Axelar

Ethereum Bridge

31 March 2022

by Ackee Blockchain

Contents

1. Document Revisions

1.0	Final report	March 31, 2022
1.1	Final report • Add Appendix D, *Fix Review*	April 7, 2022

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, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course 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, Rockaway Blockchain Fund.

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 Slither 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 fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Review team

Member's Name	Position
Dominik Teiml	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.4. 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

Axelar is a cross-chain interoperability solution. It allows users to send tokens and interact with contracts in a cross-chain manner.

Between March 22 and March 31, 2022, Axelar engaged Ackee Blockchain to conduct a security review of the Solidity CGP Gateway project. This was a follow-up from our earlier assesment, where we also reviewed this project.

Working from commit 838de95e41, we were allocated 10 engineering days and the lead auditor was Dominik Teiml.

We began our review by looking for common Solidity pitfalls. This yielded several issues such as M3: Several external calls lack existence checks and M1: Pitfalls of upgradeability. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- Is the correctness of the contract ensured?
- Do the contracts correctly use dependencies or other contracts they rely on, such as OpenZeppelin dependencies?
- Are access controls not too relaxed or too strict?
- Are the upgradeable contracts subject to common upgradeability pitfalls?
- Is the code vulnerable to re-entrancy attacks, either through ERC777-style contracts, or maliciously supplied user input?

We also created a model of and fuzzed the AdminMultisigBase. The target contract passed on interactions and invariants we identified and tested.

Our review resulted in 10 findings, ranging from Informational to High severity. The most severe one is that an observer could make incorrect decisions, since an event logs incorrect values (see H1: `AxelarGatewayMultisig.transferOperatorship` emits an event with an incorrect value).

Ackee Blockchain recommends Axelar:

- correct the incorrect event emission,
- revise the upgradeability mechanism (see M1: Pitfalls of upgradeability).
- pay special attention to edge cases such as string collision in `abi.encodePacke` (see M2: `abi.encodePacked` contains dynamic-length data),
- address all other reported issues.

Update April 7, 2022: StakerDAO provided an updated codebase that addresses issues from this report. See Appendix D for a detailed discussion of the exact status of each issue.

4. 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.

4.1. Contracts

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

EternalStorage

This contract holds all necessary state variables to carry out the storage of any contract. It is a crucial part of the known upgradeability pattern - Eternal Storage.

AdminMultisigBase

Contract mainly provides the implementation of the admin threshold consensus via the `onlyAdmin` modifier. This modifier is used later in AxelarGateway to provide voting capabilities for a number of admin-only functions.

Each admin has one vote for a given epoch and topic. A consensus is reached if the admin vote count is greater than the threshold. The function can proceed if a consensus on a selected topic is reached. After the function finishes, votes are reset. An important function `_setAdmins` is implemented here. It is used to initialize admins a threshold from the setup function in the gateway.

AxelarGateway

AxelarGateway is an abstract contract that provides definitions of admin functions. Functions are marked with the `onlyAdmin` modifier. The contract provides functionality for admins to collectively agree and approve transactions via threshold cryptography.

Apart from that, it provides implementations functions that can be executed via a command passed to `_execute` function that is implemented in single/multisig gateways.

AxelarGatewayMultisig

Inherits from the AxelarGateway and implements functions that can be executed via external calls. Those functions are executed through the `call` and have the `onlySelf` modifier. They are called through the `_execute` function, which uses dynamic dispatch for this purpose. The function `_execute` receives individual commands to be executed as parameters. It dispatches them if certain access control conditions are met. It receives an array of signatures, which are used for access control. If addresses derived from signatures get appropriate roles, then they are allowed to execute provided commands.

AxelarGatewaySinglesig

The contract has very similar functionality to AxelarGatewayMultisig On the opposite, it uses just one ECDSA signature for access control.

4.2. Actors

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

Admin

Admins, if they reach the consensus, are able to freeze and unfreeze tokens and upgrade the logic contract.

Owner

Owners are able to transfer ownership or operatorship and deploy, mint or burn tokens.

Operator

Operators can mint or burn tokens.

User

A user is somebody that requests the Axelar Network to transfer tokens to a different chain, or interact with a contract on a different chain.

