Axelar Amplifier Gateway Security Assessment

Axelar

Version 1.01 – July 26, 2024

Prepared By
Parnian Alimi
Kevin Henry

Prepared For
Christian Gorenflo
Milap Sheth

1 Executive Summary

Synopsis

During the summer of 2024, Axelar engaged NCC Group's Cryptography Services practice to conduct a security assessment of revisions to the *axelar-amplifier* gateway and its associated smart contracts. NCC Group has previously reviewed *axelar-amplifier*, and this engagement is limited to recent updates to the application, alongside the corresponding smart contracts. The application allows developers to interact across multiple chains within the Axelar network without incurring the additional cost of connecting to each chain individually. Hence, the resources are "amplified" by leveraging *axelar-amplifier*. The review was delivered by 2 consultants with a total effort of 14 person-days.

In July 2024, a retest was performed, and as a result all findings are considered "Fixed". Additionally, fixes were also provided for several non-security related comments provided in the report.

Scope

The *axelar-amplifier* was previously reviewed by NCC Group. This follow-on engagement targeted commit `5d54280` of the repository located at https://github.com/axelarnetwork/axelar-amplifier/, and was guided by the following:

- Additions and updates relating to `multisig`, `multisig-prover`, and `voting-verifier`.
- Updates to message ID format.
- No changes to the core repository were in scope.

Additionally, the corresponding smart contracts for the Amplifier Gateway were in scope targeting commit `9dae93a` of the repository located at https://github.com/axelarnetwork/axelar-gmp-sdk-solidity, focusing on the Gateway contract and its associated dependencies. A Notion document detailing the design was provided by Axelar.

Limitations

While good coverage of the in-scope code was achieved, the review only covered a portion of the complete codebase. No coverage of code outside of the Amplifier Gateway was targeted.

Key Findings

The review resulted in a five relatively low impact findings, including:

- Finding "Weighted MultiSig Validity Dependent on Signature Ordering" details implicit assumptions about multisig proof's verification process which may not be enforced.
- Finding "Gateway Contract can Be Instantiated with No Operator" describes a potential constraint violation when bootstrapping the Gateway contract for the first time.
- Finding "Message IDs Unsafely Cast from u64" highlights a small number of non-checked integer conversions.

The section Engagement Notes also details several minor notes and comments resulting from the review. Upon retesting, all findings are considered "Fixed".

Strategic Recommendations

- Ensure that any non-automated dependency management processes are well-defined. Despite some monitoring, the number of stale and vulnerable dependencies appears to have grown since the previous review.
- Ensure that validity criteria (e.g., multisig validity or batch validity) are documented and enforced at both message creation and message handling.

2 Dashboard

Target Data

Name	Axelar Amplifier Gateway
Type	Blockchain Application
Platforms	Rust, Solidity
Environment	Local

Engagement Data

Type	Security Assessment
Method	Code-assisted
Dates	2024-05-29 to 2024-06-07
Consultants	2
Level of Effort	15 person-days

Targets

Amplifier Gateway	https://github.com/axelarnetwork/axelar-amplifier/ commit `5d54280` Targeting changes to the gateway to support weighted multisig.
Solidity GMP SDK	https://github.com/axelarnetwork/axelar-gmp-sdk-solidity commmit `9dae93a` Targeting the gateway contract and its dependencies.

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	0	
Low issues	2	🟨🟨
Informational issues	3	⬜⬜⬜
Total issues	**5**	

Category Breakdown

Configuration	1	🟨
Data Validation	3	🟨⬜⬜
Patching	1	⬜

Component Breakdown

axelar-amplifier	3	🟨🟨⬜
axelar-gmp-sdk-solidity	2	⬜⬜

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Message IDs Unsafely Cast from u64	Fixed	LXK	Low
Invalid Configuration Leads to Unusable MultiSig Contract	Fixed	TPM	Low
Vulnerable and Outdated Dependencies	Fixed	WF9	Info
Gateway Contract can Be Instantiated with No Operator	Fixed	LQ2	Info
Weighted MultiSig Validity Dependent on Signature Ordering	Fixed	MX6	Info

4 Finding Details

Low Message IDs Unsafely Cast from u64

Overall Risk	Low	**Finding ID**	NCC-E010021b-LXK
Impact	Low	**Component**	axelar-amplifier
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Unchecked conversion between primitives may result in unintended behavior. In particular, conversion from a large numerical primitive (`u64`) to a smaller primitive (`u32`) could result in truncation, leading to duplicate message IDs.

