Security Audit Report

Axelar mvx-interchain-token-service

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:** ./interchain-token-service/

Final scope

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

4 inherent risks in the final scope

0 issue in the final scope

23 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	3	0	0	0	0	0
Medium	3	0	0	0	0	0
Minor	16	0	0	0	0	0

Inherent Risks

R1: Users outgoing interchain transfers might never be successfully executed on the destination blockchain and they will never be refunded on MultiversX.

This is because, for reasons explained below, outgoing interchain transfers might either (1) never be executed, or (2) be incorrectly executed, or (3) systematically fail to be executed. Moreover, in such cases the tokens would be lost as there is no refunding mechanism implemented on MultiversX.

1) An outgoing interchain transfer might never be executed if for any reason the Axelar network does not forward it to the destination blockchain.

2) An outgoing interchain transfer might be incorrectly executed if the transfer's information is altered by the Axelar network while being forwarded to the destination blockchain.

Example: The ITS Hub on the Axelar blockchain modifies the amount of the transfer based on scaling rules which depend on the token decimals (both on the source and destination blockchains) and on the blockchains requirements (e.g. the maximal allowed integers). In particular, truncations might lead to smaller transferred amounts than expected.

3) An outgoing interchain transfer might systematically fail to be executed on the destination blockchain, i.e. even if it is possible to re-try the execution, it would never succeed. There are two main types of failures:

3-a) Failures due to user errors in the interchain transfer's information.

Examples:

- The transfer's data (e.g. the destination address or smart contract call) is in a format incompatible with the destination blockchain.
- There is no Token Manager deployed on the destination blockchain for the token being transferred.

3-b) Failures that the user did not expect or could not anticipate.

Examples:

- The Token Manager of a custom token on the destination blockchain is of type *"Lock/Unlock"* and has an insufficient reserve of tokens. This is impossible to anticipate: the reserve of the Token Manager on the destination blockchain might have been sufficient at the time the interchain transfer was registered on MultiversX, but might have become too small by the time the interchain transfer arrives at destination.
- The Token Manager is of type *"Mint/Burn"* and has lost its minting role by the time the interchain transfer arrives at destination.
- The admins of the Token Manager on the destination blockchain have the ability to limit the amounts of tokens that can be transferred to that blockchain. Therefore, they could completely block interchain transfers.
- The destination address is a smart contract that is upgraded while the interchain transfer is being forwarded, and the endpoint signature becomes incompatible with the signature prescribed in the interchain transfer's information.
- The ITS on the destination blockchain has lost its right to transfer the token since the interchain transfer was initiated.
- The amount to transfer to the destination address exceeds the maximum integer that can be represented in a smart contract of the destination blockchain. For example, on Sui and Stellar, integers are `u64`, while on EVMs, integers are `u256`.

R2: Users incoming interchain transfers might never be successfully executed on MultiversX and they might never be refunded on the source blockchain.

For the same reasons described in R1 regarding outgoing interchain transfers, incoming interchain transfers might never be successfully executed.

In particular, an example specific to MultiversX of incoming interchain transfers that systematically fail to be executed, is a transfer consisting in a call to a smart contract on a different shard. Indeed, ITS only allows calls to smart contracts on the same shard.

Finally, unlike in R1, there might be a refunding mechanism implemented on the source blockchain, however if there is no such mechanism, then users would

have lost their tokens if their incoming interchain transfers are never successfully executed on MultiversX.

R3: Users' registration of a token from MultiversX might not be successfully executed on the destination blockchain.

This is because, for reasons explained below, the registration of a token on a destination blockchain might either (1) never be executed, or (2) be incorrectly executed, or (3) systematically fail to be executed.

1) The token's registration might never be executed if for any reason the Axelar network does not forward it to the destination blockchain.

Example: the ITS Hub on the Axelar blockchain currently only forwards the 1st attempt to register a token to a given destination blockchain, and does not forward any subsequent attempts. Therefore, if the 1st registration fails for any reason on the destination blockchain, then it would be impossible to attempt the registration again.

2) The token's registration might be incorrectly executed if the registration's information is altered by the Axelar network while being forwarded to the destination blockchain.

Example: The ITS smart contract on the destination blockchain might perform normalizations of the token's information (e.g. name and symbol), to comply with the blockchain's specific requirements.

3) The token's registration might systematically fail to be executed on the destination blockchain, i.e. even if it is possible to re-try the execution, it would never succeed.

Example: The registration's data (e.g. minter address, or decimals) is in a format incompatible with the destination blockchain.

R4: Users' registration of a token from a source blockchain might not be successfully executed on MultiversX.

For the same reasons described in R3 regarding outgoing token registrations, incoming token registrations from a source blockchain to MultiversX might never be successfully executed.

Here are additional examples that specifically apply to incoming registrations:

a) The token's registration might not be executed with the expected name and ticker, because the ITS performs the following normalizations on the name and ticker given as arguments, in order to comply with MultiversX requirements:

- Non-alphanumeric characters are skipped,
- The name and ticker are truncated above the maximum allowed length (20 and 10, respectively),
- The name and ticker are padded with "0" if their length is below the minimal allowed length, i.e. 3,
- For the ticker, all characters are uppercased.

b) The token's registration would fail if the token's decimals exceed the maximal allowed decimals on MultiversX, i.e. 18.

