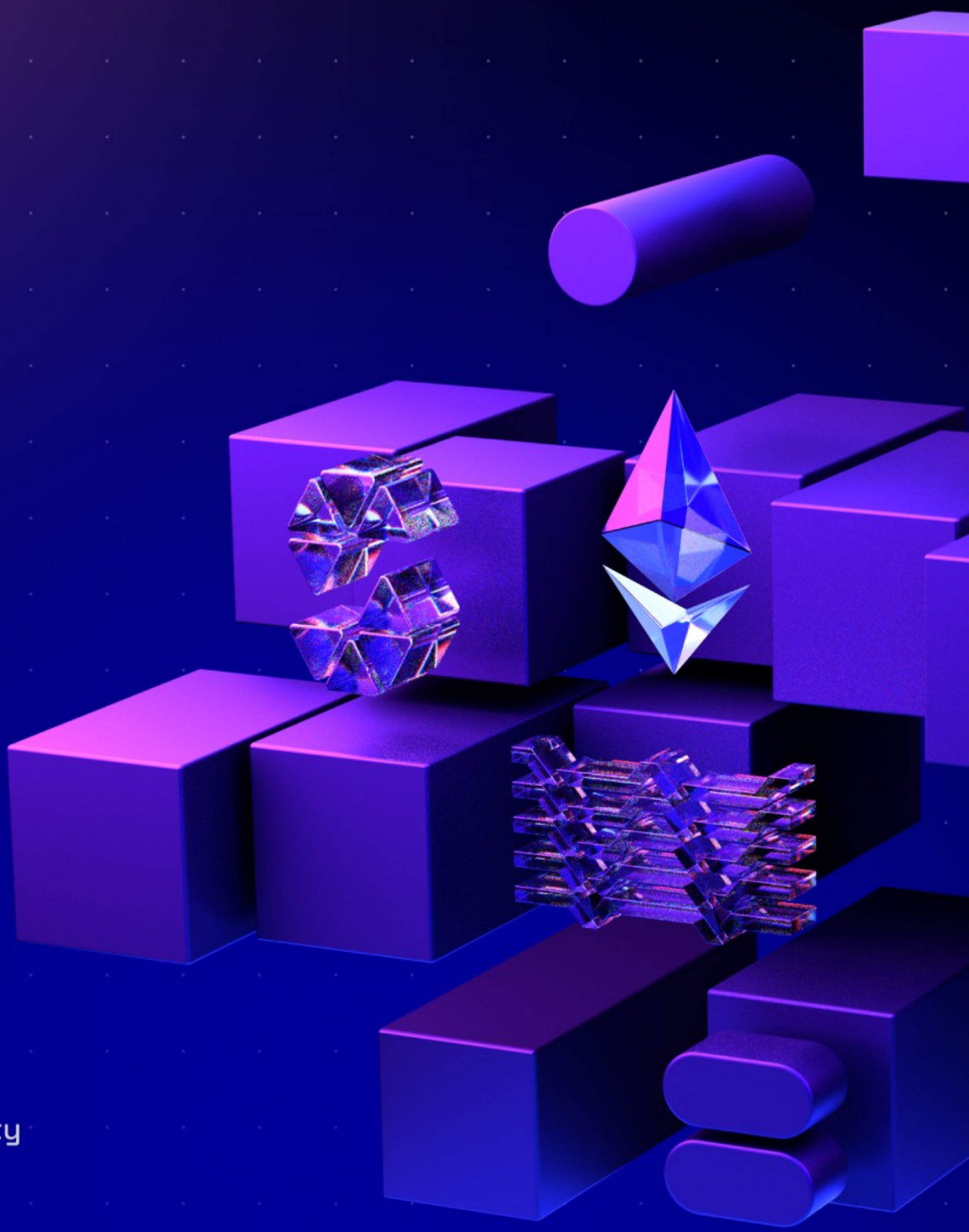
ackee
blockchain security

Axelar

Interchain Token Service

29.11.2024



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	29.11.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.