4.3. Trust model

The users have to trust that a threshold number of admins, operators, or owners will not collude to steal their funds.

5. Vulnerabilities risk methodology

Each finding contains an *Impact* and *Likelihood* ratings.

If we have found a scenario in which the issue is exploitable, it will be assigned an impact of *Critical*, *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 *Informational*.

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

5.1. Finding classification

The full definitions are as follows:

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 "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Informational

The issue is on the border-line 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.

6. Findings

This section contains the list of discovered findings. Unless overriden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of the necessary changes to the codebase. In that case, we will try to enumerate them all, making clear which solve the underlying issue better (albeit possibly only with architectural changes) than others.

Summary of Findings

	Type	Impact	Likelihood
H1: `AxelarGatewayMultisig.transferOperatorship` emits an event with an incorrect value	Logging	Medium	High
M1: Pitfalls of upgradeability	Proxy pattern	High	Low
M2: `abi.encodePacked` contains dynamic-length data	Encoding	High	Low
M3: Several external calls lack existence checks	Data validation	High	Low

	Type	Impact	Likelihood
M4: `_execute` functions set command as executed even before it gets executed	Re-entrancy, Data validation	High	Low
M5: Commands that failed can be re-run	Data validation	Medium	Medium
M6: Usage of `solc` optimizer	Compiler configuration	High	Low
W1: `AxelarGatewayMultisig` ignores epoch 0	Data validation, Documentation	Warning	N/A
W2: Cannot use multiple tokens with same symbol	Token interaction, Data validation	Warning	N/A
I1: Many operations don't emit events	Logging	Informational	N/A

Table 1. Table of Findings

H1: `AxelarGatewayMultisig.transferOperatorship` emits an event with an incorrect value

Impact:	Medium	Likelihood:	High
Target:	AxelarGatewayMultisig	Type:	Logging

Listing 1. Excerpt from AxelarGatewayMultisig.transferOperatorship

```
    function transferOperatorship(bytes calldata params, bytes32) external onlySelf {
        (address[] memory newOperators, uint256 newThreshold) = abi.decode(params, (address[], uint256));

        uint256 ownerEpoch = _ownerEpoch();

        emit OperatorshipTransferred(operators(), _getOperatorThreshold(ownerEpoch), newOperators, newThreshold);
```

Description

`AxelarGatewayMultisig.transferOperatorship` is used to create a new operator epoch and set its operators. However, when logging the event `OperatorshipTransferred`, the operator threshold corresponding to the current *owner epoch* gets logged, as opposed to the operator threshold corresponding to the current *operator epoch* (see Listing 1).

Exploit scenario

Operatorship is transferred and an incorrect event value gets logged, resulting in unintended consequences.

Recommendation

Short term, log the operator threshold corresponding to the current operator epoch in Listing 1.

Long term, ensure all log values are correct. This will ensure they are consistent with stakeholders’ expectations.

Go back to Findings Summary

M1: Pitfalls of upgradeability

Impact:	High	Likelihood:	Low
Target:	/**/*	Type:	Proxy pattern

Listing 2. Getter in the logic contract

```
bytes32 public constant CONTRACT_TYPE = keccak256("Axelar Gateway");
```

Listing 3. Require statement for Data validation

```
    require(
        AxelarGateway(newImplementation).CONTRACT_TYPE() == keccak256
("Axelar Gateway"),
        "Not a gateway contract"
    );
```

Description

There are several issues with the current upgradeability mechanism:

1. `AxelarGatewayProxy.constructor` lacks data validation for `gatewayImplementation`.

Recommendation

Add a getter to AxelarGateway that returns a hash unique to the (project, contract) tuple, and check it on proxy construction (see Listing 2 and Listing 3). This will ensure the maximum possible data validation of the logic.

2. `AxelarGatewayProxy.fallback` lacks an existence check for `implementation`

Recommendation

Add an existence check using `implementation.code.length`. This will ensure early error detection in case the implementation ceases to exist.

3. `AxelarGateway.upgrade` has insufficient data validation of `newImplementation`

Recommendation

Add a getter to AxelarGateway that returns a hash unique to the (project, contract) tuple, and check it in the `upgrade` function (see Listing 2 and Listing 3). This will ensure the maximum possible data validation of the new logic contract.

