Axelar

Amplifier Gateway

by Ackee Blockchain

14.7.2024

ackee | blockchain security

Contents

1. Document Revisions

1.0-draft	Draft report	14.4.2024
1.0	Final report	23.4.2024
2.0	Reaudit	21.6.2024
2.0	Add client's responses	14.7.2024

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	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (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	Lead 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

Axelar Amplifier Gateway is a simplified version of Axelar Gateway designed for cross-chain messaging. It can send data payloads between different chains with the help of message relayers.

The main differences from the original Gateway are:

- token transfer functionality is removed,
- message relayers are expected/allowed to use signing keys across chains,
- messages may be signed multiple times but executed only once; Amplifier Gateway is responsible for deduplication.

Revision 1.0

Axelar engaged Ackee Blockchain to perform a security review of the Axelar with a total time donation of 5 engineering days in a period between April 8 and April 14, 2024, with Michal Převrátil as the lead auditor.

The scope of the audit was `AxelarAmplifierAuth`, `AxelarAmplifierGateway` contracts and their dependencies. The audit started on the commit `4bfaec1` [1], but the commit was updated to `8ba57a8` [2] during the audit after the agreement of both parties. Except for issues reported in this document, an issue in `AxelarAmplifierGateway` was discovered by the client on the first commit during the audit and was fixed in the second commit.

We began our review with fuzz testing using the Wake testing framework. This yielded the H1 issue. Static analysis tools, namely Wake and Slither, were used to analyze the codebase, discovering the I1 issue. We proceeded with a manual review of the codebase, focusing on the following areas:

- ensuring anti-replay protection is correctly implemented in all necessary places,

- clearing out the possibility of denial of service attacks,
- assessing the roles and permissions of all actors in the system, making sure no participant can abuse their privileges,
- checking the modified Axelar Gateway implementation may be used interchangeably with the original Gateway from the 3rd party perspective.

Our review resulted in 5 findings, ranging from Info to High severity. The most severe one may cause a temporal or even permanent denial of service to the protocol (see H1).

Ackee Blockchain recommends Axelar:

- address the unfixed findings reported in this document.

See Revision 1.0 for the system overview of the codebase.

Revision 2.0

Axelar engaged Ackee Blockchain to perform a reaudit of the Axelar with a total time donation of 5 engineering days with 1 day dedicated to fuzz testing. The review was done in a period between June 17 and June 21, 2024, with Jan Kalivoda as the lead auditor.

The given commit was: `9dae93a` [3] and the scope were the changes from previous revision. Namely the following files:

- contracts/gateway/AxelarAmplifierGateway.sol
- contracts/gateway/AxelarAmplifierGatewayProxy.sol
- contracts/gateway/BaseAmplifierGateway.sol
- contracts/governance/BaseWeightedMultisig.sol

The fuzz tests from the previous revision were updated to match the new codebase and they did not yield any new issues. During the manual review, there also were not identified any significant issues. Our review resulted in 1 informational finding. The codebase is well-structured and documented.

See Revision 2.0 for the review of the updated codebase and additional information we consider essential for the current scope.

[1] full commit hash: `4bfaec1472a10e6dc8e5f7c65eca4d1d54a661b4`

[2] full commit hash: `8ba57a8d6b0e8b7fc5ce09b5aa4f9aa0734ba5a9`

[3] full commit hash: `9dae93af0b799e536005951ddc36284132813579`

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 *Fix*.

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.

	Severity	Reported	Status
H1: `validateProof` DoS	High	1.0	Fixed
W1: Static domain separator	Warning	1.0	Acknowledged
W2: Initial signers missing zero length check	Warning	1.0	Fixed
I1: Unused errors	Info	1.0	Fixed
I2: Unused function	Info	1.0	Fixed
I3: Unused function	Info	2.0	Acknowledged

Table 2. 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.

AxelarAmplifierGateway

The contract is responsible for emitting events with cross-chain messages on the source chain and approving the messages on the destination chain. A contract integrating Axelar on the destination chain can then validate the authenticity of the received message.

