# Axelar

## Interchain Token Service

by Ackee Blockchain

*11.4.2024*

# Contents


1. Document Revisions . . . 8
2. Overview . . . 9
   2.1. Ackee Blockchain . . . 9
   2.2. Audit Methodology . . . 9
   2.3. Finding classification . . . 10
   2.4. Review team . . . 12
   2.5. Disclaimer . . . 12
3. Executive Summary . . . 13
   Revision 1.0 . . . 13
   Revision 1.1 . . . 14
   Revision 2.0 . . . 15
   Revision 2.1 . . . 17
   Revision 3.0 . . . 17
   Revision 4.0 . . . 19
   Revision 4.1 . . . 20
   Revision 5.0 . . . 20
   Revision 5.1 . . . 21
   Revision 6.0 . . . 22
   Revision 6.1 . . . 23
   Revision 7.0 . . . 23
   Revision 8.0 . . . 24
   Revision 8.1 . . . 25
   Revision 8.2 . . . 25
4. Summary of Findings . . . 26
5. Report revision 1.0 . . . 33
   5.1. System Overview . . . 33

5.2. Trust Model . . . 35

L1: Missing validations . . . 36

W1: Duplicated code . . . 38

W2: Malicious token registration . . . 39

W3: Identical function body . . . 41

W4: Unused internal functions . . . 42

W5: Usage of `solc` optimizer . . . 43

I1: Redundant data validation . . . 44

I2: Missing documentation . . . 45

6. Report revision 1.1 . . . 46

6.1. System Overview . . . 46

L2: Expected revert . . . 48

L3: Missing validations . . . 50

W6: Lack of events . . . 51

W7: Duplicated code . . . 53

7. Report revision 2.0 . . . 54

7.1. System Overview . . . 54

7.2. Trust Model . . . 58

H1: Express receive double execution . . . 59

M1: Gateway token check missing . . . 61

M2: `toAddress` missing validation . . . 63

L4: `expressReceiveTokenWithData` spoofed data . . . 66

L5: `sendHash` is not unique . . . 68

W8: Express receive functions can be called by recipient . . . 71

W9: `IInterchainTokenExecutable` typo . . . 73

W10: Misleading `TokenManagerNotDeployed` error name . . . 74

W11: `LinkerRouter` initial trusted parameters cannot be set . . . 76

W12: `PREFIX_CANONICAL_TOKEN_ID` typo . . . 78
W13: Token manager implementations order validation . . . 79
W14: `requiresApproval` misleading value . . . 81
W15: Different decimals not handled . . . 82
I3: `IInterchainTokenService` event parameter typo . . . 84
I4: `InterchainToken` revert if max approval given . . . 85
I5: `StringToAddress` library unused . . . 87
I6: Use `type(uint256).max` for infinite flow limit . . . 88
I7: Token manager send function names . . . 90
I8: `LinkerRouter` remote addresses normalization . . . 92
I9: Unused functions and variables . . . 93
I10: `InterchainTokenServiceProxy` unused constructor parameter . . . 95
I11: `_executeWithToken` redundant modifier . . . 96
8. Report revision 2.1 . . . 98
8.1. System Overview . . . 98
9. Report revision 3.0 . . . 99
9.1. System Overview . . . 99
9.2. Trust Model . . . 100
H2: Wrong variable passed to hook . . . 102
H3: Tokens with callbacks can artificially increase cross-chain transfer amount . . . 104
M3: Operator slot incorrect preimage . . . 106
M4: Proposed role not cleared when accepted . . . 108
M5: Lack of destination chain validation . . . 110
M6: Incorrect accounting of flowIn for fee-on-transfer tokens . . . 111
M7: Front-running express execute with copy of gateway payload . . . 113
M8: Tokens with callbacks can break the flow accounting . . . 117

L6: Chain name validation . . . 119
W16: Return of literal instead of enum . . . 121
W17: Manager implementation zero address check . . . 122
W18: Prefix incorrectly calculated . . . 124
W19: Lack of contract prefixes in slot preimages . . . 125
W20: Code-comment discrepancy . . . 126
I12: Reentrancy lock private . . . 128
I13: Typo in function parameter name . . . 129
10. Report revision 4.0 . . . 130
10.1. System Overview . . . 130
10.2. Actors . . . 130
10.3. Trust Model . . . 130
M9: Tokens with callbacks can artificially increase cross-chain transfer amount . . . 131
W21: Token id can differ on the deployment method . . . 132
W22: Chain name data validation . . . 134
W23: Possible code injection on deployment on remote . . . 135
W24: Change in the enum order can affect access controls . . . 137
I14: Incorrect inline documentation . . . 138
I15: Ambiguous revert message . . . 139
I16: Code duplication . . . 140
11. Report revision 5.0 . . . 141
11.1. System Overview . . . 141
11.2. Actors . . . 142
11.3. Trust Model . . . 142
W25: Hardcoded metadata version/prefix . . . 143
W26: One-step role transfer . . . 144

W27: Incorrect parent contract . . . . 146
W28: A danger of the interchain service's balance drainage . . . . 147
I17: Incorrect or missing documentation . . . . 149
12. Report revision 6.0 . . . . 151
12.1. System Overview . . . . 151
12.2. Actors . . . . 157
12.3. Trust Model . . . . 157
W29: Missing symbol validation . . . . 158
I18: Missing documentation . . . . 159
I19: `InterchainToken` has code unrelated to the token logic . . . . 160
I20: Unused constant . . . . 161
I21: Inconsistent code style . . . . 162
13. Report revision 7.0 . . . . 164
13.1. System Overview . . . . 164
13.2. Actors . . . . 165
13.3. Trust Model . . . . 165
C1: The `executeWithToken` function may be called by anyone . . . . 166
H4: `executeWithToken` allows to perform interchain operations even when the protocol is paused . . . . 168
M10: ERC-20 double approval . . . . 170
M11: Optional ERC-20 functions required . . . . 172
W30: `executeWithToken` and `expressExecuteWithToken` functions do not check for the correctness of payload arguments . . . . 174
14. Report revision 8.0 . . . . 175
C2: Mint privilege . . . . 176
W31: Zero amount transfers possible . . . . 181
W32: Canonical lock/unlock fee-on-transfer tokens . . . . 182

I22: Unused using-for directives . . . 184
Appendix A: How to cite . . . 185
Appendix B: Glossary of terms . . . 186
Appendix C: Wake outputs . . . 187
C.1. H1 proof of concept . . . 187
C.2. M10 proof of concept . . . 188
C.3. C2 proof of concept . . . 190
C.4. I22 list of all unused using-for detections . . . 191

# 1. Document Revisions

| | | |
|---|---|---|
| 1.0 | Final report | 19.4.2023 |
| 1.1 | Fix review and updates revision | 12.5.2023 |
| 2.0-draft | Re-audit draft | 23.6.2023 |
| 2.0 | Re-audit final report | 26.6.2023 |
| 2.1 | Fix review | 27.6.2023 |
| 3.0-draft | Re-audit draft | 10.8.2023 |
| 3.0 | Re-audit final report | 11.8.2023 |
| 4.0 | Re-audit final report | 24.10.2023 |
| 4.1 | Issue adjustment | 31.10.2023 |
| 5.0 | Re-audit final report | 13.11.2023 |
| 5.1 | Fix review | 16.11.2023 |
| 6.0 | Re-audit final report | 12.12.2023 |
| 7.0 | Re-audit final report | 15.2.2024 |
| 8.0 | Fuzzing report | 18.3.2024 |
| 8.1 | Fix review | 21.3.2024 |
| 8.2 | Canonical fee-on-transfer tokens issue severity update | 11.4.2024 |

# 2. Overview

This document presents our findings in reviewed contracts.

## 2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

## 2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

## 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 (such as deployment scripts, compiler configuration, use of 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** | **-** |
| *Impact* | **High** | Critical | High | Medium | - |
| | **Medium** | High | Medium | Medium | - |
| | **Low** | Medium | Medium | 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 (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), 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 (see above) 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

| Member's Name | Position |
| --- | --- |
| Michal Převrátil | Auditor |
| Andrey Babushkin | Auditor |
| Štěpán Šonský | Auditor |
| Lukáš Böhm | Auditor |
| Michal Převrátil | Auditor |
| Miroslav Škrabal | Auditor |
| Jan Kalivoda | 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

## Revision 1.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 5 engineering days in a period between April 11 and April 19, 2023 and the lead auditor was Štěpán Šonský. The scope of the audit is token linker, which can deploy wrapped versions of existing tokens on multiple chains, and also that the wrapped token being deployed is cross-chain native.

The audit has been performed on the commit 9bb6e07.

We began our review by using static analysis tools, namely Wake. Then we took a deep dive into the codebase and set the following goals:

- check access controls,
- check correctness of LinkedTokenData operations using Wake fuzzer,
- check cross-chain data integrity (e.g. IDs, decimals...),
- detect possible reentrancies in the code,
- look for common issues such as data validation.

Upon conducting an in-depth analysis, our examination yielded a total of 8 findings, with the severity levels ranging from Info to Low. Overall, the codebase is incomplete, it contains a lot of "TODO" markers and unused functions (see 5.1). Also, there are a lot of code duplications, which can be easily refactored/removed to improve architecture, readability and secure maintainability. Given the current state of the codebase, we don't recommend deployment or public dissemination of the source code until the mentioned issues have been thoroughly addressed.

Ackee Blockchain recommends Axelar:

- clean the code of unused functions,
- resolve all TODOs, implement missing parts,
- be aware of potentially malicious token contracts,
- remove duplicated code,
- add detailed documentation.

See Revision 1.0 for the system overview of the codebase.

## Revision 1.1

The review was done on the given commit: `c03c4eb` with a time donation of 5 engineering days between May 1 and May 12, 2023, and the lead auditor was Štěpán Šonský.

For revision 1.1 we had the following goals:

- review the fixes implemented in response to our previous audit,
- check the implementation and logic of minting limits,
- evaluate the `InterchainToken` functions,
- analyze express logic for possible reentrancies and double-spending,
- revisit the previously audited code to ensure its interactions with the new code do not introduce new vulnerabilities.

We began with fix review, comparing codebase differences and analyzing changes and new features. The codebase analysis revealed that most of the issues identified in the previous scope have been effectively addressed, thereby enhancing the overall functionality of the contract. The TODOs from the previous version were found to be satisfactorily resolved, indicating a more mature state of the project. However, we identified a new low severity issue L2, missing validations, events and duplicated code.

Ackee Blockchain recommends addressing these in the next revision to further enhance the contract's security, functionality and readability.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

## Revision 2.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 10 engineering days in a period between June 12 and June 23, 2023 and the lead auditor was Michal Převrátil.

The scope of the audit was an interchain token service contract with its dependencies. The service contract implements the deployment of token managers responsible for managing different interchain token pairs. All interchain messages are handled by the service contract, which serves as the central point of the whole system. For an additional fee, there is an express receive functionality giving the ability to receive tokens from a relayer at his own risk. The project has been significantly rewritten since the last audit.

The audit was performed on the commit 1e40298.

We began our review with deep dive into the service deployment functions. In parallel, we implemented unit tests in the Wake framework to better understand the architecture and to test edge cases of the interchain service setup and token manager deployment. After that, we focused on different token manager implementations and their interactions with the service contract. With the system's core functionality covered, we moved on to libraries and utility contracts. We implemented a fuzz test for an ERC-20 token implementation with the EIP-2612 permit extension, which is included in the project as a dependency. We concluded our review by using static analysis tools, namely Wake and Slither, and by implementing a complex fuzz test verifying the overall system's functionality.

During the review, we paid special attention to:

- checking it is not possible for anyone else to deploy a token manager with a given `tokenId` except for the original deployer,
- making sure deployment and flow limit rules cannot be bypassed,
- looking for access control issues and trust model problems,
- ensuring tokens cannot be stolen from deployed token managers and from relayers responsible for the express receive functionality,
- checking users of the service are appropriately protected from losing their tokens.

Our review resulted in 22 findings, ranging from Info to Critical severity. The most severe one results in tokens stolen from a relayer by calling the express receive function multiple times (see C1).

Ackee Blockchain recommends Axelar:

- fix the C1 critical issue together with both medium severity issues M1 and M2,
- clean the code of unused functions,
- add detailed documentation, especially the user-facing documentation properly describing the risks and caveats of the current solution (L4, W15).

See Revision 2.0 for the system overview of the codebase.

## Revision 2.1

Axelar engaged Ackee Blockchain to perform a fix review on the commit [3bb6705](). Except for fixes reported in the previous revision, the code was significantly refactored, some features were removed while others were added. This review aimed to ensure the fixes were implemented correctly without auditing the new features.

The status of all reported issues was updated and can be found in the findings table. Some issues include client responses.

## Revision 3.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 8 engineering days in a period between July 31 and August 10, 2023 and the lead auditor was Miroslav Škrabal.

The scope of the audit was a pull request from the branch `feat/fee-on-transfer-separate` to the last audit commit. The pull request merges the changes up to the commit [412dce9]() to the audit commit from the previous revision [Revision 2.1](), i.e. the commit `3bb6705`. The pull request mainly introduces many refactorings, however, the architecture from the previous revision remains almost the same. The directories `contracts/test` and `contracts/examples` were excluded from the review. A few new contracts were added: `NoReEntrancy`, `Operatable` and a new implementation of the `TokenManager`.

During the review, we mainly focused on the `diff` against the previous revision of the protocol, which was audited and thus considered secure. We reviewed the codebase holistically and analyzed the changes in the broader context.

The majority of the review was spent on manual auditing. We noticed that the codebase lacks proper unit test coverage for certain contracts and thus we wrote a few unit tests in Wake to ensure the correctness of the implementation. This yielded a few findings, most notably H2 which was clearly a result of the missing unit tests. We also used Wake detectors to statically analyze the codebase, mainly to guide the analysis of possible reentrancies.

Because the architecture was mainly unchanged, we followed the high-level objectives as set in the previous revision. Additionally, we paid special attention to:

- logic regarding fee-on-transfer tokens,
- reentrancies through tokens with hooks,
- front-running of main transaction types (deployment, execute, express call,..),
- keccak slot addresses as used in the unstructured storage pattern.

Our review resulted in 16 findings, ranging from Info to High severity.

Based on the review we recommend Axelar following these high-level recommendations:

- ensure high test coverage for all contracts (would have prevented H2),
- be careful when using the unstructured storage pattern (would have prevented M3, W18),
- double-check the NatSpec comments when refactoring contracts (would have prevented W20),
- follow the Check-Effect-Interaction pattern (would have prevented M7),
- carefully test and consider the support of fee-on-transfer tokens (the

logic corresponding to supporting these tokens caused multiple issues),

- address all the reported issues.

## Revision 4.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 8 engineering days in a period between October 9 and October 24, 2023 and the lead auditor was Jan Kalivoda.

The scope was changed multiple times during the time slot of the audit, but all the time it was the difference between the audit commit from the previous revision Revision 3, i.e. the commit `412dce9`, and the following commits:

- `14fecc4` - First auditing week spent 1-2 MD,
- `04972f9` - First auditing week spent 1 MD + time spent on the second auditing week and then left 1 MD on the third week for another incoming commit,
- `3ef1db2` - New commit introducing new contracts usage, spent 1 MD.

Changes were mostly fixes from the Revision 3 and code refactors for simplicity (e.g. Express Service implementation for ITS). Therefore, most of the time was spent on double-checking the whole codebase and the fix review from the previous revision. However, there were also introduced new contracts that undertook a review. The last commit introduced the usage of a new contract for access control.

We began our review by using static analysis tools, namely Wake. Then we took a deep dive into the codebase and set the following goals:

- ensuring the code refactor does not introduce new vulnerabilities,

- ensuring the deployment with the token registrars is correct,
- checking if the fixes from previous revisions were implemented correctly,
- otherwise goals remained the same as in the previous revisions.

Our review resulted in 8 findings, ranging from Informational to High severity. Most of the findings are related to the new contracts.

Ackee Blockchain recommends Axelar:

- add documentation for the new code,
- address all the reported issues.

## Revision 4.1

We adjusted the severity of [M9: Tokens with callbacks can artificially increase cross-chain transfer amount] from High to Medium, based on the client's request since in the wider context it affects significantly only specific use cases and not the whole system.

## Revision 5.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 5 engineering days in a period between November 7 and November 14, 2023, and the lead auditor was Štěpán Šonský.

The first day of the audit was spent on the commit `e4f9953` and then the final commit `73b91cb` ([v1.0.0-beta.2]) was delivered by the client. We began our review by using static analysis tools, namely [Wake], then we focused on the [diff] since the last audit revision and then we did a full audit of crucial contracts with most changes - `InterchainTokenFactory`, `InterchainTokenService` and `BaseInterchainToken`. We also performed a fix review of the previous revision.

During our code review, we set the following goals:

- checking fixes from revision 4.0,
- reviewing all code changes,
- ensuring delegatecalls cannot be misused,
- ensuring the correct usage of `msg.value` and `gasValue` in interchain transfers and calls,
- ensuring that the `setup` function cannot be called multiple times,
- checking the express implementation,
- ensuring that the `expressExecute` function cannot be called multiple times,
- validating the presence and the correctness of the NatSpec documentation,
- and checking for common issues like re-entrancy, access controls and data validations.

The overall code quality and architecture have a high standard, our review resulted in 5 findings, ranging from Informational to Warning severity.

Ackee Blockchain recommends Axelar:

- unify metadata version/prefix implementation,
- consider using only two-step role transfer,
- fix and finish the NatSpec documentation.

## Revision 5.1

Ackee Blockchain performed a fix review of findings reported in Revision 5.0 on commit `0a00533` (v1.0.0-beta.3). 4/5 issues were addressed, only the W26 was acknowledged due to the required flexibility.

## Revision 6.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 5 engineering days in a period between December 4 and December 12, 2023, and the lead auditor was Štěpán Šonský.

The review was done on commit f2af0a1 (v1.0.0-beta.4 + additional changes). The diff since the last revision was analyzed and also, a complete code review was performed on contracts with crucial changes (`TokenHandler`, `InterchainTokenService`, `InterchainTokenFactory`, and `TokenManager`).

We began our review by using static analysis tools, namely Wake. During our review, we paid special attention to:

- newly introduced `TokenHandler` contract,
- checking new delegatecalls to `TokenHandler` from `InterchainTokenService`,
- checking the new deployer balance logic,
- and checking for common issues like re-entrancy, access controls and data validations.

The code quality remains high and the architecture has been improved using `TokenHandler`. Our review resulted in 5 findings, ranging from Informational to Low severity. The most severe L7 points to missing token symbol validation in `InterchainToken` contract.

Ackee Blockchain recommends Axelar:

- add `tokenSymbol` validation in `InterchainToken.init`,
- extract init logic to abstract contract,

- remove unused constant,
- add missing documentation.

## Revision 6.1

Ackee Blockchain performed a fix review of findings in Revision 6.0 on commit `b37ad32`. 3/5 issues were addressed, and two were acknowledged. We adjusted the severity of W29: Missing symbol validation from Low to Warning, based on the client's request since the token may be redeployed with another salt set in case of a mistake.

## Revision 7.0

Axelar engaged Ackee Blockchain to perform a security review of the Interchain Token Service with a total time donation of 7 engineering days in a period between January 29 and February 13, 2024, and the lead auditor was Andrey Babushkin.

The review was done on commit 0977738. The diff since the last revision was analyzed and also, a complete code review was performed on contracts with crucial changes (`TokenHandler`, `InterchainTokenService`, and `InterchainTokenFactory`).

We began our review by using static analysis tools, namely Wake. During our review, we paid special attention to:

- checking that there is no possibility of using a forged payload, token or amount to newly implemented functions,
- checking if there can be no collisions in token IDs for different tokens with the integration of gateway tokens,
- validating that canonical gateway token managers cannot be deployed by unprivileged users,

- verifying that the protocol works with non-standard ERC-20 tokens,
- validating if the protocol is prone to gateway tokens behaving incorrectly,
- and checking for common issues like reentrancy, access controls and data validations.

Our review resulted in 5 findings, ranging from Warning to Critical severity. Testing with Wake and manual code review resulted in finding the two most severe issues, C1 and H1.

These issues suggest lower code quality compared to previous revisions and a lack of adequate testing of access controls. To prevent the occurrence of such errors in the future, we recommend adding new test suites with validating permissions of access to each external function for all system actors.

Ackee Blockchain recommends Axelar:

- implement all the required access controls,
- review the integration logic of non-standard ERC-20 tokens,
- implement proper data validation of the payload to execute functions.

## Revision 8.0

Ackee Blockchain and Axelar agreed to update the fuzz test from revision 2.0 to cover changes in the codebase. The fuzzing was performed with a total donation of 5 engineering days in a period between March 4 and March 11, 2024, with Michal Převrátil as the fuzz test implementer.

The fuzz test was updated for the commit a6d33b8. During the fuzzing process, the C2 issue was discovered and disclosed to the client.

## Revision 8.1

Ackee Blockchain performed a fix review of findings in Revision 8.0.

The C2 issue was reviewed in a private repository on a codebase identical to the one in the commit a6d33b8. The fix was not implemented on the latest commit in the public repository and thus its application on future commits cannot be commented on.

A fix review for the findings W31 and I22 was performed on the commits ece8a09 and b4b0759 in the public repository. The finding W32 was acknowledged.

## Revision 8.2

The severity of the W32 finding was changed from Medium to Warning based on the client’s request since the requested change is currently out of scope for the project and the issue does not affect in-scope functionality.

