Axelar Network

Scope

The code under review can be found within the C4 Axelar Network repository.

Summary

Findings

ID	Issue	Severity
L-01	Missing EOA check for Distributor and Operator roles	Low
L-02	msg.value >= gasValue check mentioned in comments is not implemented in the code	Low
L-03	revertIfInvalidFee() check missing in function sendProposal()	Low
N-01	Modifier should be utilized for checks only and not make state changes	Non-Critical
N-02	Typo error reduces code readability	Non-Critical
N-03	Missing event emission	Non-Critical

ID	Issue	Severity
N-04	Existing function should be used maintain pattern across _processCommand() functions	Non-Critical

Analysis Report

- Approach taken in evaluating the codebase
- Architecture recommendations
- Codebase quality analysis
- Centralization risks
- Mechanism Review
  - Interchain Governance
  - Interchain Token Service
- Time Spent

Findings

QA Report

[L-01] Missing EOA check for Distributor and Operator roles

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/utils/Distributable.sol#L39

Missing EOA check for distributor

```
File: contracts/its/utils/Distributable.sol
39: function _setDistributor(address distributor_) internal {
40:         assembly {
41:             sstore(DISTRIBUTOR_SLOT, distributor_)
42:         }
43:         emit DistributorChanged(distributor_);
44:     }
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/utils/Operatable.sol#L39

Missing EOA check for Operator:

```
File: contracts/its/utils/Operatable.sol
39: function _setOperator(address operator_) internal {
40:         assembly {
41:             sstore(OPERATOR_SLOT, operator_)
42:         }
43:         emit OperatorChanged(operator_);
44:     }
```

Solution for both would be to check the extcodesize(distributor/operator) > 0. If this is true, we revert with an error mentioning that the distributor/operator is not an EOA.

[L-02] msg.value >= gasValue check mentioned in comments is not implemented in the code

There are 3 instances of this issue occurring in the following 3 functions:

1. deployRemoteCanonicalToken - https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L322
2. deployRemoteCustomTokenManager - https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L360
3. deployAndRegisterRemoteStandardizedToken - https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L413

[L-03] revertIfInvalidFee() check missing in function sendProposal()

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/interchain-governance-executor/InterchainProposalSender.sol#L80

When sending proposal for execution at single destination chain using sendProposal(), we do not check if the msg.value provided is equal to the gas required. This check has been added to the sendProposals() function but not in sendProposal().

[N-01] Modifier should be utilized for checks only and not make state changes

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/auth/MultisigBase.sol#L44

The onlySigners modifier in the link above should be made into a function since it makes changes to storage. These are the lines where state changes occur:

```
File: contracts/cgp/auth/MultisigBase.sol
53: voting.hasVoted[msg.sender] = true;
61: voting.voteCount = voteCount;
66: voting.voteCount = 0;
71: voting.hasVoted[signers.accounts[i]] = false;
```

[N-02] Typo error reduces code readability

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L559

implementations is written incorrectly as implementaions on Line 559 (as parameter) and Line 564.

```
File: contracts/its/interchain-token-service/InterchainTokenService.sol
559: function _sanitizeTokenManagerImplementation(address[] memory
implementaions, TokenManagerType tokenManagerType)
560:         internal
561:         pure
562:         returns (address implementation)
563:     {
564:         implementation = implementaions[uint256(tokenManagerType)];
565:         if (implementation == address(0)) revert ZeroAddress();
566:         if (ITokenManager(implementation).implementationType() !=
uint256(tokenManagerType)) revert InvalidTokenManagerImplementation();
567:     }
```

[N-03] Missing event emission

There are 2 instances of this:

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L613

Event is emitted in the if block but not else block.

```
File: contracts/its/interchain-token-service/InterchainTokenService.sol
609: if (expressCaller == address(0)) {
610:             amount = tokenManager.giveToken(destinationAddress,
amount);
611:             emit TokenReceived(tokenId, sourceChain,
destinationAddress, amount);
612:         } else {
613:             amount = tokenManager.giveToken(expressCaller, amount);
//@audit missing event emission here
614:         }
```

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/its/interchain-token-service/InterchainTokenService.sol#L653

In this instance, an early return misses event emission on Line 653.