Rohanske nabrezi 717/4

186 00 Prague, Czech Republic

https://ackee.xyz

hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**

   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**

   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**

   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**

   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**

   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Lukáš Rajnoha	Lead Auditor
Jan Převrátil	Auditor
Michal Převrátil	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

WARNING This is not a final report, it is not intended for publication.

Revision 1.0

Axelar engaged Ackee Blockchain Security to perform a security review of the Axelar protocol with a total time donation of 18 engineering days in a period between October 29 and November 29, 2024, with Lukáš Rajnoha as the lead auditor.

The audit was performed on the commit `7e6916`[1] and the scope was the following:

- `InterchainTokenFactory.sol`
- `InterchainTokenService.sol`
- `TokenHandler.sol`
- `utils/TokenManagerDeployer.sol`
- `utils/InterchainTokenDeployer.sol`
- `utils/Create3AddressFixed.sol`
- `utils/GatewayCaller.sol`
- `utils/Create3Fixed.sol`
- `token-manager/TokenManager.sol`
- `types/InterchainTokenServiceTypes.sol`.

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. During the review, we paid special attention to:

- newly introduced ITS Hub interoperability;
- ensuring interchain token transfers work as intended;
- looking for issues in custom token managers;
- ensuring correct mintership privileges in interchain tokens;
- ensuring access controls are not too relaxed;
- proper upgradeability patterns in ITS contracts;
- looking for common issues such as data validation.

Our review resulted in 17 findings, ranging from Info to Medium severity. The most severe one M1 concerns missing recovery mechanisms for stuck funds when interchain transfers fail while utilizing the ITS Hub. M2 and L2 regard potential issues with non-standard and fee-on-transfer `ERC-20` tokens.

Ackee Blockchain Security recommends Axelar to:

- reconsider proper recovery mechanisms for stuck funds when utilizing the ITS Hub;
- assess support for non-standard and fee-on-transfer `ERC-20` tokens;
- address all other reported issues.

See Report Revision 1.0 for the system overview and trust model.

[1] full commit hash: `7e6916d0d5bfe0c2c1bc915cc1ac5b21f82d1b36`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- `Description`
- `Exploit scenario` (if severity is low or higher)
- `Recommendation`
- `Fix` (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	2	2	5	8	17

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
M1: Funds cannot be retrieved from failed interchain transactions	Medium	1.0	Reported
M2: Tokens with callbacks can break accounting	Medium	1.0	Reported
L1: `InterchainTokenFactory.deployInterchainToken` allows deploying empty token	Low	1.0	Reported
L2: Optional `ERC-20` functions required	Low	1.0	Reported

Finding title	Severity	Reported	Status
W1: `GatewayCaller` contract ambiguously reverts when on insufficient funds	Warning	1.0	Reported
W2: Missing token zero address check in `TokenManagerProxy`	Warning	1.0	Reported
W3: Several contracts receive ether, but never send it	Warning	1.0	Reported
W4: Factory does not check if canonical token exists when registering it	Warning	1.0	Reported
W5: `TokenManager` function selector clashes	Warning	1.0	Reported
I1: `TokenManager` contains implementation details of `RolesBase`	Info	1.0	Reported
I2: Duplicit message types	Info	1.0	Reported
I3: Inconsistent usage of type for `TokenManagerType`	Info	1.0	Reported
I4: Unused code	Info	1.0	Reported
I5: Excessive inheritance in `InterchainToken`	Info	1.0	Reported
I6: Typos or missing documentation	Info	1.0	Reported

Finding title	Severity	Reported	Status
I7: `TokenManager.tokenAddress` can be marked as pure	Info	1.0	Reported
I8: `InterchainTokenService.execute` function listed among internal functions	Info	1.0	Reported

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Lukáš Rajnoha	Lead Auditor
Jan Převrátil	Auditor
Michal Převrátil	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The Interchain Token Service is a modular cross-chain token bridging protocol built on Axelar Network. It enables seamless token transfers between blockchains through token managers that handle canonical ERC-20s and `InterchainTokens` with native cross-chain capabilities. The system supports both lock/unlock and mint/burn mechanics while maintaining consistent token properties across chains. The `InterchainTokenFactory` may provide additional guarantees like fixed supply and removed deployer privileges.