# 4. Summary of Findings

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

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Solution*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

| Critical | High | Medium | Low | Warning | Info |
| --- | --- | --- | --- | --- | --- |
| 2 | 4 | 11 | 6 | 32 | 22 |

*Table 2. Findings Count by Severity*

| | Severity | Reported | Status |
| --- | --- | --- | --- |
| L1: Missing validations | Low | 1.0 | Fixed |
| W1: Duplicated code | Warning | 1.0 | No longer valid |
| W2: Malicious token registration | Warning | 1.0 | Acknowledged |
| W3: Identical function body | Warning | 1.0 | Fixed |
| W4: Unused internal functions | Warning | 1.0 | No longer valid |
| W5: Usage of `solc` optimizer | Warning | 1.0 | Acknowledged |

| | Severity | Reported | Status |
|---|---|---|---|
| I1: Redundant data validation | Info | 1.0 | Fixed |
| I2: Missing documentation | Info | 1.0 | Fixed |
| L2: Expected revert | Low | 1.1 | No longer valid |
| L3: Missing validations | Low | 1.1 | Fixed |
| W6: Lack of events | Warning | 1.1 | Partially fixed |
| W7: Duplicated code | Warning | 1.1 | No longer valid |
| H1: Express receive double execution | High | 2.0 | Fixed |
| M1: Gateway token check missing | Medium | 2.0 | Fixed |
| M2: `toAddress` missing validation | Medium | 2.0 | Fixed |
| L4: `expressReceiveTokenWithData` spoofed data | Low | 2.0 | Fixed |
| L5: `sendHash` is not unique | Low | 2.0 | Fixed |
| W8: Express receive functions can be called by recipient | Warning | 2.0 | Fixed |
| W9: `IInterchainTokenExecutable` typo | Warning | 2.0 | Fixed |
| W10: Misleading `TokenManagerNotDeployed` error name | Warning | 2.0 | Fixed |

| | Severity | Reported | Status |
|---|---|---|---|
| W11: `LinkerRouter` initial trusted parameters cannot be set | Warning | 2.0 | Partially fixed |
| W12: `PREFIX_CANONICAL_TOKEN_ID` typo | Warning | 2.0 | Fixed |
| W13: Token manager implementations order validation | Warning | 2.0 | Fixed |
| W14: `requiresApproval` misleading value | Warning | 2.0 | Fixed |
| W15: Different decimals not handled | Warning | 2.0 | Acknowledged |
| I3: `IInterchainTokenService` event parameter typo | Info | 2.0 | Fixed |
| I4: `InterchainToken` revert if max approval given | Info | 2.0 | Partially fixed |
| I5: `StringToAddress` library unused | Info | 2.0 | Fixed |
| I6: Use `type(uint256).max` for infinite flow limit | Info | 2.0 | Acknowledged |
| I7: Token manager send function names | Info | 2.0 | Not fixed |
| I8: `LinkerRouter` remote addresses normalization | Info | 2.0 | Not fixed |
| I9: Unused functions and variables | Info | 2.0 | Partially fixed |

| | Severity | Reported | Status |
|---|---|---|---|
| I10: `InterchainTokenServiceProxy` unused constructor parameter | Info | 2.0 | Fixed |
| I11: `_executeWithToken` redundant modifier | Info | 2.0 | Fixed |
| H2: Wrong variable passed to hook | High | 3.0 | Fixed |
| H3: Tokens with callbacks can artificially increase cross-chain transfer amount | High | 3.0 | Fixed |
| M3: Operator slot incorrect preimage | Medium | 3.0 | Fixed |
| M4: Proposed role not cleared when accepted | Medium | 3.0 | Fixed |
| M5: Lack of destination chain validation | Medium | 3.0 | No longer valid |
| M6: Incorrect accounting of flowIn for fee-on-transfer tokens | Medium | 3.0 | Not fixed |
| M7: Front-running express execute with copy of gateway payload | Medium | 3.0 | Fixed |
| M8: Tokens with callbacks can break the flow accounting | Medium | 3.0 | Not fixed |
| L6: Chain name validation | Low | 3.0 | Fixed |

| | Severity | Reported | Status |
|---|---|---|---|
| W16: Return of literal instead of enum | Warning | 3.0 | Fixed |
| W17: Manager implementation zero address check | Warning | 3.0 | Not fixed |
| W18: Prefix incorrectly calculated | Warning | 3.0 | No longer valid |
| W19: Lack of contract prefixes in slot preimages | Warning | 3.0 | Not fixed |
| W20: Code-comment discrepancy | Warning | 3.0 | Not fixed |
| I12: Reentrancy lock private | Info | 3.0 | Fixed |
| I13: Typo in function parameter name | Info | 3.0 | Fixed |
| M9: Tokens with callbacks can artificially increase cross-chain transfer amount | Medium | 4.0 | Fixed |
| W21: Token id can differ on the deployment method | Warning | 4.0 | Fixed |
| W22: Chain name data validation | Warning | 4.0 | No longer valid |
| W23: Possible code injection on deployment on remote | Warning | 4.0 | Not fixed |
| W24: Change in the enum order can affect access controls | Warning | 4.0 | Acknowledged |

ackee | blockchain security

| | Severity | Reported | Status |
|---|---|---|---|
| I14: Incorrect inline documentation | Info | 4.0 | Not fixed |
| I15: Ambiguous revert message | Info | 4.0 | Not fixed |
| I16: Code duplication | Info | 4.0 | Fixed |
| W25: Hardcoded metadata version/prefix | Warning | 5.0 | Fixed |
| W26: One-step role transfer | Warning | 5.0 | Acknowledged |
| W27: Incorrect parent contract | Warning | 5.0 | Fixed |
| W28: A danger of the interchain service's balance drainage | Warning | 5.0 | Fixed |
| I17: Incorrect or missing documentation | Info | 5.0 | Fixed |
| W29: Missing symbol validation | Warning | 6.0 | Fixed |
| I18: Missing documentation | Info | 6.0 | Fixed |
| I19: `InterchainToken` has code unrelated to the token logic | Info | 6.0 | Acknowledged |
| I20: Unused constant | Info | 6.0 | Fixed |
| I21: Inconsistent code style | Info | 6.0 | Acknowledged |
| C1: The `executeWithToken` function may be called by anyone | Critical | 7.0 | Fixed |

| | Severity | Reported | Status |
|---|---|---|---|
| H4: `executeWithToken` allows to perform interchain operations even when the protocol is paused | High | 7.0 | Fixed |
| M10: ERC-20 double approval | Medium | 7.0 | Fixed |
| M11: Optional ERC-20 functions required | Medium | 7.0 | Acknowledged |
| W30: `executeWithToken` and `expressExecuteWithToken` functions do not check for the correctness of payload arguments | Warning | 7.0 | Fixed |
| C2: Mint privilege | Critical | 8.0 | Fixed |
| W31: Zero amount transfers possible | Warning | 8.0 | Fixed |
| W32: Canonical lock/unlock fee-on-transfer tokens | Warning | 8.0 | Acknowledged |
| I22: Unused using-for directives | Info | 8.0 | Fixed |

*Table 3. Table of Findings*

# 5. Report revision 1.0

## 5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

### Contracts

Contracts we find important for better understanding are described in the following section.

#### InterchainTokenService

`InterchainTokenService` is the core contract of the protocol. Inhretits from `AxelarExecutable`, `EternalStorage` and `Upgradable`. The contract is used for token registration and deployment on selected chains. Axelar's Create3Deployer contract handles the creation itself. Receiving functions are protected by `onlySelf` modifier. The contract contains a lot of TODOs, namely these functions are incomplete:

- `registerOriginGatewayToken`
- `registerRemoteGatewayToken`
- `sendSelf`
- `callContractWithSelf`
- `selfSendToken`
- `selfSendTokenWithData`
- `_sendToken`
- `_sendTokenWithData`

#### InterchainToken

`InterchainToken` is not used, inherits from `ERC20` and adds 2 functions `interchainTransfer` and `interchainTransferFrom`, which aren't implemented.

#### LinkedTokenData

A library for creating and reading `bytes32 tokenData`. It uses bitmasks for various flags, e.g. `IS_ORIGIN_MASK`, `IS_GATEWAY_MASK`, `IS_REMOTE_GATEWAY_MASK`.

#### LinkerRouter

Provides supported token address validations using the `validateSender` function.

### TokenDeployer

Deploys tokens using Create3Deployer

### BytecodeServer

`BytecodeServer` holds the token creation code, which is passed to its constructor.

### ERC20BurnableMintable

ERC-20 implementation, which is used for all token deployments. Inherits from `ERC20.sol`.

### Actors

This part describes actors of the system, their roles, and permissions.

#### Owner

The owner has total control over the supported tokens and associated validations, namely the following privileges in the contracts:

#### LinkerRouter

- Add trusted address,
- remove trusted address,
- add gateway-supported chains,
- remove gateway-supported chains.

#### User

The user (any EOA or contract) can interact with the protocol in following ways:

- Deploy interchain token,
- send token,
- send token with data,
- register origin token,
- register origin token and deploy remote tokens,
- deploy remote tokens.

## 5.2. Trust Model

The Interchain Token Service inherits the security of Axelar GMP and adds some `onlyOwner` privileges on top of it. Users can register their own (potentially malicious) token.

## L1: Missing validations

*Low severity issue*

| | | | |
|---|---|---|---|
| Impact: | Low | Likelihood: | Low |
| Target: | TokenDeployer | Type: | Data validation |

### Description

The contract `TokenDeployer` constructor does not implement any data validation.

```
constructor(address deployer_, address bytecodeServer_, address
tokenImplementation_) { deployer = Create3Deployer(deployer_); bytecodeServer =
bytecodeServer_; tokenImplementation = tokenImplementation_; thisAddress =
ITokenDeployer(this); }
```

### Recommendation

Proper data validation is necessary because the contract is always used when deploying new tokens using InterchainTokenService. The most error-resistant is a contract composition with a contract ID function, which is called on a given address and compared with a value saved in the contract. The less strict way to validate contract addresses is to perform a check on whether the given address is a contract or EOA. If a random wrong address is passed inside the constructor by mistake, there is a very low probability that it will point to an existing contract and revert in such a case. The least robust validation is a zero-address check.

### Solution (Revision 1.1)

Zero-address checks have been added to `TokenDeployer` constructor. `if (deployer_ == address(0) || bytecodeServer_ == address(0) || tokenImplementation_ == address(0)) revert AddressZero();`

[Go back to Findings Summary]

ackee | blockchain security

# W1: Duplicated code

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Best practices |

## Description

InterchainTokenService function _giveTokenWithData contains the same code as _giveToken. Code duplications are generally bad practice and could lead to errors during future development.

```
    _setTokenMintAmount(tokenId, getTokenMintAmount(tokenId) + amount);
    bytes32 tokenData = getTokenData(tokenId);
    address tokenAddress = tokenData.getAddress();

    if (tokenData.isOrigin() || tokenData.isGateway()) {
        _transfer(tokenAddress, destinationaddress, amount);
    } else {
        _mint(tokenAddress, destinationaddress, amount);
    }
```

## Recommendation

Refactor the code and call _giveToken from _giveTokenWithData to improve the architecture and code readability.

## Solution (Revision 1.1)

Functions have been renamed to _transferOrMintWithData and _transferOrMint. The _transferOrMintWithData function calls _transferOrMint to avoid code duplications.

[Go back to Findings Summary]

ackee | blockchain security

# W2: Malicious token registration

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | InterchainTokenService | Type: | Trust model |

### Description

Anyone can register their own ERC-20 token implementation to the Interchain Token Service. This feature opens a large variety of potential malicious scenarios, which could affect the protocol's reputation in case it's misused.

### Vulnerability scenario

We did not identify any reentrancy scenario using malicious token implementation. However, keep in mind that attackers can be very creative in token development, e.g.:

- The attacker deploys the malicious token.
- The attacker registers the token to the Interchain Token Linker and uses it to deploy to other chains.
- Users transfer the tokens to other chains.
- The attacker rug pulls tokens from the Linker.
- Users are not able to transfer tokens back to the original chain.

### Recommendation

Axelar is not primarily responsible for preventing the introduction of malicious tokens within the protocol. However, if such an occurrence were to take place, it could potentially undermine the credibility and trust associated with the protocol. Therefore it's good to be transparent and communicate this potential risk to users.

Go back to Findings Summary

# W3: Identical function body

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | InterchainTokenService | Type: | Best practices |

## Description

`InterchainTokenService` contains two functions `_execute` and `_executeWithToken` with identical body.

```
if (!linkerRouter.validateSender(sourceChain, sourceAddress)) return;
// solhint-disable-next-line avoid-low-level-calls
(bool success, ) = address(this).call(payload);
if (!success) revert ExecutionFailed();
```

## Recommendation

Call `_execute` from `_executeWithToken`.

## Solution (Revision 1.1)

The function `_executeWithToken` calls `_execute`.

[Go back to Findings Summary]

## W4: Unused internal functions

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | InterchainTokenService | Type: | Best practices |

### Description

`InterchainTokenService` contains unused internal functions `_setTokenMintLimit` and `_callContractWithToken`

### Recommendation

Remove all unused code or implement missing logic to utilize these functions.

### Solution (Revision 1.1)

Functions `_setTokenMintLimit` and `_callContractWithToken` are now used in the code.

Go back to Findings Summary

ackee | blockchain security

## W5: Usage of `solc` optimizer

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | `**/*` | Type: | Compiler config |

### Description

The project uses `solc` optimizer. Enabling `solc` optimizer [may lead to unexpected bugs](#).

The Solidity compiler was audited in November 2018, and the audit [concluded](#) that the optimizer may not be safe.

### Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

### Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

## I1: Redundant data validation

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

### Description

In the constructor of the InterchainTokenService contract, a redundant check for a zero address is performed.

```
if (gatewayAddress_ == address(0) || gasServiceAddress_ == address(0) ||
linkerRouterAddress_ == address(0))
```

The check for a `gatewayAddress_` variable is already performed in the inherited constructor of the contract `AxelarExecutable`.

```
constructor(address gateway_) {
    if (gateway_ == address(0)) revert InvalidAddress();

    gateway = IAxelarGateway(gateway_);
}
```

### Recommendation

Remove the redundant check in the InterchainTokenService constructor to save some gas.

Go back to Findings Summary

# I2: Missing documentation

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | `**/*` | Type: | Best practices |

## Description

Although the code is realatively simple and easy to read, the project is missing detailed documentation.

## Recommendation

We strongly recommend covering the code by NatSpec. High-quality documentation has to be an essential part of any professional project.

## Solution (Revision 2.1)

The code documentation was significantly improved and can be considered sufficient.

[Go back to Findings Summary]

# 6. Report revision 1.1

## 6.1. System Overview

Description of improvements and changes in contracts and trust model for revision 1.1.

### Contracts

Updates and changes in the contracts' code we find important.

#### InterchainTokenService

`InterchainTokenService` contains a lot of refactored and new code, e.g. gateway and remote gateway logic in `_sendToken` and `_sendTokenWithData` functions.

There are new features like token mint limits per time interval, express logic and functions like `sendSelf` or `callContractWithSelf`. Also, a new security mechanism has been introduced in the function `registerSelfAsInterchainToken`. It is designed to self-register tokens, which avoids malicious actors from registering tokens and blocking their registration as an origin token in the future.

#### InterchainToken

Functions `interchainTransfer` and `interchainTransferFrom` are now implemented and using common logic from `internal` function `_interchainTransfer`.

#### LinkerRouter

The `LinkerRouter` has been slightly refactored, with no major changes.

## Actors

This part describes changes in actors, their roles, and permissions.

### Owner

The owner has now the following new abilities in the system:

- Register origin gateway token,
- register remote gateway token.

### Interchain Token

Interchain token is a new active actor in the system, it can interact with the `InterchainTokenService` in the following ways:

- Register self,
- set self mint limit,
- send self,
- call contract with self.

### User

The user (any EOA or contract) can now perform these additional operations:

- Express execute,
- express execute with token.

# L2: Expected revert

*Low severity issue*

| | | | |
|---|---|---|---|
| Impact: | Low | Likelihood: | Low |
| Target: | InterchainTokenService | Type: | Data validation |

## Description

The `_execute` function in `InterchainTokenService` contract does not `revert` in case of failed sender validation. This can lead to partial execution and an inconsistent token state and loss of user's funds.

Steps to reproduce the issue:

1. `executeWithToken` in `AxelarExecutable` gets called.
2. `validateContractCallAndMint` passes and mints tokens.
3. `_executeWithToken` in `InterchainTokenService` calls `_execute`.
4. Condition `if (!linkerRouter.validateSender(sourceChain, sourceAddress))` is met, and the function returns without revert.
5. Transaction passes, tokens are minted to the contract, but `payload` is not executed.

```
if (!linkerRouter.validateSender(sourceChain, sourceAddress)) return;
(bool success, ) = address(this).call(payload);
if (!success) revert ExecutionFailed();
```

## Recommendation

Use `revert` instead of `return` to avoid partial execution.

```
if (!linkerRouter.validateSender(sourceChain, sourceAddress)) revert
```

```
SenderValidationFailed();
```

### Update (Revision 2.0)

After a discussion with the client, the impact of this issue was reevaluated from High to Low. The main reason is ERC-20 tokens may be potentially stuck in the service contract, which does not present a significant issue as anyone can already send any ERC-20 tokens to the contract.

Go back to Findings Summary

# L3: Missing validations

*Low severity issue*

| Impact: | Low | Likelihood: | Low |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

## Description

The contract `InterchainTokenService` constructor is missing `tokenDeployerAddress_` validation.

```
constructor(
        address gatewayAddress_,
        address gasServiceAddress_,
        address linkerRouterAddress_,
        address tokenDeployerAddress_,
        string memory chainName_
    ) AxelarExecutable(gatewayAddress_) {
        if (gatewayAddress_ == address(0) || gasServiceAddress_ == address(0) ||
linkerRouterAddress_ == address(0))
            revert TokenServiceZeroAddress();

        ...
    }
```

## Recommendation

Add the zero-address check also for `tokenDeployerAddress_`.

```
if (gatewayAddress_ == address(0) || gasServiceAddress_ == address(0) ||
linkerRouterAddress_ == address(0) || tokenDeployerAddress_ == address(0))
            revert TokenServiceZeroAddress();
```

[Go back to Findings Summary]

## W6: Lack of events

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | LinkerRouter | Type: | Events |

### Description

The contract `LinkerRouter` is missing emits of events when the state changes, namely in functions `addTrustedAddress`, `removeTrustedAddress`, `addGatewaySupportedChains` and `removeGatewaySupportedChains`. Events are generally useful for monitoring contract activity, tracking changes and triggering off-chain responses.

### Recommendation

Add event emits into the mentioned functions

```
event TrustedAddressAdded(string chain, string addr);
event TrustedAddressRemoved(string chain);
event GatewaySupportedChainsAdded(string[] chainNames);
event GatewaySupportedChainsRemoved(string[] chainNames);

...

function addTrustedAddress(string memory chain, string memory addr) public
onlyOwner {
    ...
    emit TrustedAddressAdded(chain, addr);
}

function removeTrustedAddress(string calldata chain) external onlyOwner {
    ...
    emit TrustedAddressRemoved(chain);
}

function addGatewaySupportedChains(string[] calldata chainNames) external
onlyOwner {
    ...
    emit GatewaySupportedChainsAdded(chainNames);
```

```
}

function removeGatewaySupportedChains(string[] calldata chainNames) external
onlyOwner {
    ...
    emit GatewaySupportedChainsRemoved(chainNames);
}
```

### Update (Revision 2.0)

The original issue is still valid and there are more state-changing functions missing events. The functions are:

- `setFlowLimit` in the `FlowLimit` contract,
- `setPaused` in the `Pausable` contract,
- `setAdmin` in the `Adminable` contract.

### Partial solution (Revision 2.1)

Event emits were added to the `LinkerRouter`, `Adminable` and `Pausable` contracts. However, the `FlowLimit` contract still does not emit an event of a flow limit change.

[Go back to Findings Summary]

ackee blockchain security

## W7: Duplicated code

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Best practices |

### Description

`InterchainTokenService` contains following duplicated code for validation in functions `isOriginToken`, `isGatewayToken`, `getGatewayTokenSymbol`, `isCustomInterchainToken` and `_sendToken`.

```
bytes32 tokenData = getTokenData(tokenId);
if (tokenData == bytes32(0)) revert NotRegistered();
```

### Recommendation

Create a separate getter for `TokenData` including validation.

```
function getValidTokenData(bytes32 tokenId) public view returns (bytes32
tokenData) {
    tokenData = bytes32(getUint(_getTokenDataKey(tokenId)));
    if (tokenData == bytes32(0)) revert NotRegistered();
}
```

[Go back to Findings Summary](#)

ackee | blockchain security

# 7. Report revision 2.0

## 7.1. System Overview

This section contains an outline of the audited contracts. As the project was significantly rewritten, this section covers all important contracts without references to the previous revision. Note that this is meant for understandability purposes and does not replace project documentation.

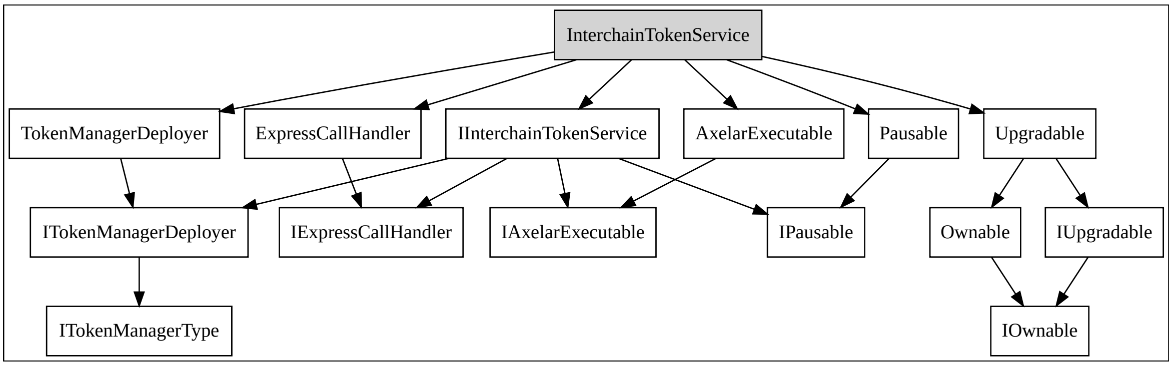
### Contracts

Contracts we find important for better understanding are described in the following section.

#### InterchainTokenService

This contract is the main entry point for the system. It is responsible for token manager deployments, including remote chain deployments, and interchain communication with service contracts on other chains. The `InterchainTokenService` contract is deployed behind a proxy, and it references all token manager types used when deploying a new token manager (also behind a proxy).

Users can deploy canonical token managers and custom token managers using the service. For each token address, only one canonical token manager can be deployed. Custom token managers do not have this restriction and can be controlled by anyone. The service contract also offers express receive functions typically called by relayers to provide tokens to a user faster if additional fees are paid. The rest of the functions serve for interactions with token managers.




#### InterchainToken

`InterchainToken` is an abstract ERC-20 token contract with [EIP-2612](https://eips.ethereum.org/EIPS/eip-2612) (permit) support. It can serve as a base contract for ERC-20 tokens, but it is not required to use it. The `InterchainToken` contract is a base contract of the `TokenManagerCanonical` contract.

The contract implements functions for interchain token transfers. However, it is possible to perform interchain transfers directly through token managers.

#### LinkerRouter

The `LinkerRouter` contract is responsible for the authorization of cross-chain messages. It defines from which addresses from which chains messages will be accepted. In a typical scenario, the `InterchainTokenService` contract will be deployed to the same address on different chains. However, deploying it to a different address and setting the given address as an authorized one is still possible. This approach is also needed when adding support for new chains with different lengths of addresses.

#### TokenManager

The `TokenManager` abstract contract is a base class for all token manager types. It handles token transfers of an underlying ERC-20 token. It is deployed behind a proxy, and the address of the implementation contract is stored in `InterchainTokenService`.

The owner of a token manager can set a new owner and configure a token flow limit. The flow limit describes the maximum amount of tokens sent or received in addition to the amount of tokens already received or sent respectively. The absolute value of the subtraction between token inflows and outflows must not exceed the flow limit (if set).

#### TokenManagerCanonical

`TokenManagerCanonical` inherits from the `InterchainToken` abstract contract, and so itself is an ERC-20 token. This token manager type is used when deploying a remote token manager for a local canonical token. However, it can still be used as a custom token manager.

#### TokenManagerLiquidityPool

The `TokenManagerLiquidityPool` implementation expects tokens stored in an external contract, a liquidity pool. The liquidity pool address is set during deployment and can be changed by the owner of the token manager.

#### TokenManagerLockUnlock

`TokenManagerLockUnlock` is similar to the `TokenManagerLiquidityPool` contract, but tokens are stored directly in the token manager. This token manager type is used when registering a local canonical token.

#### TokenManagerMintBurn

The `TokenManagerMintBurn` token manager type uses `mint` and `burn` functions to manage tokens. It expects common signatures of these functions implemented in the underlying token contract and appropriate permissions to call them.

### Actors

This part describes actors of the system, their roles, and permissions.

#### Axelar

The Axelar team is responsible for deployments of the `InterchainTokenService` contract and its proxies on supported chains. As this contract holds the addresses of all token managers, it is also responsible for token manager implementations and their upgrades. Even though canonical tokens can be registered and remotely deployed by anyone, the admin of the corresponding token managers is set to the `InterchainTokenService` contract. This means that the Axelar team can set flow limits and an address of a liquidity pool (in the case of `TokenManagerLiquidityPool`) of canonical tokens. The Axelar team can also pause the `InterchainTokenService` contract in case of an emergency.

#### Relayer

For an extra fee, a user of the interchain service can request a relayer to perform an extra receive and lend tokens on the target chain without waiting for the finality on the source chain.

#### User

A user of the interchain service can register canonical tokens on a chain and deploy corresponding canonical token managers to remote chains. However, these token managers are controlled by the `InterchainTokenService` contract, and so the user cannot set flow limits or change the address of a liquidity pool. A user can also deploy any number of custom token managers and control them. It is guaranteed that noone else can deploy token managers with the same `tokenId` on any EVM-compatible chain.

Users of the protocol can send tokens with an optional data message to any supported chain either through a token manager or an ERC-20 token contract directly if it iherits from the `InterchainToken` contract.

## 7.2. Trust Model

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 ERC-20 tokens on both a source and a destination chain. When interacting with a custom token manager pair, users have to trust the token manager admin to have linked both ERC-20 tokens correctly and that the flow limit will not be abused.

## H1: Express receive double execution

*High severity issue*

| Impact: | Medium | Likelihood: | High |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Double execution |

### Description

The functions `expressReceiveToken` and `expressReceiveTokenWithData` in the `InterchainTokenService` contract can be executed multiple times with the same arguments. Both functions transfer a given amount of tokens from a caller to an interchain recipient and overwrite an express receive slot with the address of the caller. Later, when the interchain transfer is relayed, the address stored in the express receive storage slot receives tokens instead of the interchain recipient.

An ability to overwrite the storage slot opens the possibility for an attack when a malicious user interchain transfers tokens with the express receive functionality. A relayer lends tokens on the destination chain while writing his address into the express receive storage slot. The malicious user calls the `expressReceiveToken` function with the same arguments as the relayer, effectively overwriting the relayer's address with his own. When the interchain transfer is relayed, the malicious user receives tokens instead of the relayer.

### Vulnerability scenario

Alice sends 1000 tokens from her first address on the BNB chain to her first address on the Ethereum mainnet. She pays an extra fee to receive her tokens sooner on the Ethereum mainnet using the express receive functionality. Bob is a relayer who calls the `expressReceiveToken` function with correct arguments on the Ethereum mainnet, transferring 1000 tokens from his address to Alice's first address. Alice owns another 1000 tokens on her second address on the Ethereum mainnet. She calls the `expressReceiveToken` function with the same arguments as Bob, transferring 1000 tokens from her second address to her first address. When the interchain transfer is relayed, Alice receives 1000 tokens instead of Bob. Bob loses 1000 tokens while Alice gains 1000 tokens.

See [Appendix C](#) for a proof of concept script in the [Wake](#) testing framework.

### Recommendation

Revert the transaction executing one of the `expressReceiveToken` and `expressReceiveTokenWithData` functions if the express receive storage slot already contains a non-zero address.

### Solution (Revision 2.1)

After a discussion with the client, the impact was reevaluated to medium, especially because the express receive functionality is intended to be used optionally with smaller amounts of tokens. Because of this, an attacker would not be able to steal a large amount of tokens as an explicit relayer action is required to trigger the vulnerability.

The issue was fixed by reverting the transaction if the express receive storage slot already contains a non-zero address.

[Go back to Findings Summary](#)

# M1: Gateway token check missing

*Medium severity issue*

| Impact: | Low | Likelihood: | High |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

## Description

The `registerCanonicalToken` function checks if a given ERC-20 token address is already registered at the gateway. In this case, canonical token registration fails to allow the Axelar team to register already supported tokens later. However, this check is missing in the `registerCanonicalTokenAndDeployRemoteCanonicalTokens` function, effectively letting anyone bypass the check in the `registerCanonicalToken` function.

*Listing 1. Excerpt from InterchainTokenService*

```
    function registerCanonicalToken(address tokenAddress) external notPaused returns (bytes32 tokenId) {
        (, string memory tokenSymbol, ) = _validateToken(tokenAddress);
        if (gateway.tokenAddresses(tokenSymbol) == tokenAddress) revert GatewayToken();
        tokenId = getCanonicalTokenId(tokenAddress);
        _deployTokenManager(tokenId, TokenManagerType.LOCK_UNLOCK, abi.encode(address(this).toBytes(), tokenAddress));
    }
```

*Listing 2. Excerpt from InterchainTokenService*

```
    function registerCanonicalTokenAndDeployRemoteCanonicalTokens(
        address tokenAddress,
        string[] calldata destinationChains,
        uint256[] calldata gasValues
    ) external payable notPaused returns (bytes32 tokenId) {
        tokenId = getCanonicalTokenId(tokenAddress);
        _deployTokenManager(tokenId, TokenManagerType.LOCK_UNLOCK, abi.encode(address(this).toBytes(), tokenAddress));
```

```
        (string memory tokenName, string memory tokenSymbol, uint8
tokenDecimals) = _validateToken(tokenAddress);
        bytes memory params = abi.encode('', tokenName, tokenSymbol,
tokenDecimals);
        _deployRemoteCanonicalTokens(tokenId, params, destinationChains,
gasValues);
    }
```

### Vulnerability scenario

A user registers a canonical token through the `registerCanonicalTokenAndDeployRemoteCanonicalTokens` function using the address of an already supported token by the gateway. Now, two canonical token handlers of the same ERC-20 token exist in the Axelar ecosystem. Furthermore, the Axelar team cannot register the canonical token later with possibly different parameters.

### Recommendation

Check that the address of a token is not already known to the gateway in the `registerCanonicalTokenAndDeployRemoteCanonicalTokens` function.

### Solution (Revision 2.1)

Fixed by removing the `registerCanonicalTokenAndDeployRemoteCanonicalTokens` functionality.

Go back to Findings Summary

# M2: `toAddress` missing validation

*Medium severity issue*

| Impact: | High | Likelihood: | Low |
|---|---|---|---|
| Target: | AddressBytesUtils | Type: | Data validation |

## Description

The `toAddress` function in the `AddressBytesUtils` library is missing a validation that the number of bytes to be converted to an address is exactly 20. Only the first 20 bytes are taken, or if less than 20 bytes are given, any data stored in memory immediately after the `bytesAddress` argument are used.

*Listing 3. Excerpt from [AddressBytesUtils]()*

```
    function toAddress(bytes memory bytesAddress) internal pure returns
(address addr) {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            addr := mload(add(bytesAddress, 20))
        }
    }
```

## Vulnerability scenario

There are multiple vulnerability scenarios corresponding to the `toAddress` function usages:

- A user of the interchain service mistypes the destination address by inserting less than or more than 20 bytes. Tokens are still transferred on the destination chain, taking the first 20 bytes in memory.

*Listing 4. Excerpt from [InterchainTokenService]()*

```
    function _processSendTokenPayload(string calldata sourceChain, bytes
```

```
calldata payload) internal {
        (, bytes32 tokenId, bytes memory destinationAddressBytes, uint256
amount, bytes32 sendHash) = abi.decode(
            payload,
            (uint256, bytes32, bytes, uint256, bytes32)
        );
        address destinationAddress = destinationAddressBytes.toAddress();
        ITokenManager tokenManager =
ITokenManager(getValidTokenManagerAddress(tokenId));
        address expressCaller = _popExpressSendToken(tokenId,
destinationAddress, amount, sendHash);
        if (expressCaller == address(0)) {
            amount = tokenManager.giveToken(destinationAddress, amount);
            emit TokenReceived(tokenId, sourceChain, destinationAddress,
amount, sendHash);
        } else {
            amount = tokenManager.giveToken(expressCaller, amount);
            emit ExpressExecutionFulfilled(tokenId, destinationAddress,
amount, sendHash, expressCaller);
        }
    }
```

- A custom token manager deployer mistypes the admin address by inserting less than or more than 20 bytes. The admin address is still set, taking the first 20 bytes in memory.

*Listing 5. Excerpt from [TokenManager]*

```
    function setup(bytes calldata params) external onlyProxy {
        bytes memory adminBytes = abi.decode(params, (bytes));
        address admin_;
        // Specifying an empty admin will default to the service being the
admin. This makes it easy to deploy remote canonical tokens without knowing
anything about the service address at the destination.
        if (adminBytes.length == 0) {
            admin_ = address(interchainTokenService);
        } else {
            admin_ = adminBytes.toAddress();
        }
        _setAdmin(admin_);
        _setup(params);
    }
```

- A `TokenManagerCanonical` deployer mistypes the admin address by inserting less than or more than 20 bytes. Tokens are still minted to an address, taking the first 20 bytes in memory.

*Listing 6. Excerpt from [TokenManagerCanonical]()*

```
    function _setup(bytes calldata params) internal override {
        uint256 mintAmount;
        bytes memory admin;
        //the first argument is reserved for the admin.
        (admin, name, symbol, decimals, mintAmount) = abi.decode(params, (bytes, string, string, uint8, uint256));
        _setDomainTypeSignatureHash(name);
        if (mintAmount > 0) {
            // Not sure why initial mint for an arbitrary admin address is needed natively.
            // Better to keep it simpler I think and it can be done at a higher level if needed.
            _mint(admin.toAddress(), mintAmount);
        }
    }
```

## Recommendation

Revert the transaction if the number of bytes to be converted to an address is not exactly 20.

## Solution (Revision 2.1)

Fixed by reverting the transaction if the number of bytes is not exactly 20.

[Go back to Findings Summary]()

# L4: `expressReceiveTokenWithData` spoofed data

*Low severity issue*

| Impact: | Low | Likelihood: | Low |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Access controls, data validation |

## Description

The `expressReceiveTokenWithData` function in the `InterchainTokenService` contract can be called by anyone with the arguments spoofed. As a consequence, the `IInterchainTokenExecutable.exectuteWithInterchainToken` callback function can be called with spoofed data.

*Listing 7. Excerpt from [InterchainTokenService]()*

```
588
    IInterchainTokenExecutable(destinationAddress).exectuteWithInterchainToken(s
    ourceChain, sourceAddress, data, tokenId, amount);
```

## Vulnerability scenario

A governance contract implementing the `IInterchainTokenExecutable.exectuteWithInterchainToken` callback function is deployed. ERC-20 tokens are used to vote in the contract and authorization based on the `sourceChain` and `sourceAddress` parameters is required prior to voting. Because of the possibility of executing the callback function with spoofed data, anyone can vote in someone else's name.

## Recommendation

Perform validation that such interchain transfer exists against the gateway or limit the execution of the express receive functions to a limited set of trusted addresses. If this is not an option, make sure it is clearly visible in the contract source code, in the Github repository and in the documentation that the data passed to the callback function cannot be trusted.

### Solution (Revision 2.1)

Fixed by adding a documentation comment to the `expressReceiveTokenWithData` function stating:

> Use this only if you have detected an outgoing sendToken that matches the parameters passed here.

Go back to Findings Summary

# L5: sendHash is not unique

*Low severity issue*

| Impact: | Low | Likelihood: | Low |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Architecture design |

## Description

sendHash used to identify an interchain transfer may not be unique. Currently, the sendHash value is computed from tokenId, block.number and amount:

*Listing 8. Excerpt from InterchainTokenService*

```
        bytes32 sendHash = keccak256(abi.encode(tokenId, block.number,
    amount));
```

*Listing 9. Excerpt from InterchainTokenService*

```
        bytes32 sendHash = keccak256(abi.encode(tokenId, block.number,
    amount));
```

sendHash is used to compute an express receive slot. However, the tokenId and amount parameters are already incorporated in the slot computation:

*Listing 10. Excerpt from ExpressCallHandler*

```
        slot = uint256(keccak256(abi.encode(PREFIX_EXPRESS_GIVE_TOKEN,
    tokenId, destinationAddress, amount, sendHash)));
```

*Listing 11. Excerpt from ExpressCallHandler*

```
        slot = uint256(
            keccak256(
                abi.encode(
```

```
                    PREFIX_EXPRESS_GIVE_TOKEN_WITH_DATA,
                    tokenId,
                    sourceChain,
                    sourceAddress,
                    destinationAddress,
                    amount,
                    data,
                    sendHash
                )
            )
        );
```

Multiple interchain transactions with the same parameters (`tokenId`, `sourceAddress`, `destinationaddress`, `amount`) can be included in the same block. More importantly, some layer 2 solutions (e.g. Arbitrum) may report the same `block.number` for multiple layer 2 blocks.

> Accessing block numbers within an Arbitrum smart contract (i.e., block.number in Solidity) will return a value close to (but not necessarily exactly) the L1 block number at which the Sequencer received the transaction.
>
> — Arbitrum documentation

> A single Ethereum block could include multiple Arbitrum blocks within it; however, an Arbitrum block cannot span across multiple Ethereum blocks. Thus, any given Arbitrum transaction is associated with exactly one Ethereum block and one Arbitrum block.
>
> — Arbitrum documentation

### Vulnerability scenario

A user wants to transfer 2000 tokens from Arbitrum in two transactions with express receive functionality. Both transactions are included in the same Ethereum block. Consequently, the `sendHash` value is the same for both transactions and the user express receives only the first batch of tokens. The second batch is received after reaching the finality on the source chain.

### Recommendation

Define a nonce counter within the `InterchainTokenService` contract and use it instead of `sendHash`. The nonce value together with `sourceChain` will be unique for each interchain transfer.

### Solution (Revision 2.1)

Fixed by using `commandId` instead of `sendHash` in the `InterchainTokenService` contract. `commandId` is guaranteed to be unique because it is used internally in Axelar gateways to identify an interchain message.

[Go back to Findings Summary]

# W8: Express receive functions can be called by recipient

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

## Description

Both express receive functions `expressReceiveToken` and `expressReceiveTokenWithData` can be called from the destination address resulting in the express receive amount being computed as zero. This value is later used when overwriting an express receive storage slot and when emitting an event.

*Listing 12. Excerpt from [InterchainTokenService]()*

```
    function expressReceiveToken(bytes32 tokenId, address destinationAddress, uint256 amount, bytes32 sendHash) external notPaused {
        address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValidTokenManagerAddress(tokenId));
        IERC20 token = IERC20(tokenManager.tokenAddress());
        uint256 balance = token.balanceOf(destinationAddress);
        SafeTokenTransferFrom.safeTransferFrom(token, caller, destinationAddress, amount);
        amount = token.balanceOf(destinationAddress) - balance;
        _setExpressSendToken(tokenId, destinationAddress, amount, sendHash, caller);

        emit ExpressExecuted(tokenId, destinationAddress, amount, sendHash, caller);
    }
```

*Listing 13. Excerpt from [InterchainTokenService]()*

```
    function expressReceiveTokenWithData(
        bytes32 tokenId,
        string memory sourceChain,
```

```
        bytes memory sourceAddress,
        address destinationAddress,
        uint256 amount,
        bytes calldata data,
        bytes32 sendHash
    ) external notPaused {
        address caller = msg.sender;
        ITokenManager tokenManager = ITokenManager(getValidTokenManagerAddress(tokenId));
        IERC20 token = IERC20(tokenManager.tokenAddress());
        uint256 balance = token.balanceOf(destinationAddress);
        SafeTokenTransferFrom.safeTransferFrom(token, caller, destinationAddress, amount);
        amount = token.balanceOf(destinationAddress) - balance;
        _setExpressSendTokenWithData(tokenId, sourceChain, sourceAddress, destinationAddress, amount, data, sendHash, caller);
        _passData(destinationAddress, tokenId, sourceChain, sourceAddress, amount, data);
        emit ExpressExecutedWithData(tokenId, sourceChain, sourceAddress, destinationAddress, amount, data, sendHash, caller);
    }
```

### Recommendation

While there are no security consequences, this is an unexpected edge-case scenario that can cause problems in the future. It is recommended to add a check to ensure that the caller is not the destination address.

### Solution (Revision 2.1)

Fixed by avoiding recomputing `amount` as a difference between the balance after and before the transfer.

[Go back to Findings Summary]

## W9: `IInterchainTokenExecutable` typo

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | IInterchainTokenExecutable | Type: | Code quality |

### Description

There is a typo in the `IInterchainTokenExecutable` callback function name.

*Listing 14. Excerpt from [IInterchainTokenExecutable]()*

```
function exectuteWithInterchainToken(
```

### Recommendation

Because this typo affects the selector of the function, it is recommended to fix the typo as soon as possible.

### Solution (Revision 2.1)

The typo was fixed.

[Go back to Findings Summary]()

# W10: Misleading `TokenManagerNotDeployed` error name

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | InterchainTokenService | Type: | Logic error |

## Description

The error `TokenManagerNotDeployed` is used when checking if a token manager with a given `tokenId` exists. However, the error is raised only when the token manager exists (the address contains a code), but the `tokenId()` function returns an unexpected `tokenId`. When the token manager with a given `tokenId` is not deployed, execution of the function reverts earlier because of ABI decoding failure.

*Listing 15. Excerpt from [InterchainTokenService](InterchainTokenService)*

```
    function getValidTokenManagerAddress(bytes32 tokenId) public view returns (address tokenManagerAddress) {
        tokenManagerAddress = getTokenManagerAddress(tokenId);
        if (ITokenManagerProxy(tokenManagerAddress).tokenId() != tokenId) revert TokenManagerNotDeployed(tokenId);
    }
```

## Recommendation

Perform the `tokenId()` external call as a low-level call to be able to raise the user-defined error or rename the error to `TokenIdMismatch` or a similar name. Note that under normal conditions, it should never happen that a token manager would return an unexpected `tokenId`.