4. It may be possible to call functions other than `setup`, before `setup` is called

Recommendation

Ensure that calling all state-changing public-entrypoints before `setup` results in no state changes that wouldn't be available after it is called. Alternatively, add a require to every state-changing public entrypoint that it cannot be called before `setup` is called.

5. It may be possible to front-run `setup` on the proxy contract

Recommendation

Ensure that the initialization function `setup` reverts if called multiple times. Additionally, in your deployment scripts, ensure that the call to `setup` succeeds. This will ensure that if it was front-run, the deployment will abort.

6. Function shadowing is currently used for authorization

Listing 4. Excerpt from AxelarGatewaySinglesig.setup

```
    function setup(bytes calldata params) external override {
        // Prevent setup from being called on a non-proxy (the
implementation).
        if (implementation() == address(0)) revert NotProxy();
```

Listing 5. Excerpt from AxelarGatewayMultisig.setup

```
    function setup(bytes calldata params) external override {
        // Prevent setup from being called on a non-proxy (the
implementation).
        if (implementation() == address(0)) revert NotProxy();
```

Listing 6. Excerpt from AxelarGatewayProxy.setup

```
    function setup(bytes calldata params) external {}
```

Description

Currently, the `setup` functions only have access control to ensure it is not called on the logic, but lack access control to ensure it is not called by an attacker on the proxy (see Listing 4 and Listing 5). This is currently done by shadowing the `setup` function in the proxy (see Listing 6). This pattern is error-prone, and should be avoided.

Recommendation

Use a traditional way of access control, by require the setup function to be called only once on a single proxy instance.

Go back to Findings Summary

M2: abi.encodePacked contains dynamic-length data

Impact:	High	Likelihood:	Low
Target:	/**/*	Type:	Encoding

Listing 7. Excerpt from AxelarGateway._getIsContractCallApprovedKey

```
    function _getIsContractCallApprovedKey(
        bytes32 commandId,
        string memory sourceChain,
        string memory sourceAddress,
        address contractAddress,
        bytes32 payloadHash
    ) internal pure returns (bytes32) {
        return
            keccak256(
                abi.encodePacked(
                    PREFIX_CONTRACT_CALL_APPROVED,
                    commandId,
                    sourceChain,
                    sourceAddress,
                    contractAddress,
                    payloadHash
                )
            );
    }
```

Description

There are numerous places where dynamic-length data (such as `bytes` or `string`) is used as an argument to `abi.encodePacked` (see Listing 7). This builtin function does not perform the regular encoding of dynamic-length data. Having multiple such data slots could lead to a collision.

Exploit scenario

Consider the function in Listing 7 and consider the following possible inputs for demonstration purposes:

sourceChain	sourceAddress	abi.encodePacked
"ETH"	"2deadbeef"	"ETH2deadbeef"
"ETH2"	"deadbeef"	"ETH2deadbeef"

Both inputs will result in the same value of `abi.encodePacked`, even though they refer to difference source chains and addresses.

Recommendation

Short term, use `abi.encode` over `abi.encodePacked`. This will ensure such a collision cannot occur.

Long term, ensure the contracts are resilient against attackers crafting any form of malicious input.

Go back to Findings Summary

M3: Several external calls lack existence checks

Impact:	High	Likelihood:	Low
Target:	DepositHandler	Type:	Data validation

Listing 8. Excerpt from DepositHandler.execute

```
    {
        (success, returnData) = callee.call(data);
    }
```

Listing 9. Excerpt from AxelarGateway._callERC20Token

```
    function _callERC20Token(address tokenAddress, bytes memory
callData) internal returns (bool) {
        (bool success, bytes memory returnData) = tokenAddress.call
(callData);
        return success && (returnData.length == uint256(0) || abi
.decode(returnData, (bool)));
    }
```

Description

Several places in the code lack existence checks in external calls (see Listing 8 and Listing 9). While Solidity performs existence checks on all high-level calls, using a low-level call bypasses that check. On the EVM, calling accounts that don't contain code with arbitrary calldata results in a successful call. Calling a contract is usually intended to bring about a side-effect, and reverting early in that case will mean the undesired behavior is not propagated to the system.