```
File: contracts/its/interchain-token-service/InterchainTokenService.sol
652: if (expressCaller != address(0)) {
653:                 amount = tokenManager.giveToken(expressCaller,
amount);
654:                 return; //@audit early return misses event emission
655:           }
656:         }
657:       amount = tokenManager.giveToken(destinationAddress, amount);
658:
IInterchainTokenExpressExecutable(destinationAddress).executeWithInterchai
nToken(sourceChain, sourceAddress, data, tokenId, amount);
659:       emit TokenReceivedWithData(tokenId, sourceChain,
destinationAddress, amount, sourceAddress, data);
660:     }
```

[N-04] Existing function should be used maintain pattern across _processCommand() functions

https://github.com/code-423n4/2023-07-axelar/blob/2f9b234bb8222d5fbe934beafede56bfb4522641/contracts/cgp/governance/AxelarServiceGovernance.sol#L84

The _getProposalHash() function is used to get the proposal hash in the _processCommand() function in the InterchainGovernance.sol contract but not in the _processCommand() function in the AxelarServiceCommand.sol contract (which inherits from InterchainGovernance.sol).

We can take a look at the difference in pattern below:

```
File: contracts/cgp/governance/InterchainGovernance.sol
125: bytes32 proposalHash = _getProposalHash(target, callData,
nativeValue);
```

```
File: contracts/cgp/governance/AxelarServiceGovernance.sol
84: bytes32 proposalHash = keccak256(abi.encodePacked(target, callData,
nativeValue));
```

Analysis Report

Approach taken in evaluating the codebase

To start evaluating this project, I identified the high level functionalities and its corresponding contracts involved in the scope (more on this in the Mechanism Review section below). The evaluation of the contracts (for each of a,b,c) was done in a base-to-derived contract manner. The functionalities are as follows: a. Interchain Governance b. Interchain Token/Message transfer c. Helper utility contracts involved in

facilitating a and b as well as to upgrade certain contracts (I will not be diving deeper into this separately but will refer to the contracts wherever necessary).

Additionally, I approached this codebase from an attacker's mindset. Through this type of mentality, I could recognise three points where failure/potential bugs could exist:

1. The endpoints that customers interact with
2. The internal endpoints that validators interact with
3. The internal workings and function flow that connects 1 to 2.

This diagram serves as a great reference to visualize these endpoints:
https://cdn.discordapp.com/attachments/1126884239330263072/1130039751173484604/gmp-diagram.webp

Architecture recommendations

The architecture of the protocol is quite simple and easy to understand. The developer team has done an amazing job in modularising the code into smaller chunks and storing it in separate files. The only recommendation I would have is to divide the InterchainTokenService.sol contract into two files. The pattern of this contract is as follows:

```
=> represents "calling"
Public/External functions => Internal functions (within the same contract)
```

Having the public/external functions and the internal functions in two separate files would greatly reduce the size of the file from 533 nSLOC to half of it, making the code more readable, modular and less concentrated (just like the rest of the contracts).

Codebase quality analysis

Overall, the codebase is structured in an easy to understand manner. The documentation and Natspec comments provided helps to recognize the endpoint (both customer and validator side) and intermediary contracts involved in facilitating interchain transfers. There are minor edge case issues that can lead to big exploits. Some of these issues (that I have included in my findings) are invalid implicit type conversion, improper validation of create2 return value, strict equality check, freezing governance functions when malicious signer votes when vote count is one less than threshold etc. Additionally, there are numerous instances across the codebase where a caller's ETH can get locked. Other than this, the developer team has done a good job with handling storage variables (For example, having storage structs, enums and mappings in their own separate files, updating boolean and uint256 Voting struct members in the onlySigners() modifier correctly etc).

Centralization risks

There seem to be several centralization points in the protocol, which I believe decrease the decentralisation in an interoperable network. They are here as follows:

1. Owner - Whitelisting addresses indirectly adds centralization, which can compromise the whole protocol if owner's private key gets leaked.

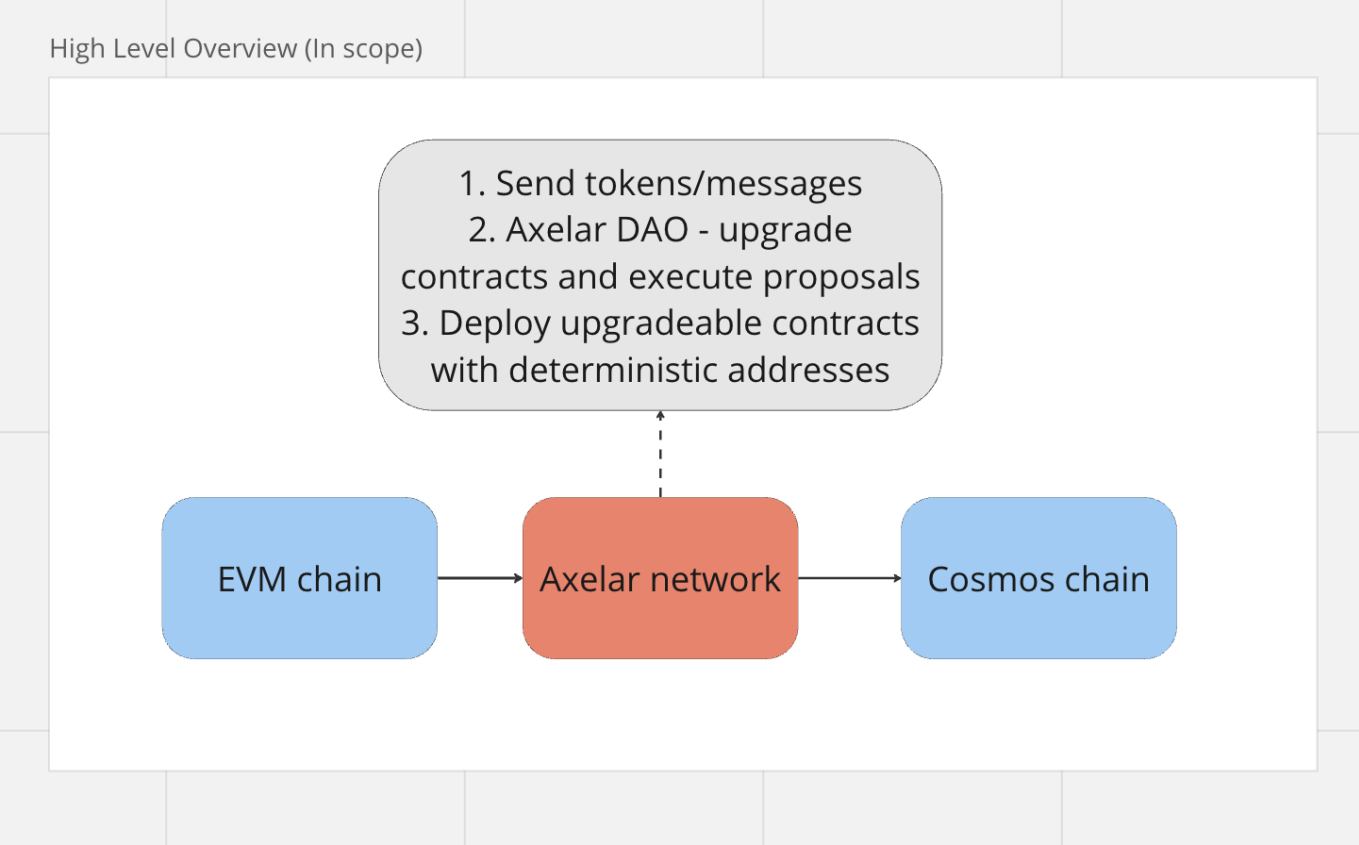
2. Distributor - Being allowed to mint and burn tokens
3. Operator - Allowed to set flow limit and liquidity pool

If any of these centralized entities have leaked private keys, it can destabilize and halt the protocol since they have control over important settings of the protocol like flow in and flow out limits, the address of the liquidity pool, minting and burning tokens etc. A multi-sig should be used for configuration operations like these in order to stay one step away from losing control over the protocol.

Mechanism Review

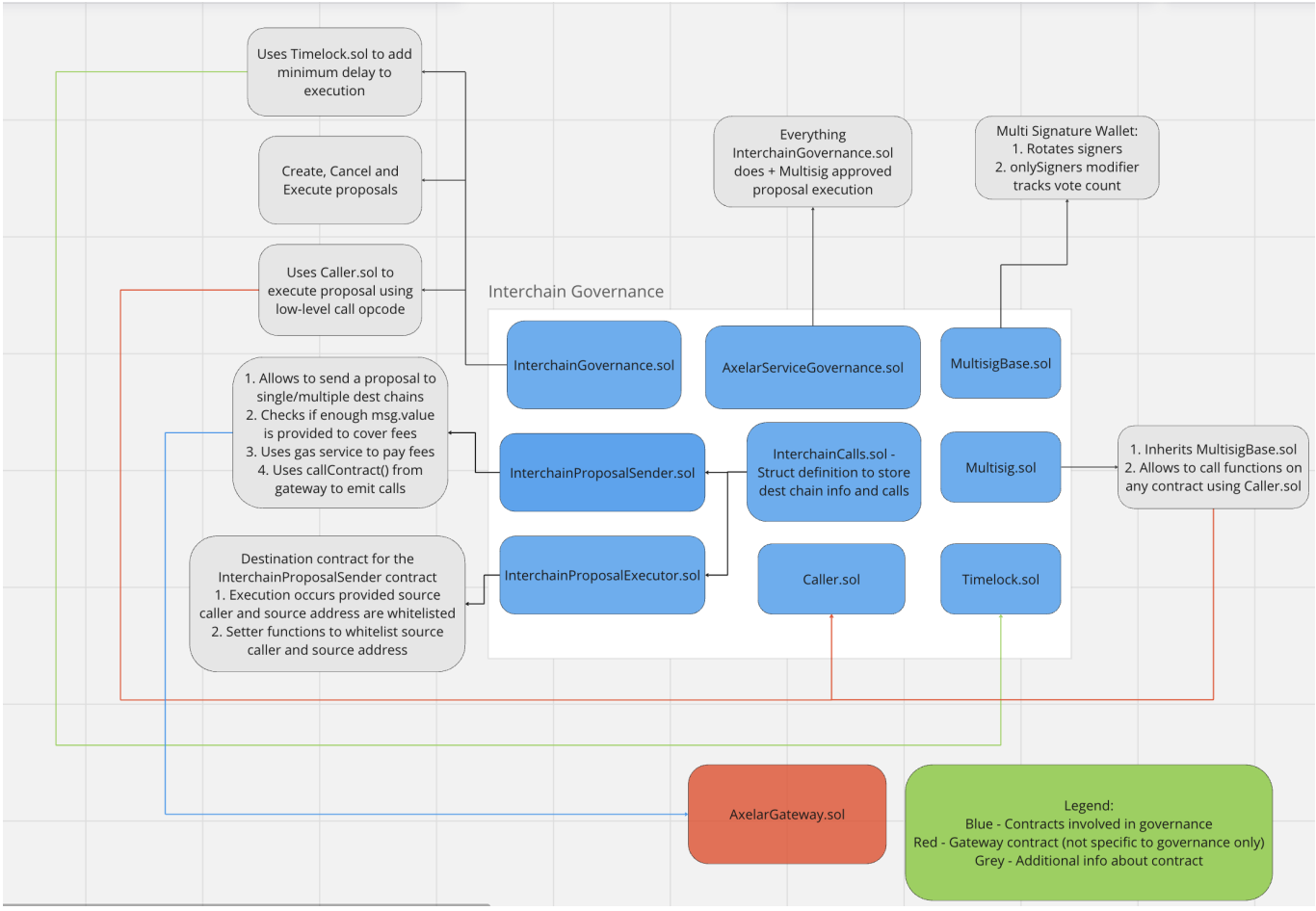
To understand the mechanism of this protocol, we need to look at the functionalities in scope: a. Interchain Governance b. Interchain Token Service c. Helper utility contracts involved in facilitating a and b as well as to upgrade certain contracts (I will not be diving deeper into this separately but will refer to the contracts wherever necessary).

Here is a mindmap to get an extremely high level overview of this protocol with the above mentioned functionalities in scope: https://drive.google.com/file/d/1YEMRzE9MChjO1_6l-ru1RUhUzhuJqxyn/view?usp=sharing



1. Interchain Governance

Here is a mindmap that explains the inner workings and important function calls of the Interchain Governance mechanism. I'll be referring to this while explaining: https://drive.google.com/file/d/1ocbESssCLWZi4XtIl-GrM41xWzdrCng_/view?usp=sharing



The diagram provides a high-level overview of how the main contracts are connected. The heaviest contracts (functionality-wise) involved are InterchainGovernance.sol, MultisigBase.sol, InterchainProposalSender.sol and InterchainProposalExecutor.sol. The rest of the blue box contracts (refer to the legend in diagram for more info) are either helper contracts or contracts inheriting from these heavier contracts to extend functionality. To simply explain the whole process:

1. InterchainGovernance.sol contract can create, cancel and execute proposals. a. Creating and cancelling of proposals happens through specific commandIds, which are basically their index in the enum they are stored in. To create and cancel, you can pass in the respective commandId (0 for create, 1 for cancel) as parameter value to the _execute() function. b. Execution of the proposal happens through the executeProposal() function, which finalizes the timelock (i.e. sets the hash of the proposal to 0) and executes the proposal using the low-level call in the Caller.sol contract. Note, the proposals have a minimum delay before they are ready to execute. This minimum delay is added by the Timelock.sol contract while scheduling the proposal.

2. AxelarServiceGovernance.sol - Does everything InterchainGovernance.sol can do, the only addition being execution of multisig proposals, which are handled by the signers (rotated every epoch) set in the the MultisigBase.sol contract.

3. Multisig.sol - Inherits from MultisigBase.sol and allows signers to execute any function in any target contract.

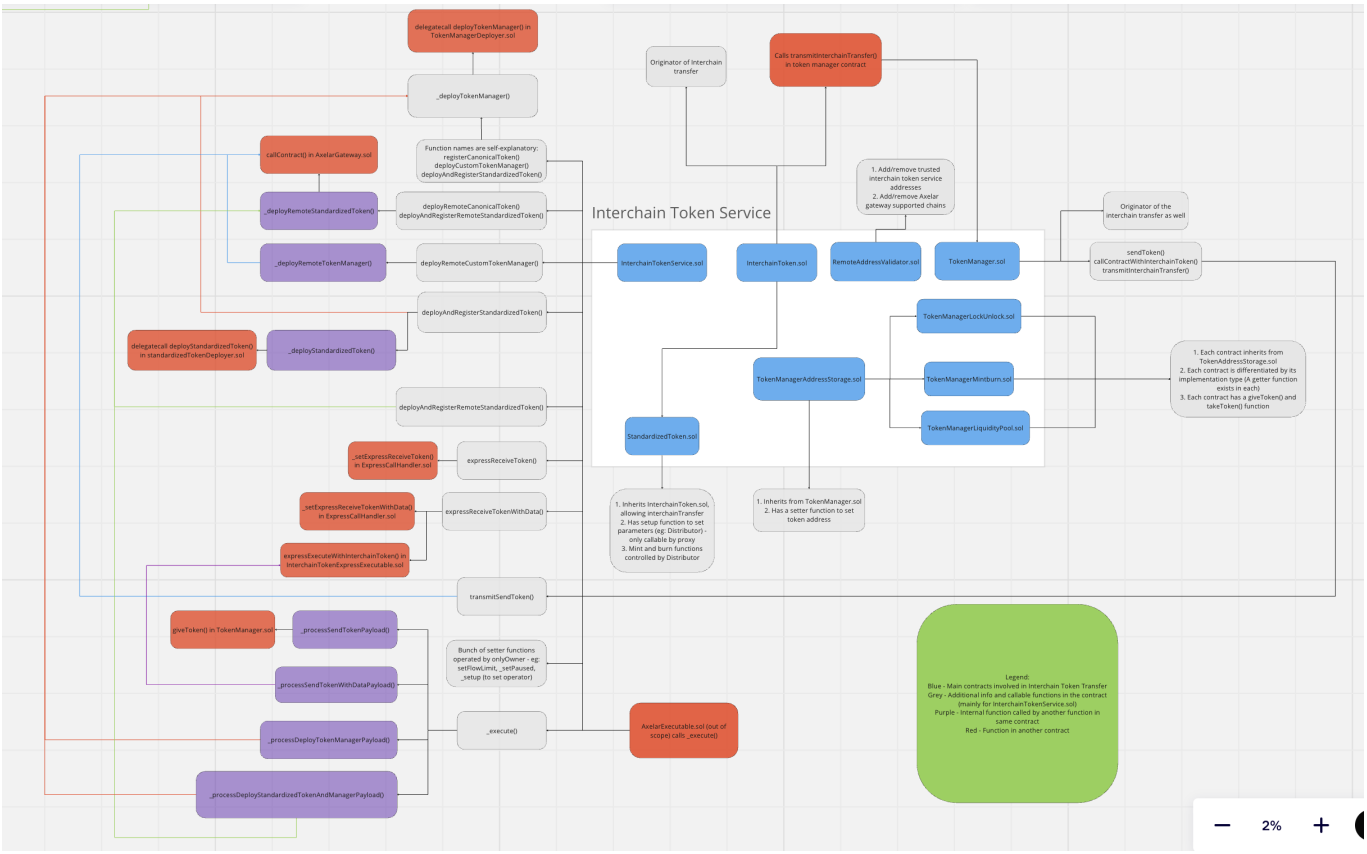
4. InterchainProposalSender.sol and InterchainProposalExecutor.sol play hand-in-hand in the sending and execution of proposals. The proposal sender is a contract on the source chain, which communicates with the proposal executor on the destination chain through the AxelarGateway.sol

contract. Note in order for execution to occur on the destination chain, the source caller and source address need to be whitelisted by the owner.

5. InterchainCalls.sol - This is a contract that stores the struct definition for interchain calls, which includes destination chain specific info and calls to be executed. It is used in the proposal sender and executor contracts.

2. Interchain Token Service

Here is a mindmap that explains the inner workings and important function calls of the Interchain Token Service. I'll be referring to this while explaining:
https://drive.google.com/file/d/1AQl1QwuYzXASMtsqL6VfIGoQza0Fygv_/view?usp=sharing



The diagram provides a high-level overview of how the main functions communicate. Refer to the legend to understand what each color represents. The way to read this mindmap would be to start from the top left corner.

1. The first contract we come across is the InterchainTokenService.sol contract. This is by far the heaviest contract (both functionality-wise and nSLOC-wise). The first level in the hierarchy (side-ways) are the callable functions in the contract. Each function either directly or indirectly (through internal functions - level 2 in side-ways hierarchy) deploys contracts on both source and destination chains. What differentiates the contracts deployed on the source and destination chains is the "remote" keyword used in the function names. The function names and comments provided by the team are self-explanatory so I will not be delving much deeper into explaining what each function does.

2. InterchainToken.sol and TokenManager.sol are both the originators of interchain transfer calls (since the functions in both contracts are external, thus call can originate from either). The

InterchainToken.sol calls the TokenManager.sol, which further calls the InterchainTokenService.sol contract. Here is how the flow looks:

```
InterchainToken.sol => TokenManager.sol => InterchainTokenService.sol
(function transmitSendToken()) => AxelarGateway.sol (function
callContract())
```

They both call the transmitSendToken() function to directly/indirectly (check disgram) to emit an event in the callContract function in the AxelarGateway.sol contract. The execution on the destination chain follows as mentioned by this example provided in the third paragraph of the Overview section in the README.md file: https://github.com/code-423n4/2023-07-axelar#overview

3. RemoteAddressValidator.sol - The main function of this contract is to add/remove trusted interchain token service addresses and add/remove Axelar supported gateway chains. This contract also provides a function, which can be used to validate if the sender is a valid interchain token service address.

4. StnadardizedToken.sol - Inherits from InterchainToken.sol to allow for interchain transfer (although this can be considered as originator of interchain transfer as well, I have not marked it in the diagram). This contract provides a setup function (Callable by onlyProxy) that sets up certain parameters like the distributor, tokenManager etc. Additionally, the distributor can use the token minting and burning functions.

5. TokenAddressStorage.sol - This can be considered as originator of interchain transfer since it inherits from the TokenManager.sol contract (not marked in diagram though). It has a setter function to set the token address as well. Although this contract is not important by itself, it is used by its child contracts - TokenManagerLockUnlock.sol, TokenManagerMintBurn.sol and TokenManagerLiqudiityPool.sol.

6. TokenManagerLockUnlock.sol, TokenManagerMintBurn.sol and TokenManagerLiqudiityPool.sol - These are different implementation types of the token manager and are differentiated by their implementation number (i.e. their index in the TokenManagerType enum located in the ITokenManagerTypes interface). Additionally, each contract has their own giveToken() and takeToken() functions that interact with the FlowLimit.sol contract.

7. FlowLimit.sol - This contract makes sure that while giving and taking tokens, the flow limit set by the operator is adhered to in order to prevent exploit attacks.

Time spent:

90 hours