Description

Within the `voting-verifier` , three instances of potentially unsafe conversion to `u32` were observed. For example, the following function computes a message ID from its inputs:

```
fn message_id(id: &str, index: u64, msg_id_format: &MessageIdFormat) ->
↪ nonempty::String {
    let tx_hash = Keccak256::digest(id.as_bytes()).into();
    match msg_id_format {
        MessageIdFormat::HexTxHashAndEventIndex => HexTxHashAndEventIndex {
            tx_hash,
            event_index: index as u32,
        }
        .to_string()
        .parse()
        .unwrap(),
        MessageIdFormat::Base58TxDigestAndEventIndex => Base58TxDigestAndEventIndex {
            tx_digest: tx_hash,
            event_index: index as u32,
        }
        .to_string()
        .parse()
        .unwrap(),
    }
}
```

Figure 1: contracts/voting-verifier/src/contract.rs

Per the Rust documentation:

> Casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate

Therefore, it is possible to force a collision in message IDs by using an event index larger than `u32::MAX` . The same behavior is present in `message()` in *contracts/voting-verifier/src/query.rs*. Comments elsewhere in the code suggest this may be a known issue, but that it is inherited from a dependency:

```
// it's parsed into u64 instead of u32 (https://github.com/axelarnetwork/axelar-amplifier/
↪ blob/bf0b3049c83e540989c7dad1c609c7e2ef6ed2e5/contracts/voting-verifier/src/
↪ events.rs#L162)
// here in order to match the message type defined in the nexus module. Changing nexus to
↪ use u32 instead is not worth the effort.
fn parse_message_id(message_id: &str) -> Result<(nonempty::Vec<u8>, u64), ContractError> {
    let id = HexTxHashAndEventIndex::from_str(message_id)
        .change_context(ContractError::InvalidMessageId(message_id.into()))?;
    let tx_id = nonempty::Vec::<u8>::try_from(id.tx_hash.to_vec())
        .change_context(ContractError::InvalidMessageId(message_id.into()))?;

    Ok((tx_id, id.event_index.into()))
}
```

Figure 2: contracts/nexus-gateway/src/nexus.rs

For safety, a check could be added to ensure that the index is in the correct range prior to casting it, or the parameter could be swapped to a `u32` for the caller to decide how to handle overflow. The only non-test usage of the `message_id()` function is the `messages()` function in the same file, which accepts a `len` parameter as a `u64` and creates distinct message IDs for `len` messages. It appears that this instance could be switched to `u32` without any additional code updates to partially mitigate the issue.

On the receiving side, the event index is parsed from a string as a `u32` and will throw an error if overflow occurs.

Recommendation

Either align the types used for the event index to uniformly use `u32` or add validation logic to ensure that truncation is explicitly handled during conversion.

Location

- *contracts/voting-verifier/src/contract.rs*
- *contracts/voting-verifier/src/query.rs*

Retest Results

2024-07-19 – Fixed

Commit `f6096f8` addresses this issue by updating the `message_id()` function to take a `u32` and to perform this conversion when calling the function to using `try_from()` so that an exception will be triggered. Additionally, the Clippy rule `cast_possible_truncation = "deny"` has been adopted to prevent similar issues in the future.

These changes effectively implement the recommendations above and this finding is considered “Fixed”.

Low

Invalid Configuration Leads to Unusable MultiSig Contract

Overall Risk	Low	**Finding ID**	NCC-E010021b-TPM
Impact	Low	**Component**	axelar-amplifier
Exploitability	Low	**Category**	Configuration
		Status	Fixed

Impact

A zero block expiry configuration results in signing sessions that will always fail.