BaseWeightedMultisig

`BaseWeightedMultisig` implements multisig verification, where each signer has a weight. Multiple sets of signers may be defined in a queue based on the retention period. Signer sets may be rotated by adding a new set and removing the oldest one.

AxelarAmplifierAuth

The contract is a wrapper around `BaseWeightedMultisig` that exposes signer rotation and multisig verification functionality to the public. Signer sets may only be rotated by the owner of the `AxelarAmplifierAuth` contract.

Actors

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

Gateway signers

Gateway signers are responsible for approving incoming cross-chain messages. All currently active signer sets are allowed to sign messages. Additionally, the latest signer set is allowed to rotate the signers. There is a minimum delay between the rotation of the signers.

Rotation operator

The rotation operator may trigger the signer set rotation, ignoring the minimum delay between rotations and ignoring the condition that only the latest signer set may rotate the signers. The rotation operator must still provide rotation operation signatures from a valid signer set.

5.2. Trust Model

Axelar Amplifier cross-chain messaging users have to trust Axelar Amplifier Gateway signers to approve only valid messages.

H1: `validateProof` DoS

High severity issue

Impact:	High	Likelihood:	Medium
Target:	BaseWeightedMultisig	Type:	Denial of service

Description

The function `_validateSignatures` is used to verify that the recovered addresses from message data signatures reflect the valid signers set. Not all signers from the valid signers set have to sign the message data, depending on the configured threshold.

In order to achieve signature verification efficiency, the signers as well as the signatures have to be sorted in ascending order.

Listing 1. Excerpt from [BaseWeightedMultisig]()

```
        for (uint256 i; i < signaturesLength; ++i) {
            address recoveredSigner = ECDSA.recover(messageHash, signatures[i]);
            WeightedSigner memory weightedSigner = signers[signerIndex];

            // looping through remaining signers to find a match
            for (; signerIndex < signersLength && recoveredSigner != weightedSigner.signer; ++signerIndex) {}

            // checking if we are out of signers
            if (signerIndex == signersLength) revert MalformedSignatures();

            // accumulating signatures weight
            totalWeight = totalWeight + weightedSigner.weight;

            // weight needs to reach or surpass threshold
            if (totalWeight >= weightedSigners.threshold) return;

            // increasing signers index if match was found
            ++signerIndex;
```

```
166         }
```

Due to an issue in the inner for loop (line 153), the signature verification fails on the first address of a signer who did not sign the message data. The `validateProof` function calling `_validateSignatures` is used to verify incoming cross-chain messages and rotate to a new set of signers.

The issue was discovered with fuzz testing, using the Wake testing framework.

Exploit scenario

The aforementioned issue may affect the protocol in different ways.

In the first scenario, a malicious signer may cause a full denial of service if their address is low enough (in the sorted list of signers) so that the accumulated weight of the signers with lower addresses is lower than the threshold.

In the second scenario, the issue significantly limits the protocol's performance and defeats the purpose of weighted multisig because the first `N` signers in the sorted list are the only ones who decide if the signature verification is successful or not. Any (even temporary) absence of an honest signer with a low address will cause a full denial of service.

The issue is even more severe given the fact the audited contracts are not currently upgradeable.

Recommendation

Remove the optimization caching the `weightedSigner` address and use `signers[signerIndex]` directly instead.

Fix 1.0

The issue was fixed in the commit `8ba57a8` provided during the audit. The fix follows the recommendation and removes the `weightedSigner` caching.

Go back to Findings Summary

W1: Static domain separator

Impact:	Warning	Likelihood:	N/A
Target:	BaseWeightedMultisig	Type:	Signature replay

Description

The `BaseWeightedMultisig` contract accepts `domainSeparator` as a parameter in the constructor and stores it as an immutable value.

In the case of a chain hard fork into multiple chains, the hardcoded domain separator would open a possibility for signature replay attacks on the hard-forked chains.