Code Issues & Recommendations

C1: Execution of incoming interchain transfer might be locked forever and the user's tokens are lost

Severity: Critical **Status:** Fixed

Location

```
interchain-token-service/src/proxy_its.rs
   execute_with_token_callback
```

Description

Current behavior: When an incoming interchain transfer with smart contract call is executed, the execution is made through an asynchronous call with callback. In order to protect against concurrent executions, a lock is created before the asynchronous call, and is cleared in the callback.

However, it is possible that the callback `execute_with_token_callback` fails, but then the lock would not be cleared. In turn, the user would never be allowed to re-try executing the interchain transfer, and he would effectively have lost his tokens.

The callback might fail for various reasons, e.g. due to unforeseen scenarios. Below we detail two examples of such scenarios which can make the callback fail. Both assume that the smart contract being called is on another shard.

1) Hitting flow limits when sending back tokens to the Token Manager in the callback: In the callback, in case the asynchronous call has failed, the tokens are sent back to the Token Manager by calling its endpoint `take_token`. However, this endpoint can fail due to *flow limits*, i.e. limits on the amount of tokens that is currently allowed to be deposited in the Token Manager for outgoing interchain transfers. More precisely, we have the following requirement in `take_token`:

```
require!(
    user_amount <= flow_limit &&
    user_amount + (total_flow_in - total_flow_out) <= flow_limit,
    "Flow limit exceeded"
);
```

Therefore, since the destination smart contract of the interchain transfer is on another shard, other transactions might be executed before the callback is reached, which could make the above requirement fail in the following cases:

- The maximal amount `flow_limit` that can be deposited in a single transaction might have been reduced, and if it leads to `user_amount > flow_limit`, then the callback would fail.
- The total amount of tokens `total_flow_in` deposited in the Token Manager for outgoing interchain transfers might have increased, and if it leads to `user_amount + (total_flow_in - total_flow_out) > flow_limit`, then the callback would fail.

2) ITS or Token Manager is forbidden to transfer the token in the callback: If the ITS or the Token Manager is frozen for the token by the time the callback is executed, or transfer roles have been set for the token, then the callback would fail.

Expected behavior: The lock on an interchain transfer with smart contract call should be present only during the period where it is actually being executed, that is, from the time the asynchronous call is launched, until the callback is executed.

This is to ensure that the user can re-try to execute the interchain transfer in case a previous execution has failed.

Worst consequence: A user loses his funds, because of a failure of the callback `execute_with_token_callback`.

Moreover, when this issue is combined with C3: Attacker might be able to delay interchain transfer by repeatedly providing insufficient gas for asynchronous call, then **a malicious attacker could make any user lose his funds** from an interchain transfer with a call to a smart contract on a another shard, by executing the interchain transfer with insufficient gas for the asynchronous call. We describe the precise steps that the attacker would follow in the example below.

Example: A malicious user Alice is insolvent in a Lending Borrowing protocol which is not in the same shard as ITS, and Bob from another blockchain makes an interchain transfer of 10,000$ worth of tokens with a smart contract call to liquidate Alice. To prevent the liquidation, Alice proceeds as follows:

- She executes Bob's interchain transfer with insufficient gas for the asynchronous call.
- While the asynchronous call is ongoing, she records an outgoing interchain transfer such that the flow limit in the Token Manager is saturated.
- The callback then hits the flow limit and fails. Therefore the liquidation can't be re-tried, and Bob has lost 10,000$ worth of tokens.

Recommendation

We suggest following the recommendation to C2: Performing asynchronous calls for interchain transfers with smart contract calls is unnecessary and too complex, which also resolves this issue.

Alternatively, in case the above recommendation does not fit with the project's needs, we suggest reaching out to the auditor to discuss about alternative solutions.

C2: Performing asynchronous calls for interchain transfers with smart contract calls is unnecessary and too complex

Severity: Major **Status:** Fixed

Location

```
interchain-token-service/src/proxy_its.rs
   executable_contract_execute_with_interchain_token
```

Description

Current behavior: Incoming interchain transfers with smart contract calls are performed asynchronously. However, performing asynchronous calls (1) is unnecessary, and (2) results in a highly complex mechanism.

1) Asynchronous calls are unnecessary: As explained in the section "Expected behavior", projects integrating with Axelar ITS can be assumed to be on the same shard as ITS, thus they can be called synchronously.

2) The resulting execution mechanism is highly complex:

- In order to prevent replays of the interchain transfer while an asynchronous execution is ongoing, a lock is created before the asynchronous call, and removed in the callback. This lock mechanism has led to the issue C3: Attacker might be able to delay interchain transfer by repeatedly providing insufficient gas for asynchronous call.
- In case the asynchronous call has failed, the tokens need to be sent back to the Token Manager. This mechanism has led to the issue C1: Execution of incoming interchain transfer might be locked forever and the user's tokens are lost.
- In order to allow re-trials of a failed interchain transfer, the Gateway message is checked before the asynchronous call, and only validated in the callback if the execution was successful, to prevent replays. On other non-sharded blockchains, e.g. on Ethereum, the message is simply directly validated before executing the transfer.
- Sufficient gas should be carefully reserved for executing the callback as well as for finishing the execution after registering the asynchronous call.