Trust Model

The Interchain Token Service inherits the security of Axelar GMP and the new ITS Hub. Users of the protocol have to trust the Axelar infrastructure to perform interchain relayings. In the context of this project, users have to trust the Axelar team to deploy and setup the `InterchainTokenService` contract and its proxies correctly. Users have to trust canonical `ERC-20` tokens on source chains.

Fuzzing

The fuzzing was performed with a total donation of 3 engineering days with Michal Převrátil as the fuzz test implementer. During the fuzzing process, no additional issues were discovered.

A manually-guided differential stateful fuzz test was developed during the review to test the correctness and robustness of the system. The differential fuzz test keeps its own Python state according to the system's specification. Assertions are used to verify the Python state against the on-chain state in contracts. The list of all implemented execution flows and invariants is available in Appendix B. These tests covered:

- interchain token deployments and canonical token registrations via the `InterchainTokenFactory` contract;
- interchain transfers in the `InterchainTokenService` contract, simulating both GMP and ITS Hub cross-chain relaying.

Findings

The following section presents the list of findings discovered in this revision.

M1: Funds cannot be retrieved from failed interchain transactions

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	various	Type:	Trust model

Description

Interchain transactions can result in irretrievable locked funds if they fail on the ITS Hub, as the infrastructure doesn't implement fund recovery mechanisms. For example, if a token is not deployed on the destination chain or if issues arise during interchain token amount conversions, the transaction fails without the ability to recover the burned (or locked) tokens.

Exploit scenario

A user may inadvertently lose funds if failures occur during the transaction relaying process. For instance:

1. A user initiates an interchain transfer from chain A to chain C via the ITS Hub.
2. The transaction succeeds on chain A, and the tokens are burned on the source chain.
3. The transaction fails during relaying on the ITS Hub (for example, because there is no token contract on chain C).
4. The transaction is not successfully relayed to chain C, which means the equivalent tokens are not minted on the destination chain.
5. Since the tokens were burned on chain A but not reminted on chain C, the user's funds were lost and remain unrecoverable at the current implementation.

Recommendation

Consider enhancing the ITS Hub to prevent irretrievable fund losses in interchain transactions that fail during relay on the ITS Hub. Implement recovery mechanisms or fallback procedures to mitigate the risk of irretrievable funds. If implementing comprehensive recovery mechanisms is not feasible, document these design limitations transparently.

Go back to Findings Summary

M2: Tokens with callbacks can break accounting

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	TokenHandler	Type:	Logic error

Description

The `TokenHandler` contract contains the `_transferTokenFromWithFee` helper function, which triggers a token transfer for tokens with an on-transfer fee and returns the amount transferred, excluding the fee amount. The return amount is calculated as a difference in the sender's balance before and after the transaction. To avoid problems with reentering this function via token callbacks, the `noReEntrancy` modifier guard is used. However, the sender's balance can also be increased via a direct transfer in the token callback. A malicious user can transfer tokens to themselves via a callback and increase their balance. This issue has two implications:

- breaking flow accounting with (`flowIn` and `flowOut`);
- transferring the full token amount (including the fee) cross-chain.

Exploit scenario

A malicious user initiated a cross-chain transfer. The ITS call includes the `TokenHandler.takeToken` function, which calls the `_transferTokenFromWithFee` function and returns its returned value as the amount to be transferred. Thus, more tokens will be transferred cross-chain than they actually should be. An analogous scenario can happen on the receiving side of the cross-chain transfer, where the full amount can be delivered to the receiving account (only routed via the `TokenHandler.giveToken` function instead).

Recommendation

Fixing the issue requires knowing the exact fee for the given token. Proposing a simple solution to the issue is thus quite hard. Therefore, consider how important it is to support fee-on-transfer tokens.

[Go back to Findings Summary]

L1: `InterchainTokenFactory.deployInterchainToken` allows deploying empty token

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	InterchainTokenFactory.deployInterchainToken	Type:	Data validation

Description

Method `deployInterchainToken` of `InterchainTokenFactory` contract allows token deployment without both the initial supply and minter. Deploying such a token is only a loss of funds for the deployer because the token cannot be used. ITS will only mint new tokens on interchain transactions, but because the token has no supply, no such transactions can be made.

Exploit scenario

1. The deployer deploys a token without an initial supply and minter — by accident or because he expects to be the minter by default.
2. The token is successfully deployed, but since it contains no supply, it can't be used and only pollutes the state space.
3. The deployer loses the full gas fees for the deployment rather than the partial gas fees for trying to mint

Recommendation

Add a check to revert the transaction if both the initial supply and minter are zero.

[Go back to Findings Summary]

L2: Optional `ERC-20` functions required

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	InterchainTokenFactory	Type:	Data validation

Description

The solution assumes that canonical `ERC-20` tokens registered in the `InterchainTokenService` implement optional functions:

- `name()`
- `symbol()`
- `decimals()`.

However, some `ERC-20` tokens may not implement these functions or may implement them with different return types as the ITS expects.

Exploit scenario

A user decides to register the canonical `MKR` token and deploy a token manager for it. `MKR` returns `bytes32` for the `name` and `symbol` functions instead of `string`, which the ITS expects. This causes an ABI decoding revert.

Recommendation

Consider adjusting the codebase to add support for non-standard tokens like `MKR`.

[Go back to Findings Summary]

W1: `GatewayCaller` contract ambiguously reverts when on insufficient funds

Impact:	Warning	Likelihood:	N/A
Target:	GatewayCaller	Type:	Data validation

Description

The `GatewayCaller.callContract` function forwards `gasValue` amount of ether to `gasService`, however it does not check if `msg.value` is at least of `gasValue`. If `msg.value` is lower than `gasValue` (unless the contract has additional funds), the function reverts without additional information.

Recommendation

Consider adding a new error (i.e., `NotEnoughFunds`), which is thrown when `msg.value` is lower than `gasValue` (or possibly `address(this).balance`) so that the user is properly notified about the cause of the revert.

Go back to Findings Summary

W2: Missing token zero address check in `TokenManagerProxy`

Impact:	Warning	Likelihood:	N/A
Target:	TokenManagerProxy	Type:	Data validation

Description

When registering a canonical token as an interchain token, a token manager (`TokenManagerProxy`) is deployed for the token. The `TokenManagerProxy.<constructor>` decodes the `tokenAddress` from the params function parameter using the `TokenManager.getTokenAddressFromParams` function. The constructor, however, does not contain any check if this address is non-zero. The non-zero address check is also absent in the `InterchainTokenFactory` and `InterchainTokenService` (possible entry point contracts for registering a canonical interchain token).

Listing 1. Excerpt from [TokenManagerProxy]()

```
/**
 * @notice Constructs the TokenManagerProxy contract.
 * @param interchainTokenService_ The address of the interchain token service.
 * @param implementationType_ The token manager type.
 * @param tokenId The identifier for the token.
 * @param params The initialization parameters for the token manager contract.
 */
constructor(address interchainTokenService_, uint256 implementationType_, bytes32 tokenId, bytes memory params) {
    if (interchainTokenService_ == address(0)) revert ZeroAddress();

    interchainTokenService = interchainTokenService_;
    implementationType = implementationType_;
    interchainTokenId = tokenId;

    address implementation_ = _tokenManagerImplementation(interchainTokenService_, implementationType_);
    if (implementation_ == address(0)) revert InvalidImplementation();

```

```
    (bool success, ) =
implementation_.delegatecall(abi.encodeWithSelector(IProxy.setup.selector,
params));
    if (!success) revert SetupFailed();

    tokenAddress =
IBaseTokenManager(implementation_).getTokenAddressFromParams(params);
}
```

Recommendation

The issue is closely tied with W4, which concerns not checking if the registered canonical token exists. Resolving that issue also simultaneously resolves this one. Alternatively, consider adding a zero address check for the `tokenAddress` when deploying a new `TokenManagerProxy` proxy contract.

Go back to Findings Summary

W3: Several contracts receive ether, but never send it

Impact:	Warning	Likelihood:	N/A
Target:	TokenHandler	Type:	Code quality

Description

Several contracts in the codebase can receive ether, but never send it.

The `TokenHandler` contract can receive ether but never sends it. Even though ether should not be sent to the contract under normal circumstances, in a rare case it would happen the ether will be locked inside the contract without any means to retrieve it. The payable functions allowing this are:

- `takeToken`
- `postTokenManagerDeploy`.

The `TokenManager` contract can receive ether via the `multicall` function which is `payable`. The contract, however, does not contain any function that sends ether, nor does the `TokenManagerProxy` which delegate-calls the `TokenManager` implementation.

Similarly, the `TokenManagerDeployer` contract marks the only `deployTokenManager` function as `payable` but the contract does not send ether.

Issues were detected using Wake static analysis.

Recommendation

Consider removing `payable` from functions in the mentioned contracts where possible.

Go back to Findings Summary

W4: Factory does not check if canonical token exists when registering it

Impact:	Warning	Likelihood:	N/A
Target:	InterchainTokenFactory	Type:	Data validation

Description

The `InterchainTokenFactory.registerCanonicalInterchainToken` function does not check if the registering token actually exists. Other factory functions such as `deployRemoteCanonicalInterchainToken` or `deployRemoteInterchainToken` check for the existence of the token using the following code:

Listing 2. Excerpt from InterchainTokenFactory

```
// The 3 lines below will revert if the token does not exist.
string memory tokenName = token.name();
string memory tokenSymbol = token.symbol();
uint8 tokenDecimals = token.decimals();
```

A similar check is missing in the `registerCanonicalInterchainToken` function.

Recommendation

Consider adding the missing check to the `registerCanonicalInterchainToken` function.

Go back to Findings Summary

W5: `TokenManager` function selector clashes

Impact:	Warning	Likelihood:	N/A
Target:	TokenManager	Type:	Code quality

Description

New token managers are deployed as the `TokenManagerProxy` contracts, which delegate-call to the pre-deployed `TokenManager` implementation contract. Both contracts include `address public immutable interchainTokenService;` in their bytecode. Thus, the selector in the proxy shadows the implementation selector. This shadowing also applies to other getters in `TokenManager`, namely:

- `tokenAddress`
- `interchainTokenId`
- `implementationType`.

These functions are, however, implemented always to revert when called from within the contract itself as it is designed only to be delegate-called by proxies:

Listing 3. Excerpt from [TokenManager]()

```
/**
 * @notice Reads the token address from the proxy.
 * @dev This function is not supported when directly called on the implementation. It
 * must be called by the proxy.
 * @return tokenAddress_ The address of the token.
 */
function tokenAddress() external view virtual returns (address) {
    revert NotSupported();
}
```

In contrast, the `interchainTokenService` function does not follow this pattern and is fully implemented and callable from both the implementation and proxy contract. Although functional, mixing design approaches may be confusing for external viewers. Additionally, the `implementationType` function documentation does not mention that the function is intended to be used only from the implementation contract, while the other functions do.

Recommendation

Consider modifying the `TokenManager.interchainTokenService` function to be a `pure` function which always reverts to unify with the other getter functions. Finally add explanation of the intent in the code documentation.

Go back to Findings Summary

I1: `TokenManager` contains implementation details of `RolesBase`

Impact:	Info	Likelihood:	N/A
Target:	TokenManager, RolesBase	Type:	Code quality

Description

The bit shifting operation `1 << X` is an implementation detail of the `RolesBase` contract, which is exposed in functions `_addAccountRoles` or `_transferAccountRoles`. Thanks to this design, bit shifting is used 5 times in the `TokenManager` contract. Functions with an interface similar to `_addRole` and `_transferRole`, which would accept multiple `uint8` role values, could be added to abstract away this implementation detail, making the code of `TokenManager` clearer.

Recommendation

Consider improving the function interface for multiple roles to make the code clearer. Function overloading for `_addRole` and `_transferRole` functions could be used to add at least a variant taking two `uint8` role arguments, which would be sufficient in the current codebase to abstract the bit shifting.

Go back to Findings Summary

I2: Duplicit message types

Impact:	Info	Likelihood:	N/A
Target:	various	Type:	Unused code

Description

Message types are defined in two places:

- as constants:

Listing 4. Excerpt from [InterchainTokenService]()

```
uint256 private constant MESSAGE_TYPE_INTERCHAIN_TRANSFER = 0;
uint256 private constant MESSAGE_TYPE_DEPLOY_INTERCHAIN_TOKEN = 1;
uint256 private constant MESSAGE_TYPE_DEPLOY_TOKEN_MANAGER = 2;
uint256 private constant MESSAGE_TYPE_SEND_TO_HUB = 3;
uint256 private constant MESSAGE_TYPE_RECEIVE_FROM_HUB = 4;
```

- as enum:

Listing 5. Excerpt from [InterchainTokenServiceTypes]()

```
enum MessageType {
    INTERCHAIN_TRANSFER,
    DEPLOY_INTERCHAIN_TOKEN,
    DEPLOY_TOKEN_MANAGER
}
```

Recommendation

Choose one of the options and use it everywhere and remove the other one.

[Go back to Findings Summary]()

I3: Inconsistent usage of type for `TokenManagerType`

Impact:	Info	Likelihood:	N/A
Target:	various	Type:	Code quality

Description

For `enum TokenManagerType` type `TokenManagerType` and `uint256` are used inconsistently. That can lead to unnecessary confusion. Outside of contract `InterchainTokenService.sol` is always used `uint256`, which makes casting necessary when comparing it to enum constants.

Examples where type `TokenManagerType` is used:

Listing 6. Excerpt from [InterchainTokenService]()

```
function deployTokenManager(
    bytes32 salt,
    string calldata destinationChain,
    TokenManagerType tokenManagerType,
    bytes calldata params,
    uint256 gasValue
) external payable whenNotPaused returns (bytes32 tokenId) {
```

Listing 7. Excerpt from [InterchainTokenService]()

```
function _deployRemoteTokenManager(
    bytes32 tokenId,
    string calldata destinationChain,
    uint256 gasValue,
    TokenManagerType tokenManagerType,
    bytes calldata params
) internal {
```

Examples where type `uint256` is used:

Listing 8. Excerpt from [InterchainTokenService]()

```
function tokenManagerImplementation(uint256 /*tokenManagerType*/) external view returns (address) {
```

Listing 9. Excerpt from [TokenHandler]()

```
function postTokenManagerDeploy(uint256 tokenManagerType, address tokenManager) external payable {
```

Listing 10. Excerpt from [TokenManagerDeployer]()

```
function deployTokenManager(
    bytes32 tokenId,
    uint256 implementationType,
    bytes calldata params
) external payable returns (address tokenManager) {
```

Recommendation

Unite the usage of types for `TokenManagerType`. Move the type to the dedicated `InterchainTokenServiceTypes.sol` and use it everywhere because it's more explicit, and casting is not needed.

[Go back to Findings Summary]()

I4: Unused code

Impact:	Info	Likelihood:	N/A
Target:	various	Type:	Unused code

Description

Following errors are not used:

Listing 11. Excerpt from [ITokenManagerDeployer]()

```
error AddressZero();
```

Listing 12. Excerpt from [ITokenManager]()

```
error TakeTokenFailed();
error GiveTokenFailed();
error NotToken(address caller);
error ZeroAddress();
error AlreadyFlowLimiter(address flowLimiter);
error NotFlowLimiter(address flowLimiter);
```

Listing 13. Excerpt from [IInterchainTokenService]()

```
error InvalidTokenManagerImplementationType(address implementation);
```

Listing 14. Excerpt from [IInterchainTokenFactory]()

```
error InvalidChainName();
```

Listing 15. Excerpt from [IInterchainTokenFactory]()

```
error NotOperator(address operator);
error NotServiceOwner(address sender);
```

Following contracts are not used in `using-for` directives:

Listing 16. Excerpt from [TokenHandler]

```
using SafeTokenTransfer for IERC20;
```

Listing 17. Excerpt from [InterchainToken]

```
using AddressBytes for bytes;
```

Recommendation

Please review all the unused errors and `using-for` directives and either use them in the corresponding place or remove them to simplify the codebase.

Go back to Findings Summary

I5: Excessive inheritance in `InterchainToken`

Impact:	Info	Likelihood:	N/A
Target:	InterchainToken	Type:	Code quality

Description

On line 19, the `ERC20` in the inheritance list of `InterchainToken` is excessive because `InterchainToken` already extends `ERC20Permit`, which itself already extends the `ERC20` contract.

Listing 18. Excerpt from [InterchainToken]()

```
19 contract InterchainToken is InterchainTokenStandard, ERC20, ERC20Permit,
   Minter, IInterchainToken {
```

Recommendation

Remove the excessive item `ERC20` from the inheritance list.

[Go back to Findings Summary]()

I6: Typos or missing documentation

Impact:	Info	Likelihood:	N/A
Target:	various	Type:	Code quality

Description

Various files contain typos or missing documentation: - Documentation in `TokenHandler` mentions `This interface` instead of `This contract`:

Listing 19. Excerpt from [TokenHandler]()

```
/**
 * @title TokenHandler
 * @notice This interface is responsible for handling tokens before
initiating an interchain token transfer, or after receiving one.
 */
contract TokenHandler is ITokenHandler, ITokenManagerType, ReentrancyGuard,
Create3AddressFixed {
```

- Constant in `ERC20Permit` is undocumented:

Listing 20. Excerpt from [ERC20Permit]()

```
if (uint256(s) >
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) revert
InvalidS();
```

- `TokenManager` has a missing `@dev note` that the intent is to only be called from TM proxy:

Listing 21. Excerpt from [TokenManager]()

```
/**
 * @notice Returns implementation type of this token manager.
 * @return uint256 The implementation type of this token manager.
 */
function implementationType() external pure returns (uint256) {
    revert NotSupported();
```

```
}
```

Recommendation

Correct the typos and add the missing documentation.

Go back to Findings Summary

I7: `TokenManager.tokenAddress` can be marked as pure

Impact:	Info	Likelihood:	N/A
Target:	TokenManager	Type:	Code quality

Description

`TokenManager.tokenAddress` function is marked as `view virtual`:

Listing 22. Excerpt from [TokenManager]()

```
/**
 * @notice Reads the token address from the proxy.
 * @dev This function is not supported when directly called on the implementation. It
 * must be called by the proxy.
 * @return tokenAddress_ The address of the token.
 */
function tokenAddress() external view virtual returns (address) {
    revert NotSupported();
}

/**
 * @notice A function that returns the token id.
 * @dev This will only work when implementation is called by a proxy, which stores the tokenId as an immutable.
 * @return bytes32 The interchain token ID.
 */
function interchainTokenId() public pure returns (bytes32) {
    revert NotSupported();
}

/**
 * @notice Returns implementation type of this token manager.
 * @return uint256 The implementation type of this token manager.
 */
function implementationType() external pure returns (uint256) {
    revert NotSupported();
}
```

Similarly, as `interchainTokenId` and `implementationType` getter functions, it is expected to be shadowed by the `TokenManagerProxy` contract and serve just as a placeholder (so that the functions are callable in the `TokenManager` contract). All of these three functions are implemented as `public immutable` variables in the `TokenManagerProxy`:

Listing 23. Excerpt from [TokenManagerProxy]()

```
contract TokenManagerProxy is BaseProxy, ITokenManagerProxy {
    bytes32 private constant CONTRACT_ID = keccak256('token-manager');

    address public immutable interchainTokenService;
    uint256 public immutable implementationType;
    bytes32 public immutable interchainTokenId;
    address public immutable tokenAddress;
```

Since all of these values will be hard coded during the `TokenManagerProxy` contract deployment, all three functions can be marked as `pure`.

Recommendation

Consider marking the `TokenManager.tokenAddress` as `pure`, the same as the `interchainTokenId` and `interchainTokenService` functions.

[Go back to Findings Summary]()

I8: `InterchainTokenService.execute` function listed among internal functions

Impact:	Info	Likelihood:	N/A
Target:	InterchainTokenService	Type:	Code quality

Description

The `execute` function in the `InterchainTokenService` contract is placed among internal functions, even though it is `external`.

Recommendation

Move the function out of `internal` functions block and place it to an appropriate position within the code.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Axelar: Interchain Token Service, 29.11.2024.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Execution of interchain transfer	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Token balances are correct accross connected chains	1.0	Success

Table 5. Wake fuzzing invariants

B.2. Detectors

```
wake detect unused-using-for

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
contract TokenHandler is ITokenHandler, ITokenManagerType, ReentrancyGuard, Create3AddressFixed {
    using SafeTokenTransferFrom for IERC20;
    using SafeTokenCall for IERC20;
    using SafeTokenTransfer for IERC20;

    /**
     * @notice This function gives token to a specified address from the token manager.
contracts/TokenHandler.sol

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
 * @dev This contract also inherits Minter and Implementation logic.
 */
contract InterchainToken is InterchainTokenStandard, ERC20, ERC20Permit, Minter, IInterchainToken {
    using AddressBytes for bytes;

    string public name;
    string public symbol;
contracts/interchain-token/InterchainToken.sol
```

Figure 1. Unused using for *detector*

```
wake detect locked-ether

[HIGH][MEDIUM] Contract receives ether but never sends it. [locked-ether]
 * @title TokenHandler
 * @notice This interface is responsible for handling tokens before initiating an interchain token transfer,
 */
contract TokenHandler is ITokenHandler, ITokenManagerType, ReentrancyGuard, Create3AddressFixed {
    using SafeTokenTransferFrom for IERC20;
    using SafeTokenCall for IERC20;
    using SafeTokenTransfer for IERC20;
contracts/TokenHandler.sol
    This function can receive ether.
         * @return uint256 The amount of token actually taken, which could be different for certain token typ
         */
        // slither-disable-next-line locked-ether
        function takeToken(bytes32 tokenId, bool tokenOnly, address from, uint256 amount) external payable re
            address tokenManager = _create3Address(tokenId);
            (uint256 tokenManagerType, address tokenAddress) = ITokenManagerProxy(tokenManager).getImplementa
    contracts/TokenHandler.sol
    This function can receive ether.
         * @param tokenManager The address of the token manager.
         */
        // slither-disable-next-line locked-ether
        function postTokenManagerDeploy(uint256 tokenManagerType, address tokenManager) external payable {
            // For lock/unlock token managers, the ITS contract needs an approval from the token manager to
            if (tokenManagerType == uint256(TokenManagerType.LOCK_UNLOCK) || tokenManagerType == uint256(Tok
                ITokenManager(tokenManager).approveService();
    contracts/TokenHandler.sol
```

Figure 2. Sample from Locked ether *detector*

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz