Security Audit Report

Axelar mvx-gateway
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:** ./gateway/

Final scope

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

3 inherent risks in the final scope

0 issue in the final scope

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

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

Inherent Risks

R1: Users have no guarantee that the messages they send to the Gateway will reach the target smart contract on the destination chain.

This is because there is trust that sufficiently many Axelar validators sign the message and that relayers forward the message:

- Sufficiently many Axelar validators should sign the message, otherwise the message won't reach the Gateway on the destination chain.
- Relayers should pick the message from the source chain and inform Axelar validators about it, and once the message is signed by validators, relayers should forward it to the destination chain. Therefore, in the absence of relayers, the message would not reach the destination smart contract on the destination chain.

R2: Users have no guarantee that the messages which are approved in the Gateway are the messages which were effectively created by accounts from another chain.

This is because a message can be approved in the Gateway if it is signed by sufficiently many Axelar validators, and moreover there should be relayers to forward the message from the source blockchain to Axelar network and then to the Gateway. Therefore:

- An erroneous message which was not created on another chain might be approved if for any reason sufficiently many Axelar validators signed it, e.g. if these validators made mistakes or were manipulated.
- A valid message created on another chain might never reach the Gateway if insufficiently many Axelar validators signed it, or if it was not forwarded by relayers.

R3: The following parameters might be set to harmful values: the size of a validator set, the number of past validator sets that can approve messages, and the delay before which a new validator set can be registered.

This is because Axelar validators are allowed to set these parameters to any values, without any kind of protections, which could be harmful:

- If the number of validators in a set is too big, then it might be too gas costly to check all their signatures when approving messages. This could prevent users' messages from further being processed, until they are approved by another set with a reasonable number of validators.
- If the number of past validator sets that can approve messages is too big (e.g. 1B), then even after adding numerous new validator sets, an old set, possibly made of unwanted and malicious validators, would still be allowed to approve messages.
- If the delay before which a new validator set can be registered is too small (e.g. 0 seconds), then new validators might be added too frequently, making it hard to keep track of active validator sets: in this case relayers might fail to make users' message approved in the Gateway. By contrast, if the delay is too big (e.g. 100 years), then the most recent validator set would be unable to elect a new validator set in case it plans to go inactive and needs to be replaced.

In practice however, the risk is low, because the choices of parameters result from a consensus between multiple validators of the Axelar blockchain.

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

Code Issues & Recommendations

C1: Can create a non-authorized set of signers with same hash as an authorized set of signers

Severity: Critical **Status:** Fixed

Location

`gateway/src/auth.rs`
`get_signers_hash`

Description

Current behavior: In the smart contract, a set of signers is identified by applying the function `get_signers_hash` to the set of signers, represented by the struct `WeightedSigners`. However, this identification can be manipulated, i.e. it is possible to modify the set of signers such that `get_signers_hash` produces the same result for the new set of signers and for the original one.

This is because the function `get_signers_hash` concatenates the **top-level encodings** of signers' information, and hashes the concatenation:

```
fn get_signers_hash(signers: WeightedSigners)
    let mut encoded = ManagedBuffer::new();

    // all type -> buffer conversions below are top-level encodings
    for signer in signers.signers.iter() {
      encoded.append(signer.signer.as_managed_buffer());
      encoded.append(signer.weight.to_bytes_be_buffer());
    }

    encoded.append(signers.threshold.to_bytes_be_buffer());
    encoded.append(signers.nonce.as_managed_buffer());

    self.crypto().keccak256(encoded)
}
```

Namely, the top-level encoding uniquely characterizes a value of a certain type only if it is isolated from other values, but can be ambiguous for types with

dynamic length when concatenated with other values. In our context, it can be possible to modify the number of signers in the set as well as the fields of type `BigUint` (the signers' weights and the quorum threshold) in a way that keeps the same result after applying `get_signers_hash`. An example is provided below.

The consequence is that, if a signer belonging to an authorized set of signers gets compromised, then **he will very likely be able to approve any message he wants**. This is because he could design an invalid set of signers that leads to the same hash, in which the compromised signer has a huge weight and is the only signer.

Expected behavior: The mechanism used to identify a set of signers should be resilient against manipulations, meaning that it should be computationally infeasible to find different sets of signers producing the same result after applying the function `get_signers_hash`.

Worst consequence: A signer belonging to an authorized set of signers gets compromised, and succeeds to approve and execute any message while bypassing Axelar's consensus mechanism. In turn, he can manipulate any smart contract that integrates the Gateway. For example, the attacker could drain all the tokens of Axelar's ITS smart contract.

Example: Consider a valid set of 3 signers Alice, Bob and Carol with addresses `[a, b, c]` each with weight `100`, a threshold of `200` and a nonce of `1`. Abbreviating "top-level encoding" as `TL`, the set of signers is recorded as the hash of the concatenation: `TL(a)@TL(100)@TL(b)@TL(100)@TL(c)@TL(100)@TL(200)@TL(1)`.

Let's say that Bob is malicious. He builds the following different set of signers:

- The addresses are `[a, b]`,
- The weight of `a` is `100`,
- The weight of his address `b` is the (huge) `BigUint` obtained by concatenating `TL(100)`, `TL(c)`, `TL(100)` as well as all bytes of `TL(200)` except the last one,
- The threshold is the last byte of `TL(200)`,
- The nonce is the same: `1`.

This set of signers produces the same concatenation as the initial one, hence produces the same hash, and therefore is considered valid by the Gateway.

From now on, Bob can reach the threshold (which is very small) only with his address `b` (which has a huge weight), and thus Bob can approve and execute any message.

Note that a similar attack would be feasible if the malicious user was rather Alice or Carol.

Recommendation

In `get_signers_hash`, when concatenating the fields of the struct `WeightedSigners`, we suggest using a nested encoding rather than a top-level encoding, by using the helper `dep_encode` from MultiversX Rust framework. Indeed, the nested encoding is exactly designed to avoid ambiguity when decoding multiple consecutive values. The way it works is by adding the length of a value as a prefix to the value, in case its type has dynamic length.

Moreover, we recommend adding a unit test showing that the non-authorized set of signers from the above example would not be considered as valid by the Gateway.

C2: Can create a non-authorized message with same hash as an authorized message

Severity: Medium **Status:** Fixed

Location

`gateway/src/lib.rs`
`message_hash`

Description

Current behavior: In the smart contract, a message is identified as follows:

- The message's information (`source_chain` , `message_id` , `source_address` , `destination_address` , `payload_hash`) is compressed using the function `message_hash` : it hashes the concatenation of the **top-level encodings** of the message's information.
- The *command ID* `source_chain@"_"@message_id` , where `source_chain` and `message_id` are **top-level encoded**, is used as a key to store the above hashed message's information.

However, this identification can be manipulated, i.e. it is possible to modify the message's information in a way that the hash `message_hash` and the command ID stay the same.

This is because of the use of top-level encoding, which uniquely characterizes a value of a certain type only if it is isolated from other values, but can be ambiguous for types with dynamic length when concatenated with other values. In our context, we have the following constraints to modify the message's information while keeping the same `message_hash` and command ID:

- The first 2 information `source_chain` and `message_id` are of type `ManagedBuffer` , whose top-level encoding can have arbitrary length. Therefore these fields can be modified. However, they can be modified only in a way that produces the same command ID: `source_chain@"_"@message_id` . Therefore, the length of the concatenation `source_chain@message_id` can't be changed.
- The last 2 information `contract_address` and `payload_hash` have top-level encoding with fixed length, therefore they can't be modified, otherwise

`message_hash` would be different.

- The 3rd information `source_address` is of type `ManagedBuffer`, however it can't be modified. This is because from the previous points: (a) the first 2 information can be modified only in a way that preserves the length of `source_chain@message_id`, (b) the last 2 information have fixed length. Therefore, in order to keep `message_hash` unchanged, `source_address` must stay the same.

It is thus possible to obtain a message that would be considered as approved in the Gateway and could be executed, even if it was not effectively approved by Axelar validators and was not effectively created on another chain.

In practice however, the constraints on how a message can be modified are very limiting and likely prevent validating a harmful message:

- For one thing, it is likely impossible to modify `source_chain` and `message_id` in a way that doesn't change the concatenation `source_chain@"_"@message_id`, because from Axelar's conventions neither `source_chain` and `message_id` are ever supposed to contain the "_" character.
- For another thing, `destination_address` and `payload_hash` can't be modified, so all the information about the message's validation must be unchanged, namely the smart contract which is allowed to validate the message, the specific endpoint allowed to call the validation endpoint, and the arguments that should be provided to that endpoint.

Expected behavior: The mechanism used to identify a message should be resilient against manipulations, meaning that it should be computationally infeasible to find different messages that produce the same command ID and produce the same result after applying the function `message_hash`.

Recommendation

At a high-level, we recommend using nested encodings instead of top-level encodings for defining `message_hash` and the command ID. Indeed, the nested encoding is exactly designed to avoid ambiguities when decoding multiple consecutive values. The way it works is by adding the length of a value as a prefix to the value, in case its type has dynamic length.

More precisely, we recommend making the following changes:

1) In `message_hash`, we suggest nested encoding each message information rather than top-level encoding it, by using the helper `dep_encode` from MultiversX Rust framework.

2) We suggest making the command ID into a struct `CommandId` whose fields are `source_chain` and `message_id`, as the fields of a struct are always nested encoded:

```
pub struct CommandId {
   source_chain: ManagedBuffer,
   message_id: ManagedBuffer
}

fn messages(id: CommandId) -> SingleValueMapper<MessageState>;
```

We can then remove some unnecessary logic around command IDs: the function `message_to_command_id`, and the field `command_id` from the events `message_approved_event` and `message_executed_event`.

Note: In principle, modifying the command ID as above guarantees that `source_chain` and `message_id` can't be manipulated, and since no other fields can be manipulated, it might seem unnecessary to also modify `message_hash`, i.e. to also avoid using top-level encodings there. However, using top-level encodings in the context of information identification is generally a bad practice, in particular it could lead to manipulations in future versions of the Gateway if there are modifications of the message's information or of the order in which information are concatenated. This is why we advise using nested encodings in `message_hash` as well.

C3: Typo in variable name "enfore_rotation_delay”

Severity: Minor **Status:** Fixed

Location

```
gateway/src/lib.rs
    rotate_signers
```

Description

In the function `rotate_signers`, there is a typo in the variable’s name `enfore_rotation_delay`.

Recommendation

In `rotate_signers`, we recommend correcting the typo by renaming the variable into `enforce_rotation_delay`.

Test Issues & Recommendations

T1: Erroneous comment in test 'Message approved'

Severity: Minor **Status:** Fixed

Location

`tests/gmp/gateway.test.ts`

Description

In the test *'Message approved'* of the file *tests/gmp/gateway.test.ts*, there is an erroneous comment when checking the final state of the smart contract:

```
// Message was executed
```

Indeed, at this point of the test the message is approved, but not executed.

Recommendation

We suggest correcting the comment, e.g. into:

```
// Message was approved
```