Description

When a signing session starts, its expiry is set to the current block height plus a block expiry duration that is set by the configuration. This session is then stored in the `SIGNING_SESSIONS` map:

```rust
pub fn start_signing_session(
    deps: DepsMut,
    env: Env,
    verifier_set_id: String,
    msg: MsgToSign,
    chain_name: ChainName,
    sig_verifier: Option<Addr>,
) -> Result<Response, ContractError> {
    let config = CONFIG.load(deps.storage)?;
    let verifier_set = get_verifier_set(deps.storage, &verifier_set_id)?;

    let session_id = SIGNING_SESSION_COUNTER.update(
        deps.storage,
        |mut counter| -> Result<Uint64, ContractError> {
            counter = counter
                .checked_add(Uint64::one())
                .map_err(ContractError::Overflow)?;
            Ok(counter)
        },
    )?;

    let expires_at = env
        .block
        .height
        .checked_add(config.block_expiry)
        .ok_or_else(|| {
            OverflowError::new(
                OverflowOperation::Add,
                env.block.height,
                config.block_expiry,
            )
        })?;
```

Figure 3: contracts/multisig/src/contract/execute.rs

The session's `expires_at` will be set to the current block height if `config.block_expiry` is zero.

Once the session starts, every submitted signature is processed in the `submit_signature()` function, which calls `validate_session_signature()` to ensure that the current block height is before the session's expiry block:

```rust
pub fn validate_session_signature(
    session: &SigningSession,
    signer: &Addr,
    signature: &Signature,
    pub_key: &PublicKey,
    block_height: u64,
    sig_verifier: Option<SignatureVerifier>,
) -> Result<(), ContractError> {
    if session.expires_at < block_height {
        return Err(ContractError::SigningSessionClosed {
            session_id: session.id,
        });
    }
```

Figure 4: contracts/multisig/src/signing.rs

If session's `expires_at` is the block height when it started, all signature submissions will fail.

Recommendation

Consider whether a minimum should be enforced on the `block_expiry` configuration when the MultiSig contract is instantiated:

```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
    deps: DepsMut,
    _env: Env,
    _info: MessageInfo,
    msg: InstantiateMsg,
) -> Result<Response, axelar_wasm_std::ContractError> {
    cw2::set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;

    let config = Config {
        governance: deps.api.addr_validate(&msg.governance_address)?,
        rewards_contract: deps.api.addr_validate(&msg.rewards_address)?,
        block_expiry: msg.block_expiry,
    };
```

Figure 5: contracts/multisig/src/contract.rs

Location

- *contracts/multisig/src/contract.rs*

Retest Results

2024-07-19 – Fixed

As described in the Client Response, commit `62a8c6f` now uses `nonempty::Uint64` to ensure that the `block_expiry` cannot be set to 0, and as an additional safety measure any migration from a configuration where this value is set to 0 will be bumped to 10 after migration.

This change effectively implements the recommendations above, and this finding is consider "Fixed".

Client Response

`InstantiateMsg` for the contract and the Config struct that keeps the `block_expiry` value now uses our `nonempty::Uint64` value that doesn't allow the 0 value. So, it's impossible to set the expiry to 0. Additionally, the contract migration sets the block expiry to 10 instead of 0 if it finds it (we have not deployed the contract with 0 anywhere, but just in case)

Info

Vulnerable and Outdated Dependencies

Overall Risk	Informational	**Finding ID**	NCC-E010021b-WF9
Impact	Low	**Component**	axelar-amplifier
Exploitability	Low	**Category**	Patching
		Status	Fixed

Impact

Attackers may use public security advisories to identify and exploit vulnerabilities within the application. Even if vulnerabilities are not exploitable, the presence of outdated or vulnerable dependencies may affect the reputation and the perceived security posture of the application.