Listing 2. Excerpt from BaseWeightedMultisig

```
    /// @dev The domain separator for the signer proof
    /// @return The domain separator for the signer proof
    bytes32 public immutable domainSeparator;

    /// @dev The required delay between rotations
    /// @return The minimum delay between rotations
    uint256 public immutable minimumRotationDelay;

    /// @param previousSignersRetentionEpochs The number of epochs to keep
previous signers valid for signature verification
    /// @param domainSeparator_ The domain separator for the signer proof
    /// @param minimumRotationDelay_ The minimum delay between rotations
    constructor(
        uint256 previousSignersRetentionEpochs,
        bytes32 domainSeparator_,
        uint256 minimumRotationDelay_
    ) {
        previousSignersRetention = previousSignersRetentionEpochs;
        domainSeparator = domainSeparator_;
        minimumRotationDelay = minimumRotationDelay_;
    }
```

Recommendation

Consider caching the chain ID part of the domain separator as well as the domain separator itself. If it’s not applicable to use `block.chainId`, pass the chain ID part to the constructor and allow changing its value through a privileged function.

Client’s response

Acknowledged by the client.

> We aren’t too concerned about replay prevention on hard fork. The Axelar verifiers will decide which fork to support for their own node for voting on events and that’ll be the fork that will be fully supported. For the other fork, we can update the gateway to use a different domain separator, or more likely deploy a new gateway contract entirely and add it to Axelar as a new chain.
>
> — Axelar

Go back to Findings Summary

W2: Initial signers missing zero length check

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

Description

The `AxelarAmplifierAuth` constructor accepts a dynamic array with encoded initial sets of weighted signers.

Listing 3. Excerpt from AxelarAmplifierAuth

```
    constructor(
        address owner_,
        bytes32 domainSeparator_,
        uint256 previousSignersRetention_,
        uint256 minimumRotationDelay_,
        bytes[] memory initialSigners
    ) Ownable(owner_) BaseWeightedMultisig(previousSignersRetention_,
domainSeparator_, minimumRotationDelay_) {
        uint256 length = initialSigners.length;

        for (uint256 i; i < length; ++i) {
            WeightedSigners memory signers = abi.decode(initialSigners[i],
(WeightedSigners));

            _rotateSigners(signers, false);
        }
    }
```

However, it is not checked if the length of the `initialSigners` array is greater than zero. Supplying an empty array will lead to an unusable contract (permanent denial of service) because the contract cannot rotate to new signers without previous valid signers set.

Recommendation

Add a zero length check for `initialSigners` to the constructor.

Fix 2.0

The contract is no longer in the codebase.

Go back to Findings Summary

I1: Unused errors

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

Description

The `IAxelarAmplifierGateway` interface contains user-defined errors that are not used in the codebase.

See Appendix C for the list of all unused errors.

Recommendation

Remove the unused errors to improve code quality and readability.

Fix 2.0

The unused errors were removed from the interface.

Go back to Findings Summary

I2: Unused function

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

Description

The function `rotateSigners` is no longer used in the code. Its usage was replaced with the `rotateSignersWithDelay` function.

Listing 4. Excerpt from AxelarAmplifierAuth

```
    function rotateSigners(bytes calldata newSigners) external onlyOwner {
        WeightedSigners memory signers = abi.decode(newSigners,
(WeightedSigners));

        _rotateSigners(signers, false);
    }
```

Recommendation

Consider removing the function to improve code quality and readability.

Fix 2.0

The contract is no longer in the codebase. Also, the currently used `rotateSigners` function is the only one and relevant.

Go back to Findings Summary

6. Report revision 2.0

The AxelarAmplifierAuth contract was removed and the BaseAmplifierGateway contract was added. The AxelarAmplifierGateway now holds all the logic by inheriting from BaseAmplifierGateway, BaseWeightedMultisig and also Upgradable (Axelar contract for upgradeability).

I3: Unused function

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

Description

The BaseWeightedMultisig contract contains the `timeSinceRotation` that calculates the time passed since the last rotation and the function is not used in the code.

