Security Audit Report

Axelar mvx-token-manager

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:** `6ccc55290af7c2e3a14909e2bb331b113eef8ab3`
- **MultiversX smart contract path:** ./token-manager/

Final scope

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

0 inherent risk in the final scope

0 issue in the final scope

9 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	6	0	0	0	0	0
Minor	2	0	0	0	0	0

Code Issues & Recommendations

C1: At most 1 external account should be able to mint tokens

Severity: Major **Status:** Fixed

Location

`token-manager/src/lib.rs`
`deploy_interchain_token`

Description

Current behavior: It is possible that multiple addresses are minters at the same time, i.e. they are able to mint tokens in the Token Manager.

Namely, as long as the token identifier is not issued, it is possible to call `deploy_interchain_token` multiple times, and each time the internal method `add_minter` would grant the mint role to the address given as optional argument (or the zero address if the optional argument was not provided).

```
fn deploy_interchain_token(minter: Option<ManagedAddress>, ...) {
    ...
    add_minter(self.blockchain().get_sc_address());
    if minter.is_some() {
        add_minter(minter.unwrap());
    } else { add_minter(ManagedAddress::zero()); }
    ...
}

fn add_minter(minter: ManagedAddress) {
    add_role(minter, Roles::MINTER);
}

fn add_role(address: ManagedAddress, new_roles: Roles) {
    account_roles(address).update(|roles| {
        roles.insert(new_roles)
    });
}
```

Since the token issuance is made via an asynchronous call to the metachain, it can take multiple blocks to be effective, thus leaving time to call `deploy_interchain_token` multiple times.

Moreover, this situation can happen both if the token is deployed directly on MultiversX, or if it is deployed from another blockchain.

Expected behavior: According to the rules of the Axelar network which are already enforced on other blockchains, there should be at most 1 external address allowed to mint tokens.

Worst consequence: Projects whitelist multiple minters on MultiversX, increasing the risk that one minter is corrupted and mints tokens which are not backed by tokens on other blockchains, making users lose funds.

Recommendation

We recommend ensuring that only 1 external account can be granted the minter role. For this, we introduce a storage `minter_address`, which is changed each time a minter is set, in particular when a minter transfers his minter role.

Then, in the endpoint `deploy_interchain_token`:

- We don't grant the minter role to the `minter` argument if a minter was already set in `minter_address`.
- We don’t grant the minter role to the Token Manager itself, because this is unnecessary (see C8: Adding Token Manager as minter in "deploy_interchain_token" is useless) and would lead to 2 minters.

C2: No flow limit is enforced when flow limiter asks for the most restrictive flow limit

Severity: Medium **Status:** Fixed

Location

`token-manager/src/flow_limit.rs`

Description

Current behavior: When the flow limit is set to `0`, then in fact it is interpreted as if there was no limit at all on the interchain transfers. Indeed, the method `add_flow_out_raw` responsible for verifying that limits are not exceeded would return early in this case:

```
fn add_flow_out_raw(flow_out_amount: BigUint) {
  let flow_limit = self.flow_limit().get();
  if flow_limit == 0 {
      return;
  }
  ...
}
```

Expected behavior: If the flow limiter sets the flow limit to `0`, then the effective limit should be `0`, i.e. users shouldn't be able to transfer any amount of tokens, as this is the intention of the flow limiter.

Worst consequence: A project needs to temporarily shut down interchain transfers for his token, in order to prevent an ongoing issue from escalating. For this, the flow limiter sets the flow limit to `0`, thinking that it would fully block interchain transfers, but in fact the opposite occurs: users can transfer arbitrary amounts of tokens, which might aggravate the issue.

Recommendation

We suggest changing the type of the storage `flow_limit` to `Option<BigUint>`, instead of `BigUint`. Then:

- If the flow limit is `None`, then there is no limit on the interchain transfers,
- If the flow limit is `Some(x)`, e.g. `Some(0)`, then the effective limit is `x`.

C3: "token_id" might change if issued several times and all previous users lose their funds

Severity: Medium **Status:** Fixed

Location

`token-manager/src/lib.rs`

Description

Current behavior: The token `token_id` of the Token Manager might change. Namely, as long as it is not set, the endpoint `deploy_interchain_token` can be called an arbitrary number of times: each call issues a token, and stores the resulting identifier in `token_id` in the callback `deploy_token_callback`. Therefore, `token_id` is overwritten in each callback.

However, this means that if between two callbacks, some tokens are minted and sent to users, then these tokens would become valueless as they would not be recognized by the Token Manager any longer.

Expected behavior: Once the token `token_id` of the Token Manager is set in storage, it should never change, to guarantee that this token will be forever recognized by the smart contract and hence that users holding this token will be able to perform interchain transfers.

Worst consequence: Some users on MultiversX hold valueless tokens that they can't transfer to other blockchains.

Example: Consider an existing bridge between several blockchains other than MultiversX. The project which initially created this bridge plans to extend to MultiversX, and therefore performs a remote deployment of a Native Interchain Transfer followed by a minting of an initial supply on MultiversX. The following sequence occurs on MultiversX:

- The Token Manager is successfully deployed.
- A 1st transaction to issue the token is triggered.
- 2 blocks later, a 2nd transaction to issue the token is accidentally triggered.
- The callback of the 1st issuance is reached, setting the token in storage.
- The project mints an initial supply of tokens and sends them to some users.

- The callback of the 2nd issuance is reached, overwriting the token in storage. In turn, the tokens previously sent to users have no value any longer.

Recommendation

In the callback `deploy_token_callback`, we recommend setting the storage `token_id` only if it is empty, by using `set_if_empty`.

C4: Funds can't be deposited in a Token Manager of type Lock/Unlock

Severity: Medium **Status:** Fixed

Description

Current behavior: It is impossible to fund the Token Manager, i.e. to deposit tokens in it. This is because there are no endpoints to fund the Token Manager, and moreover the ITS deploys each Token Manager as a non-payable smart contract.

This is problematic for a Token Manager of type “Lock/Unlock”, as then it means that the Token Manager has no tokens initially, hence incoming interchain transfers would be impossible, and the only way to use the bridge initially would be to do outgoing interchain transfers. However, some projects might have wanted to allow users to bridge their funds to MultiversX from the start.

Expected behavior: It should be possible to fund a Token Manager of type “Lock/Unlock”. This is because initially, such a Token Manager would have no funds, hence initial deposits would be necessary if it is desired to allow users from other blockchains to do interchain transfers towards MultiversX.

Recommendation

We suggest adding an endpoint `donate_tokens` that accepts only the token of the Token Manager, and verifies that the Token Manager is of type “Lock/Unlock”.

C5: No check that user EGLD amount for issuing token is 0.05 EGLD and user would lose the extra EGLD

Severity: Medium **Status:** Fixed

Location

```
token-manager/src/lib.rs
    deploy_interchain_token
```

Description

Current behavior: When calling the endpoint `deploy_interchain_token` to issue the Token Manager's token, there is no check that the EGLD received is 0.05 EGLD, i.e. the cost of the issuance.

If the user provided more EGLD e.g. by accident, then 0.05 EGLD would effectively be used for the issuance, but the user would lose the extra EGLD.

Expected behavior: The endpoint `deploy_interchain_token` should exclusively accept a payment of 0.05 EGLD, as this is the cost for issuing a token.

Recommendation

In the endpoint `deploy_interchain_token`, we recommend verifying that the amount of received EGLD equals `DEFAULT_ESDT_ISSUE_COST`, i.e. 0.05 EGLD.

C6: User is not refunded of EGLD issuance cost if issuance fails

Severity: Medium **Status:** Fixed

Location

```
token-manager/src/lib.rs
    deploy_token_callback
```

Description

Current behavior: If the issuance of the Token Manager's token fails, then the user who paid for the issuance cost is not refunded, because the callback `deploy_token_callback` does not perform any refunding.

Expected behavior: If the issuance of the Token Manager's token fails, the user who paid for the issuance cost should be refunded.

Recommendation

In the callback `deploy_token_callback`, if the issuance has failed, we recommend refunding the issuance cost to the user. For this, the user address should be forwarded as an argument to `deploy_token_callback` from the endpoint `deploy_interchain_token`.

Moreover, in case `deploy_interchain_token` is called by the ITS, then it should rather receive the user as a new argument, forwarded from the ITS Factory to the ITS, and from the ITS to the Token Manager.

C7: "upgrade" endpoint has unnecessary arguments and can lead to more than 1 operator in Token Manager

Severity: Medium **Status:** Fixed

Location

`token-manager/src/lib.rs`
`upgrade`

Description

Current behavior: The `upgrade` endpoint takes several arguments, which are unnecessary because the parameters of the smart contract were already set at deployment:

```
fn upgrade(
  interchain_token_service: ManagedAddress,
  implementation_type: TokenManagerType,
  interchain_token_id: ManagedByteArray,
  params: DeployTokenManagerParams,
) { self.init(...) }
```

Most of these arguments would not be set in storage in `init`, because they are set using `set_if_empty` and were already set at deployment. However, there are two exceptions:

- The operator role is granted to the address given as argument `params`,
- The flow limiter role and the operator role are granted to the ITS address given as argument `interchain_token_service`.

```
 fn init(...) {
  ...
  self.add_role(operator, Roles::FLOW_LIMITER | Roles::OPERATOR);
  self.add_role(interchain_token_service, Roles::FLOW_LIMITER |
Roles::OPERATOR);
  ...
}
```

Therefore, the Token Manager might be left with unintended flow limiters and operators, in particular with more than 1 operator.

Expected behavior: The `upgrade` endpoint should have no arguments, as it is not supposed to perform any logic and to set any storage.

In particular, from Axelar specifications, there should be at most 1 operator per Token Manager, thus no additional operator should be set when upgrading the smart contract.

Recommendation

We recommend removing all arguments from the `upgrade` endpoint and not performing any logic inside it.

C8: Adding Token Manager as minter in "deploy_interchain_token" is useless

Severity: Minor **Status:** Fixed

Location

```
token-manager/src/lib.rs
    deploy_interchain_token
```

Description

In the endpoint `deploy_interchain_token`, the line

```
self.add_minter(self.blockchain().get_sc_address());
```

is unnecessary, because it is useless to grant the minter role to the Token Manager. Indeed, a minter is an account which can call the endpoint `mint`, however the Token Manager has no way to call that endpoint and is not supposed to call it.

Note: This line of code was copied from the Solidity code, where it made sense because the Token Manager is distinct from the token's ERC20 smart contract.

Recommendation

We recommend deleting the useless line from the endpoint `deploy_interchain_token`.

C9: Misleading endpoint name "invalid_token_identifier”

Severity: Minor **Status:** Fixed

Location

`invalid_token_identifier`

Description

The endpoint `invalid_token_identifier` returns the token identifier, if already set, of the Token Manager, and returns `None` otherwise. Therefore, the naming `invalid_token_identifier` is slightly misleading, because the endpoint does not return information about the validity of the token.

Recommendation

We suggest renaming the endpoint `invalid_token_identifier` e.g. into `get_opt_token_identifier`. We would then also rename the endpoint `token_manager_invalid_token_identifier` of the ITS Factory e.g. into `token_manager_get_opt_token_identifier`.