PUBLIC

Code Assessment

of the Governance Relay Smart Contracts

July 31, 2024

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 Governance Relay according to Scope to support you in forming an opinion on their security risks.

SparkDAO implements a relay for governance actions that allows for the execution of governance proposals across chains.

The most critical subjects covered in our audit are access control and functional correctness. The general subjects covered are unit testing, specification and trustworthiness. Security regarding all the aforementioned subjects is high.

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 assessment was performed on the source code files inside the 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	03 July 2024	b6ce510c899f1d582b03dade9222e2db8c81f20e	Initial Version
2	31 July 2024	5ed105fd33b5524cb6e493e9fdc56985c4a2391a	Removal of action queuing

For the solidity smart contracts, the compiler version `0.8.25` was chosen. The files below were in scope:

```
src:
    Executor.sol
    interfaces:
        IExecutor.sol
```

2.1.1 Excluded from scope

Generally, all files not mentioned above are out of scope. The configuration is out of scope. The bridges are out of scope.

Note that the codebased is based on Aave's governance bridges. While the full codebase was reviewed, efficiency, design choices and other parts not affecting security are out of scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) 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 contract that allows for processing cross-chain execution of governance proposals.

2.2.1 General Overview

Governance will make decisions on Ethereum Mainnet. Some of these will require execution on other chains (e.g. Arbitrum). The `Executor` contract will allow for receiving cross-chain messages through bridges so that the governance actions can be executed.

More specifically, the expected setup and execution flow is the following:

1. Governance sends a message to the desired chain with a dedicated forwarder using the chain-specific library from the `xchain-helpers` library.

2. The message is send through the respective bridge.
3. The corresponding chain-specific receiver contract (defined in `xchain-helpers` aswell) forwards the payload to the `Executor`.
4. The governance actions are queued and will eventually be executed.

2.2.2 Executor

The `Executor` receives the cross-chain messages from a dedicated receiver. Namely, the `queue` function should be the entrypoint for receiving messages. Thus, the payload generation (not part of the scope) should create messages so that `queue` is invoked. When queuing messages for execution, a set of actions is published. Each such action consists of the following parameters:

- Target: Target of the call.
- Value: Value of the call.
- Function Signature: Function signature (as string)
- Call data: The arguments passed to the call with the function signature (however, it might include the function selector if the function signature is an empty string).
- Call or Delegatecall: Flag whether the action should be executed with a call or delegatecall.
- Execution time: The timestamp at which the action can be executed. That is the timestamp of `queue` plus the currently set `delay`.

The set of actions is assigned an ID under which all actions are stored in the submitted order. Note that each action is marked as queued. No identical action can be queued (e.g. duplicate action submitted in the set of actions). Further, note that the action set expires after the `gracePeriod` has passed (after earliest execution time).

Once the `delay` has passed and if the set of actions has not expired, been already executed or cancelled, arbitrary users can execute the actions as a batch with `execute`. That will mark the set as executed and proceed executing the individual actions which dequeues the individual action. Then, it operates in one of the two modes (depending on the call type flag):

1. Call: Perform a call directly on the target contract (with the given parameters).
2. Delegatecall: Perform a call to `this.executeDelegateCall` with the value sent along with the self-call (so that the target can safely use `msg.value`) and the target and calldata passed as arguments. That, ultimately, will delegatecall into the target with the corresponding calldata.

Note that if one execution reverts, the full transaction will revert.

Queued transactions can not only be executed but also can be cancelled by a guardian with `cancel` (typically that should occur while the execution time has not been reached). That will tag the action set as cancelled and will dequeue each action.

Further, the contract offers functions callable only by a self-call through `execute` to faciliate updating local parameter:

1. `updateDelay`: Updates the delay and is only callable by governance. Expected to be callable only by self-call through `execute()`.
2. `updateGracePeriod`: Updates the grace period and is only callable by governance. Expected to be callable only by self-call through `execute()`.

Additionally, `receiveFunds` is implemented which can receive the native token. `getActionsSetById` and `getCurrentState` return an array of actions and the current status of the set, respectively.

2.2.3 Changes in Version 2

Queuing of actions has been removed (i.e. `isActionQueued`). Note that it served as a duplicate detection mechanism. However, only for a batch of actions.

2.2.4 Roles and Trust Model

The following roles are defined:

1. Governance on L1: Fully trusted. It can govern the protocol on L2 through the governance contract (potentially) arbitrarily.
2. Bridges and related contracts: Fully trusted. This includes not allowing to re-enter the message relay. Further, replay protection for messages is expected to happen either in the receiver or in the bridge itself (common case). In case of a malicious bridge, a governance takeover on L2 could occur.
3. Governed contracts: Expected to work correctly.
4. `GUARDIAN_ROLE`: Trusted. The guardian is trusted to behave honestly. Otherwise all proposals could get cancelled.
5. `SUBMISSION_ROLE`: Fully trusted. Can queue arbitrary messages. Expected to be the corresponding receiver from `xchain-helpers`.
6. `DEFAULT_ADMIN_ROLE`: Fully trusted. Expected to only be self after the setup (deployer expected to revoke his privileges).
7. Users: Untrusted.

Note that the deployer is given the default admin role. After the setup is completed but before the integration into the system, the deployer is expected to revoke their permissions.

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 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 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.

6.1 Guardian Powers

Note Version 1

Typically, a guardian is a privileged address that can execute a specific set of governance actions. For the governance relay, that corresponds to canceling queued actions. That implies that a guardian can cancel the revocation of their role. Thus, a guardian could DoS governance on an L2 for arbitrary time - however, at the cost of gas required for the cancellation.

Users and governance should be aware of such a possibility.