Description

During a previous engagement, a similar finding was presented to document vulnerable and outdated dependencies. Axelar has configured a GitHub Action to forward security advisories to a Slack channel for monitoring, however, access to repository settings, such as Dependabot, were not available for formal review. For completeness, `cargo audit` results are documented here.

The following crates result in `cargo audit` **vulnerabilities**:

- `h2 0.3.24`
- `mio 0.8.8`
- `quinn-proto 0.10.4`
- `rsa 0.8.2`
- `rustls 0.19.1`, `0.21.7`
- `shlex 1.2.0`
- `tungstenite 0.20.0`
- `webpki 0.21.4`
- `whoami 1.4.1`

The following crates result in `cargo audit` *warnings*:

- `difference 2.0.0`
- `dirs 5.0.1`
- `yaml-rust 0.4.5`
- `ahash 0.7.6`, `0.8.3`
- `elliptic-curve 0.13.5`
- `gateway-api 0.1.0`
- `move-bytecode-verifier 0.1.0`
- `move-command-line-common 0.1.0`
- `move-coverage 0.1.0`
- `move-ir-to-bytecode 0.1.0`
- `move-symbol-pool 0.1.0`
- `rustls-webpki 0.101.5`

Of particular note is the inclusion of `rustls`, which is a recent high impact vulnerability on a direct dependency (via a feature of `ethers`) of *axelar-amplifier*.

Recommendation

- Ensure that vulnerable dependency notifications are working and that appropriate processes are in place to respond to them.
- Consider using a tool such as `cargo deny` to enforce an allow list of vulnerable dependencies. This would enforce a proactive action to continue in the presence of a vulnerable crate, rather than a reactive action to respond to a vulnerable crate.

Location

.github/workflows/dependabot-vulns-to-slack.yaml

Retest Results

2024-07-19 – Fixed

Commit `ada2dbe` updates several dependencies, including `ethers`, `tokio` and `sui` related packages.

This finding is informational, and the identified vulnerable crates do not appear to affect Amplifier. While the recommendation to continue actively monitoring dependencies remains, this finding is considered “Fixed” for the purpose of retesting.

Dependency management is a continuous process, whereas a security review represents a point-in-time evaluation. While this finding is considered “Fixed”, the retest uncovered the following new vulnerable packages that have been reported *after* the commit reviewed here:

- `curve25519-dalek 3.2.0`: https://rustsec.org/advisories/RUSTSEC-2024-0344
- `zerovec 0.10.2`: https://rustsec.org/advisories/RUSTSEC-2024-0347
- `zerovec-derive 0.10.2`: https://rustsec.org/advisories/RUSTSEC-2024-0346

Axelar has confirmed that they were made aware of these via their existing monitoring processes and will schedule fixes as appropriate.

Client Response

We’ve updated the dependencies to the extend that’s possible. The remaining flagged vulnerabilities are either still not patched or we rely on a specific version of a dependency. None of them affects us though

Info

Gateway Contract can Be Instantiated with No Operator

Overall Risk	Informational	**Finding ID**	NCC-E010021b-LQ2
Impact	None	**Component**	axelar-gmp-sdk-solidity
Exploitability	None	**Category**	Data Validation
		Status	Fixed

Impact

The Amplifier Gateway contract can be instantiated without an operator address, which may leave it in an undefined state.

Description

The Amplifier Gateway provides a `_setup()` function which is used to bootstrap the contract, or to reconfigure it after an upgrade.

```
    /**
     * @notice Internal function to set up the contract with initial data. This function is also called during upgrades.
     * @dev The setup data consists of an optional new operator, and a list of signers to rotate too.
     * @param data Initialization data for the contract
     * @dev This function should be implemented in derived contracts.
     */
    function _setup(bytes calldata data) internal override {
        (address operator_, WeightedSigners[] memory signers) = abi.decode(data, (address, WeightedSigners[]));

        if (operator_ != address(0)) {
            _transferOperatorship(operator_);
        }

        for (uint256 i = 0; i < signers.length; i++) {
            _rotateSigners(signers[i], false);
        }
    }
```