Expected behavior: The execution of an interchain transfer with a smart contract call should be done synchronously. This is because:

1) According to the Axelar team, there is no need to support smart contracts on other shards.

Indeed, as there are currently no smart contracts deployed on MultiversX which integrate the ITS protocol, it is possible to require that future smart contracts should be deployed in the same shard as ITS if they want to integrate with it, so that interchain transfers can be done synchronously.

This way, even projects on other shards would be able to indirectly receive interchain transfers by deploying a proxy smart contract on the shard of ITS. This proxy would receive the interchain transfers in an endpoint `executeWithInterchainToken`, and if needed, handle the logic to forward the tokens to the protocol's smart contracts on other shards.

2) Synchronous calls would significantly simplify the code, thereby reducing the risk of introducing issues in future changes.

Recommendation

We recommend executing incoming interchain transfers with smart contract calls synchronously, i.e. by using `execute_on_dest_context` instead of `register_promise`, effectively forcing smart contracts integrating with Axelar ITS to be on the same shard.

This approach incidentally solves other issues of this report:

- C3: Attacker might be able to delay interchain transfer by repeatedly providing insufficient gas for asynchronous call,
- C1: Execution of incoming interchain transfer might be locked forever and the user's tokens are lost.

In turn, we can simplify the smart contract as follows:

- The lock mechanism can be deleted.
- The callback `execute_with_token_callback` can be deleted, in particular the logic to send back tokens to the Token Manager.
- The message can be directly validated before calling the smart contract.
- The mechanism for reserving gas can be deleted, including the constants `EXECUTE_WITH_TOKEN_CALLBACK_GAS` and `KEEP_EXTRA_GAS`.

C3: Attacker might be able to delay interchain transfer by repeatedly providing insufficient gas for asynchronous call

Severity: Major **Status:** Fixed

Location

`interchain-token-service/src/proxy_its.rs`

Description

Current behavior: When an incoming interchain transfer with smart contract call is executed, the execution is made through an asynchronous call with callback. In order to protect against concurrent executions, a lock is created before the asynchronous call, and is cleared in the callback.

However, there is no way to guarantee that sufficient gas is provided to the asynchronous call for successfully executing the endpoint of the destination smart contract. Indeed, this gas is obtained from the gas left in the transaction when registering the asynchronous call, but this gas left could be too small, as the caller provided an arbitrary amount of gas to the transaction.

Consequently, if the gas is insufficient and if the smart contract is on a different shard than ITS, then users must wait during a few blocks (typically around 30 seconds) before they can attempt to re-execute the interchain transfer.

In the worst case, a malicious actor could delay the execution of an interchain token transfer of another user over an extended period of time, by repeatedly performing the interchain transfer with insufficient gas.

Expected behavior: The protocol should ensure that the incoming interchain transfer can be called with a sufficient amount of gas for the asynchronous call.

Worst consequence: An attacker prevents an incoming interchain transfer with a call to a smart contract on a different shard from being executed over an arbitrarily long period of time. For this, he executes the call with insufficient gas, and repeats this each time the callback is reached.

Note however that, in order to perform such an attack, the attacker would need to be the first to re-execute the interchain transfer each time the callback is reached.

Example: A cross-chain liquidation of 10000$ arrives to MultiversX, and the borrower succeeds to prevent the liquidation over an extended period of time, until a point where he has become so insolvent that liquidating him leaves bad debt in the protocol.

Recommendation

We suggest following the recommendation to C2: Performing asynchronous calls for interchain transfers with smart contract calls is unnecessary and too complex, which also resolves this issue.

Alternatively, in case the above recommendation does not fit with the project's needs, we suggest reaching out to the auditor to discuss about alternative solutions.

C4: A wrong refund address is provided to Gas Service in some cases

Severity: Major **Status:** Fixed

Location

`interchain-token-service/src/proxy_gmp.rs`

Description

Current behavior: When a user performs an outgoing interchain call, he can pre-pay the gas for the interchain call by sending tokens which are forwarded to the Gas Service smart contract. Along the gas payment, a refund address is provided to the Gas Service smart contract. This way, in case there are gas tokens remaining at the end of the interchain call, the relayer managing the Gas Service smart contract can refund that address.

However, the provided refund address might be erroneous. Namely, it is always the caller of the ITS smart contract, which in the following cases would not be the user, but rather the ITS Factory:

- When the outgoing interchain call consists in linking a custom token,
- When the outgoing interchain call consists in deploying a token on a destination blockchain.

In these cases, the relayer executing the interchain call would be misled into thinking that the address to refund is the ITS Factory, and in turn, the user might not be refunded.