## Solution (Revision 2.1)

Fixed by renaming the error to `TokenManagerDoesNotExist`.

Go back to Findings Summary

# W11: `LinkerRouter` initial trusted parameters cannot be set

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | LinkerRouter | Type: | Logic error |

## Description

The parameters `trustedChainNames` and `trustedAddresses` in the `LinkerRouter` constructor cannot be set to non-empty arrays. The reason is that the `LinkerRouter` contract is expected to be deployed behind a proxy, and the contract's admin is set when deploying the proxy contract. Because setting trusted addresses is an admin-only operation, execution of the constructor fails with non-empty parameters as the admin is the zero address during the implementation contract deployment.

*Listing 16. Excerpt from [LinkerRouter]()*

```
    constructor(address _interchainTokenServiceAddress, string[] memory trustedChainNames, string[] memory trustedAddresses) {
        if (_interchainTokenServiceAddress == address(0)) revert ZeroAddress();
        interchainTokenServiceAddress = _interchainTokenServiceAddress;
        uint256 length = trustedChainNames.length;
        if (length != trustedAddresses.length) revert LengthMismatch();
        interchainTokenServiceAddressHash = keccak256(bytes(_lowerCase(interchainTokenServiceAddress.toString())));
        for (uint256 i; i < length; ++i) {
            addTrustedAddress(trustedChainNames[i], trustedAddresses[i]);
        }
    }
```

*Listing 17. Excerpt from [LinkerRouter]()*

```
    function addTrustedAddress(string memory chain, string memory addr) public onlyOwner {
        if (bytes(chain).length == 0) revert ZeroStringLength();
```

```
        if (bytes(addr).length == 0) revert ZeroStringLength();
        remoteAddressHashes[chain] = keccak256(bytes(_lowerCase(addr)));
        remoteAddresses[chain] = addr;
    }
```

## Recommendation

Remove the `trustedChainNames` and `trustedAddresses` parameters from the `LinkerRouter` constructor and if needed, set them using the `setup` function invoked by the proxy without access controls checked.

## Partial solution (Revision 2.1)

The parameters were removed from the `LinkerRouter` constructor and can now be set using the `setup` function. However, the parameters are still set using the `addTrustedAddress` function. Because the `addTrustedAddress` function checks access controls, deployment of `LinkerRouterProxy` may revert if the configured admin is not the deployer (`msg.sender`) of the proxy contract.

[Go back to Findings Summary]

# W12: PREFIX_CANONICAL_TOKEN_ID typo

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Code quality |

## Description

There is a typo in the value of the `PREFIX_CANONICAL_TOKEN_ID` constant.

*Listing 18. Excerpt from [InterchainTokenService]()*

```
    bytes32 internal constant PREFIX_CANONICAL_TOKEN_ID = keccak256('its-
cacnonical-token-id');
```

## Recommendation

Fix the typo as soon as possible as it may cause backward compatibility issues.

## Solution (Revision 2.1)

The typo was fixed by using a different value (`its-standardized-token-id`).

[Go back to Findings Summary]()

# W13: Token manager implementations order validation

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

## Description

There is no validation that token manager implementation addresses are passed in the correct/expected order to the `InterchainTokenService` constructor. Passing the implementations in an incorrect order makes the `InterchainTokenService` contract dysfunctional and requires a new contract deployment.

*Listing 19. Excerpt from [InterchainTokenService]*

```
    constructor(
        address deployer_,
        address bytecodeServer_,
        address gateway_,
        address gasService_,
        address linkerRouter_,
        address[] memory tokenManagerImplementations,
        string memory chainName_
    ) TokenManagerDeployer(deployer_, bytecodeServer_) AxelarExecutable(gateway_) {
        if (linkerRouter_ == address(0) || gasService_ == address(0)) revert ZeroAddress();
        linkerRouter = ILinkerRouter(linkerRouter_);
        gasService = IAxelarGasService(gasService_);

        if (tokenManagerImplementations.length != uint256(type(TokenManagerType).max) + 1) revert LengthMismatch();

        // use a loop for the zero address checks?
        if (tokenManagerImplementations[uint256(TokenManagerType.LOCK_UNLOCK)] == address(0)) revert ZeroAddress();
        implementationLockUnlock = tokenManagerImplementations[uint256(TokenManagerType.LOCK_UNLOCK)];
```

```
        if (tokenManagerImplementations[uint256(TokenManagerType.MINT_BURN)]
== address(0)) revert ZeroAddress();
        implementationMintBurn = tokenManagerImplementations[
uint256(TokenManagerType.MINT_BURN)];
        if (tokenManagerImplementations[uint256(TokenManagerType.CANONICAL)]
== address(0)) revert ZeroAddress();
        implementationCanonical = tokenManagerImplementations[
uint256(TokenManagerType.CANONICAL)];
        if (tokenManagerImplementations[
uint256(TokenManagerType.LIQUIDITY_POOL)] == address(0)) revert
ZeroAddress();
        implementationLiquidityPool = tokenManagerImplementations[
uint256(TokenManagerType.LIQUIDITY_POOL)];
```

### Recommendation

While the type of a token manager is accessible from the `TokenManagerProxy` proxy contract, it is helpful to keep the token manager type even in the implementation contract. Store the token manager type in the `TokenManager` contract and check if the type matches the expected type in the `InterchainTokenService` constructor.

### Solution (Revision 2.1)

Fixed by providing the token manager type in token manager implementations and checking the type in the `InterchainTokenService` constructor.

Go back to Findings Summary

# W14: `requiresApproval` misleading value

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | TokenManagerMintBurn | Type: | Logic error |

### Description

The function `requiresApproval` defined in the `ITokenManager` interface may return misleading values in the case of `TokenManagerMintBurn`. The `false` value is returned in this case, but the correct value depends on the implementation of an underlying ERC-20 token. Some implementations may require the allowance set when burning tokens, while others may not.

Furthermore, the function is only used in the `InterchainToken` contract; in this case, the correct value depends on the implementation of the interchain token itself.

### Recommendation

Consider moving the `requiresApproval` function into the `InterchainToken` contract and making the function virtual, as only this contract uses the function. At the same time, the implementation of this contract may require different values returned.

### Solution (Revision 2.1)

The `requiresApproval` function was removed from the `ITokenManager` interface and a new virtual function `tokenManagerRequiresApproval` with the same functionality was added to the `InterchainToken` contract.

[Go back to Findings Summary]

# W15: Different decimals not handled

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | InterchainTokenService | Type: | Arithmetics, Data validation |

## Description

When registering canonical tokens and deploying remote canonical tokens, it is guaranteed that both token contracts have identical decimals. However, custom token manager pairs may be deployed with different token decimals. The `InterchainTokenService` contract does not handle different token decimals and token amounts cannot even be recomputed on a backend because token amounts are encoded into generic messages for Axelar General Message Passing. Using custom token manager pairs with different decimals leads to a loss of tokens for a user performing an interchain transfer or to a loss for an admin managing token managers.

## Recommendation

Encode source token decimals into a payload sent to a destination chain.

*Listing 20. Excerpt from [InterchainTokenService]()*

```
        bytes memory payload = abi.encode(SELECTOR_SEND_TOKEN, tokenId,
    destinationAddress, amount, sendHash);
```

*Listing 21. Excerpt from [InterchainTokenService]()*

```
            bytes memory payload = abi.encode(
                SELECTOR_SEND_TOKEN_WITH_DATA,
                tokenId,
                destinationAddress,
                amount,
                sourceAddress.toBytes(),
                data,
```

```
                    sendHash
                );
```

On the destination chain, compare decimals decoded from the payload with decimals of the destination token. Either recompute the amount of tokens given to a destination user or revert on different decimals.

### Solution (Revision 2.1)

The finding was acknowledged by the client with the following comment:

> Different decimals for the same token aren’t supported. Due to the permissionless nature of the system, there are many ways a user can create invalid links. So, it’s not too useful to add on-chain checks that increase gas usage.
>
> — Axelar Team

Go back to Findings Summary

## I3: `IInterchainTokenService` event parameter typo

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | IInterchainTokenService | Type: | Code quality |

### Description

There is a typo in the last parameter of the `IInterchainTokenService.TokenSent` event.

*Listing 22. Excerpt from [IInterchainTokenService]()*

```
    event TokenSent(bytes32 tokenId, string destinationChain, bytes
destinationAddress, uint256 indexed amount, bytes32 sendHahs);
```

### Recommendation

Fix the typo.

### Solution (Revision 2.1)

The `sendHash` parameter was removed from the `TokenSent` event, so the typo no longer exists.

[Go back to Findings Summary]()

## I4: `InterchainToken` revert if max approval given

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainToken | Type: | Integer overflow |

### Description

The `InterchainToken` transfer functions `interchainTransfer` and `interchainTransferFrom` revert on overflow when a sender already has max approval given to the token manager.

*Listing 23. Excerpt from [InterchainToken]()*

```
        if (tokenManager.requiresApproval()) {
            _approve(sender, address(tokenManager), allowance[sender][
address(tokenManager)] + amount);
        }
```

*Listing 24. Excerpt from [InterchainToken]()*

```
        if (tokenManager.requiresApproval()) {
            _approve(sender, address(tokenManager), allowance[sender][
address(tokenManager)] + amount);
        }
```

### Recommendation

Do not increase the allowance to the token manager when the allowance is set to `type(uint256).max`.

### Partial solution (Resivision 2.1)

The issue was fixed in the `interchainTransfer` function by increasing the allowance by an amount that does not overflow. However, the issue is still present in the `interchainTransferFrom` function.

[Go back to Findings Summary]

## I5: `StringToAddress` library unused

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | LinkerRouter | Type: | Code quality |

### Description

The `StringToAddress` library is imported in the `LinkerRouter` contract through a using-for directive, but it is never used in the contract.

*Listing 25. Excerpt from [LinkerRouter](LinkerRouter)*

```
using StringToAddress for string;
```

### Recommendation

Remove the using-for directive referencing the `StringToAddress` library.

### Solution (Revision 2.1)

The usage of the `StringToAddress` library was removed from the `LinkerRouter` contract.

[Go back to Findings Summary](Go back to Findings Summary)

## I6: Use `type(uint256).max` for infinite flow limit

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | FlowLimit | Type: | Code quality |

### Description

The `FlowLimit` contract uses zero as a value indicating there is no flow limit. However:

- this is against conventions as an infinite allowance is typically expressed using `type(uint256).max`,
- it is not possible to completely disable a single token manager by setting the flow limit to zero.

*Listing 26. Excerpt from [FlowLimit]*

```
    function _addFlow(uint256 slotToAdd, uint256 slotToCompare, uint256
flowAmount) internal {
        uint256 flowLimit = getFlowLimit();
        if (flowLimit == 0) return;
```

### Recommendation

Consider using `type(uint256).max` as a special value indicating an infinite flow limit.

### Solution (Revision 2.1)

The finding was acknowledged by the client with the following comment:

> Setting flow limits to uint256 max increases gas cost for all deployments. Using the 0 default is simpler, and we'll document the behaviour so apps set it correctly.

> — Axelar Team

Go back to Findings Summary

## I7: Token manager send function names

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

### Description

The functions `sendToken` and `callContractWithInterchainToken` perform almost the same operation, with the latter also executing a callback with user-defined payload, but the names of the functions are completely different. This is against good coding practices and current conventions in the Axelar ecosystem, where `IAxelarExecutable` functions are named `execute` and `executeWithToken`.

*Listing 27. Excerpt from TokenManager*

```
    function sendToken(string calldata destinationChain, bytes calldata
destinationAddress, uint256 amount) external payable virtual {
        address sender = msg.sender;
        amount = _takeToken(sender, amount);
        _addFlowOut(amount);
        _transmitSendToken(sender, destinationChain, destinationAddress,
amount);
    }

    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);
        _transmitSendTokenWithData(sender, destinationChain,
destinationAddress, amount, data);
    }
```

### Recommendation

Consider using the same prefix for both function names. For example, consider renaming the second function to `sendTokenWithPayload` or `sendTokenWithData`.

[Go back to Findings Summary]

# I8: LinkerRouter remote addresses normalization

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | LinkerRouter | Type: | Code quality |

## Description

Remote addresses are accepted as a string in the `LinkerRouter.addTrustedAddress` function. These addresses are then normalized (lower-cased) and checksummed for comparison purposes using Keccak-256. However, a human-readable version of the address is stored unnormalized.

*Listing 28. Excerpt from [LinkerRouter]()*

```
    function addTrustedAddress(string memory chain, string memory addr)
public onlyOwner {
        if (bytes(chain).length == 0) revert ZeroStringLength();
        if (bytes(addr).length == 0) revert ZeroStringLength();
        remoteAddressHashes[chain] = keccak256(bytes(_lowerCase(addr)));
        remoteAddresses[chain] = addr;
    }
```

## Recommendation

Consider storing remote addresses in the `remoteAddresses` mapping normalized, i.e., lower-cased.

[Go back to Findings Summary]()

# I9: Unused functions and variables

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | ExpressCallHandler, InterchainTokenService, LinkerRouter | Type: | Code quality |

## Description

In the `ExpressCallHandler` contract, both mappings `expressGiveToken` and `expressGiveTokenWithData` are unused.

There are multiple unused functions in the `InterchainTokenService` contract:

- `transmitSendTokenWithToken`,
- `transmitSendTokenWithDataWithToken`,
- `approveGateway`.

The `LinkerRouter` contract defines the public mapping `supportedByGateway` and functions `addGatewaySupportedChains` and `removeGatewaySupportedChains` to modify the mapping, but values stored in the mapping are not used in the project.

## Recommendation

Leaving unused functions and variables in the code is not a good practice. Either remove these functions and variables or implement missing features using them.

## Partial solution (Revision 2.1)

All functions and variables except `supportedByGateway` mapping and `addGatewaySupportedChains` and `removeGatewaySupportedChains` functions were removed from the project.

[Go back to Findings Summary]

## I10: `InterchainTokenServiceProxy` unused constructor parameter

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenServiceProxy | Type: | Code quality |

### Description

There is an unused parameter (`setupParams`) in the `InterchainTokenServiceProxy` constructor.

*Listing 29. Excerpt from [InterchainTokenServiceProxy]()*

```
    constructor(
        address implementationAddress,
         address owner,
         bytes memory /*setupParams*/
     )
         // Pass the setup through in case the implementation changes in the future to override the setup? This avoids changing the proxy bytecode
         FinalProxy(implementationAddress, owner, new bytes(0)) // solhint-disable-next-line no-empty-blocks
     {}
```

### Recommendation

Either remove the parameter or pass it to the `FinalProxy` constructor.

### Solution (Revision 2.1)

The parameter was removed from the constructor.

[Go back to Findings Summary]()

ackee | blockchain security

# I11: `_executeWithToken` redundant modifier

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

## Description

There is the redundant modifier `onlyRemoteService(sourceChain, sourceAddress)` used in the `InterchainTokenService._executeWithToken` function:

*Listing 30. Excerpt from [InterchainTokenService]()*

```
function _executeWithToken(
    string calldata sourceChain,
    string calldata sourceAddress,
    bytes calldata payload,
    string calldata /*symbol*/,
    uint256 /*amount*/
) internal override onlyRemoteService(sourceChain, sourceAddress) {
    _execute(sourceChain, sourceAddress, payload);
}
```

The same modifier is already present on the `_execute` function.

## Recommendation

Consider removing the modifier from the `_executeWithToken` function. Also, note that this function should never be executed and it should be safe and a cleaner solution to always revert from the function.

## Solution (Revision 2.1)

The `_executeWithToken` function was removed from the `InterchainTokenService` contract, using the default `AxelarExecutable` implementation that does not perform any action.

Go back to Findings Summary

# 8. Report revision 2.1

## 8.1. System Overview

Except for fixes of issues reported in the previous revision, there are a few significant changes:

- `TokenManagerCanonical` was removed,
- new interchain tokens `StandardizedToken`, `StandardizedTokenLockUnlock` and `StandardizedTokenMintBurn` were added,
- `StandardizedTokenProxy` was introduced as a proxy for standardized tokens,
- new utility contracts `Implementation`, `Multicall` and `StandardizedTokenDeployer` were added,
- `TokenManagerDeployer` was significantly simplified,
- register and deploy functions in the `InterchainTokenService` contract were renamed and modified,
- the `InterchainTokenService` contract now inherits from `Multicall`.

9. Report revision 3.0

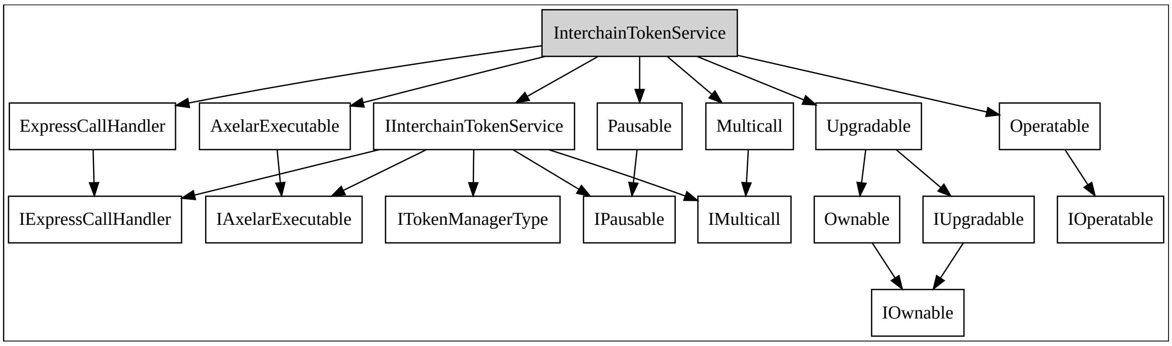
## 9.1. System Overview

The contracts and architecture remained to a large extent the same as in the previous revision. The main changes are:

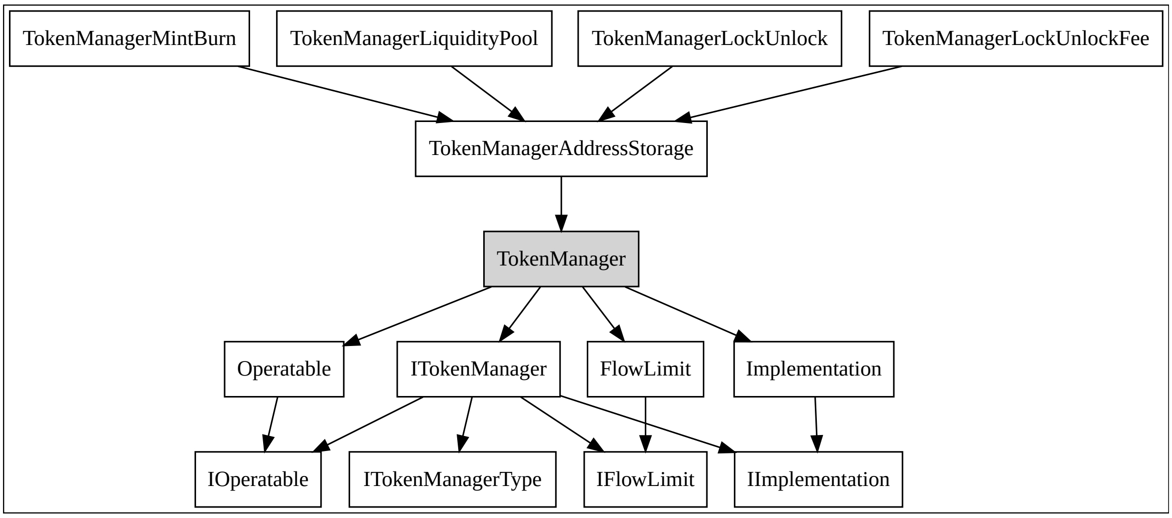
- `LinkerRouter` was renamed to `RemoteAddressValidator` and received a couple of optimizations,
- new `TokenManager` for fee-on-transfer tokens was added,
- new utility contracts like `Operatable` and `NoReEntrancy` were added,
- a lot of code improvements and optimizations were made.

We provide inheritance diagrams for the most important contracts below:

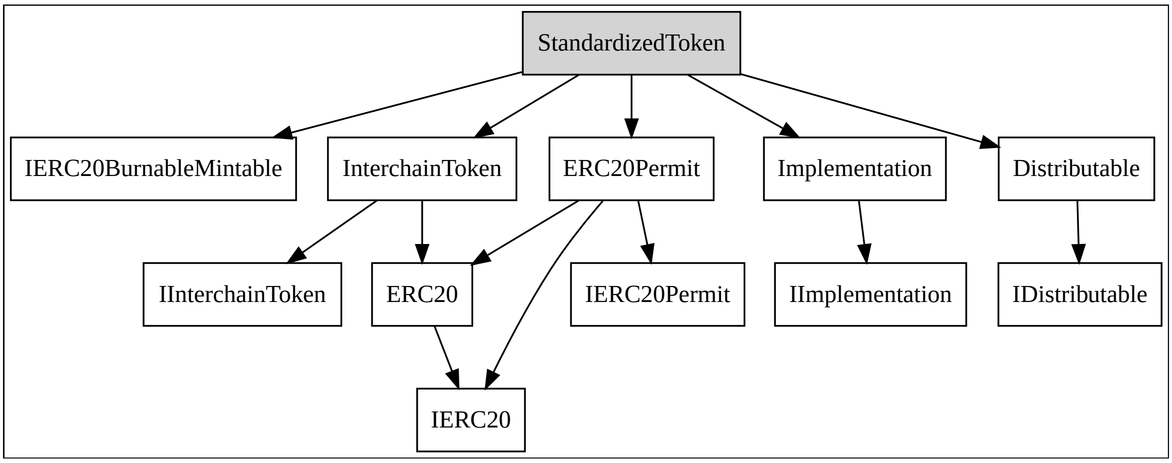
### InterchainTokenService




### TokenManager




### StandardizedToken




### Actors

This revision did not introduce new actors.