Figure 6: contracts/gateway/AxelarAmplifierGateway.sol

The highlighted portion of the code is used to ensure that the provided operator address is well-defined. However, the function does not revert if the provided operator is invalid, it simply leaves the original operator in place. Therefore, if an operator is provided with `address(0)` during the initial call to `_setup()`, then the contract will remain configured with `_operator = address(0)`. All other operator-related checks in the code suggest that this case should not occur and should therefore be prevented. The check could be refactored to `revert InvalidOperator()` in the case that the provided operator is `address(0)`, as performed elsewhere:

```
    /**
     * @notice Transfer the operatorship to a new address.
     * @param newOperator The address of the new operator.
     */
    function transferOperatorship(address newOperator) external onlyOperatorOrOwner {
        if (newOperator == address(0)) revert InvalidOperator();

        _transferOperatorship(newOperator);
    }
```

Figure 7: contracts/gateway/AxelarAmplifierGateway.sol

If the implemented behavior is intentional (e.g., to allow a 0 input to preserve the existing operator), then consider adding an additional check to ensure that the operator is already set to a non-zero address before proceeding.

Recommendation

Consider reverting with an error if `_setup()` would leave the operator address set to `address(0)`.

Location

contracts/gateway/AxelarAmplifierGateway.sol

Retest Results

2024-07-19 – Fixed

Commit `1db893b` adds additional comments in the code clarifying that is intentional that the contract can be instantiated with no operator, and the check to prevent transferring ownership to `address(0)` has been removed.

Based on the intended behavior described by Axelar, the revised approach now consistently enforces the intended behavior, and this finding is considered “Fixed”.

Furthermore, as the identified behavior was intentional, this finding has been re-rated from its original rating of “low” to “informational”.

Client Response

This is intentional. The gateway operator is optional. So it can be not set if preferred for that deployment. Note that the gateway owner can set it again via an upgrade as well.

We’ll add a comment to clarify that it’s optional. We’ll also update `transferOperatorship` to allow `address(0)` to allow explicitly removing an already set operator.

Info

Weighted MultiSig Validity Dependent on Signature Ordering

Overall Risk	Informational	**Finding ID**	NCC-E010021b-MX6
Impact	None	**Component**	axelar-gmp-sdk-solidity
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

MultiSig validation returns early once the threshold is met and reverts if a single error is encountered. Therefore, an invalid signature may or may not cause proof validation to fail depending on its location (address) within the signer list.

Description

The function `_validateSignatures()` takes a list of weighted signers, sorted by address, and determines if a sufficient threshold of signers, based on weight, have approved the message. The core logic follows:

```
207         for (uint256 i; i < signaturesLength; ++i) {
208             address recoveredSigner = ECDSA.recover(messageHash, signatures[i]);
209 
210             // looping through remaining signers to find a match
211             for (; signerIndex < signersLength && recoveredSigner !=
                ↳ signers[signerIndex].signer; ++signerIndex) {}
212 
213             // checking if we are out of signers
214             if (signerIndex == signersLength) revert MalformedSignatures();
215 
216             // accumulating signatures weight
217             totalWeight = totalWeight + signers[signerIndex].weight;
218 
219             // weight needs to reach or surpass threshold
220             if (totalWeight >= weightedSigners.threshold) return;
221 
222             // increasing signers index if match was found
223             ++signerIndex;
224         }
```

Figure 8: contracts/governance/BaseWeightedMultisig.sol

On line 208, the address of the signer is recovered and validated. **The call to `ECDSA.recover()` will revert if the resulting signature is invalid.** On line 220, the function returns early if the total weight of the signers thus far is greater than the threshold.

The above function is exclusively called by `_validateProof()`, which provides some example mock data in its documentation:

```
    /**
     * @notice This function takes dataHash and proof data and reverts if proof is invalid
     * @param dataHash The hash of the message that was signed
     * @param proof The multisig proof data
     * @return isLatestSigners True if the proof is from the latest signer set
     * @dev The proof data should have signers, weights, threshold and signatures encoded
     *      The signers and signatures should be sorted by signer address in ascending order
     *      Example: abi.encode([0x11..., 0x22..., 0x33...], [1, 1, 1], 2, [signature1, signature3])
     */
    function _validateProof(bytes32 dataHash, Proof calldata proof) internal view returns (bool isLatestSigners) {
```

Figure 9: contracts/governance/BaseWeightedMultisig.sol

The above assumes that the set of signatures is produced via `optimize_signatures()` in *axelar-amplifier*, which produces the minimal set of signatures that will exceed the threshold. However, a maliciously constructed proof may contain a non-optimized set of signers. In the above 2-of-3 example a set of consisting of `{valid sig1, valid sig2, invalid sig3}` would validate, but a similar set `{valid sig1, invalid sig2, valid sig3}` would not validate.

The above validation process optimizes for gas cost in the expected positive use case but may be considered unintuitive based on the actual validation behavior. While there is no explicit attack enabled by the above behavior, it may be beneficial to clearly document the intended/implemented behavior so that unexpected situations do not arise. The following questions should be clear to a user of the contract:

1. Is a multisig always valid if at least the weight of the valid signatures surpasses the threshold?
2. Is a multisig always invalid if there is an invalid signature present?
3. Is a multisig invalid if it is not consistent with the output of `optimize_signatures()`?

It follows that tests should be in place to detect deviations from the expected behavior.

Recommendation

Clearly document the intended/implemented behavior of multisig proof validation so that unexpected situations do not arise.

Location

- *axelar-gmp-sdk-solidity/contracts/governance/BaseWeightedMultisig.sol*
- *axelar-amplifier/contracts/multisig/src/multisig.rs*

Retest Results

2024-07-19 – Fixed

Commit `1db893b` adds a comment clarifying the intended behavior and now correctly enforces it during validation:

> The proof is only valid if the signers weight crosses the threshold and there are no redundant signatures

Furthermore, the function `_validateSignatures()` now returns an error if there are redundant signatures present:

```
    if (totalWeight >= weightedSigners.threshold) {
        // validate the proof if there are no redundant signatures
        if (i + 1 == signaturesLength) return;

        revert RedundantSignaturesProvided(i + 1, signaturesLength);
    }
```

These changes clarify and enforce the expected behavior, and this finding is considered "Fixed".

Note that this finding was originally rated "medium" but has been re-rated as "informational" as there was no identifiable attack enabled by the previous behavior.

Client Response

The intended behaviour is the more intuitive one, validate if sufficient valid signatures crossing the threshold were submitted. The requirement on ordering is merely for an optimization.

As a good practice, we can add a check to revert if there are any leftover signatures when the threshold is crossed to make it strict.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Engagement Notes

This section consists of notes and observations that do not represent security issues or that did not warrant a standalone finding.

Amplifier

Bad Link in Amplfier Documentation

The README.md file in *doc/* links to http://localhost:3000/contributing/documentation.html at `localhost:3000` instead of the GitHub-hosted documentation under the *Contributing* header.

Retest Results: Commit `25a65aa` corrects the bad link.

Duplicate Code for Governance Address Checks

Several of the contracts within *axelar-amplifier* contain functions that may only be called by the governance address. Such operations are gated by the function `require_governance()`:

```rust
// TODO: this type of function exists in many contracts. Would be better to implement this
// in one place, and then just include it
pub fn require_governance(deps: &DepsMut, sender: Addr) -> Result<(), ContractError> {
    let config = CONFIG.load(deps.storage)?;
    if config.governance != sender {
        return Err(ContractError::Unauthorized);
    }
    Ok(())
}
```

Figure 10: contracts/voting-verifier/src/execute.rs

The TODO in the code acknowledges that this function is duplicated across all relevant contracts, which may lead to divergent behavior in the future. In particular, it was observed that:

1. The implementation in *multisig-prover/src/execute.rs* is structured differently, although logically equivalent.
2. Only the implementation in *voting-verifier/src/execute.rs* contains a test to validate its behavior.

While no vulnerability exists, the current approach may be considered fragile, as changes to one of the functions will not be reflected globally, and test coverage may not catch unintended changes in behavior. Therefore, it is recommended to address the open TODO item and to ensure that the unified approach is adequately tested.

A similar comment applies to the function `require_admin()`, which contains two distinct (but logically equivalent) implementations with no tests.

Retest Results:

The following response was provided:

> Contracts now use an overhauled permission check system that is based on a derive macro and enforces setting permissions on every message variant. So address checks follow now a unified behaviour. We are currently in the process of migrating all contracts to this new model.

Commits `87ef101` and `a75b744` introduce the macro mentioned above, which include calls to the new functions `permission_control::set_admin()` and `permission_control::set_governance()` as appropriate at instantiation, as well as usage of the `ensure_permission!()` macro to enforce checks where relevant. This approach ensures permissions can be defined, applied, and tested consistently across contracts.

Documentation does not Match the Implementation

The `WeightedSigner` structure has a `signer` and a `weight` parameter, which does not match its documentation:

```
/**
 * @notice This struct represents the weighted signers payload
 * @param signers The list of signers
 * @param weights The list of weights
 * @param threshold The threshold for the signers
 */
struct WeightedSigner {
    address signer;
    uint128 weight;
}
```

Figure 11: contracts/multisig-prover/src/encoding/abi/solidity/WeightedMultisigTypes.sol

Retest Results: The referenced contract is no longer present in the repository, thereby negating this note.

Solidity GMP SDK

Usage of Slither Static Analyzer

It was observed that the SDK includes a GitHub action to automatically run Slither, a static analysis tool for smart contracts. Within the reviewed code, a single Slither hit is suppressed in the function `executeCalls()` *contracts/governance/InterchainMultisig.sol*. This suppresses a warning related to reentrancy and appears to be a safe exception. The continued use of Slither is recommended

Minor Optimization During Signer Validation

The function `_validateSigners()` takes a list of signers and determines if it is well-formed, including a check that the weights of the signers are sufficient to meet the threshold. The final check in the function is:

```
    uint128 threshold = weightedSigners.threshold;
    if (threshold == 0 || totalWeight < threshold) revert InvalidThreshold();
```

While the total weight of the signers must be computed by iterating over the complete list, the value of `threshold` is provided as part of the `weightedSigners` parameter. Therefore, the check for `threshold == 0` could be performed prior to iterating over the signers.

Duplicate Code for Message Validation

It was observed that the `_isMessageApproved()` function implements logic to determine if a given message is present in the list of approved messages. Immediately afterwards, the function `_validateMessage()` performs an identical check directly, rather than leveraging the `_isMessageApproved()` function. There are no issues with the current approach, but there is an implied dependency in that both of these pieces of code must be updated in tandem if the validity criteria are changed. One could instead call `isMessageApproved()` to avoid the duplication, and potentially rely on an optimizing compiler to inline the function call for the exact same gas cost.

Potentially Inaccurate Code Comments

The documentation for the function `_validateProof()` suggests that the proof data is of the form `abi.encode([0x11..., 0x22..., 0x33...], [1, 1, 1], 2, [signature1, signature3])`. While this is conceptually correct, the `WeightedSigners` struct contains an array of

`WeightedSigner` objects, which in turn contain the `signer` and its `weight` (i.e., the `signer` and `weight` are packed in the same array, not separately as depicted). The `WeightedSigners` also contains a `nonce` not shown in the above. While the meaning is generally clear, the explicit notation using `abi.encode(...)` may mislead a reader into assuming the stated format would be parsed correctly.

7 Contact Info

The team from NCC Group has the following primary members:

- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Kevin Henry – Consultant
  kevin.henry@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Axelar has the following primary members:

- Christian Gorenflo
  christian@interoplabs.io
- Milap Sheth
  milap@interoplabs.io
- Liana Spano
  liana@interoplabs.io