Expected behavior: For any outgoing interchain call, the refund address that ITS provides to the Gas Service smart contract should be the address of the account performing the interchain call.

This is to ensure that, if there are gas tokens left remaining at the end of the interchain call, the relayer would refund the correct address.

Worst consequence: The relayer refunds the ITS Factory instead of the user.

Recommendation

We recommend providing the correct address as the argument `refund_address` of the proxy endpoints `pay_native_gas_for_contract_call` and `pay_gas_for_contract_call`.

In particular, in case the ITS Factory is calling the ITS, i.e. calling the endpoints endpoints `link_token` or `deploy_interchain_token`, it should also forward the address of the account performing the transaction. Therefore, the endpoints `link_token` and `deploy_interchain_token` would take that address as a new argument.

C5: The mechanism around trusted blockchains and trusted addresses is unnecessarily complex

Severity: Medium **Status:** Fixed

Location

`interchain-token-service/src/address_tracker.rs`

Description

Current behavior: Incoming and outgoing interchain calls are accepted only if they are made from/to trusted addresses on trusted blockchains. Among trusted addresses and blockchains, the Axelar Hub on Axelar plays a special role, i.e. an interchain call from/to any blockchain can be made by wrapping the call inside a call to the Axelar Hub on Axelar, which then handles the forwarding to the destination blockchain.

Since the implementation of the ITS Hub has been completed, all calls are now supposed to go through it. The logic for sending messages directly to other trusted addresses on trusted blockchains was kept only for backward compatibility in existing implementations of the ITS protocol, e.g. on Ethereum, for applications that perform interchain calls in this way.

However, on MultiversX, no applications are already integrating with Axelar, therefore it is unnecessary to enable direct calls from/to trusted addresses on trusted blockchains, i.e. we can assume that all calls will go through the Axelar Hub. Therefore, the following is unnecessary:

- Recording trusted addresses and trusted blockchains, except the Axelar Hub on Axelar.
- Having logic to directly receive or send messages from/to blockchains other than Axelar.

Expected behavior: According to the Axelar team, all incoming and outgoing interchain calls should go through the Axelar Hub on the Axelar blockchain, in order to significantly simplify the code.

Such a simplification was already implemented in other integrations such as in Stellar ITS.

Recommendation

We recommend performing the following changes:

- Deleting the storage `trusted_address`, and instead having a single `UnorderedSetMapper` `trusted_chains` for trusted blockchains.
- Deleting the constant `ITS_HUB_ROUTING_IDENTIFIER`, and instead having a new storage `its_hub_address` to record the address of the Axelar Hub on Axelar, set in `init`.
- Renaming `set_trusted_address`, `remove_trusted_address` and `is_trusted_address` into `set_trusted_chain`, `remove_trusted_chain` and `is_trusted_chain`.
- For incoming calls, we verify that the source address is `its_hub_address`, and simplify `get_execute_params` as we don't need to consider the case where the source blockchain is different than Axelar:

```
fn get_execute_params(
  source_address: ManagedBuffer,
  source_chain: ManagedBuffer,
  payload: ManagedBuffer
) -> (u64, ManagedBuffer, ManagedBuffer) {
  let message_type = self.get_message_type(payload);
  require!(message_type == MESSAGE_TYPE_RECEIVE_FROM_HUB);
  require!(source_chain == ITS_HUB_CHAIN_NAME);
  require!(source_address == self.its_hub_address().get());

  let data = SendToHubPayload::abi_decode(payload);
  require!(self.is_trusted_chain(data.destination_chain));
  let message_type = self.get_message_type(data.payload);

  return (message_type, data.destination_chain, data.payload);
}
```

- For outgoing calls, similarly, we send the call to `its_hub_address`, and simplify `get_call_params` as we don't need to consider the case where the destination blockchain is different than Axelar:

```
fn get_call_params(
    destination_chain: ManagedBuffer,
    payload: ManagedBuffer
) -> (ManagedBuffer, ManagedBuffer, ManagedBuffer) {
    require!(destination_chain != *ITS_HUB_CHAIN_NAME);
    require!(self.is_trusted_chain(destination_chain));

    let hub_address = self.its_hub_address.get();
    let data = SendToHubPayload {
      message_type: MESSAGE_TYPE_SEND_TO_HUB,
      destination_chain,
      payload,
    };
    return (ITS_HUB_CHAIN_NAME, hub_address, data.abi_encode())
}
```

C6: Only EGLD should be accepted to pay gas for outgoing interchain transfers

Severity: Medium **Status:** Fixed

Location

`interchain-token-service/src/user_functions.rs`
`get_transfer_and_gas_tokens`

Description

Current behavior: When a user initiates an outgoing interchain transfer, he can pay the gas for the relayer with any token, i.e. EGLD or ESDT tokens.

However, relayers currently only forward interchain calls whose gas is paid in EGLD.

Therefore, if users pay gas in ESDT tokens, their interchain transfers might never be executed and so they would lose their tokens.

Moreover, the current logic for supporting ESDT tokens as gas payments is not fully functional: it does not work when the token to transfer is EGLD. Indeed, in this case, the method `get_transfer_and_gas_tokens` returns that the token to transfer is an ESDT (not EGLD) with identifier *"EGLD-000000"*, which would not be recognized by ITS, and therefore the transaction would fail.

Expected behavior: According to the Axelar team, users should only be allowed to pay the gas for outgoing interchain calls with EGLD, because relayers currently only execute Gateway messages whose gas is paid in EGLD.

In particular, gas should be paid in EGLD for outgoing interchain transfers, as is already the case for other outgoing interchain calls: linking of tokens and remote deployments of tokens.

Likewise, on other blockchains, only the native token of the blockchain (e.g. ETH for Ethereum) can be used to pay gas for outgoing interchain transfers.

Recommendation

For outgoing interchain transfers, we recommend enforcing that the gas is paid in EGLD, by doing the following simplifications:

- We delete the method `pay_gas_for_contract_call` for paying gas with ESDT tokens,
- We simplify the struct `TransferAndGasTokens` by deleting the field `gas_token`,
- We simplify the method `get_transfer_and_gas_tokens` such that if the received payment is an array of tokens, it has length 2 and the 2nd payment is EGLD.

```
fn get_transfer_and_gas_token(gas_amount: BigUint) {
    match payments {
        EgldOrMultiEsdtPayment::Egld(value) => {
            ...
        }
        EgldOrMultiEsdtPayment::MultiEsdt(esdts) => {
            let [transfer, gas] = self.call_value().multi_esdt();
            require!(transfer.token_nonce == 0);
            require!(gas.token_identifier == ESDT_EGLD_IDENTIFIER);
            require!(gas.amount == gas_amount);
            return TransferAndGasTokens {
                transfer_token: transfer.token_identifier,
                transfer_amount: transfer.amount,
                gas_amount,
            };
        }
```

C7: Outgoing interchain transfer might be recorded with metadata different than the one asked by the user

Severity: Medium **Status:** Fixed

Location

`interchain-token-service/src/remote.rs`
`decode_metadata`

Description

Current behavior: When submitting an outgoing interchain transfer, the user provides a `ManagedBuffer` argument `metadata`, which represents two pieces of information: the version of the interchain call (a `u32`), and the smart contract data to execute on the destination blockchain (a `ManagedBuffer`).

However, in case the decoding of the metadata fails, a default metadata is used:

```
fn decode_metadata(
    raw_metadata: ManagedBuffer
) -> (MetadataVersion, ManagedBuffer) {
   let decoded_metadata = Metadata::top_decode(raw_metadata);
   if decoded_metadata.is_err() {
       return (MetadataVersion::ContractCall, ManagedBuffer::new())
   }
   ...
}
```

This would for example happen in the case where `metadata` has length smaller than `4`, as then `top_decode` would fail to extract a `u32` version from it.

Consequently, malformed metadata could be submitted and the interchain transfer would still be executed, with some information that might not be the ones intended by the caller.

Expected behavior: When a user submits an outgoing interchain transfer, the interchain transfer should be done with the exact metadata provided by the user. In particular, if the user provided a malformed metadata, then the transaction should revert.

Recommendation

In the method `decode_metadata`, after calling the method `top_decode`, if the decoding returned an error, we suggest making the transaction fail.

In particular, metadata of length between `0` and `3` bytes would not be accepted.

C8: Versioning logic for outgoing interchain transfers is unnecessary

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/user_functions.rs
   interchain_transfer
```

Description

Current behavior: When submitting an outgoing interchain transfer, the user provides a `ManagedBuffer` argument `metadata`, which represents two pieces of information: the version `version` of the interchain call, and the smart contract data `data` to execute on the destination blockchain.

This is both unnecessary and complex:

- *Unnecessary:* Indeed, only one version of interchain transfer is supported for MultiversX, namely normal interchain transfers.
- *Complex:* It forces the caller to properly merge `data` and `version` into a single argument `metadata`, and also leads to sophisticated decoding logic inside the smart contract.

Note: the versioning logic was introduced to mimic the Solidity code, where it was initially planed to also support express interchain calls.

Expected behavior: According to the Axelar team, it is expected that only one version of interchain transfers will be supported. Therefore, the versioning logic for outgoing interchain transfers should be removed, as it would simplify the smart contract.

For example, such an approach was taken in the Stellar implementation of ITS.

Recommendation

We recommend replacing the `metadata` argument with a simple `data` argument, which would be the (possibly empty) data for executing a smart contract call on the destination blockchain. Since the version part of the

metadata would disappear, we can further delete the method `decode_metadata`, the enum `MetadataVersion`, and the const `LATEST_METADATA_VERSION`.

C9: The function "fixed_bytes_append" is unnecessarily complex

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/abi.rs
    fixed_bytes_append
```

Description

Current behavior: The method `fixed_bytes_append` processes a byte array `data` of arbitrary size, and adds it into another byte array `result`, padding the result with zeros on the right so that the final number of bytes is a multiple of `32`.

However, the code of `fixed_bytes_append` is unnecessarily complex for the reasons explained below:

1) There is no need for the modulo operation in the following:

```
match bytes.len() % 32 {
  0 => 32,
  x => x,
}
```

Indeed, at this point in the code, the length `bytes.len()` must be non-zero and is at most 32, therefore the above code is equivalent to returning `bytes.len()` directly.

2) The following condition is useless:

```
let to_copy = match i == len - 1 {
    false => 32,
    true => match bytes.len() % 32 { ... }
};
```

Indeed, the number of bytes to copy is always `bytes.len()`: it is `32` in case the batch being processed is not the last one (in this case the batch has length `32`), and otherwise it is `bytes.len()` from the previous point.

Expected behavior: The function `fixed_bytes_append` should be as simple as possible.

Recommendation

We recommend simplifying the function `fixed_bytes_append` as follows:

```
fn fixed_bytes_append(
    result: &mut ManagedBuffer,
    data: ManagedBuffer
) {
    data.for_each_batch::<32, _>(|bytes| {
        let mut padded = [0u8; 32];
        padded[..bytes.len()].copy_from_slice(bytes);
        result.append_bytes(&padded);
    });
}
```

C10: Unnecessarily complex way to convert byte array into u32 in "take_usize"

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/abi.rs
    take_usize
```

Description

Current behavior: In the method `take_usize`, given a byte array of `32` bytes whose last `4` bytes represent a `usize` integer, the conversion into a `usize` integer is made by doing:

```
((slice[28] as usize) << 24) + ((slice[29] as usize) << 16)
+ ((slice[30] as usize) << 8) + (slice[31] as usize)
```

However, this could be made simpler by using existing helpers from the Rust framework:

```
u32::from_be_bytes(slice[28..32].try_into().unwrap()) as usize
```

Expected behavior: The conversions from a byte array to a `usize` integer in the method `take_usize` should be as simple as possible, and use helpers from the Rust framework instead of custom implementations whenever possible.

Recommendation

We recommend importing the necessary helper from the Rust framework:

```
use core::convert::TryInto;
```

Then, in the method `take_usize`, we can convert the byte array into a `usize` integer by doing:

```
u32::from_be_bytes(slice[28..32].try_into().unwrap()) as usize
```

C11: Unnecessarily complex and misleading processing of 32-byte array in "head_append"

Severity: Minor **Status:** Fixed

Location

`interchain-token-service/src/abi.rs`
`head_append`

Description

Current behavior: In the method `head_append`, to encode a 32-byte array, the method `fixed_bytes_append` is used. However, using this method is unnecessarily complex and misleading:

- *Unnecessarily complex:* `fixed_bytes_append` implements logic to handle arrays of dynamic size, by processing the array in batches of 32 bytes. However, there is no need for such logic when dealing with a fixed-size 32-byte array. Instead, we could simply convert the array directly into a `ManagedBuffer` using the helper `as_managed_buffer` from the Rust framework.
- *Misleading:* `fixed_bytes_append` pads the array on the right so that the number of bytes is a multiple of 32. This is misleading when dealing with arrays of fixed sizes, as the ABI encoding of fixed-size data should not perform any padding on the right. Fortunately, no padding would occur in practice for a 32-byte array (since 32 is a multiple of 32).

Expected behavior: The processing of 32-byte arrays in `head_append` should be as simple as possible, and use helpers from the Rust framework whenever possible, instead of custom methods designed to handle arrays of dynamic size.

Recommendation

In the method `head_append`, we recommend replacing the line

```
Token::Bytes32(data) =>
  Self::fixed_bytes_append(acc, data.as_managed_buffer())
```

with the line:

```
Token::Bytes32(data) => acc.append(data.as_managed_buffer())
```

C12: Function "ascii_to_u8" is unnecessarily complex

Severity: Minor **Status:** Fixed

Location

`interchain-token-service/src/constants.rs`

Description

Current behavior: The function `ascii_to_u8` converts a string representing a token's decimals into an actual `u8` integer. However, the conversion is done in an unnecessarily complex way, i.e. by loading batches of 32 bytes:

```
fn ascii_to_u8(&self) -> u8 {
  let mut result: u8 = 0;
  self.for_each_batch::<32, _>(|batch| {
    for &byte in batch {
      if byte == 0 {
        break;
      }
      result *= 10;
      result += (byte as char).to_digit(16).unwrap() as u8;
     }
  });
  result
}
```

Namely, the decimals of a token on MultiversX must be between 0 and 18, hence the decimals' string representation is encoded on at most 2 bytes. In turn, it is unnecessary to parse the string by loading batches of 32 bytes as if the length of the string could be arbitrarily big.

Expected behavior: The function `ascii_to_u8` can be simplified and should be simplified.

Recommendation

We recommend simplifying the method `ascii_to_u8` as follows:

```
fn ascii_to_u8(&self) -> u8 {
  let mut result: u8 = 0;
  let mut byte_array = [0u8; 2];
  let _ = self.load_slice(0, &mut byte_array);
  for byte in byte_array {
    result *= 10;
    result += (byte as char).to_digit(16).unwrap() as u8;
  }
  result
}
```

C13: Unnecessary argument "initial_offset" in "raw_abi_decode”

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/abi.rs
    raw_abi_decode
```

Description

The argument `initial_offset` of the method `raw_abi_decode` is useless, because it is always `0` when `raw_abi_decode` is called, and so it is equivalent to having no initial offset at all.

Recommendation

We recommend deleting the argument `initial_offset` from the method `raw_abi_decode`.

C14: Unnecessary endpoint "call_contract_with_interchain_token”

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/user_functions.rs
    call_contract_with_interchain_token
```

Description

The endpoint `call_contract_with_interchain_token` is unnecessary, as it is a weaker version of the endpoint `interchain_transfer`: the former is restricted to the `ContractCall` version, while the latter can be used with any version.

Recommendation

We recommend deleting the endpoint `call_contract_with_interchain_token`.

C15: Unnecessarily big cap values used in helper "abi_decode" for structs “SendToHubPayload” and “DeployInterchainTokenPayload”

Severity: Minor **Status:** Fixed

Location

`interchain-token-service/src/abi_types.rs`

Description

Current behavior: For each struct that needs to be decoded from the ABI format, there is a decoding method `abi_decode`. It takes a payload as argument, decodes one by one each field of the struct, and places these fields in an array `result` of fixed size given by a cap `CAP`. For example:

```
impl AbiEncodeDecode for SendToHubPayload {
  ...
  fn abi_decode(payload: ManagedBuffer) -> Self {
    // Initialize the "result" array with cap 4
    let mut result = ArrayVec::<Token, CAP 4>::new();

    // Decode each field from payload in the "result" array
    Self::raw_abi_decode(payload, &mut result);

    // Pop each field from the "result" array
    let payload = result.pop();
    let destination_chain = result.pop();
    let message_type = result.pop();

    SendToHubPayload {message_type, destination_chain, payload}
  }
}
```

However, for the following structs, the cap is unnecessarily big, i.e. bigger than the number of fields in the struct:

- The struct `SendToHubPayload` has 3 fields, but the cap is 4.
- The struct `DeployInterchainTokenPayload` has 6 fields, but the cap is 9.

In turn, these unnecessarily big caps are slightly misleading for the code reader.

Expected behavior: In each method `abi_decode` associated to a struct to decode from the ABI format, the array `result` should have a size capped to the number of fields of the struct, as this is the exact number of entries that this array is expected to contain.

Recommendation

Both for the structs `SendToHubPayload` and `DeployInterchainTokenPayload`, in their method `abi_decode`, we suggest correcting the cap of the array `result` so that it matches the number of fields in the struct.

C16: The struct "SendToHubPayload" is used for receiving payloads from the Axelar Hub

Severity: Minor **Status:** Fixed

Location

`interchain-token-service/src/executable.rs`

Description

Current behavior: The same struct `SendToHubPayload` is used both for outgoing interchain calls to the Axelar Hub, and incoming interchain calls from the Axelar Hub.

Therefore, the name of the struct is confusing when dealing with incoming interchain calls, and also the struct's field `destination_chain` is confusing when used to represent the source blockchain in the method `get_execute_params`:

```
fn get_execute_params(...) {
    ...
    // Decoding an incoming payload as if it was an outgoing payload
    let data = SendToHubPayload::<Self::Api>::abi_decode(payload);
    ...
    // "destination_chain" is in fact the original source chain
    require(self.is_trusted_address(data.destination_chain, ...));
    ...
    return (message_type, data.destination_chain, data.payload)
}
```

Expected behavior: To avoid confusions between outgoing and incoming interchain calls, a distinct struct should be introduced for each type of call. This is the convention taken on other blockchains integrated in Axelar. For example:

- In Stellar, different structs `SendToHub` and `ReceiveFromHub` are used for outgoing and incoming calls (link).
- In Solidity, transfers are not represented as structs but rather as tuples, and the variable names clearly reflect whether the call is outgoing or incoming. For example, for outgoing calls, the destination blockchain is referred to as

`destinationChain` (link), while for incoming calls, the source blockchain is referred to as `originalSourceChain` (link).

Recommendation

We recommend introducing a struct `ReceiveFromHub` for incoming interchain transfers:

```
struct ReceiveFromHub {
  message_type: BigUint,
  original_source_chain: ManagedBuffer,
  payload: ManagedBuffer
}
```

C17: Endpoint "invalid_token_manager_address" has misleading name and return type

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/proxy_its.rs
    invalid_token_manager_address
```

Description

Current behavior: The endpoint `invalid_token_manager_address` returns the Token Manager of a token if it exists, or the zero address if there is no such Token Manager. Therefore:

- The endpoint's name is misleading, because the endpoint does not return information about the validity of the argument `token_id`.
- The endpoint's return type is misleading, because the zero address is a valid address on MultiversX, hence it does not explicitly signal the absence of a Token Manager.

Expected behavior: The name and return type of an endpoint should reflect the purpose of this endpoint.

Recommendation

We recommend renaming the endpoint `invalid_token_manager_address` e.g. into `get_opt_token_manager_address`, and changing its return type to `Option<ManagedAddress>`.

Consequently, the endpoints `deploy_interchain_token` and the method `check_token_minter` in the ITS Factory should be adapted to handle this new return type.

C18: Inconsistent logic to pay gas for interchain calls with ESDT and with EGLD

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/proxy_gmp.rs
    gas_service_pay_native_gas_for_contract_call
```

Description

Current behavior: In the method `gas_service_pay_gas_for_contract_call`, the logic to pay gas for outgoing interchain calls is not the same whether the gas is paid in ESDT or in EGLD:

- For an ESDT, the token is directly forwarded to an endpoint of the Gas Service smart contract,
- For EGLD, a helper function `gas_service_pay_native_gas_for_contract_call` is called, which forwards the EGLD to an endpoint of the Gas Service smart contract. Note that this helper is not used anywhere else.

However, there is no reason for treating the two cases differently.

Expected behavior: The logic to pay gas should be consistent whether the gas is paid in ESDT or in EGLD.

Recommendation

In the method `gas_service_pay_gas_for_contract_call`, whether the payment is an ESDT or is EGLD, we recommend directly forwarding it to the Gas Service smart contract.

In turn, we can delete the helper function `gas_service_pay_native_gas_for_contract_call`.

C19: Useless "sender" argument in function "interchain_token_id"

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/user_functions.rs
   interchain_token_id
```

Description

The argument `sender` in the endpoint `interchain_token_id` is unnecessary, because each time the endpoint is called, this argument is the zero address.

Indeed, all tokens being deployed on MultiversX are deployed through the ITS Factory, and in this case, the `sender` used to compute the interchain token ID is the zero address.

Note: The argument `sender` was introduced to mimic the Solidity code. There, the argument used to be useful when anyone (not only the ITS Factory) was allowed to deploy a token.

Recommendation

We suggest removing the argument `sender` from the endpoint `interchain_token_id`. Then, the endpoint would build the interchain token ID simply by doing:

```
Hash256("its-interchain-token-id"@salt)
```

Moreover, the issue C20: No explicit check that the caller of "link_token" is the ITS Factory must also be solved, i.e. we should explicitly check that the caller is the ITS Factory in the endpoint `link_token`.

Indeed, there is currently no such explicit check, but only an implicit one: since `interchain_token_id` depends on the token's deployer `sender` which can only be the ITS Factory, `link_token` would fail for any other caller. As we would now remove the argument `sender`, an explicit check in `link_token` becomes necessary, to prevent anyone from being able to link an existing token he did not created to other blockchains.

C20: No explicit check that the caller of "link_token" is the ITS Factory

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/user_functions.rs
    link_token
```

Description

Current behavior: In the endpoint `link_token` for linking a custom token to a destination blockchain, there is no explicit check that the caller is the ITS Factory.

Fortunately, if the caller is not the ITS Factory, the transaction would fail, but this is for a subtle reason:

- Only the ITS Factory can register new token IDs,
- When new token IDs are registered, they are derived from the address of the caller, which by the point above must be the ITS Factory,
- When `link_token` is called, the ID of the token to link is similarly derived from the caller address. Therefore, if the caller is not the ITS Factory, it would lead to a non-existing token ID, and the transaction would fail.

However, the absence of an explicit check could confuse the code reader and heightens the risk of introducing errors in future modifications of the code.

Expected behavior: Users should deploy and link tokens only via the ITS Factory. Therefore, the endpoint `link_token` should require that the caller is the ITS Factory.

Recommendation

In the endpoint `link_token`, we recommend adding an explicit check that the caller is the ITS Factory.

C21: Misleading variable name "destination_address”

Severity: Minor **Status:** Fixed

Location

```
interchain-token-service/src/proxy_gmp.rs
    get_call_params
```

Description

In the method `get_call_params`, the variable name `destination_address` is used twice, and the 1st occurrence is misleading because it is used not for an address, but for a mapper which stores an address.

```
let destination_address = self.trusted_address(destination_chain);
require!(!destination_address.is_empty(), "Untrusted chain");
let destination_address = destination_address.get();
```

Recommendation

In the method `get_call_params`, we recommend renaming the 1st variable currently named `destination_address`, e.g. into `destination_address_mapper`.

C22: Unused methods, event and struct

Severity: Minor **Status:** Fixed

Description

The following are unused:

- The method `with_every_role`,
- The method `gateway_is_message_executed`,
- The event `emit_standardized_token_deployed_event`,
- The struct `StandardizedTokenDeployedEventData`.

Recommendation

We recommend deleting the method `with_every_role`, the method `gateway_is_message_executed`, the event `emit_standardized_token_deployed_event`, and the struct `StandardizedTokenDeployedEventData`.

C23: Obsolete comment in "init”

Severity: Minor **Status:** Fixed

Location

`interchain-token-service/src/lib.rs`
`init`

Description

The following comment in `init` is obsolete:

```
// from _setup function below
```

Indeed, the function `_setup` is not present in the code, rather it is a helper in the Solidity code of ITS.

Recommendation

We recommend deleting the obsolete comment from `init`.