Exploit scenario

A contract that is expected to be at that particular address self-destructs. The call to it returns success, which can lead to unintended consequences further down the line.

Recommendation

Short term, add existence checks using `account.code.length`. This will ensure that undesired behavior is not propagated to the system.

Long term, do existence checks in all cases when using low-level calls, or document why the contract must exist, or a void call is expected behavior. This will ensure there are no surprises for the stakeholders of the system.

References

M1: Pitfalls of upgradeability

Go back to Findings Summary

M4: `_execute` functions set command as executed even before it gets executed

Impact:	High	Likelihood:	Low
Target:	AxelarGatewaySinglesig, AxelarGatewayMultisig	Type:	Re-entrancy, Data validation

Listing 10. Excerpt from AxelarGatewaySinglesig._execute

```
            // Prevent a re-entrancy from executing this command before it can be marked as successful.
            _setCommandExecuted(commandId, true);
            (bool success, ) = address(this).call(abi.encodeWithSelector(commandSelector, params[i], commandId));
```

Listing 11. Excerpt from AxelarGatewayMultisig._execute

```
            // Prevent a re-entrancy from executing this command before it can be marked as successful.
            _setCommandExecuted(commandId, true);
            (bool success, ) = address(this).call(abi.encodeWithSelector(commandSelector, params[i], commandId));
```

Description

`AxelarGatewaySinglesig._execute` and `AxelarGatewayMultisig._execute` set the command to be executed as executed even before the command call itself (see Listing 10 and Listing 11). This is used to prevent a re-entrnacy from re-executing the command. However, it creates data inconsistency.

Exploit scenario

Somewhere during the command call, some contract queries whether the command has been executed. The contract returns `true` with no way to distinguish it from an already executed transaction, leading to unintended consequences.

Recommendation

Short term, consider creating an enum with an Executing value denoting the transaction is currently being executed. This will allow the calling contract to determine its appropriate behavior.

Long term, ensure the contract returns data consistently with stakeholders' expectations.

References

M5: Commands that failed can be re-run

Go back to Findings Summary

M5: Commands that failed can be re-run

Impact:	Medium	Likelihood:	Medium
Target:	AxelarGatewaySinglesig, AxelarGatewayMultisig	Type:	Data validation

Listing 12. Excerpt from AxelarGatewaySinglesig._execute

```
            (bool success, ) = address(this).call(abi
    .encodeWithSelector(commandSelector, params[i], commandId));
            _setCommandExecuted(commandId, success);
```

Listing 13. Excerpt from AxelarGatewayMultisig._execute

```
            (bool success, ) = address(this).call(abi
    .encodeWithSelector(commandSelector, params[i], commandId));
            _setCommandExecuted(commandId, success);
```

Description

`AxelarGatewaySinglesig._execute` and `AxelarGatewayMultisig._execute` set the inexecutability of a future transaction based on the status of its call (see Listing 12 and Listing 13). Hence commands that failed for whatever reason (due to an out-of-gas exception, a revert, an assertion failure or other) can be replayed.

Exploit scenario

A command is meant by the operators to be executed in a short timestamp. For whatever reason, the call fails. An attacker can replay this command in the future, when it is not intended to be executed.

Recommendation

Short term, always set the command executed to `true`. This will prevent attackers from replaying commands, and allow the operators to sign a new command, if they still wish to execute it.

Long term, ensure the contracts behave in a safe way under all possible circumstances.

References

M4: `_execute` functions set command as executed even before it gets executed

Go back to Findings Summary

M6: Usage of `solc` optimizer

Impact:	High	Likelihood:	Low
Target:	`/**/*`	Type:	Compiler configuration

Description

The project uses the `solc` optimizer. Enabling the `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

W1: `AxelarGatewayMultisig` ignores epoch 0

Impact:	Warning	Likelihood:	N/A
Target:	AxelarGatewayMultisig	Type:	Data validation, Documentation

Listing 14. Excerpt from AxelarGatewayMultisig._areValidPreviousOwners

```
    function _areValidPreviousOwners(address[] memory accounts) internal view returns (bool) {
        uint256 ownerEpoch = _ownerEpoch();
        uint256 recentEpochs = OLD_KEY_RETENTION + uint256(1);
        uint256 lowerBoundOwnerEpoch = ownerEpoch > recentEpochs ? ownerEpoch - recentEpochs : uint256(0);

        --ownerEpoch;
        while (ownerEpoch > lowerBoundOwnerEpoch) {
            if (_areValidOwnersInEpoch(ownerEpoch--, accounts)) return true;
        }
```

Listing 15. Excerpt from AxelarGatewayMultisig.setup

```
        uint256 ownerEpoch = _ownerEpoch() + uint256(1);
        _setOwnerEpoch(ownerEpoch);
        _setOwners(ownerEpoch, ownerAddresses, ownerThreshold);

        uint256 operatorEpoch = _operatorEpoch() + uint256(1);
        _setOperatorEpoch(operatorEpoch);
        _setOperators(operatorEpoch, operatorAddresses, operatorThreshold);
```

Description

`AxelarGatewayMultisig._areValidPreviousOwners` and `AxelarGatewayMultisig._areValidRecentOperators` are functions that check whether a list of accounts constitures a valid owner / operator threshold set in the last recent owner / operator epochs.

They do this using a `while` loop that repeatedly decrements `ownerEpoch` (see Listing 14). However, since the inequality `ownerEpoch > lowerBoundOwnerEpoch` (whose RHS can be 0) is a strict one, the 0 epoch will never be used, even if it is one of the last 16 epochs.

This is not a critical vulnerability as of now, because epochs are currently initialized to 1 (see Listing 15). However, if this assumption is dropped, it can be difficult to spot the behavior of the target functions, especially since this behavior is not documented.

Recommendation

Short term, document the fact that these algorithms skip the 0 epoch.

Long term, document all assumptions functions make. This will make refactoring code easier.

Go back to Findings Summary

W2: Cannot use multiple tokens with same symbol

Impact:	Warning	Likelihood:	N/A
Target:	/**/*	Type:	Token interaction, Data validation

Listing 16. Excerpt from AxelarGateway._deployToken

```
    function _deployToken(
        string memory name,
        string memory symbol,
        uint8 decimals,
        uint256 cap,
        address tokenAddress
    ) internal {
        // Ensure that this symbol has not been taken.
        if (tokenAddresses(symbol) != address(0)) revert
TokenAlreadyExists(symbol);
```

Description

In the Ethereum ecosystem, tokens are usually manipulated by their address, with few assumptions on the value of the symbol. In Axelar Network, tokens are denoted by their symbol. If the intentions are to be able to support multiple tokens with the same symbol, this can lead to undefined behavior.

Recommendation

Short term, document the constraint that all tokens on a chain must have distinct symbols.

Long term, document all assumptions the protocol is making about external contracts.

Go back to Findings Summary

I1: Many operations don't emit events

Impact:	Informational	Likelihood:	N/A
Target:	AxelarGateway	Type:	Logging

Listing 17. Excerpt from AxelarGatewayMultisig.burnToken

```
    function burnToken(bytes calldata params, bytes32) external onlySelf {
        (string memory symbol, bytes32 salt) = abi.decode(params, (string, bytes32));

        _burnToken(symbol, salt);
    }
```

Listing 18. Excerpt from AxelarGateway._burnToken

```
    function _burnToken(string memory symbol, bytes32 salt) internal {
        address tokenAddress = tokenAddresses(symbol);

        if (tokenAddress == address(0)) revert TokenDoesNotExist(symbol);

        if (_getTokenType(symbol) == TokenType.External) {
            DepositHandler depositHandler = new DepositHandler{ salt: salt }();

            (bool success, bytes memory returnData) = depositHandler.execute(
                tokenAddress,
                abi.encodeWithSelector(
                    IERC20.transfer.selector,
                    address(this),
                    IERC20(tokenAddress).balanceOf(address(depositHandler))
                )
            );

            if (!success || (returnData.length != uint256(0) && !abi.decode(returnData, (bool))))
                revert BurnFailed(symbol);

            depositHandler.destroy(address(this));
        } else {
            BurnableMintableCappedERC20(tokenAddress).burn(salt);
        }
    }
```

Description

Many important operations in the system lack logging (see Listing 17 and Listing 18). This can make it difficult to observe and debug the contract, and make incident analysis difficult.

Recommendation

Short term, add event emissions to all current operations such as `burnToken`.

Long term, log all important operations. This will ensure all stakeholders can be effectively informed about everything happening in the system.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar 2, April 2, 2022.

If an individual issue is referenced, please use the following identifier:

`ABCH-{project_identifer}-{finding_id}`,

where `{project_identifier}` for this project is `AXELAR-02` and `{finding-id}` is the (severity, count) combination that appears as the prefix of the issue. For example, to cite an issue with a prefix `M3`, we would use `ABCH-AXELAR-02-M3`.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Public entrypoint

An `external` or `public` function.

Publicly-accessible function/entrypoint

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

Appendix C: Non-Security-Related Recommendations

C.1. Use `keccak` over byte-literals in constants

Listing 19. Excerpt from BurnableMintableCappedERC20

```
    // keccak256('token-frozen')
    bytes32 private constant PREFIX_TOKEN_FROZEN =
        bytes32(0x1a7261d3a36c4ce4235d10859911c9444a6963a3591ec5725b96871d9810626b);

    // keccak256('all-tokens-frozen')
    bytes32 private constant KEY_ALL_TOKENS_FROZEN =
        bytes32(0x75a31d1ce8e5f9892188befc328d3b9bd3fa5037457e881abc21f388471b8d96);
```

Listing 20. Excerpt from ERC20Permit

```
    // keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)')
    bytes32 private constant DOMAIN_TYPE_SIGNATURE_HASH =
        bytes32(0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f);

    // keccak256('Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)')
    bytes32 private constant PERMIT_SIGNATURE_HASH =
        bytes32(0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9);
```

Several contracts use `bytes32` literals for outputs of hash functions (see Listing 19 and Listing 20). Since 0.6.12, the solc compiler can evaluate `keccak256` of string literals at compile-time.

Changing these literals to calls to `keccak256` will make remove the possibility for human error, make auditing the contracts easier and make the code consistent with other contracts.

C.2. SELECTOR_* constant variables should be called COMMAND_HASH_*

Listing 21. Excerpt from AxelarGateway

```
    bytes32 internal constant SELECTOR_BURN_TOKEN = keccak256('burnToken');
    bytes32 internal constant SELECTOR_DEPLOY_TOKEN = keccak256('deployToken');
    bytes32 internal constant SELECTOR_MINT_TOKEN = keccak256('mintToken');
    bytes32 internal constant SELECTOR_APPROVE_CONTRACT_CALL = keccak256('approveContractCall');
    bytes32 internal constant SELECTOR_APPROVE_CONTRACT_CALL_WITH_MINT = keccak256('approveContractCallWithMint');
```

Listing 22. Excerpt from AxelarGatewayMultisig._execute

```
            bytes32 commandHash = keccak256(abi.encodePacked(commands[i]));

            if (commandHash == SELECTOR_DEPLOY_TOKEN) {
                if (!areValidRecentOwners) continue;

                commandSelector = AxelarGatewayMultisig.deployToken.selector;
            } else if (commandHash == SELECTOR_MINT_TOKEN) {
                if (!areValidRecentOperators && !areValidRecentOwners) continue;

                commandSelector = AxelarGatewayMultisig.mintToken.selector;
            } else if (commandHash == SELECTOR_APPROVE_CONTRACT_CALL) {
```

AxelarGateway contains definitions of hashes that can be used to denote the command to execute (see Listing 21). However, in Solidity, selector is most commonly used for the method id. Furthermore, since the `_execute` function later uses the names `command` and `commandHash` (see Listing 22), renaming these will make the code more readable and consistent.

Appendix D: Fix Review

On April 7, 2022, ABCH reviewed Axelar's fixes for the issues identified in this report.

In particular, we reviewed tag v3.1.0 with commit 4067ed6c8f.

Compared to the scope commit, this tag sets out to tackle the following problems:

- H1: `AxelarGatewayMultisig .transferOperatorship` emits an event with an incorrect value
- M2: `abi.encodePacked` contains dynamic-length data
- not possible to freeze external ERC20 tokens

We have found that the commits successfully solve the two reported issue, do not introduce any vulnerabilities, and successfully solve the third issue. We recommend Axelar address all other reported issues.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq