does not fully implement the `ILayerZeroReceiver` interface as defined by LayerZero V2. Specifically, it's missing the `nextNonce()` function:

```
function nextNonce(uint32 _eid, bytes32 _sender) external view returns (uint64);
```

According to the LayerZero documentation, this function returns the next expected nonce for messages from a specific source endpoint and sender. While the function can return 0 to indicate no ordering enforcement, it must still be present in the implementation to satisfy the interface requirements.

The missing function serves to:

- Indicate whether the OApp expects ordered message execution
- Allow off-chain executors to determine ordering requirements
- Return 0 by default if ordering is not enforced

For a more complete interface, consider implementing the `OAppReceiver` interface which extends `ILayerZeroReceiver` with additional function definitions like `isComposeMsgSender()` and `oAppVersion()`.

Code corrected:

In the updated code, `LZReceiver` was changed to inherit from `OApp` which properly implements the `ILayerZeroReceiver` interface including the missing `nextNonce()` function.

Note that `OApp` inherits from both `OAppSender` and `OAppReceiver`. Since this is a receiver only contract inheriting from `OAppReceiver` directly may be more appropriate. Both `OAppSender` and `OAppReceiver` inherit from `OAppCore` (which provides the peer configuration functionality). The main difference is that when inheriting from `OAppReceiver`, `oAppVersion()` returns (0, 2), correctly indicating no sender functionality, while inheriting from `OApp` causes `oAppVersion()` to return (1, 2), suggesting sending capability that isn't used.

Since Version 4 `LZReceiver` inherits from `OAppReceiver` to return the correct `oAppVersion()`.

7 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.

7.1 Fee Calculation Trusts Bridge Endpoints

Note Version 2

The LZForwarder and ArbitrumForwarder trust their respective bridge endpoints for the fee amount returned (`ILayerZeroEndpoint.estimateFees()` 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.

7.2 LayerZero Configuration Considerations

Note Version 2

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.

7.3 LayerZero Initial Trust Model

Note Version 2

The trust in LayerZero can be minimized by setting custom configurations. However, the respective contracts' deployment process might allow for temporarily trusting LayerZero governance fully (while no configurations have been made).

Consider the following example:

1. The `LZReceiver` is deployed with correct construction parameters on the receiving chain, whitelisting the expected origin as a peer.
2. LayerZero v2 can now manipulate the default receiver libraries to trivially allow for forged messages in `EndpointV2.verify`.
3. The message can now be executed with `lzReceive`.

As long as the receiver library configuration is not fully configured, such manipulations could occur.

To summarize, due to the time window between deployment and the actual configuration, LayerZero could in theory attack the `LZReceiver`. Note that similarly, such potential exists for users of the `LZForwarder` library depending on the contract using the library.

7.4 LayerZero V2 Considerations

Note **Version 2**

Users of the LayerZero forwarder and receiver should be aware of the considerations below:

- *Execution Order*: According to the design of LayerZero V2, the delivery of messages on the destination is not guaranteed to be in the same order as they were dispatched on the source. Consequently, in case multiple sends are relayed to the same destination, they might be reordered and lead to unexpected results.
- *Censorship*: Denial-of-Service and censorship are possible since it is not guaranteed the DVN will verify the send messages in time.
- *Sandwiching*: The execution of receiving messages is permissionless. Hence, anyone can trigger the execution once the message is verified. Consequently, the reception can be sandwiched by an attacker with other operations for MEVs or attacks with flashloans in particular.
- *Refund*: When sending messages a refund address can be provided. The library LZForwarder hardcodes the refund address to `msg.sender`. This refund target should be able to receive the refund (e.g. by implementing `receive()` or `fallback()` if it is a contract or an EOA using EIP-7702). Otherwise, sending may revert if there is a refund.
- *Alternative Native Token*: LayerZero implements endpoints with alternative native tokens (e.g. EndPointV2Alt for EVM chains) where the native token has no significant value. Note that chains with such endpoints are unsupported.

7.5 LzToken Can Be Changed in LayerZero EndpointV2

Note **Version 4**

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.