PUBLIC

Code Assessment

of the Sky LZ Governance Relay Smart Contracts

June 23, 2026

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

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

Sky implements governance relay contracts leveraging the sky-oapp-governance bridge to relay governance calls from L1 to L2s.

The most critical subjects covered in our audit are access control and functional correctness. Security regarding all the aforementioned subjects is high.

The general subjects covered include documentation and usage considerations. The report contains four notes and one informational finding related to defense in depth.

In summary, we find that the codebase provides a high 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 following smart contracts were in scope:

```
src/
    L1GovernanceRelay.sol
    L2GovernanceRelay.sol
deploy/
    GovernanceRelayInit.sol
    GovernanceRelayDeploy.sol
```

The assessment was performed on the source code files inside the Sky LZ Governance Relay 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	24 Sep 2025	d3e3df4db417f196fdd56123e7dbb462d04f32ef	Initial Version
2	22 Jun 2026	8718d1c269703144234fcd9049eaff2c4eb787da	Relay Timelock

For the solidity smart contracts, the compiler version `0.8.22` was chosen and `evm_version` was set to `shanghai`.

2.1.1 Excluded from scope

Any other files are not in scope of this review.

The Governance Relay is expected to work with Sky LayerZero Governance OApp, which was covered in another review by ChainSecurity.

LayerZero V2 is out of scope and assumed to function correctly according to its documentation.

2.2 System Overview

This system overview describes the latest version (Version 2) of the contracts as defined in the Assessment Overview.

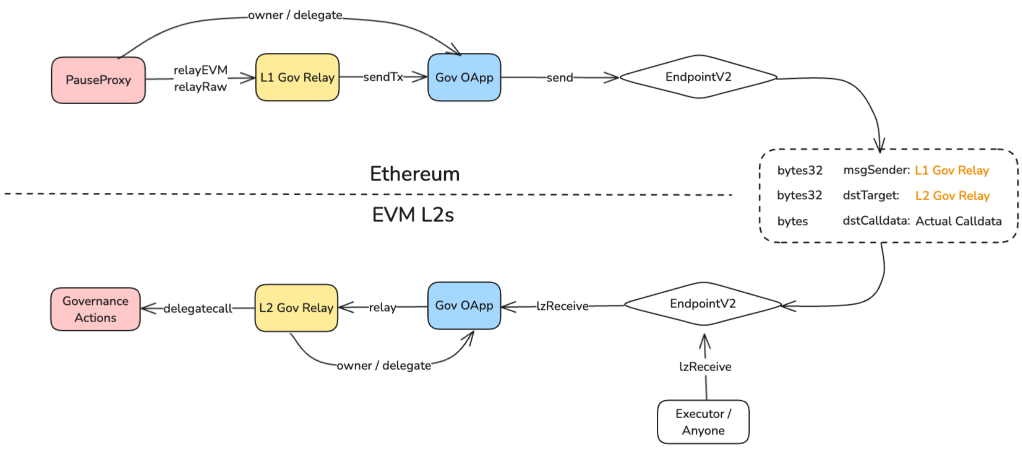
Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Sky offers Governance Relay contracts that works with Sky LayerZero Governance OApps to relay governance messages cross-chain.

2.2.1 Cross-chain Governance Overview

The Governance Relay contracts provide a generic cross-chain governance framework to relay a governance action from L1 to L2. An overview of the cross-chain governance workflow is outlined below:

1. *L1 Governor*: Instructs the L1 Governance Relay to send a message to a foreign domain.
2. *L1 Governance Relay*: Encodes message and instructs the L1 Governance OApp to send a message on its behalf. Fee tokens are attached or approved.
3. *L1 Governance OApp*: Performs authorization check and instructs the LayerZero V2 Endpoint to send the message.
4. *LayerZero V2*: Handles the message. Eventually, the message is verified on the LayerZero V2 endpoint, marking its validity. Note that this step includes on- and off-chain components.
5. *Destination Chain OApp*: The execution of the cross-chain message is triggered, which instructs the L2 Governance Relay to relay the message to the governed contract.
6. *L2 Governance Relay*: Performs validation of caller and source, then queues the governance action with a timelock. After the configured `delay` elapses, and within the `gracePeriod` window, anyone may execute it, which triggers a delegatecall to perform the actual governance action. Whitelisted guardians may veto queued actions before execution.



2.2.2 L1GovernanceRelay

The L1GovernanceRelay follows the standard deny and rely authorization mechanism. To initiate a cross-chain governance message, the following payable entrypoints can be used by the wards:

- `relayEVM()`: Relays a message to another EVM-compatible chain. The message fields are provided in detail which includes the foreign contract address to be delegatecalled with calldata. The message is encoded into a `TxParams` and sent with `l1Oapp`.
- `relayRaw()`: Relays a message to any chain with off-chain encoded `TxParams`. This is expected to be used to send to a non-EVM-compatible chain.

Governance may choose to pay fees in native token or lzToken in case it is configured. Further, a refund address can be provided. To receive a potential native token refund, a payable `receive()` fallback is implemented.

Further, the following setters are provided for the wards:

- `file()`: Sets the `lzToken` and the `l1Oapp` addresses in case of a change or upgrade.
- `reclaim()`: Sends specified amount of native token to a specified receiver.
- `reclaimLzToken()`: Sends specified amount of `lzToken` to a specified receiver.

2.2.3 L2GovernanceRelay

The L2GovernanceRelay works on-behalf-of the L1 Governance to govern other L2 contracts. It is expected to be fully setup on construction to allow self-configuration. Inbound messages are not executed immediately but queued behind a timelock, and whitelisted guardians (`bud`) may veto them. It exposes the following entrypoints:

- `relay()`: Validates that it is being called by the `l2Oapp` and the message is sent from the L1 Governance Relay at the immutable `l1Eid`. Then it queues a new action with `executionTime = block.timestamp + delay`.
- `exec()`: Permissionless. Once an action is `Ready` (after `delay` and within `gracePeriod`), executes it with a delegatecall according to the message. Spells are assumed stateless and must treat `msg.sender` as untrusted.
- `getActionState() / getActionById()`: View the state (`Queued`, `Ready`, `Executed`, `Canceled` or `Expired`) and content of an action. `delay` is snapshotted per action at queue time, while `gracePeriod` is read live.
- `cancel()`: Callable by a guardian (`bud`). Advances a monotonic checkpoint that vetoes every unexecuted action with `id <= canceledId_`, so a single guardian can cancel the whole queue.
- `skip()`, `nilify()`, `burn()` and `clear()`: Callable by a guardian (`bud`). Thin forwarders to the LayerZero Endpoint to unstick or suppress the inbound message channel when an unverifiable message stalls the queue. For these the relay must be the `delegate` of `l2Oapp` on the Endpoint.
- `file()`: Can only be called by itself to set the `l2Oapp`, `l1GovernanceRelay`, `delay` or `gracePeriod` (which must stay `>= MINIMUM_GRACE_PERIOD`). Namely the execution needs to start from `relay()` where the `messageAuth` is checked.
- `kiss()` / `diss()`: Can only be called by itself to add or remove a guardian from the `bud` whitelist.

2.2.4 Deployment

The L1 Governance Relay is expected to be deployed with the only ward switched to a specified owner (e.g. the Pause Proxy). And the L2 Governance Relay is expected to be deployed with correct configuration.

The L1 Governance Relay will be initialized with the `l1Oapp` while leaving the `lzToken` as `address(0)`. Namely sending a message with `lzToken` fees is prevented. It is further added to the chainlog with the key "LZ_GOV_RELAY". The L2 Governance Relay does not need any initialization, as its configuration (`l1Eid`, `l2Oapp`, `l1GovernanceRelay`, `delay`, `gracePeriod` and an initial `bud` guardian set) is seeded on construction.

2.3 Trust Model

Wards of L1GovernanceRelay: fully trusted; assumed to be the L1 governance contract (e.g. Pause Proxy) who has full control over the contract. It is expected all the cross-chain messages are carefully inspected and analyzed.

Guardians (``bud``) of L2GovernanceRelay: partially trusted; whitelisted addresses that can veto queued actions via `cancel` and unstick or suppress the inbound LayerZero channel via `skip`, `nilify`, `burn` and `clear`. A single guardian can cancel the entire in-flight queue or destroy inbound messages, and is therefore a potential denial-of-service vector for remote governance. Guardians cannot, however, execute or alter governance actions.

Governance OApps: fully trusted; assumed to be correctly configured, otherwise, cross-chain message can be blocked due to missing peer or authorization.

LayerZero: LayerZero is out of scope for this review and is trusted to behave correctly and deliver messages to the correct destination, the LayerZero executor is trusted to always deliver messages with the correct provided message value and gas limit. Depending on the exact configuration, LayerZero might be fully trusted to not censor messages or similar. However, the configuration is out-of-scope for this review.

The governed contracts would be in danger in case LayerZero behaves incorrectly, for example due to a compromise, an L2 finality issue, or any other failure. Because the `l2Oapp.lzReceive()`

function cannot be paused in emergency, compromised DVNs and executors may verify and execute unintended or malicious governance calls.

In the context of Sky Governance, it is assumed:

1. On L1, the PauseProxy is the ward of L1 Governance Relay.
2. On L1, the PauseProxy is the owner and delegate of the L1 GovernanceOAppSender to setup the `canCallTarget` mapping and LayerZero related configurations.
3. On L2, the L2 Governance Relay is the owner and delegate of the L2 GovernanceOAppReceiver to setup LayerZero related configurations.

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 our findings. 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 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.

6.1 `srcSender` May Contain Garbage Leading Bytes

Informational **Version 1** **Acknowledged**

CS-SLGR-001

The L2GovernanceRelay's `messageAuth` modifier verifies that the message originates from the L1GovernanceRelay and when casting the bytes32 `srcSender` to an address, it ignores the leading 12 bytes. Consequently, if the message is malformed (i.e. by a compromised DVN), it will still be accepted. Note that a legitimate L1 GovernanceOAppSender cannot send such a malformed message.

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 Guardians Can Suppress Individual Inbound Messages

Note Version 2

`nilify()`, `burn()` and `clear()` each act on a single inbound message identified by its nonce, leaving other messages in the channel unaffected: `burn()` permanently deletes an already-committed message, `nilify()` blocks a verified message from executing until it is re-verified, and `clear()` consumes a verified message as if it had been delivered without invoking the receiver, so its governance action is dropped. A guardian (`bud`) can thus suppress one specific governance action while earlier and later ones continue to be delivered. `skip()` is more limited, acting only on the next contiguous nonce, and cannot target an arbitrary message. This differs from the contract's `cancel()`: `cancel(canceledId_)` vetoes every queued action up to and including `canceledId_` and therefore cannot drop a single action in isolation, whereas these endpoint functions target a message by its nonce.

7.2 Increasing `gracePeriod` Can Revive Expired Actions

Note Version 2

`getActionState()` reads `gracePeriod` live rather than snapshotting it per action, so a `file("gracePeriod", ...)` retroactively shifts the expiry of every action already in the queue. Consequently, increasing `gracePeriod` can return an action that had already expired from the `Expired` state back to `Ready`, after which it can again be executed through the permissionless `exec()`.

7.3 LzToken Is Not Set After Initialization

Note Version 1

When the L1GovernanceRelay is initialized, the `lzToken` address is not set, hence calling `reclaimLzToken()` or sending messages with `lzToken` as fees will revert.

7.4 Msg.Value Is Not Accepted by L2GovernanceRelay

Note Version 1

The L2GovernanceRelay does not implement a `receive()` function to receive native token, and `relay()` is not payable. Consequently if a cross-chain message requires certain `msg.value` to be attached or delivered, it is not accepted by the L2GovernanceRelay.