## 9.2. Trust Model

This revision did not introduce new trust assumptions. Yet, we would like to point on the following considerations:

- `InterchainTokenService` is upgradeable,

- `InterchainTokenService` is pausable and thus the users can be DoSed,
- anyone can perform the express calls (i.e. not only Axelar-operated services) and thus the data cannot be trusted.

# H2: Wrong variable passed to hook

*High severity issue*

| Impact: | Medium | Likelihood: | High |
|---|---|---|---|
| Target: | InterchainToken.sol | Type: | Contract logic |

## Description

The `interchainTransferFrom` has the following implementation:

*Listing 31. Excerpt from [InterchainToken]()*

```
        uint256 _allowance = allowance[sender][msg.sender];

        if (_allowance != type(uint256).max) {
            _approve(sender, msg.sender, _allowance - amount);
        }

        _beforeInterchainTransfer(msg.sender, destinationChain, recipient, amount, metadata);

        ITokenManager tokenManager_ = tokenManager();
        tokenManager_.transmitInterchainTransfer{ value: msg.value }(sender, destinationChain, recipient, amount, metadata);
```

It can be seen that the function allows `msg.sender` with sufficient `allowance` to transfer from `sender` to a `recipient` and the `destinationChain`.

The problematic part is that `msg.sender` is passed to the `_beforeInterchainTransfer` hook instead of the `sender` variable. The hook can process the data arbitrarily, but the main functionality that it should provide is setting allowance from the entity that provides the tokens to the respective `TokenManager`:

*Listing 32. Excerpt from [InterchainToken]()*

```
     * @notice A method to be overwritten that will be called before an interchain transfer. You can approve the tokenManager here if you need and want to, to allow users for a 1-call transfer in case of a lock-unlock token manager.
     * @param from the sender of the tokens. They need to have approved `msg.sender` before this is called.
     * @param destinationChain the string representation of the destination chain.
```

In this scenario, the token provider is `sender` and not `msg.sender`.

Additionally, if the `sender` actually gave allowance to the `TokenManager` (and thus the call passes), then this hook will incorrectly increase the allowance of the `msg.sender` to the `TokenManager`, which is not the intended action by the `msg.sender`.

Most likely, this is a copy-paste error, caused by copying the line from the `interchainTransfer` function.

### Vulnerability scenario

Alice gives allowance to Bob. Bob wants to make an interchain transfer of these tokens. He follows the project's documentation that states that he does not need to give allowance to the manager. However, the call fails on insufficient allowance. Bob loses money on gas and the tokens don't arrive at the destination.

### Recommendation

Approve the correct address in the hook.

[Go back to Findings Summary]()

# H3: Tokens with callbacks can artificially increase cross-chain transfer amount

*High severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | Medium |
| Target: | TokenManagerLiquidityPool.sol | Type: | Contract logic |

## Description

The `_takeToken` function in `TokenManagerLiquidityPool` has the following implementation:

*Listing 33. Excerpt from [TokenManagerLiquidityPool]()*

```
    function _takeToken(address from, uint256 amount) internal override
noReEntrancy returns (uint256) {
        IERC20 token = IERC20(tokenAddress());
        address liquidityPool_ = liquidityPool();
        uint256 balance = token.balanceOf(liquidityPool_);

        SafeTokenTransferFrom.safeTransferFrom(token, from, liquidityPool_,
amount);

        // Note: This allows support for fee-on-transfer tokens
        return IERC20(token).balanceOf(liquidityPool_) - balance;
```

As can be seen, the function has a reentrancy lock to prevent reentering and repeatedly increasing the `balanceOf(liquidityPool_)`. However, the `balanceOf(liquidityPool_)` can also be directly increased in the token callback by a deposit to the liquidity pool. As a result, the final return statement `IERC20(token).balanceOf(liquidityPool_) - balance` can return a higher value than the actual input amount.

### Exploit scenario

Alice initiates a cross-chain call, the relevant token supports callbacks. Alice's tokens are transferred to the manager and at this moment her contract is given execution via the token callback. Alice takes her remaining tokens and deposits them into the corresponding liquidity pool. This increases the value returned `IERC20(token).balanceOf(liquidityPool_) - balance` from the function (and thus the value used in the cross-chain call). After the transaction finishes, Alice creates a new transaction where she withdraws the deposited amount from the liquidity pool (this amount wasn't transferred from her, but was directly deposited). Thus she achieved sending more tokens than she had to lock in the pool.

### Recommendation

Because the function supports also the fee-on-transfer tokens it is very hard to propose a solution (as we need to know how high the fee is). The problem is that the attacker can use the `feeAmount-1` to increase the value in the same way as described above (i.e., we can't just assert that the returned value is `leq` than the input amount). But even adding this simple check would limit how much the attacker can manipulate the amount. As such, we recommend considering how important it is to support the fee-on-transfer tokens.

Go back to Findings Summary

# M3: Operator slot incorrect preimage

*Medium severity issue*

| Impact: | High | Likelihood: | Low |
|---|---|---|---|
| Target: | Operatable.sol | Type: | Contract logic |

## Description

The `OPERATOR_SLOT` is defined as follows:

*Listing 34. Excerpt from [Operatable]()*

```
    // uint256(keccak256('operator')) - 1
    uint256 internal constant OPERATOR_SLOT =
0xf23ec0bb4210edd5cba85afd05127efcd2fc6a781bfed49188da1081670b22d7;
```

However, if the `keccak` function is recomputed with the specified value, a different value is returned:

```
>>> hex(int(keccak256(b"operator").hex(), 16)-1)
'0x46a52cf33029de9f84853745a87af28464c80bf0346df1b32e205fc73319f621'
```

If the function is rerun with the input `admin`, it returns:

```
>>> hex(int(keccak256(b"admin").hex(), 16)-1)
'0xf23ec0bb4210edd5cba85afd05127efcd2fc6a781bfed49188da1081670b22d7'
```

As can be seen, this is exactly the value used in the contract. This is especially problematic because the `Operatable` contract is supposed to be imported and inherited. If any of the contracts that inherit from `Operatable` also define the `admin` slot (which is likely as it is a frequent role in contracts), then the `OPERATOR_SLOT` (or the `ADMIN_SLOT`) will be overwritten.

## Vulnerability scenario

A contract imports the `Operatable` contract and inherits from it. Additionally, it defines an admin role. Firstly, the admin role is stored. Then the operator role is stored.

Then, after a while, the admin calls an admin function, but the call reverts because his slot was overwritten in storage.

## Recommendation

Recompute the `OPERATOR_SLOT` using the correct string.

[Go back to Findings Summary]

# M4: Proposed role not cleared when accepted

*Medium severity issue*

| Impact: | High | Likelihood: | Low |
|---|---|---|---|
| Target: | Operatabale.sol, Distributable.sol | Type: | Contract logic |

## Description

The `Operatable` and `Distributable` contracts allow for 2-step transfer of the roles they define. In this process, the new proposed operator/distributor are defined and if the new address accepts this proposal, they get the role.

However, the `_set` functions do not clear the proposed slots:

*Listing 35. Excerpt from [Operatable]()*

```
    function _setOperator(address operator_) internal {
        assembly {
            sstore(OPERATOR_SLOT, operator_)
        }
        emit OperatorshipTransferred(operator_);
    }
```

See the analogical function from OpenZeppelin:

```
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}
```

Additionally, both the mentioned contracts contain a function for a direct transfer without the 2-step process. This function does not clear the proposed slot either:

*Listing 36. Excerpt from Operatable*

```solidity
    function transferOperatorship(address operator_) external onlyOperator {
        _setOperator(operator_);
    }
```

### Exploit scenario

Operator A proposes Operator B as the new operator. B is an address controlled by A. Then, after some time A calls `transferOperatorship` with C as argument. Everyone thinks that C is the new operator and nothing can be changed without his action. However, in a convenient situation B call `acceptOperatorship` and becomes the operator.

### Recommendation

Follow the same pattern as the Ownable2Step from OpenZeppelin.

Go back to Findings Summary

# M5: Lack of destination chain validation

*Medium severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | Low |
| Target: | InterchainTokenService.sol, RemoteAddressValidator.sol | Type: | Data validation |

## Description

The `InterchainTokenService` contract mediates the cross-chain token transfers. One of the parameters of the transfer is the destination chain. The destination chain is represented as a string. During the transfers, this string isn't validated, i.e. the service does not validate that the destination is actually supported.

The `RemoteAddressValidator` contains the mapping `supportedByGateway`, however, this mapping is not read during the transfers.

## Vulnerability scenario

A user decides to perform a cross-chain transfer of his tokens to the destination chain D. D is not supported by the gateway. Additionally, the token manager that he uses is of the `LOCK/UNLOCK` type. As a result, his tokens are locked on the source chain and never arrive at D.

## Recommendation

Add the supported chains to the mapping and when a cross-chain transfer is performed, check that the destination chain is supported.

[Go back to Findings Summary]

## M6: Incorrect accounting of flowIn for fee-on-transfer tokens

*Medium severity issue*

| Impact: | Medium | Likelihood: | Medium |
|---|---|---|---|
| Target: | TokenManagerLiquidityPool.sol, TokenManagerLockUnlockFee.sol | Type: | Contract logic |

### Description

The `_giveToken` functions in `TokenManagerLiquidityPool` and `TokenManagerLockUnlockFee` both end with the following return statement:

*Listing 37. Excerpt from [TokenManagerLiquidityPool]()*

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

The expression inside the statement subtracts the balance before the transfer from the current balance, this allows for supporting fee-on-transfer tokens. The result of this return is then added to `flowIn`.

However, for fee-on-transfer tokens, the amount that is added is smaller than the original amount that arrived in the cross-chain transfer. Thus a small discrepancy in the flow accounting is created.

### Exploit scenario

Bob sends a fee-on-transfer token to Alice via a cross-chain call. Suppose, that if the actual transferred amount was added to the current `flowIn` accounting, then the difference of the flows for the current epoch would revert. However, since the amount added to `flowIn` is smaller than the actual transferred amount, the difference of the flows for the current epoch is still within bounds. This means that the call incorrectly passes.

### Recommendation

When performing the accounting of incoming flows, use the actual transferred amount incoming from the source chain.

[Go back to Findings Summary]

# M7: Front-running express execute with copy of gateway payload

*Medium severity issue*

| Impact: | High | Likelihood: | Low |
|---|---|---|---|
| Target: | InterchainTokenProxy.sol | Type: | Front-running |

## Description

The `InterchainTokenService` exposes the `expressReceiveTokenWithData` function:

*Listing 38. Excerpt from [InterchainTokenProxy]()*

```
    ) external {
        if (gateway.isCommandExecuted(commandId)) revert AlreadyExecuted(commandId);

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

        SafeTokenTransferFrom.safeTransferFrom(token, caller, destinationAddress, amount);

        _expressExecuteWithInterchainTokenToken(tokenId, destinationAddress, sourceChain, sourceAddress, data, amount);

        _setExpressReceiveTokenWithData(tokenId, sourceChain, sourceAddress, destinationAddress, amount, data, commandId, caller);
    }
```

There are multiple things to be observed regarding the function:

- it is external without access control,

- it transfers tokens to the `destinationAddress`,
- it transfers the control to the `destinationAddress` via the call `_expressExecuteWithInterchainTokenToken`,
- it sets the corresponding express receive slot on the last line.

From that, it can be seen that the function does not follow the CEI pattern as it transfers the control to the `destinationAddress` before modifying the storage slot corresponding to the express receive. This leads to an exploit described in the next section.

### Exploit scenario

The attack is initiated by a cross-chain token transfer with data from the attacker. This cross-chain transfer has one precondition (which makes this attack low likelihood): the express call transaction and the `execute` transaction to the gateway on the destination chain (containing the command for the attacker's cross-chain call) end up being in the mempool at the same time. The express call is expected to be much faster than the gateway transaction, however, the precondition could be met for example in the following scenarios: express service downtime or and more likely, the express call transaction is sent with a very low gas price.

The following steps suppose that the precondition is met:

1. Attacker observes the mempool and sees that both the relevant transaction are in the mempool. He takes the `execute` transaction to the gateway and creates a new transaction which copies the payload of the `execute` transaction to his contract. This contract is the contract to which the cross-chain call is supposed to be made, i.e., the `destinationAddress`.
2. The attacker then bundles these transactions and sends them to Flashbots. The first transaction is the `copy` transaction, the second one is

the expressExecute one. The transaction to the gateway is no longer relevant.

3. Once the expressExecute transaction gets executed, it transfers the tokens to the destinationAddress and at some point, it also transfers the execution to the destinationAddress as explained earlier.

4. Once the destinationAddress receives the control, it executes the copied payload from the execute transaction. This payload contains the attacker's cross-chain call. This will trigger the execute function on the InterchainTokenService, which in turn will call _processSendTokenWithDataPayload. This function checks if the corresponding express receive slot is address(0):

*Listing 39. Excerpt from InterchainTokenProxy*

```
            address expressCaller = _popExpressReceiveTokenWithData(
                tokenId,
                sourceChain,
                sourceAddress,
                destinationAddress,
                amount,
                data,
                commandId
            );
            if (expressCaller != address(0)) {
                amount = tokenManager.giveToken(expressCaller, amount);
                return;
            }
        }
```

+ 5. However, it was already explained that the slot is set after the call to the destinationAddress. This means that this call will also transfer tokens to the destinationAddress and the attacker will receive the tokens twice.

It was shown that the attacker will receive the tokens twice and that the express service will not be refunded.

### Recommendation

Follow the CEI pattern in the `expressReceiveTokenWithData` function.

[Go back to Findings Summary]

# M8: Tokens with callbacks can break the flow accounting

*Medium severity issue*

| Impact: | Medium | Likelihood: | Medium |
|---|---|---|---|
| Target: | TokenManagerLiquidityPool.sol, TokenManagerLockUnlockFee.sol | Type: | Contract logic |

## Description

The `_giveToken` functions in `TokenManagerLiquidityPool` and `TokenManagerLockUnlockFee` both end with the following return statement:

*Listing 40. Excerpt from [TokenManagerLiquidityPool]()*

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

The expression inside the statement subtracts the balance before the transfer from the current balance, this allows for supporting fee-on-transfer tokens. The result of this return is then added to `flowIn`.

To avoid problems with reentering this function via callbacks and increasing the `balanceOf(to)` a reentrancy lock is used. However, the `to` balance can also be increased via a direct transfer in the callback. Thus a malicious user can transfer tokens to themselves via a callback and increase their balance (e.g. via `transferFrom` from a different account to actually increase the balance), this will cause the accounting of `flowIn` to be incorrect. The amount added to `flowIn` will be higher than the actual amount from the cross-chain call.

### Exploit scenario

Bob sends a fee-on-transfer token to Alice via a cross-chain call. On the destination chain, the corresponding token supports callbacks. This allows Alice to take over the execution and transfer additional tokens to herself. This in turn artificially increases the `flowIn` accounting. The worst-case effect can be that other users will be DoSed for the current epoch as the difference between the outgoing and incoming flows will be too large.

### Recommendation

When performing the accounting of incoming flows, use the actual transferred amount incoming from the source chain.

Go back to Findings Summary

## L6: Chain name validation

*Low severity issue*

| | | | |
|---|---|---|---|
| Impact: | Low | Likelihood: | Low |
| Target: | RemoteAddressValidator.sol | Type: | Data validation |

### Description

The `RemoteAddressValidator` allows for adding and removing supported chains. The chains are added using a string representing the chain name:

*Listing 41. Excerpt from [RemoteAddressValidator]()*

```
    function addGatewaySupportedChains(string[] calldata chainNames)
external onlyOwner {
        uint256 length = chainNames.length;
        string calldata chainName;
        for (uint256 i; i < length; ++i) {
            chainName = chainNames[i];
            supportedByGateway[chainName] = true;
```

The process is analogical for removing chains.

Neither of the functions recognizes that an empty string was passed in, which can lead to reporting invalid information to outside contracts.

### Vulnerability scenario

The problem is more severe in the case of removing the chains. Suppose that a gateway no longer supports a chain and that this information is to be stored in the validator contract. Due to a bug in frontend or deployment script, one of the chain names in the array is passed in as empty.

Because the string is not checked for length, the empty string goes unnoticed. As a result, outside contracts querying the address validator for the chain status can incorrectly suppose the destination is still supported, although it is not.

### Recommendation

Add the length validation to the mentioned functions.

[Go back to Findings Summary]

# W16: Return of literal instead of enum

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | token-manager/implementations/* | Type: | Code quality |

### Description

The implementations of the `TokenManager` have the function `implementationType`, which returns a literal, see for example `TokenManagerLiquidityPool`:

*Listing 42. Excerpt from [TokenManagerLiquidityPool](TokenManagerLiquidityPool)*

```
    function implementationType() external pure returns (uint256) {
        return 3;
    }
```

This could become problematic once a new type is added. If it is not added at the end of the enum, then the getters must be modified. Additionally, this approach requires counting the type's position in the enum.

### Recommendation

Consider returning the enum value directly; the TokenManagerLiquidityPool case would be handled as `return uint256(TokenManagerType.LIQUIDITY_POOL);`.

[Go back to Findings Summary](Go back to Findings Summary)

# W17: Manager implementation zero address check

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | TokenManagerProxy.sol, InterchainTokenService.sol | Type: | Data validation |

## Description

The `constructor` of `TokenManagerProxy,`, through a sequence of calls, calls the `getImplementation` function on the `InterchainTokenService`:

*Listing 43. Excerpt from [TokenManagerProxy]()*

```
        address impl =
_getImplementation(IInterchainTokenService(interchainTokenServiceAddress_),
implementationType_);

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

The `getImplementation` function is implemented as follows:

*Listing 44. Excerpt from [InterchainTokenService]()*

```
    function getImplementation(uint256 tokenManagerType) external view
returns (address tokenManagerAddress) {
        if (tokenManagerType > uint256(type(TokenManagerType).max)) revert
InvalidImplementation();
        if (TokenManagerType(tokenManagerType) ==
TokenManagerType.LOCK_UNLOCK) {
            return implementationLockUnlock;
        } else if (TokenManagerType(tokenManagerType) ==
TokenManagerType.MINT_BURN) {
            return implementationMintBurn;
        } else if (TokenManagerType(tokenManagerType) ==
TokenManagerType.LOCK_UNLOCK_FEE_ON_TRANSFER) {
```

```
            return implementationLockUnlockFee;
        } else if (TokenManagerType(tokenManagerType) ==
TokenManagerType.LIQUIDITY_POOL) {
            return implementationLiquidityPool;
        }
    }
```

If a new type is added to the enum but the corresponding type is not added to the `else-if` chain, then this functions returns `address(0)`.

In the `TokenManagerProxy` constructor', a `delegatecall` is done on the retrieved implementation address. However, `delegatecall` on an account with no code returns true. Thus, the bug goes unnoticed.

### Recommendation

Add a zero address check to the constructor of `TokenManagerProxy`.

[Go back to Findings Summary]

# W18: Prefix incorrectly calculated

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | ExpressCallHandler.sol | Type: | Contract logic |

## Description

The `PREFIX_EXPRESS_RECEIVE_TOKEN_WITH_DATA` is defined as follows:

*Listing 45. Excerpt from [ExpressCallHandler]()*

```
    // uint256(keccak256('prefix-express-give-token-with-data'));
    uint256 internal constant PREFIX_EXPRESS_RECEIVE_TOKEN_WITH_DATA =
0x3e607cc12a253b1d9f677a03d298ad869a90a8ba4bd0fb5739e7d79db7cdeaad;
```

However, if the `keccak` function is recomputed with the specified value, a different value is returned:

```
>>> hex(int(keccak256(b"prefix-express-give-token-with-data").hex(), 16))
'0x3e607cc12a253b1d9f677a03d298ad869a90a8ba4bd0fb5739e7d79db7cdeaaf'
```

The value in the contract ends with `d`, and the correct value ends with `f`.

## Recommendation

Redefine the slot to use the correct value.

[Go back to Findings Summary]()

# W19: Lack of contract prefixes in slot preimages

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | `**/*` | Type: | Contract logic |

## Description

A large number of the protocol's contracts use the unstructured storage pattern. A `keccak` hash of a slot string is computed, and the result is used as a storage address for a given variable. However, the addresses are computed without fully qualifying the relevant variable; see the example:

*Listing 46. Excerpt from [Operatable]()*

```
// uint256(keccak256('operator')) - 1
uint256 internal constant OPERATOR_SLOT =
0xf23ec0bb4210edd5cba85afd05127efcd2fc6a781bfed49188da1081670b22d7;
// uint256(keccak256('proposed-operator')) - 1
uint256 internal constant PROPOSED_OPERATOR_SLOT =
0x18dd7104fe20f6107b1523000995e8f87ac02b734a65cf0a45fafa7635a2c526;
```

The `PROPOSED_OPERATOR_SLOT` is defined as `keccak256('proposed-operator')) - 1`. This is not ideal from the long-term perspective as it increases the probability of a clash if some other contract defines the same slot.

## Recommendation

Rather than deriving the string used in `keccak` just from the variable name, it should be derived from the contract name as well. For example, the `PROPOSED_OPERATOR_SLOT` could be defined as `keccak256('Operatable.proposed-operator')) - 1`. This will ensure that a collision can happen only if the slot would be defined again in the same contract and thus the probability of a collision decreases.

[Go back to Findings Summary]()

## W20: Code-comment discrepancy

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | NoReEntrancy.sol, InterchainTokenService.sol, TokenManagerProxy.sol, InterchainToken.sol | Type: | Documentation |

### Description

The `NoReEntrancy` contract has the following incorrect NatSpec comment:

*Listing 47. Excerpt from [NoReEntrancy]()*