However, it could be used in the `_updateRotationTimestamp` function instead of doing the same without it.

```
function _updateRotationTimestamp(bool enforceRotationDelay) internal {
    uint256 lastRotationTimestamp_ =
_baseWeightedMultisigStorage().lastRotationTimestamp;
    uint256 currentTimestamp = block.timestamp;

    if (enforceRotationDelay && (currentTimestamp - lastRotationTimestamp_) <
minimumRotationDelay) { ①
        revert InsufficientRotationDelay(
            minimumRotationDelay,
            lastRotationTimestamp_,
            currentTimestamp - lastRotationTimestamp_
        );
    }
    ...
```

① The `(currentTimestamp - lastRotationTimestamp_)` statement can be replaced by the value returned from the function.

Recommendation

Utilize the unused function to improve code quality and readability.

Client’s response

Acknowledged by the client.

> The function was added as a convenient query for users. We could reuse it internally, although it increases the gas cost in the revert case due to reading from storage again. While not the worst situation, it still seems unnecessary since we’d be reading storage again.
>
> — Axelar

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Axelar: Amplifier Gateway, 14.7.2024.

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

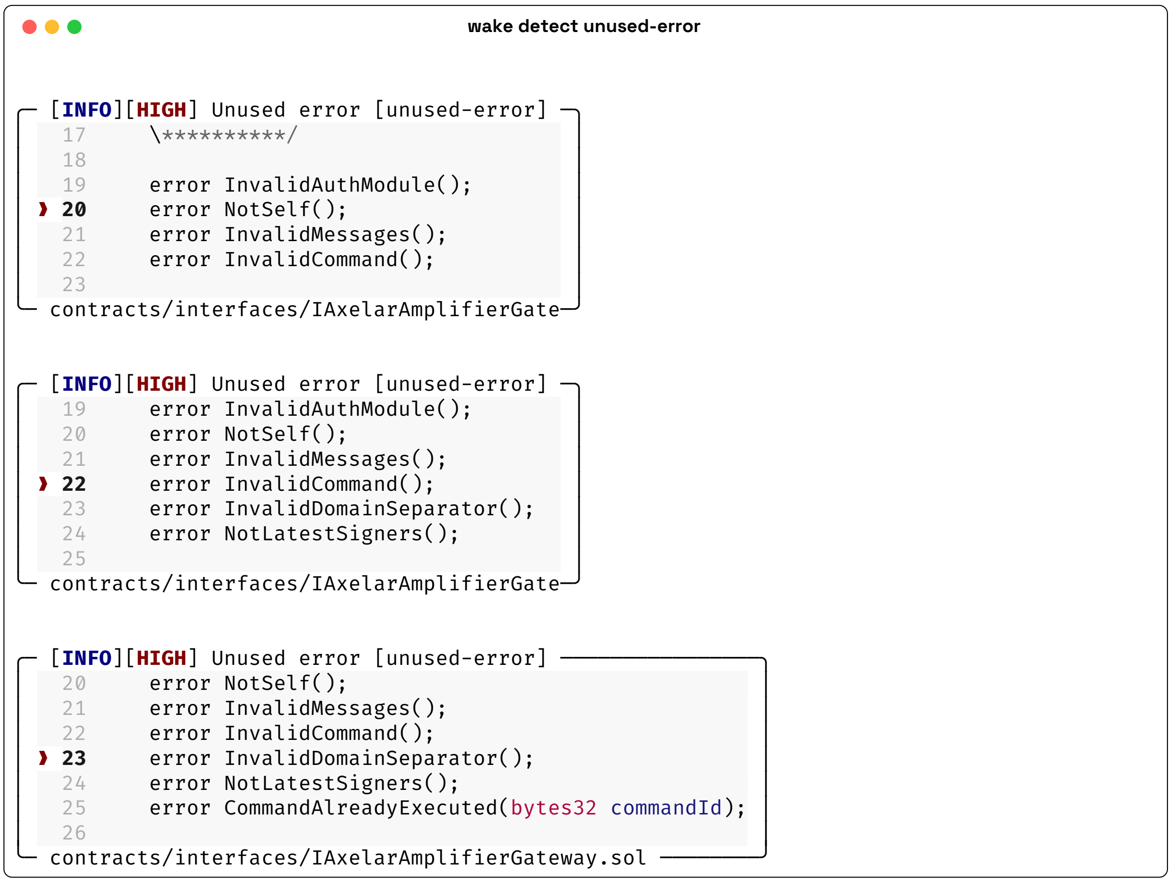
This section presents the outputs of the Wake tool.

C.1. Tests

The following part of a fuzz test prepared during the Revision 1.0 review discovered the H1 finding. The full source code of the test is available at https://github.com/Ackee-Blockchain/tests-axelar-amplifier-gateway.

```
@flow(weight=300)
def flow_approve_messages(self):
    messages = []
    for _ in range(1, 10):
        if random.random() < 0.1 and len(self.approved_messages) + len(
self.executed_messages) > 0:
            messages.append(random.choice(list(self.approved_messages |
self.executed_messages)))
        else:
            messages.append(Message(
                random_bytes(32).hex(),
                "chain2",
                str(random_address()),
                random_address(),
                bytes(random_bytes(32)),
            ))

    data_hash = keccak256(abi.encode_packed(uint8(CommandType.ApproveMessages),
abi.encode(messages)))

    i = random_int(0, len(self.signers) - 1)
    signers = self.signers[i]

    signers_subset = random.sample(signers.signers, random_int(1,
len(signers.signers)))
    signers_subset.sort(key=lambda s: s.signer)
    payload = abi.encode_packed(self.domain_separator,
keccak256(abi.encode(signers)), data_hash)

    proof = abi.encode(Proof(
        signers,
        [bytearray(Account(s.signer).sign(payload)) for s in signers_subset],
    ))

    tx = self.gw.approveMessages(messages, proof, from_=random_account(),
confirmations=0)

    if len(self.signers) - i - 1 > self.signers_retention:
        assert tx.raw_error.data == AxelarAmplifierAuth.InvalidSigners.selector
    elif sum(s.weight for s in signers_subset) < signers.threshold:
        assert tx.raw_error.data ==
AxelarAmplifierAuth.LowSignaturesWeight.selector
    else:
        assert tx.raw_error is None

        for message in messages:
            if message not in self.executed_messages:
                self.approved_messages.add(message)
```

C.2. I1 list of all unused errors


wake detect unused-error

```
╭─ [INFO][HIGH] Unused error [unused-error] ─╮
│  17      \**********/                      │
│  18                                        │
│  19      error InvalidAuthModule();        │
│❱ 20      error NotSelf();                  │
│  21      error InvalidMessages();          │
│  22      error InvalidCommand();           │
│  23                                        │
╰─ contracts/interfaces/IAxelarAmplifierGate─╯

╭─ [INFO][HIGH] Unused error [unused-error] ─╮
│  19      error InvalidAuthModule();        │
│  20      error NotSelf();                  │
│  21      error InvalidMessages();          │
│❱ 22      error InvalidCommand();           │
│  23      error InvalidDomainSeparator();   │
│  24      error NotLatestSigners();         │
│  25                                        │
╰─ contracts/interfaces/IAxelarAmplifierGate─╯

╭─ [INFO][HIGH] Unused error [unused-error] ──────────────╮
│  20      error NotSelf();                                │
│  21      error InvalidMessages();                        │
│  22      error InvalidCommand();                         │
│❱ 23      error InvalidDomainSeparator();                 │
│  24      error NotLatestSigners();                       │
│  25      error CommandAlreadyExecuted(bytes32 commandId);│
│  26                                                      │
╰─ contracts/interfaces/IAxelarAmplifierGateway.sol ──────╯
```

ackee | blockchain security

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain