Security Audit Report

Axelar mvx-governance

MultiversX smart contract

by ARDA

on April 21, 2025

Table of Contents

Disclaimer

The report makes no statements or warranties, either expressed or implied, regarding the security of the code, the information herein or its usage. It also cannot be considered as a sufficient assessment regarding the utility, safety and bugfree status of the code, or any other statements.

This report does not constitute legal or investment advice. It is for informational purposes only and is provided on an "as-is" basis. You acknowledge that any use of this report and the information contained herein is at your own risk. The authors of this report shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Terminology

Code: The code with which users interact.

Inherent risk: A risk for users that comes from a behavior inherent to the code's design.

Inherent risks only represent the risks inherent to the code's design, which are a subset of all the possible risks. **No inherent risk doesn’t mean no risk.** It only means that no risk inherent to the code's design has been identified. Other kind of risks could still be present. For example, the issues not fixed incur risks for the users, or the upgradability of the code might also incur risks for the users.

Issue: A behavior unexpected by the users or by the project, or a practice that increases the chances of unexpected behaviors to appear.

Critical issue: An issue intolerable for the users or the project, that must be addressed.

Major issue: An issue undesirable for the users or the project, that we strongly recommend to address.

Medium issue: An issue uncomfortable for the users or the project, that we recommend to address.

Minor issue: An issue imperceptible for the users or the project, that we advise to address for the overall project security.

Objective

Our objective is to share everything we have found that would help assessing and improving the safety of the code:

1. The **inherent risks** of the code, labelled R1, R2, etc.
2. The **issues** in the **code**, labelled C1, C2, etc.
3. The **issues** in the **testing** of the code, labelled T1, T2, etc.
4. The **issues** in the **other** parts related to the code, labelled O1, O2, etc.
5. The **recommendations** to address each issue.

Audit Summary

Initial scope

- **Repository:** https://github.com/multiversx/sc-axelar-cgp-rs
- **Commit:** `4b05e701ae050b6d10e936e43c289413d901a585`
- **MultiversX smart contract path:** ./governance/

Final scope

- **Repository:** https://github.com/multiversx/sc-axelar-cgp-rs
- **Commit:** `b863a1ba7fe8180e63961f721a63c6d53d818137`
- **MultiversX smart contract path:** ./governance/

1 inherent risk in the final scope

0 issue in the final scope

6 issues reported in the initial scope and 0 remaining in the final scope:

Severity	Reported			Remaining		
	Code	Test	Other	Code	Test	Other
Critical	0	0	0	0	0	0
Major	1	0	0	0	0	0
Medium	4	1	0	0	0	0
Minor	0	0	0	0	0	0

Inherent Risks

R1: The minimal time lock for proposals might be set to harmful values.

This is because Axelar validators are allowed to set the minimal time lock to any value, without any kind of protections, which could be harmful:

- If the minimal time lock is too short, e.g. 0 seconds, then it would allow executing some proposals as soon as they are submitted, leaving no time for users to react to the implications of the proposal, or for cancelling the proposal in case it was malformed or unwanted.
- If the minimal time lock is too long, e.g. 1000 years, then no governance proposals can ever be executed.

In practice however, the risk is low, because the choice of the minimal time lock results from a consensus between multiple validators of the Axelar blockchain.

Finally, the reason why Axelar does not implement a protection at the smart contract level is to follow the same convention as all the Governance smart contracts deployed on other chains.

Code Issues & Recommendations

C1: Executor of governance proposal does not get his EGLD back if the execution fails

Severity: Major **Status:** Fixed

Location

```
governance/src/lib.rs
    execute_proposal_callback
```

Description

Current behavior: If the execution of a proposal fails, the EGLD sent by the user to the endpoint `execute_proposal` are not sent back to him. This is because the execution of the proposal consists in an asynchronous call to the endpoint of the destination smart contract, however in case the asynchronous call fails, the callback `execute_proposal_callback` does not refund the user.

Expected behavior: When executing a proposal, the user expects his EGLD to be spent only if the execution is successful, and otherwise to be refunded.

Worst consequence: Users lose significant amounts of EGLD while repeatedly attempting to execute a proposal.

Example: A governance proposal to send 100 EGLD to an endpoint of a protocol is executed: the caller sends 100 EGLD to `execute_proposal`, however the execution of the protocol's endpoint fails (for any reason), and the caller has lost his 100 EGLD.

Recommendation

For each user, we suggest introducing a new storage `refund_token`. Then:

- In the callback `execute_proposal_callback`, if the asynchronous call has failed, we add the EGLD amount to `refund_token`.
- We have a new endpoint `withdraw_refund` that a user can call to withdraw his EGLD, and which clears the storage `refund_token`.

Moreover, we would add a unit test to verify that the refunding works well.

C2: Can’t control minimum gas amount for proposal's asynchronous call

Severity: Medium **Status:** Fixed

Location

```
governance/src/lib.rs
    execute_proposal
```

Description

Current behavior: The creator of a proposal has no control on the gas that will be reserved for the proposal’s asynchronous call, and so he has no way to ensure that the transaction will be executed successfully on the destination shard.

This is because anyone is allowed to execute the proposal by calling the endpoint `execute_proposal`, and the asynchronous call will be executed no matter the amount of gas provided by the caller.

Consequently, the asynchronous call might fail if the gas provided is insufficient, and it won’t be possible to re-try executing the proposal until the callback `execute_proposal_callback` is reached, due to the protection against replays.

Therefore, if the destination smart contract is on another shard, `execute_proposal_callback` would be reached at least 4 blocks later, therefore the proposal can’t be re-executed for at least ~24 seconds (given the current block duration of 6 seconds).

Expected behavior: The Governance smart contract should provide the proposal’s creator with enough control on the proposal to ensure that it can be successfully executed, in particular he should be able to control the gas to reserve for the asynchronous call.

Worst consequence: A malicious user prevents a proposal that calls a smart contract on another shard from ever being successfully executed: he calls the endpoint `execute_proposal` with insufficient gas for the asynchronous call, and repeats this operation as soon as the callback `execute_proposal_callback` is reached.

Note however that this attack would be quite hard to maintain over time as the malicious user would need to be systematically faster for executing his transaction than other honest users who concurrently call `execute_proposal` with sufficient gas.

Recommendation

We recommend adding a field `min_gas_for_async_call` to the struct `DecodedCallData`, allowing the proposal's creator to specify a minimal amount of gas to reserve for the asynchronous call. Then, in `execute_proposal`, we verify that the gas `gas_limit` allocated to the asynchronous call is at least `min_gas_for_async_call`:

```
let gas_limit = gas_left - EXECUTE_PROPOSAL_CALLBACK_GAS -
KEEP_EXTRA_GAS
require!(gas_limit >= min_gas_for_async_call, "Insufficient gas for
execution");
```

C3: “execute_proposal” does not accept ESDT tokens and might prevent executing proposals that transfer ESDT tokens

Severity: Medium **Status:** Fixed

Location

```
governance/src/lib.rs
    execute_proposal
```

Description

Current behavior: The endpoint `execute_proposal` only accepts EGLD as payment, but not ESDT tokens. Therefore, it is not possible to execute a proposal while sending the ESDT tokens that are necessary for the execution.

If in addition the smart contract is deployed as non-payable, it would become very impractical to execute proposals which need to transfer some ESDT tokens. The only way would be to execute a first proposal that retrieves ESDT tokens by calling another smart contract, and then to execute a second, separate proposal to transfer the tokens.

Expected behavior: Like on Ethereum, the Governance smart contract on MultiversX should handle the transfer of any kind of token, thus `execute_proposal` should be able to receive EGLD and ESDT tokens to ease the execution of proposals that transfer tokens.

Recommendation

We recommend making `execute_proposal` payable by any token instead of only EGLD. For this, we can replace the endpoint annotation `#[payable("EGLD")]` by `#[payable("*")]`.

Moreover, in `execute_proposal_callback`, in case the asynchronous call has failed, we would let the caller get back his EGLD or ESDT tokens, following the recommendation in C1: Executor of governance proposal does not get his EGLD back if the execution fails.

C4: Proposal can be resubmitted although it is not yet successfully executed or cancelled

Severity: Medium **Status:** Fixed

Location

`governance/src/lib.rs`

Description

Current behavior: A proposal can be submitted although the same proposal was already submitted and is not yet successfully executed or cancelled. Indeed:

- For operator proposals, the same proposal can be resubmitted anytime because no verification is performed when submitting such proposals.
- For proposals with timelock, there is a check in the method `process_command` that no timelock is currently set for the proposal, however there is a special case where the timelock would not be set although the proposal is not yet successfully executed or cancelled. Namely, while the proposal is being asynchronously executed, the timelock is temporarily cleared, and is only restored in the callback if the call has failed. Therefore, while the asynchronous call is ongoing, it is possible to resubmit the proposal.

This is problematic because the Governance smart contract only maintains at most one copy of any given proposal, i.e. one copy for a given hash of the proposal's data (information about the transfer and smart contract call). Therefore, if the Axelar Governance has voted twice on the same proposal and the 2nd instance is submitted while the 1st instance is not yet successfully executed, then the proposal would be executed only once.

Expected behavior: As long as a proposal is not successfully executed or cancelled, it should not be possible to resubmit the same proposal, because the Governance smart contract can only maintain one copy of any given proposal at the same time.

For this reason, in the Solidity implementation of ITS (link), a resubmission would fail until the proposal is successfully executed or cancelled.

Recommendation

The solution recommended below also solves the issue C5: Proposal should be cancellable as long as it is not successfully executed. It relies on the following new storages:

- `timelock_proposals_submitted` : The set of proposals with timelock submitted in the smart contract.
- `operator_proposals_submitted` : The set of operator proposals submitted in the smart contract.
- `timelock_proposals_being_executed` : The set of proposals with timelock whose execution is currently ongoing.
- `operator_proposals_being_executed` : The set of operator proposals whose execution is currently ongoing.

Then:

1) When submitting a proposal, we require that it is not yet submitted and not being executed, and mark it as submitted. Incidentally, for proposals with timelock, we can remove the requirement that no timelock is already set.

2) When cancelling a proposal, we call a new helper `remove_proposal` : it removes the proposal from the list of submitted proposals, and clears the storages `time_lock_eta` (for proposals with timelock) or `operator_approvals` (for operator proposals).

3) When executing a proposal, we require that it is submitted and not being executed. Then, we mark it as being executed before the asynchronous call, and unmark it in the callback, no matter if the asynchronous execution succeeded or not.

Moreover, if it succeeded, then in the callback, we clear the proposal by calling `remove_proposal` . In particular, we would not clear anymore the storages `time_lock_eta` (for proposals with timelock) or `operator_approvals` (for operator proposals) before the asynchronous call, and likewise we would not restore them in the callback.

C5: Proposal should be cancellable as long as it is not successfully executed

Severity: Medium **Status:** Fixed

Location

`governance/src/lib.rs`

Description

Current behavior: If a cancellation command is submitted while a Governance proposal is currently being executed, then the cancellation command would be consumed, but if the execution of the proposal fails, then the proposal would be considered as pending again, i.e. it would not be cancelled and it would be possible to re-try its execution.

More precisely, the execution of a proposal happens through an asynchronous call which can take a few blocks, and when the execution fails, the callback restores the proposal's information, i.e. `time_lock_eta` (for normal proposal) or `operator_approvals` (for operator proposal), no matter if a cancellation command was executed in the meantime.

Moreover, once the cancellation command is executed, it is marked as validated in the Gateway, thus this command can't be executed again. Re-doing a cancellation might then be heavy: the Axelar Governance would need to reach consensus again and post a new cancellation command on MultiversX.

Expected behavior: Proposals should be cancellable as long as they are not successfully executed.

Worst consequence: A malicious user prevents the cancellation of a proposal which performs a call to a smart contract on another shard. For this, he would trigger the execution just before the cancellation is executed.

Example: At the end of the following sequence, a proposal's cancellation is consumed but the proposal was not cancelled.

1. The Axelar Governance submits a proposal in the Governance smart contract. The proposal consists in calling a smart contract on another shard.

2. The Axelar Governance decides to cancel the proposal, e.g. because it is no longer relevant and the call to the target smart contract would fail.
3. However, just before the cancellation command is executed, someone triggers the execution of the proposal.
4. While the asynchronous call is being executed, the cancellation command is executed, and is consumed.
5. In the callback, as the asynchronous call has failed, the proposal is restored. Therefore it is possible to re-execute the proposal.
6. However, the cancellation command can't be re-executed because it is marked as validated in the Gateway.

Recommendation

We suggest following the recommendation of C4: Proposal can be resubmitted although it is not yet successfully executed or cancelled, as it also resolves this issue.

Test Issues & Recommendations

T1: Missing unit test

Severity: Medium **Status:** Fixed

Description

Current behavior: There is no unit test for executing a proposal that transfers ESDT tokens.

Expected behavior: A unit test should be implemented to test the case where a proposal consists in transferring ESDT tokens, because it is a functionality that the Governance smart contract is expected to handle.

Recommendation

We suggest adding a unit test for executing a proposal that transfers multiple ESDT tokens.