```
/**
 * @title Pausable
 * @notice This contract provides a mechanism to halt the execution of specific functions
 * if a pause condition is activated.
 */
contract NoReEntrancy is INoReEntrancy {
```

The NatSpec is clearly copied from `Pausable`.

The `deployAndRegisterStandardizedToken` function in `InterchainTokenService` has the following incorrect NatSpec comment:

*Listing 48. Excerpt from [InterchainTokenService]()*

```
    /**
     * @notice Used to deploy a standardized token alongside a TokenManager. If the `distributor` is the address of the TokenManager (which
     * can be calculated ahead of time) then a mint/burn TokenManager is used. Otherwise a lock/unlcok TokenManager is used.
```

However, the function always deploys the `MINT_BURN` token manager:

*Listing 49. Excerpt from [InterchainTokenService]*

```
        _deployTokenManager(tokenId, TokenManagerType.MINT_BURN,
    abi.encode(msg.sender.toBytes(), tokenAddress));
```

The `InterchainToken` contract has the following incorrect NatSpec comment:

*Listing 50. Excerpt from [InterchainToken]*

```
 * @dev You can skip the `tokenManagerRequiresApproval()` function altogether
 if you know what it should return for your token.
```

However, the `tokenManagerRequiresApproval()` function is not present in the contract, nor the interface.

The `TokenManagerProxy` contract has the following incorrect NatSpec comment:

*Listing 51. Excerpt from [TokenManagerProxy]*

```
 * @dev This contract is a proxy for token manager contracts. It implements
 ITokenManagerProxy and
 * inherits from FixedProxy from the gmp sdk repo
 */
contract TokenManagerProxy is ITokenManagerProxy {
```

However, the contract doesn't inherit from the `FixedProxy`.

### Recommendation

Fix all the incorrect comments and ensure that the code matches the developers' expectations.

[Go back to Findings Summary]

## I12: Reentrancy lock private

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | NoReEntrancy.sol | Type: | Code quality |

### Description

The `NoReEntrancy` contract provides the logic for reentrancy locks. The lock storage slot is private as it is accessible only through assembly and private functions. However, in certain situations, it is beneficial to have the lock public.

For example, view functions are often not guarded with the lock; as such, the protocols can be reentered through them. This can lead to the infamous read-only reentrancy hacks, e.g., the Curve read-only reentrancy hack.

If the lock is public, the calling contract can read it first and, based on that, decide whether to call the view function or not.

### Recommendation

Consider adding a view function for reading the lock.

[Go back to Findings Summary]

# I13: Typo in function parameter name

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService.sol | Type: | Code quality |

## Description

The function `_sanitizeTokenManagerImplementation` has the following parameter:

*Listing 52. Excerpt from [InterchainTokenService]()*

```
    function _sanitizeTokenManagerImplementation(
        address[] memory implementaions,
        TokenManagerType tokenManagerType
    ) internal pure returns (address implementation) {
```

The parameter `implementaions` is mistyped.

## Recommendation

Rename the parameter to `implementations`.

Go back to Findings Summary

# 10. Report revision 4.0

## 10.1. System Overview

The system components and their functionalities were mainly unchanged since the previous revision. Three new contracts and one significant dependency were introduced.

### TokenManagerMintBurnFrom

The contract is the same as TokenManagerMintBurn with a difference in the `_takeToken` function. It uses `burnFrom` instead of `burn` and thus requires allowance for `amount` on `from`.

### StandardizedTokenRegistrar

The contract allows the deployment of standardized tokens. It handles input parameters and calls the Interchain Token Service to deploy the token. It is a single-purpose contract for additional flexibility.

### CanonicalTokenRegistrar

Similar to StandardizedTokenRegistrar but handles canonical tokens. It also deploys and registers standardized tokens on the remote.

## 10.2. Actors

This revision did not introduce new actors.

## 10.3. Trust Model

This revision did not introduce new trust assumptions.

ackee | blockchain security

# M9: Tokens with callbacks can artificially increase cross-chain transfer amount

*Medium severity issue*

| Impact: | Medium | Likelihood: | Medium |
|---|---|---|---|
| Target: | TokenManagerLockUnlockFee.sol | Type: | Contract logic |

*Listing 53. Excerpt from TokenManagerLockUnlockFee._takeToken*

```
    function _takeToken(address from, uint256 amount) internal override
noReEntrancy returns (uint256) {
        IERC20 token = IERC20(tokenAddress());
        uint256 balanceBefore = token.balanceOf(address(this));

        token.safeTransferFrom(from, address(this), amount);

        return token.balanceOf(address(this)) - balanceBefore;
    }
```

## Description

The description matches the following issue: H3: Tokens with callbacks can artificially increase cross-chain transfer amount. However, it affects a different contract.

## Recommendation

Fix it in the same way as for the `TokenManagerLiquidityPool` contract.

Go back to Findings Summary

## W21: Token id can differ on the deployment method

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | StandardizedTokenRegistrar.sol, CanonicalTokenRegistrar.sol | Type: | Contract logic |

*Listing 54. Excerpt from* [*StandardizedTokenRegistrar.deployStandardizedToken*]()

```
    function deployStandardizedToken(
        bytes32 salt,
        string calldata name,
        string calldata symbol,
        uint8 decimals,
        uint256 mintAmount,
        address distributor
    ) external payable {
        address sender = msg.sender;
        salt = getStandardizedTokenSalt(sender, salt);
        bytes32 tokenId = service.getCustomTokenId(address(this), salt);

        service.deployAndRegisterStandardizedToken(salt, name, symbol,
decimals, mintAmount, distributor);
```

*Listing 55. Excerpt from* [*InterchainTokenService.deployAndRegisterStandardizedToken*]()

```
    function deployAndRegisterStandardizedToken(
        bytes32 salt,
        string calldata name,
        string calldata symbol,
        uint8 decimals,
        uint256 mintAmount,
        address distributor
    ) external payable notPaused {
        bytes32 tokenId = getCustomTokenId(msg.sender, salt);
        _deployStandardizedToken(tokenId, distributor, name, symbol,
```

```
    decimals, mintAmount, msg.sender);
```

### Description

We have two options for how to deploy a standardized token. The first one is with the Interchain Token Service and the second is via the new `StandardizedTokenRegistrar` contract.

Seeing the snippets above, the deployment is almost the same but not entirely if we consider the same input parameters. For the `StandardizedTokenRegistrar` contract there is a different salt calculation.

```
function getStandardizedTokenSalt(address deployer, bytes32 salt) public view
returns (bytes32) {
    return keccak256(abi.encode(PREFIX_STANDARDIZED_TOKEN_SALT, chainNameHash,
deployer, salt));
}
```

This intermediate step results in a different token id, depending on the deployment method.

Respectively applies also to the `CanonicalTokenRegistrar` contract.

### Recommendation

Ensure this is intended and if so, document the difference.

Go back to Findings Summary

## W22: Chain name data validation

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | StandardizedTokenRegistrar.sol, CanonicalTokenRegistrar.sol | Type: | Data validation |

*Listing 56. Excerpt from [StandardizedTokenRegistrar.constructor]()*

```
    constructor(address interchainTokenServiceAddress, string memory chainName_) {
        if (interchainTokenServiceAddress == address(0)) revert ZeroAddress();
        service = IInterchainTokenService(interchainTokenServiceAddress);
        chainName = chainName_;
        chainNameHash = keccak256(bytes(chainName_));
    }
```

### Description

It is possible to pass an empty string or any arbitrary value to the new token deployers. This can affect salt in deployments.

Since the salt is already different from Interchain Token Service (see W21: Token id can differ on the deployment method), it is considered only as a warning.

### Recommendation

The Interchain Token Service is fetching the chain name value from the `RemoteAddressValidator` contract and the contract is handling empty string input. Consider doing it same for token registrars.

Go back to Findings Summary

# W23: Possible code injection on deployment on remote

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | CanonicalTokenRegistrar.sol | Type: | Code injection |

*Listing 57. Excerpt from* *CanonicalTokenRegistrar.deployAndRegisterRemoteCanonicalToken*

```
    function deployAndRegisterRemoteCanonicalToken(bytes32 salt, string calldata destinationChain, uint256 gasValue) external payable {
        // This ensures that the token manages has been deployed by this address, so it's safe to trust it.
        bytes32 tokenId = service.getCustomTokenId(address(this), salt);
        IERC20Named token = IERC20Named(service.getTokenAddress(tokenId));
        // The 3 lines below will revert if the token manager does not exist.
        string memory tokenName = token.name();
        string memory tokenSymbol = token.symbol();
        uint8 tokenDecimals = token.decimals();

        // slither-disable-next-line arbitrary-send-eth
        service.deployAndRegisterRemoteStandardizedToken{ value: gasValue }(
            salt,
            tokenName,
            tokenSymbol,
            tokenDecimals,
```

## Description

Unlike the standardized token, the canonical token can be registered with a custom implementation. Therefore, some functions (`decimals`, `name`, …) can deliver arbitrary values or inject a code. This can be potentially a problem in the deployment on remote (see Listing 57), leading to changed values during execution or doing entirely something different from the context of the registrar contract.

### Recommendation

Ensure this is not an issue, consider removing the external calls.

[Go back to Findings Summary]

## W24: Change in the enum order can affect access controls

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | `**/*` | Type: | Access controls |

### Description

Currently, the roles for the project are defined followingly:

```
enum Roles {
    DISTRIBUTOR,
    OPERATOR,
    FLOW_LIMITER
}
```

Changing the order (removing an old one and adding another, etc.) changes also roles.

### Recommendation

Be extremely aware of this design decision and carefully update roles in the future.

[Go back to Findings Summary]

## I14: Incorrect inline documentation

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | TokenManagerMintBurnFrom.sol | Type: | Documentation |

### Description

The new `TokenManagerMintBurnFrom` has incorrect NatSpec comment:

*Listing 58. Excerpt from [TokenManagerMintBurnFrom]()*

```
/**
 * @title TokenManagerMintBurn
 * @notice This contract is an implementation of TokenManager that mints and burns a specific token on behalf of the interchain token service.
 * @dev This contract extends TokenManagerAddressStorage and provides implementation for its abstract methods.
 * It uses the Axelar SDK to safely transfer tokens.
 */
contract TokenManagerMintBurnFrom is TokenManagerMintBurn {
```

The NatSpec is clearly copied from `TokenManagerMintBurn`.

### Recommendation

Fix the incorrect code comment.

[Go back to Findings Summary]()

# I15: Ambiguous revert message

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainToken.sol | Type: | Best practice |

*Listing 59. Excerpt from [InterchainToken.interchainTransferFrom]()*

```
        if (_allowance != UINT256_MAX) {
            _approve(sender, msg.sender, _allowance - amount);
        }
```

## Description

The Interchain Token is decreasing approval on interchain transfer. However, it doesn't check if the allowance is smaller than the amount, so as a result, if the allowance is insufficient, the user gets an underflow error.

## Recommendation

Add a require/if statement for better error handling.

[Go back to Findings Summary]()

ackee | blockchain security

# I16: Code duplication

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | Operatable.sol, Distributable.sol, TokenManager.sol | Type: | Best practice |

## Description

The `Roles` contract provides the `_addRole` function which already creates an array and adds `uint8` element to it. This function can be used for many occurrences in the codebase, where is just one role added.

The same rule applies to the `_removeRole` function.

## Recommendation

Replace the `_addRoles` (`_removeRoles`) function with the `_addRole` (`_removeRole`) function for cases where only one role is added (removed).

[Go back to Findings Summary]

# 11. Report revision 5.0

## 11.1. System Overview

Since the previous version, several changes were made to the system. Some interfaces were removed, several components and functions were renamed. The biggest change is the introduction of the InterchainTokenFactory component, which is responsible for creating new interchain tokens, previously known as standardized tokens. Contracts with breaking changes are listed in the following subsections, other contracts contain minor non-breaking changes like renaming and slight refactoring.

### InterchainTokenFactory

The contract is responsible for creating new interchain tokens and their corresponding token managers with different token types. It inherits from `IInterchainTokenFactory`, `ITokenManagerType`, `Multicall` to support multi-calls for calling multiple functions in one transaction to save gas, and `Upgradable`. The deployment of tokens is handled by the InterchainTokenService. The address of new tokens is computed using the salt hash, which is based on the address of the deployer, a chain name, and the salt provided by the user. The deployment is possible both on the current chain and remote chains. The contract also serves as an entry point for registering canonical tokens. Finally, the contract allows to transfer of tokens.

### InterchainTokenService

`InterchainTokenService` is the core contract of the protocol. The contract newly inherits from `Create3Address`, `ExpressExecutorTracker`, `InterchainAddressTracker` and `IInterchainTokenService`. It was changed to support the new InterchainTokenFactory and multiple functions were renamed. Additionally, the `expressExecute()` function was added to support the express execution of interchain transfers. Functions responsible for interchain transfers and remote contract calls are now marked `payable` to support paying gas fees for remote transfers. The `_setup()` function was extended to support more setup parameters. To prevent unauthorized access to the external `setup()` function, this function is shadowed in the `BaseProxy` from Axelar's GMP SDK. The `execute()` function was extended to support express transfers.

### BaseInterchainToken

This new contract serves as an example implementation of the interchain token standard. It inherits all the required functions from the `ERC20` contract to support the ERC-20 standard. Additionally, it shows the implementation of the `IInterchainTokenStandard` interface with two additional functions: `interchainTransfer()` for interchain transfers and `interchainTransferFrom()` for interchain transfers with an allowance.

### InterchainTokenFactoryProxy

A basic proxy contract for the InterchainTokenFactory. It inherits from `Proxy`, where all proxy functions are defined, and defines the `CONTRACT_ID` constant to identify the contract and the constructor to set the implementation address along with the owner.

## 11.2. Actors

This revision did not introduce new actors.

## 11.3. Trust Model

This revision did not introduce new trust assumptions.

ackee | blockchain security

## W25: Hardcoded metadata version/prefix

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | Distributable.sol, Operatable.sol | Type: | Best practices |

*Listing 60. Excerpt from [InterchainTokenService](#)*

```
        uint32 prefix = 0;
```

*Listing 61. Excerpt from [InterchainTokenService](#)*

```
        uint32 version;
        (version, metadata) = _decodeMetadata(metadata);
        if (version > 0) revert InvalidMetadataVersion(version);
```

### Description

Version/prefix in metadata is hardcoded to `0` in two places in the code. It can lead to inconsistency while updating the value. Also, the naming inconsistency `prefix` for encoding and `version` from decoding is confusing.

### Recommendation

Use constant or immutable state variable for the metadata version/prefix and use the same variable naming for encoding/decoding.

[Go back to Findings Summary](#)

# W26: One-step role transfer

| Impact: | Warning | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | Distributable.sol, Operatable.sol | Type: | Access controls |

*Listing 62. Excerpt from [Distributable]()*

```
    function transferDistributorship(address distributor_) external
onlyRole(uint8(Roles.DISTRIBUTOR)) {
        _transferRole(msg.sender, distributor_, uint8(Roles.DISTRIBUTOR));
    }
```

*Listing 63. Excerpt from [Operatable]()*

```
    function transferOperatorship(address operator) external onlyRole(
uint8(Roles.OPERATOR)) {
        _transferRole(msg.sender, operator, uint8(Roles.OPERATOR));
    }
```

## Description

The `DISTRIBUTOR` and `OPERATOR` roles can be transferred using a one-way process. Therefore the roles can be accidentally transferred to an invalid address, which would cause irreversible loss of control over the role in contracts that extend `Distributable` or `Operatable`. However the contracts can be redeployed with different salt, therefore the damage is recoverable.

## Recommendation

Always use the two-step role transfer, even if the likelihood of the accident is low.

## Solution (Revision 5.1)

The client acknowledged this issue due to the required flexibility. Client's comment: ”We already provide both one-step and two-step transfers and let the user decide. Two-step transfers are tricky if transferring role to a contract who can’t issue an accept ownership call, or is tedious to perform (e.g. when transferring to a governance contract)”.

Go back to Findings Summary

## W27: Incorrect parent contract

| | | | |
|---|---|---|---|
| Impact: | Warning | Likelihood: | N/A |
| Target: | TokenManagerLockUnlockFee.sol | Type: | Documentation |

### Description

The `TokenManagerLockUnlockFee` contract has three parent contracts: `TokenManager`, `ReentrancyGuard` and `ITokenManagerLockUnlock`, however, in the context of the presence of the `ITokenManagerLockUnlockFee` interface we conclude that the `ITokenManagerLockUnlock` is used incorrectly. While those interfaces are identical and only differ in the interface name, future changes and consequent discrepancies between these interfaces may cause unexpected results.

### Recommendation

We recommend either changing the parent interface from `ITokenManagerLockUnlock` to `ITokenManagerLockUnlockFee`, or removing the `ITokenManagerLockUnlockFee` from the codebase since this interface is identical to `ITokenManagerLockUnlock` and is not used anywhere else.

[Go back to Findings Summary](#)

## W28: A danger of the interchain service's balance drainage

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService.sol | Type: | Documentation |

### Description

The InterchainTokenService contract allows users to execute interchain operations and pay gas for these operations. The gas value that the user is willing to pay to a gas service contract is given as an argument to external functions since it simplifies the usage of `msg.value` in the case of multi-call. More specifically, these functions are `deployTokenManager()`, `deployInterchainToken()`, `expressExecute()`, `interchainTransfer()`, `callContractWithInterchainToken()` and `transmitInterchainTransfer()`. The gas service is then called with the value set to `gasValue` provided by the user, and the refund address is set to the caller's address. This poses the risk of the drainage of the contract's balance in the case if the contract holds Ether: the user may set the gas value to the contract's balance and the gas service will refund to `tx.origin` the whole balance of the contract without the gas price required for the interchain operation. This potential issue may be extended to utilize the multi-call functionality of the contract and to reuse the `msg.value` to bypass potential gas value checks in the called functions.

### Recommendation

This issue is not critical since the contract does not hold Ether by design, and the unintentional transfers to the contract are a part of the risk according to Axelar's team:

> Ackee Blockchain: `InterchainTokenService` contract is not supposed to hold any Ether, right? Or is there any chance of

> some leftovers? Axelar: It’s not expected to. Any amount sent by the user unintentionally to it is considered at risk. While it would be nice to prevent unintentional sends, this was done to keep the multicall simpler.

We recommend explicitly stating in the documentation that the contract is not intended to hold Ether and describe the risks of losing any assets stored in the contract. Such an explicit statement will be helpful for the users of the contract and for the developers who will work on extending the system in the future.

Go back to Findings Summary

# I17: Incorrect or missing documentation

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | `**/*` | Type: | Documentation |

### Description

Multiple contracts have either incorrect documentation or no documentation at all. The following functions and contracts are affected: - `contracts/executable/InterchainTokenExecutable.sol` and `contracts/executable/InterchainTokenExpressExecutable.sol`: missing NatSpec docs. - `contracts/interchain-token/InterchainToken.sol::setup()`: the documentation says that `mintAmount` and `mintTo` are a part of `params`, however, in the code, these parameters are not expected. - `contracts/interfaces/IDistributable.sol::acceptDistributorship()`: missing `fromDistributor` parameter description. - `contracts/interfaces/IInterchainTokenFactory.sol`: missing NatSpec docs. - `contracts/interfaces/IInterchainTokenService.sol`: functions `interchainTransfer()`, `callContractWithInterchainToken()` do not have NatSpec docs. - `contracts/interfaces/IOperatable.sol::acceptOperatorship()`: missing `fromOperator` parameter description. - `contracts/proxies/InterchainTokenServiceProxy.sol::constructor()`: missing `setupParams` parameter description. - `contracts/token-manager/TokenManagerLiquidityPool.sol:_setup()`: the documentation says that `params_` should contain the token address and the liquidity pool address, however, in the code, the `params_` parameter contains additional bytes in the beginning. - `contracts/token-manager/TokenManagerLockUnlock.sol:_setup()`: the documentation says that `params_` should contain the token address, however, in the code, the `params_` parameter contains additional bytes in the beginning. - `contracts/token-manager/TokenManagerLockUnlockFee.sol:_setup()`: the documentation says that `params_` should contain the token address, however, in the code, the `params_` parameter contains additional bytes in the beginning. - `contracts/token-manager/TokenManagerMintBurn.sol:_setup()`: the documentation says that `params_` should contain the token address, however, in the code, the `params_` parameter contains additional bytes in the beginning. - `contracts/utils/Distributable.sol::acceptDistributorship()`: missing `fromDistributor` parameter description. - `contracts/utils/FlowLimit.sol::_setFlowLimit()`: missing `tokenId` parameter description. - `contracts/utils/InterchainTokenDeployer.sol::deployedAddress()`: missing `salt` parameter description. - `contracts/utils/Operatable.sol::acceptOperatorship()`: missing `fromOperator` parameter description. - `contracts/InterchainTokenFactory.sol`: missing NatSpec docs. - `contracts/InterchainTokenService.sol`: functions `contractCallValue()`, `expressExecute()`, `callContractWithInterchainToken()`, `_setup()`, `_sanitizeTokenManagerImplementation()`, `contractCallWithTokenValue()`, `expressExecuteWithToken()`, `executeWithToken()`, `_decodeMetadata()` are missing NatSpec docs. - `contracts/InterchainTokenService.sol::execute()`: missing `commandId` parameter description. - `contracts/InterchainTokenService.sol::_processInterchainTransferPayload()`: missing `expressExecutor` and `messageType` parameters description. - `contracts/InterchainTokenService.sol::TOKEN_FACTORY_DEPLOYER`: a typo, `…was deployed too…` should be `…was deployed to…`.

### Recommendation

We strongly recommend covering the code by NatSpec. High-quality documentation has to be an essential part of any professional project.

[Go back to Findings Summary]

# 12. Report revision 6.0

## 12.1. System Overview

In this revision, several changes in logic and refactoring were introduced. All changes are described in detail in the following sections. Contracts were also thoroughly documented using the NatSpec format, and those changes are not described in this document, nor simple changes in function declarations. This includes the following contracts:

- `InterchainTokenExecutable`
- `InterchainTokenExpressExecutable`
- `IAddressTracker`
- `IDistributable`
- `IERC20BurnableFrom`
- `IERC20MintableBurnable`
- `IERC20Named`
- `IFlowLimit`
- `IInterchainTokenDeployer`
- `IInterchainTokenExecutable`
- `IInterchainTokenExpressExecutable`
- `IInterchainTokenStandard`
- `IOperatable`
- `ITokenManagerDeployer`
- `ITokenManagerImplementation`
- `ITokenManagerType`

- `Distributable`
- `FlowLimit`
- `Operatable`
- `RolesConstants`

Also, multiple contracts and interfaces were removed from the repository, namely:

- `ITokenManagerLiquidityPool`
- `ITokenManagerLockUnlock`
- `ITokenManagerLockUnlockFee`
- `ITokenManagerMintBurn`
- `InterchainTokenFactoryProxy`
- `InterchainTokenProxy`
- `InterchainTokenServiceProxy`
- `TokenManagerLiquidityPool`
- `TokenManagerLockUnlock`
- `TokenManagerLockUnlockFee`
- `TokenManagerMintBurn`
- `TokenManagerMintBurnFrom`

### InterchainTokenFactory

The main functional change was the introduction of the gateway tokens handling logic. In the `deployInterchainToken()` function, if the `distributor` is set to the zero address, a zero-length array of bytes is passed to `_deployInterchainToken()`, and a new zero-address flow limiter is added. Additionally, the deployer's (`msg.sender`) balance gets set to the `initialSupply` of the token.

In `registerCanonicalInterchainToken()`, a new condition was added to check if the passed token address is not the address of an existing gateway token. If it is, the function reverts.

In `interchainTransfer()`, the balance of `msg.sender` is validated against the `amount` of tokens to be transferred. If the balance is not sufficient, the function reverts.

For balance tracking and the check if the token address is a gateway token, two new internal functions were added: `_isGatewayToken()` and `_setDeployerTokenBalance()`.

The interface `IInterchainTokenFactory` was modified to reflect the changes in the contract. All functions are now thoroughly documented using the NatSpec format.

### InterchainTokenService

Two new immutable variables were added to the contract: `tokenManager` and `tokenHandler`. The constant `MESSAGE_TYPE_INTERCHAIN_TRANSFER_WITH_DATA` was removed and the constants `MESSAGE_TYPE_DEPLOY_INTERCHAIN_TOKEN` and `MESSAGE_TYPE_DEPLOY_TOKEN_MANAGER` were updated to have new values. The `onlyTokenManager` modifier was removed. The contract does not distinguish between different token manager types.

The `_expressExecute()` implementation was changed to reflect the introduction of the new [TokenHandler]() class, which handles transfer operations. The contract now uses the `delegatecall` to the `tokenHandler` contract to execute the `transferTokenFrom` from `msg.sender` to `destinationAddress` with the `amount` of tokens. The `InterchainTransferReceived` event is emitted after the successful transfer.

The `_processInterchainTransferPayload()` function now supports additional `data` in the payload. If the length of the `data` is greater than zero, the function calls the `executeWithInterchainToken()` function on the `IInterchainTokenExecutable` contract.

The `_deployTokenManager()` function now calls the `approveService()` function on the `TokenManager` contract for `LOCK_UNLOCK` and `LOCK_UNLOCK_FEE` token manager types.

The `_decodeMetadata()` function implementation was simplified, as well as the `_transmitInterchainTransfer()` function. Functions `interchainTransfer()`, `callContractWithInterchainToken` and `transmitInterchainTransfer()` were updated to reflect the changes in the `_transmitInterchainTransfer` implementation and to utilize the newly introduced `_takeToken()` function. Additionally, the `transmitInterchainTransfer()` is now only callable by a token.

Finally, functions `_takeToken()` and `_giveToken()` were added to the contract. The `_takeToken()` function is used to transfer tokens from the `msg.sender`, while the `_giveToken()` function is used to transfer tokens to the `msg.sender`. In the previous revision, this functionality was a part of the `TokenManager` contract.

The interface `IInterchainTokenService` was modified to reflect the changes in the contract. All functions are now thoroughly documented using the NatSpec format.

### TokenHandler

The newly introduced `TokenHandler` contract is used to handle token transfers before or after making an interchain transfer. It has two main functions, namely `giveToken()` and `takeToken()`. The need for this contract arose from the fact that multiple token types can be used in the interchain transfer (`MINT_BURN`, `LOCK_UNLOCK` and `LOCK_UNLOCK_FEE`). The contract is assumed to be called using the `delegatecall` from `InterchainTokenService`.

The new interface `ITokenHandler` was introduced to reflect the changes in the `TokenHandler` contract. All functions are now thoroughly documented using the NatSpec format.

### BaseInterchainToken

This abstract contract introduced two virtual view functions, namely `interchainTokenId()` and `interchainTokenService()`. The `interchainTransfer()` and `interchainTransferFrom()` functions were modified to use the InterchainTokenService instead of TokenManager to handle interchain transfers. All functions are now thoroughly documented using the NatSpec format.

### InterchainToken

The initialization flag was introduced that is stored in the contract storage on the `INITIALIZED_SLOT`. The `setup()` function was renamed to `init()`, and the code was modified to set the flag to `true` after the initialization. Additionally, if the `distributor` is set to be the zero address, it is still added using the `_addDistributor()` function for users to easily check that no custom distributor is set. Since the contract inherits from `BaseInterchainToken`, the `interchainTokenId()` and `interchainTokenService()` functions were implemented. All functions are now thoroughly documented using the NatSpec format.

### IBaseTokenManager

This contract is a new interface that defines the functions that are common to all token managers. These functions are `interchainTokenId()`, `tokenAddress()` and `getTokenAddressFromParams()`. All functions are now thoroughly documented using the NatSpec format.

## IInterchainToken

Several new errors were added, as well as declarations of `interchainTokenService()`, `interchainTokenId()` and `init()`. The view `tokenManager()` function was removed. All functions are now thoroughly documented using the NatSpec format.

## Proxy

This file serves as an alias for the `Proxy` contract from the Axelar GMP SDK.

## TokenManager

The `onlyToken()` modifier was removed. Functions `tokenAddress()`, `interchainTokenId()` and `implementationType()` now revert, since they are intended to be called on the token manager proxy.

In the `setup()` function, the operator now can be set to the zero address. Functions `transmitInterchainTransfer()`, `giveToken()` and `takeToken()` were removed. External `addFlowIn()`, `addFlowOut()` and `isFlowLimiter()` functions were introduced. `addFlowLimiter()` and `removeFlowLimiter()` functions now accept the zero address as an argument due to the introduction of the zero address operators and flow limiters. The new `approveService()` function approves the InterchainTokenService contract for the maximum spending allowance.

The `TokenManagerProxy` contract was modified to reflect the changes in the `TokenManager` contract. The interfaces `ITokenManager` and `ITokenManagerProxy` were modified to reflect the changes in the contract and its proxy. All functions are now thoroughly documented using the NatSpec format.

## InterchainTokenDeployer

The `deployInterchainToken()` function now uses the ERC-1167 minimal clones to create a new token contract using the Create3 method. After the successful creation of a token contract, the deployer calls the initializer of a newly created token.

## 12.2. Actors

This revision did not introduce new actors.

## 12.3. Trust Model

This revision did not introduce new trust assumptions.

## W29: Missing symbol validation

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainToken.sol | Type: | Data validation |

*Listing 64. Excerpt from [InterchainToken]()*

```
        if (tokenId_ == bytes32(0)) revert TokenIdZero();
        if (bytes(tokenName).length == 0) revert TokenNameEmpty();

        name = tokenName;
         symbol = tokenSymbol;
         decimals = tokenDecimals;
         tokenId = tokenId_;
```

### Description

The `init` function in the [InterchainToken]() contract validates only `tokenId_` and `tokenName`.

### Recommendation

We recommend validating `tokenSymbol` as well for more robust validation.

```
if (bytes(tokenSymbol).length == 0) revert TokenSymbolEmpty();
```

### Solution (Revision 6.1)

The issue was fixed by adding a check for `tokenSymbol`.

[Go back to Findings Summary]()

# I18: Missing documentation

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenFactory.sol | Type: | Documentation |

### Description

In the InterchainTokenFactory contract, the `deployerTokenBalance` function does not have the NatSpec docstring.

### Recommendation

We recommend covering the code by NatSpec. High-quality documentation has to be an essential part of any professional project.

### Solution (Revision 6.1)

The issue is not relevant since the client ”removed this balance tracking altogether due to the complexity”.

Go back to Findings Summary

## I19: `InterchainToken` has code unrelated to the token logic

| | | | |
|---|---|---|---|
| Impact: | Info | Likelihood: | N/A |
| Target: | InterchainToken.sol | Type: | Best practices |

### Description

The InterchainToken contract is upgradeable and initializable. The initialization part of the contract is baked into the core logic code and is a part of the implementation contract. This approach defies the "Separation of Concerns" principle and increases the code complexity.

### Recommendation

We recommend moving the initialization logic into a separate contract and making the InterchainToken contract inheriting from it. In particular, the following may be moved to a separate contract:

- Functions: `_isInitialized`, `_initialize`
- Variable: `INITIALIZED_SLOT`
- Error: `AlreadyInitialized`

### Solution (Revision 6.1)

The client acknowledged the issue. The client's comment: "Not using it elsewhere at the moment".

Go back to Findings Summary

ackee | blockchain security

# I20: Unused constant

| Impact: | Info | Likelihood: | N/A |
| --- | --- | --- | --- |
| Target: | TokenManager.sol | Type: | Code quality |

### Description

In the TokenManager contract, the constant `LATEST_METADATA_VERSION` is not used and duplicates the same variable from the InterchainTokenService contract.

### Recommendation

Remove the unused variable from the TokenManager contract.

### Solution (Revision 6.1)

The issue was fixed by removing the constant.

Go back to Findings Summary

# I21: Inconsistent code style

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenFactory.sol | Type: | Code quality |

*Listing 65. Excerpt from [InterchainTokenFactory]()*

```
    function interchainTokenSalt(bytes32 chainNameHash_, address deployer,
bytes32 salt) public pure returns (bytes32) {
        return keccak256(abi.encode(PREFIX_INTERCHAIN_TOKEN_SALT,
chainNameHash_, deployer, salt));
    }
```

*Listing 66. Excerpt from [InterchainTokenFactory]()*

```
    function canonicalInterchainTokenSalt(bytes32 chainNameHash_, address
tokenAddress) public pure returns (bytes32 salt) {
        salt = keccak256(abi.encode(PREFIX_CANONICAL_TOKEN_SALT,
chainNameHash_, tokenAddress));
    }
```

## Description

NIT, the `interchainTokenSalt` function in [InterchainTokenFactory]() contract uses an unnamed return variable, but `canonicalInterchainTokenSalt` and other functions use a named return variable.

## Recommendation

Unify the code style and use the named return variable in the `interchainTokenSalt` function.

## Solution (Revision 6.1)

The issue was acknowledged with the client's comment: "`salt` is also an argument, we can consider a different name perhaps".

Go back to Findings Summary

# 13. Report revision 7.0

## 13.1. System Overview

This revision mainly focuses on the integration of gateway tokens to the token service and implementation of the `executeWithToken` and `expressExecuteWithToken` functions. All changes are described in detail in the following sections.

### InterchainTokenFactory

A new function `registerGatewayToken` was introduced to the `InterchainTokenFactory` contract. This function allows the registration of gateway tokens. The `onlyOwner` modifier is used to restrict access to the function.

The corresponding interface, `IInterchainTokenFactory` was also updated to include the changes in the `InterchainTokenFactory` contract.

### InterchainTokenService

The `executeWithToken` and `expressExecuteWithToken` functions were implemented. These functions are called by the Axelar Gateway when an interchain call is made with a token payload. Two new internal functions were also introduced handling the main logic of token calls, namely, `_callContractWithToken` and `_execute`.

The corresponding interface, `IInterchainTokenService` was also updated with a newly introduced `PostDeployFailed` error type.

### TokenHandler

The handler was changed to support the new token type, namely, gateway tokens. In this revision, the constructor was defined to store the address of the Axelar Gateway. A new function, `postTokenManagerDeploy` was implemented that is called after the `TokenManager` contract is deployed. This function is used to set the correct allowances of the target token for the InterchainTokenService contract. In the case of gateway tokens, the allowance is set to the maximum value.

The corresponding interface, `ITokenHandler` was also updated to include the external getter function for the address of the Axelar Gateway and a new error type.

## 13.2. Actors

### Gateway Tokens

In this revision, the new token type, namely, gateway tokens, was introduced. Gateway tokens are used to represent legacy tokens that were registered on the gateway. These tokens can only be registered by the Axelar team, or the InterchainTokenFactory owner, and are considered trusted and known.

## 13.3. Trust Model

The aforementioned gateway tokens are considered trusted. They are registered in the InterchainTokenService contract by the Axelar team. Their behavior is considered to follow the ERC-20 standard and is not validated.

# C1: The `executeWithToken` function may be called by anyone

*Critical severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | High |
| Target: | InterchainTokenService.sol | Type: | Access Control |

### Description

In [InterchainTokenService](), the function `executeWithToken` is declared as external with no modifiers. This function, however, should only be available for calling from a remote InterchainTokenService, otherwise this may lead to a loss of funds.

### Exploit scenario

Eve sends a message with any token attached to the InterchainTokenService through the Axelar Gateway using the General Message Passing protocol. The Gateway approves the call and sends the call with a `payload` crafted by Eve to the `executeWithToken` function. The `payload` is an ABI-encoded string with the following values:

- `messageType`: `MESSAGE_TYPE_INTERCHAIN_TRANSFER`;
- `tokenId`: The ID of any token held by the InterchainTokenService;
- `destinationAddress`: The address of an Eve-controlled address enabled to receive tokens;
- `amount`: The amount of tokens to be received;
- `data`: An empty byte-string.

This payload propagates through the `executeWithToken`, `_execute` and finally to `_processInterchainTransferPayload`, which sends the requested `amount` of the token to the destination address and emits the `InterchainTransferReceived` event.

Eve repeats the steps above for every token held by `InterchainTokenService` and on multiple chains. This way, Eve steals all assets stored in the contract.

### Recommendation

Add the `onlyRemoteService(sourceChain, sourceAddress)` modifier to the `executeWithToken` function.

### Solution (Revision 8.0)

The `onlyRemoteService(sourceChain, sourceAddress)` modifier was added to the `executeWithToken` function.

[Go back to Findings Summary]

## H4: `executeWithToken` allows to perform interchain operations even when the protocol is paused

*High severity issue*

| | | | |
|---|---|---|---|
| Impact: | High | Likelihood: | Medium |
| Target: | InterchainTokenService.sol | Type: | Access Control |

### Description

In InterchainTokenService, the `executeWithToken` function is a part of the core functionality of the InterchainTokenService. The protocol has a feature to pause the execution of all core functions. However, the `executeWithToken` function does not have the correct `whenNotPaused` modifier allowing for the execution of core operations even in the paused state. Together with C1, this issue makes it impossible to stop the adversary's actions and prevent more damage to the protocol's treasury.

### Exploit Scenario

Eve performs the steps described in detail in C1. The Axelar team receives a notification that an attack is taking place on the protocol. The team decides to do the emergency shutdown of the `InterchainTokenService` protocol to prevent further damage. However, the vulnerable function lacks the required modifier and still allows the execution of interchain transfers. Eve successfully finishes the attack.

### Recommendation

Add the `whenNotPaused` modifier to the `executeWithToken` function of the `InterchainTokenService` protocol.

### Solution (Revision 8.0)

The `whenNotPaused` modifier was added to the `executeWithToken` function.

Go back to Findings Summary

# M10: ERC-20 double approval

*Medium severity issue*

| Impact: | Medium | Likelihood: | Medium |
|---|---|---|---|
| Target: | TokenHandler | Type: | Non-standard tokens |

## Description

The `TokenHandler` contract contains a logic to give an ERC-20 approval to the Axelar gateway.

*Listing 67. Excerpt from [TokenHandler]()*

```
    function _approveGateway(address tokenAddress, uint256 amount) internal
{
IERC20(tokenAddress).safeCall(abi.encodeWithSelector(IERC20.approve.selector
, gateway, amount));
    }
```

The logic is called through a delegatecall so that the approval is given from `InterchainTokenService`.

*Listing 68. Excerpt from [InterchainTokenService]()*

```
        (success, returnData) = tokenHandler.delegatecall(
abi.encodeWithSelector(ITokenHandler.postTokenManagerDeploy.selector,
tokenManagerType, tokenManager_)
        );
```

The approval call may be executed in multiple transactions when deploying multiple gateway token managers for the same token. However, some ERC-20 tokens may require setting the allowance to zero before setting a new allowance. This is the case for USDT, for example.

The full proof of concept code in the Wake testing framework is enclosed in Appendix C.

### Exploit scenario

Axelar deploys a canonical gateway token manager for USDT. The transaction executes an approval from `InterchainTokenService` to the Axelar gateway on USDT. Since this is the first approval executed on USDT with these parameters, the `approve` call succeeds (the previous allowance was zero). Later, an Axelar user decides to deploy a non-canonical gateway token manager for USDT. The transaction performs the same approval call. It fails because the previous allowance was non-zero.

### Recommendation

Use OpenZeppelin's `safeIncreaseAllowance` or `forceApprove` from the `SafeERC20` library.

### Solution (Revision 8.0)

An additional check for the zero allowance was added. The `approve` function is now only called when the current allowance is zero. Since the allowance for the gateway by InterchainTokenService can only be changed through the `TokenHandler` contract, using this simple condition instead of `forceApprove` from the `SafeERC20` library does not affect the protocol and does not introduce any new issue.

Go back to Findings Summary

# M11: Optional ERC-20 functions required

*Medium severity issue*

| Impact: | Medium | Likelihood: | Low |
|---|---|---|---|
| Target: | InterchainTokenFactory, InterchainTokenService | Type: | Non-standard tokens |

## Description

The solution assumes that ERC-20 tokens to be added to `InterchainTokenService` implement optional functions `name()`, `symbol()`, `decimals()`.

*Listing 69. Excerpt from [InterchainTokenFactory]()*

```
            tokenName = token.name();
            tokenSymbol = token.symbol();
            tokenDecimals = token.decimals();
```

*Listing 70. Excerpt from [InterchainTokenService]()*

```
        if (tokenManagerType == uint256(TokenManagerType.GATEWAY)) {
            symbol = IERC20Named(tokenAddress).symbol();
        }
```

However, some ERC-20 tokens may not implement these functions or may implement them with different return types.

## Exploit scenario

A user decides to deploy a gateway token manager for the MKR token. MKR returns `bytes32` for `name()` and `symbol()` instead of expected string. This causes an ABI decoding revert.

#### Recommendation

Consider adding an extra mapping from the token address to the token symbol to support non-standard tokens like MKR.

#### Solution (Revision 8.0)

The issue was acknowledged by the team with the comment:

> We explicitly only support tokens that have ERC20 metadata in ITS factory. I'm aware of the MKR issue, but it's not a major token to justify adding workarounds, and a user can always use the gateway directly. We'll be documenting unsupported tokens in our docs.

Go back to Findings Summary

# W30: `executeWithToken` and `expressExecuteWithToken` functions do not check for the correctness of payload arguments

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService.sol | Type: | Input Validation |

### Description

In InterchainTokenService, the functions `executeWithToken` and `expressExecuteWithToken` do not have validations if the passed `amount` and `symbol` input arguments correspond to those encoded in the `payload`. The logic implies that the only caller is the `InterchainTokenService`, which always sends the correct arguments. However, together with C1 this lack of validation allows for sending an arbitrary token from the source chain for any token on the destination chain. While the mentioned issue is easily fixed, the absence of adequate validations may lead to further issues in the future when new logic is added.

### Recommendation

We recommend adding additional checks to the aforementioned functions to validate the input payload and compare it to other input arguments.

### Solution (Revision 8.0)

The issue was fixed by adding validation to the `executeWithToken`, `expressExecuteWithToken` and `contractCallWithTokenValue` functions. The validation checks if the `amount` and a token address corresponding to `symbol` from input arguments match those encoded in the `payload`.

Go back to Findings Summary

# 14. Report revision 8.0

This report revision was dedicated to the Interchain Token Service fuzzing.

ackee | blockchain security

## C2: Mint privilege

*Critical severity issue*

| Impact: | High | Likelihood: | High |
|---|---|---|---|
| Target: | InterchainToken, TokenHandler, InterchainTokenService | Type: | Access control, Data validation |

### Description

The project allows for the deployment of different token managers of different types bound to ERC-20 tokens. One of the types is the mint/burn token manager type, which is most commonly used with the `InterchainToken` ERC-20 contract and can be automatically deployed through `InterchainTokenFactory` or `InterchainTokenService`.

In order for an interchain token connection to work properly, the project must mint tokens to the interchain transfer recipient in the case of the mint/burn type. This is achieved by giving the mint permission to the `InterchainTokenService` contract. The mint logic is invoked through the `TokenHandler` contract using a delegatecall.

*Listing 71. Excerpt from [InterchainToken]()*

```
        /**
         * @dev Set the token service as a minter to allow it to mint and burn tokens.
         * Also add the provided address as a minter. If `address(0)` was provided,
         * add it as a minter to allow anyone to easily check that no custom minter was set.
         */
        _addMinter(interchainTokenService_);
        _addMinter(minter);
```

*Listing 72. Excerpt from [InterchainTokenService]*

```
        (bool success, bytes memory data) = tokenHandler.delegatecall(
            abi.encodeWithSelector(ITokenHandler.giveToken.selector, tokenManagerType, tokenAddress, tokenManager_, to, amount)
        );
```

*Listing 73. Excerpt from [TokenHandler]*

```
    function giveToken(
        uint256 tokenManagerType,
        address tokenAddress,
        address tokenManager,
        address to,
        uint256 amount
    ) external payable returns (uint256) {
        if (tokenManagerType == uint256(TokenManagerType.MINT_BURN) || tokenManagerType == uint256(TokenManagerType.MINT_BURN_FROM)) {
            _giveTokenMintBurn(tokenAddress, to, amount);
            return amount;
        }
```

*Listing 74. Excerpt from [TokenHandler]*

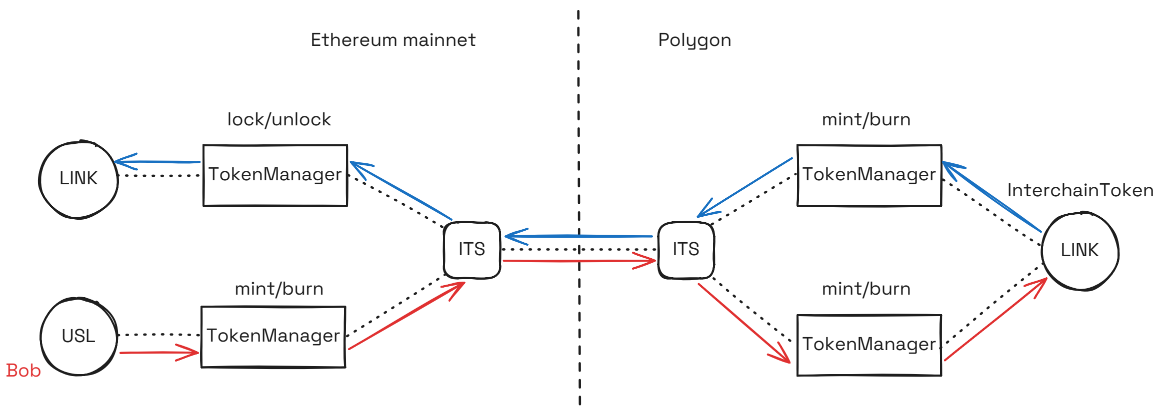
```
    function _giveTokenMintBurn(address tokenAddress, address to, uint256 amount) internal {
        IERC20(tokenAddress).safeCall(abi.encodeWithSelector(IERC20MintableBurnable.mint.selector, to, amount));
    }
```

Given the fact the mint privilege is not bound to a specific token manager but to the `InterchainTokenService` contract, it is possible to exploit the mint privilege by deploying a token manager of the mint/burn type bound to an already existing token that has the mint privilege given to `InterchainTokenService`. The issue is not only limited to the `InterchainToken` contract but to all tokens bound to a mint/burn token manager because the token must always give minting access to `InterchainTokenService`.

## Exploit scenario

Alice registers a canonical token manager for the LINK token on the Ethereum mainnet and deploys a connected remote token manager and an interchain token on the Polygon chain. The token manager on the Polygon chain is of the mint/burn type, and the interchain token is given minting access to the `InterchainTokenService` contract on Polygon.

Bob deploys a useless token (USL) on the Ethereum mainnet with any preferred token manager type and deploys a connected remote token manager of the mint/burn type linked to the LINK interchain token on the Polygon chain. Two different mint/burn token managers are now linked with the LINK interchain token on Polygon. One is bound to a legit LINK token on the Ethereum mainnet, and the other is to a useless token on the Ethereum mainnet. Bob then mints 1000 USL tokens on the Ethereum mainnet and transfers it to the interchain token on Polygon. Bob receives 1000 LINK tokens on the Polygon chain in return. Then, Bob can use the legit interchain connection to send 1000 LINK tokens from Polygon and receive 1000 LINK tokens on the Ethereum mainnet. Bob received 1000 LINK tokens for 1000 USL tokens.




A proof of concept code snippet in the Wake testing framework is enclosed in Appendix C.

## Disclosure process

The issue was discovered with the project already deployed on all major EVM-compatible chains; however, it was during a soft launch phase of the project. The vulnerability was first disclosed to Ackee Blockchain's Ethereum tech lead to quickly confirm the severity and validity of the finding. Then, the issue was disclosed to Axelar's management and development team. The client reacted promptly, confirming the issue and preparing a hotfix to be deployed on the chains concerned.

The hotfix was discussed and reviewed by the Ackee Blockchain team, and a discussion with Axelar followed to ensure no attack vectors were left open. Both teams also checked and verified no funds were endangered or exploited. The vulnerability fix was deployed within 12 hours of the initial disclosure on all affected chains.

The following image shows the reviewed hotfix.

```
Unstaged changes (1)
modified   contracts/InterchainTokenService.sol
@@ -281,17 +281,5 @@ contract InterchainTokenService is
             uint256 gasValue
         ) external payable whenNotPaused returns (bytes32 tokenId) {
-            address deployer = msg.sender;
-
-            if (deployer == interchainTokenFactory) deployer = TOKEN_FACTORY_DEPLOYER
-
-            tokenId = interchainTokenId(deployer, salt);
-
-            emit InterchainTokenIdClaimed(tokenId, deployer, salt);
-
-            if (bytes(destinationChain).length == 0) {
-                _deployTokenManager(tokenId, tokenManagerType, params);
-            } else {
-                _deployRemoteTokenManager(tokenId, destinationChain, gasValue, tokenMa
-            }
+            revert('');
         }

@@ -737,10 +725,5 @@ contract InterchainTokenService is
          */
         function _processDeployTokenManagerPayload(bytes calldata payload) internal {
-            (, bytes32 tokenId, TokenManagerType tokenManagerType, bytes memory params
-                payload,
-                (uint256, bytes32, TokenManagerType, bytes)
-            );
-
-            _deployTokenManager(tokenId, tokenManagerType, params);
+            revert('');
         }
```

### Solution (Revision 8.1)

The issue was fixed in a private repository reflecting the state from the commit a6d33b8 in the public repository.

The fix kept the original mint/burn logic exclusively for `InterchainToken` instances deployed together with a token manager. Deploying just a token manager of the original mint/burn type is no longer possible. Instead, a new custom mint/burn token manager type was introduced. The mint permission is now bound to the token manager contract, and the mint logic is invoked through the token manager contract.

The logic for the mint/burnFrom token manager type was migrated by analogy, i.e. the mint permission is now bound to the token manager contract. As a consequence, possibly existing mint/burnFrom tokens and token managers will stop working. This is an expected behavior as the mint/burnFrom token manager was not being used prior to the fix according to the client.

Go back to Findings Summary

## W31: Zero amount transfers possible

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenService | Type: | Data validation |

### Description

The Interchain Token Service does not perform any check that the zero amount of tokens is requested to be transferred. An exception to this is an interchain transfer through a gateway token manager. The transfer is internally performed using the Axelar gateway, which contains a check for the zero amount.

### Recommendation

Add a check performing a revert on the zero amount for protocol consistency and to prevent unintended protocol interactions.

### Solution (Revision 8.1)

The finding was fixed in the commit ece8a09 by adding an extra check for the zero value to `InterchainTokenService._transmitInterchainTransfer`.

Go back to Findings Summary

ackee | blockchain security

## W32: Canonical lock/unlock fee-on-transfer tokens

| Impact: | Warning | Likelihood: | N/A |
|---|---|---|---|
| Target: | InterchainTokenFactory | Type: | Non-standard tokens |

### Description

The `InterchainTokenFactory` contract allows for the deployment of a canonical token manager for a given token. The canonical token manager can be trusted (as long as the underlying token can be trusted) because the token manager deployer gains no privileges over the token manager. The deployed canonical token manager is always of the lock/unlock type and can be deployed only once per token address.

*Listing 75. Excerpt from [InterchainTokenFactory](InterchainTokenFactory)*

```
    function registerCanonicalInterchainToken(address tokenAddress) external
payable returns (bytes32 tokenId) {
        bytes memory params = abi.encode('', tokenAddress);

        if (_isGatewayToken(tokenAddress)) revert
GatewayToken(tokenAddress);

        bytes32 salt = canonicalInterchainTokenSalt(chainNameHash,
tokenAddress);
        tokenId = interchainTokenService.deployTokenManager(salt, '',
TokenManagerType.LOCK_UNLOCK, params, 0);
    }
```

Furthermore, the Interchain Token Service project distinguishes between lock/unlock tokens and lock/unlock fee-on-transfer tokens. Lock/unlock token managers do not expect a fee to be taken on a transfer, so registering a fee-on-transfer token with the canonical token manager would lead to accounting issues when Interchain Token Service would use the full amount of tokens (including the fee that should be subtracted). Additionally, registering a fee-on-transfer token as canonical by accident or intentionally (by a malicious user) binds the token with an incorrect token manager type. This cannot be easily fixed even after an upgrade.

#### Exploit scenario

A fee-on-transfer token is registered with a canonical token manager by the token owner or by a malicious actor. Due to the incorrect token manager type, no fees are paid by the user on the given chain.

#### Recommendation

Allow choosing the correct token manager type with a new `tokenId` derivation path or clearly state that fee-on-transfer tokens are not supported in this scenario.

#### Solution (Revision 8.1)

The client acknowledged the finding with the following response:

> Lock/unlock fee-on-transfer tokens are not supported by the ITS factory to be registered in a canonical way due to being out of scope. Custom token managers should be used for those.
>
> — Axelar

Go back to Findings Summary

# I22: Unused using-for directives

| Impact: | Info | Likelihood: | N/A |
|---|---|---|---|
| Target: | `**/*` | Type: | Code quality |

### Description

The codebase contains unused using-for directives. See Appendix C for the list of all unused directives.

### Recommendation

Remove all unused using-for directives to improve code quality and maintainability.

### Solution (Revision 8.1)

The issue was fixed in the commit b4b0759 by removing all but one reported unused using-for directives. The unused using-for in `InterchainToken` was kept to avoid using different versions of the contract (it is not upgradeable).

Go back to Findings Summary

ackee | blockchain security

# Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar: Interchain Token Service, 11.4.2024.

ackee | blockchain security

# Appendix B: Glossary of terms

The following terms might be used throughout the document:

**Superclass/Ancestor of C**

A contract that C inherits/derives from.

**Subclass/Child of C**

A contract that inherits/derives from C.

**Syntactic contract**

A Solidity contract. May have an inheritance chain, and may be deployed.

**Deployed contract**

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

**Init/initialization function**

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

**External entrypoint**

A `public` or `external` function.

**Public/Publicly-accessible function/entrypoint**

An `external` or `public` function that can be successfully executed by any network account.

**Mutating function**

A non-`view` and non-`pure` function.

# Appendix C: Wake outputs

This section presents the outputs of the Wake tool.

## C.1. H1 proof of concept

```
def test_express_receive(setup_services):
    service1, service2 = setup_services
    assert isinstance(service1, InterchainTokenService)
    assert isinstance(service2, InterchainTokenService)

    owner = chain1.accounts[0].address
    attacker1 = chain1.accounts[1].address
    attacker2 = chain1.accounts[2].address

    token1 = ERC20MintableBurnable.deploy("Test", "TST", 18, chain=chain1)
    token2 = ERC20MintableBurnable.deploy("Test", "TST", 18, chain=chain2)
    salt = keccak256(b"salt")

    service1.deployCustomTokenManagerMintBurn(salt, bytes(owner), token1)
    service1.deployRemoteCustomTokenManagers(
        salt,
        ["chain2"],
        [ITokenManagerType.TokenManagerType.MINT_BURN],
        [service1.getParamsMintBurn(bytes(owner), token2.address)],
        [0],
    )
    token_id = service1.getCustomTokenId(owner, salt)

    token1.mint(attacker1, 1_000)
    token2.mint(owner, 1_000)
    token2.mint(attacker2, 1_000)

    token_manager1 =
TokenManager(service1.getValidTokenManagerAddress(token_id), chain=chain1)
    token1.approve(token_manager1, 1_000, from_=attacker1)
    send_hash = keccak256(Abi.encode(["bytes32", "uint256", "uint256"],
[token_id, chain1.blocks["pending"].number, 1_000]))

    token2.approve(service2, 1_000, from_=owner)
    service2.expressReceiveToken(token_id, attacker1, 1_000, send_hash,
from_=owner)
```

```
    assert token1.balanceOf(attacker1) == 1_000  # has not changed
    assert token2.balanceOf(attacker1) == 1_000  # owner lent attacker1 1_000
tokens
    assert token2.balanceOf(owner) == 0
    assert token2.balanceOf(attacker2) == 1_000  # has not changed

    token2.approve(service2, 1_000, from_=attacker2)
    service2.expressReceiveToken(token_id, attacker1, 1_000, send_hash,
from_=attacker2)
    assert token1.balanceOf(attacker1) == 1_000  # has not changed
    assert token2.balanceOf(attacker1) == 2_000  # owner lent attacker1 1_000
tokens, attacker2 lent attacker1 additional 1_000 tokens
    assert token2.balanceOf(owner) == 0
    assert token2.balanceOf(attacker2) == 0  # has not changed

    token_manager1.sendToken("chain2", bytes(attacker1), 1_000, from_=attacker1)

    assert token1.balanceOf(attacker1) == 0
    assert token2.balanceOf(attacker1) == 2_000
    assert token2.balanceOf(owner) == 0
    assert token2.balanceOf(attacker2) == 1_000
```

## C.2. M10 proof of concept

```
USDT = IERC20("0xdAC17F958D2ee523a2206206994597C13D831ec7")

@default_chain.connect(fork="http://localhost:8545")
@on_revert(lambda e: print(e.tx.call_trace))
def test_allowance():
    a = default_chain.accounts[0]
    start_nonce = a.nonce

    deploy_payload = Abi.encode(
        ["string", "string", "uint8", "uint256", "address", "uint256"],
        ["Token", "TKN", 18, 2**256-1, USDT, 2**256-1],
    )

    gw = MockGateway.deploy()
    gw.deployToken(
        deploy_payload,
        random_bytes(32),
    )
```

```python
    token_handler = TokenHandler.deploy(gw)
    manager_deployer = TokenManagerDeployer.deploy()
    token_deployer = get_create_address(a, start_nonce + 8)
    factory = InterchainTokenFactory(
        Proxy.deploy(
            get_create_address(a, start_nonce + 9),
            a,
            b"",
        ).address,
    )
    token_manager = get_create_address(a, start_nonce + 10)
    its = InterchainTokenService(Proxy.deploy(
        InterchainTokenService.deploy(
            manager_deployer,
            token_deployer,
            gw,
            Address(1),
            factory,
            "chain1",
            token_manager,
            token_handler,
        ),
        a,
        b"",
    ))
    interchain_token = InterchainToken.deploy(its)
    assert InterchainTokenDeployer.deploy(interchain_token).address ==
token_deployer
    InterchainTokenFactory.deploy(its)
    assert TokenManager.deploy(its).address == token_manager

    for _ in range(2):
        its.deployTokenManager(
            random_bytes(32),
            "",
            InterchainTokenService.TokenManagerType.GATEWAY,
            Abi.encode(["string", "address"], ["", USDT]),
            0
        )
```

## C.3. C2 proof of concept

```
its = self.deployments[chain1].its
a1 = random_account(chain=chain1)
a2 = random_account(chain=chain1)
factory = self.deployments[chain1].factory
LINK = IERC20("0x514910771af9ca656af840dff83e8264ecf986ca", chain=chain1)

token_id = factory.registerCanonicalInterchainToken(LINK, from_=a1).return_value
manager = its.validTokenManagerAddress(token_id)
mint_erc20(LINK, a1, 10_000_000)
LINK.transfer(manager, 10_000_000, from_=a1)
assert factory.deployRemoteCanonicalInterchainToken(
    "chain1", LINK, "chain2", 0, from_=a1
).return_value == token_id

dest_token = self.deployments[chain2].its.validTokenAddress(token_id)

salt = random_bytes(32)
useless = ERC20.deploy("useless", "USL", 18, chain=chain1)
token_id2 = its.deployTokenManager(
    salt, "", InterchainTokenService.TokenManagerType.LOCK_UNLOCK,
    abi.encode(b"", useless), 0, from_=a2
).return_value
assert its.deployTokenManager(
    salt, "chain2", InterchainTokenService.TokenManagerType.MINT_BURN,
    abi.encode(b"", dest_token), 0, from_=a2
).return_value == token_id2

assert token_id != token_id2

print("before", LINK.balanceOf(a2))

mint_erc20(useless, a2, 10_000)
useless.approve(its, 10_000, from_=a2)

its.interchainTransfer(
    token_id2,
    "chain2",
    bytes(a2.address),
    10_000,
    b"",
    0,
```

```
    from_=a2
)

self.deployments[chain2].its.interchainTransfer(
    token_id,
    "chain1",
    bytes(a2.address),
    10_000,
    b"",
    0,
    from_=a2.address
)

print("after", LINK.balanceOf(a2))
```

## C.4. I22 list of all unused using-for detections

```
wake detect unused-using-for

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
  19  * @notice This contract is responsible for deploying new interchain to
  20  */
  21 contract InterchainTokenFactory is IInterchainTokenFactory, ITokenManag
❯ 22     using AddressBytes for bytes;
  23     using AddressBytes for address;
  24     using SafeTokenTransfer for IInterchainToken;
  25
source-v2/contracts/InterchainTokenFactory.sol

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
  21 contract InterchainTokenFactory is IInterchainTokenFactory, ITokenManag
  22     using AddressBytes for bytes;
  23     using AddressBytes for address;
❯ 24     using SafeTokenTransfer for IInterchainToken;
  25     using SafeTokenTransferFrom for IInterchainToken;
  26     using SafeTokenCall for IInterchainToken;
  27
source-v2/contracts/InterchainTokenFactory.sol

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
  22     using AddressBytes for bytes;
  23     using AddressBytes for address;
  24     using SafeTokenTransfer for IInterchainToken;
❯ 25     using SafeTokenTransferFrom for IInterchainToken;
  26     using SafeTokenCall for IInterchainToken;
  27
  28
source-v2/contracts/InterchainTokenFactory.sol

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
  23     using AddressBytes for address;
  24     using SafeTokenTransfer for IInterchainToken;
  25     using SafeTokenTransferFrom for IInterchainToken;
❯ 26     using SafeTokenCall for IInterchainToken;
  27
  28     IInterchainTokenService public immutable interchainTokenService;
  29
source-v2/contracts/InterchainTokenFactory.sol
```

wake detect unused-using-for

```
┌ [WARNING][LOW] Unused contract in using-for directive [unused-using-for] ┐
│        InterchainAddressTracker,
│        IInterchainTokenService
│    {
│ ❯      using StringToBytes32 for string;
│        using Bytes32ToString for bytes32;
│        using AddressBytes for bytes;
│
└ source-v2/contracts/InterchainTokenService.sol ──────────────────────────┘

┌ [WARNING][LOW] Unused contract in using-for directive [unused-using-for] ┐
│        IInterchainTokenService
│    {
│        using StringToBytes32 for string;
│ ❯      using Bytes32ToString for bytes32;
│        using AddressBytes for bytes;
│        using AddressBytes for address;
│
└ source-v2/contracts/InterchainTokenService.sol ──────────────────────────┘

┌ [WARNING][LOW] Unused contract in using-for directive [unused-using-for] ┐
│        using Bytes32ToString for bytes32;
│        using AddressBytes for bytes;
│        using AddressBytes for address;
│ ❯      using SafeTokenTransferFrom for IERC20;
│        using SafeTokenCall for IERC20;
│
│
└ source-v2/contracts/InterchainTokenService.sol ──────────────────────────┘

┌ [WARNING][LOW] Unused contract in using-for directive [unused-using-for] ┐
│        using AddressBytes for bytes;
│        using AddressBytes for address;
│        using SafeTokenTransferFrom for IERC20;
│ ❯      using SafeTokenCall for IERC20;
│
│        IAxelarGateway public immutable gateway;
│
└ source-v2/contracts/InterchainTokenService.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
│ ❯      using AddressBytes for bytes;
│
│        string public name;
│
└ source-v2/contracts/interchain-token/InterchainToken.sol ────────────────┘
```

# Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain