

LAYERZERO · SECURITY REVIEW

LayerZero

OneSig on Canton.

A security review of OneSig on Canton.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
LayerZero	OneSig on Canton	Roberto Reigada, Felipe Gomez	July 1 – September 4, 2026	September 4, 2026

§ 00

Contents.

54 pages: overview, confidence ranking, scope, 32 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
§ 04	Findings	P. 06
	Main Review · 24 findings	P. 08
	Remediation Review 1 · 7 findings	P. 41
	Remediation Review 3 · 1 finding	P. 52
§ 05	Disclaimer	P. 53
§ 06	About Guardian	P. 54

§ 01

Overview.

Guardian reviewed OneSig on Canton across 4 review rounds. The final report includes 5 Medium, 23 Low, 4 Informational findings.

AUDIT FIRM	REPOSITORY
Guardian	https://github.com/LayerZero-Labs/audit-external
CLIENT	METHODOLOGY
LayerZero	Smart contract · Manual · Main Review · Remediation Review 1 · Remediation Review 2 · Remediation Review 3
PROJECT	MAIN REVIEW COMMIT
OneSig on Canton	2117dd3a11f4
PREPARED BY	REMEDATION REVIEW 1 COMMIT
Roberto Reigada, Felipe Gomez	662a11b7e41a
AUDIT WINDOW	REMEDATION REVIEW 2 COMMIT
July 1 – September 4, 2026	b921ee00fb93
FINAL REPORT	REMEDATION REVIEW 3 COMMIT
September 4, 2026	c544d8413d6a

1.1 Confidence Ranking

5 / 5

Very High Confidence

Guardian assigns a Confidence Ranking of 5 out of 5, combining finding severity, remediation outcome, and the reviewer’s assessment of latent risk. The full scale is on the next page.

1	2	3	4	5
VERY LOW CONFIDENCE	LOW CONFIDENCE	MODERATE CONFIDENCE	HIGH CONFIDENCE	VERY HIGH CONFIDENCE

1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED
Critical	0	0	0	0	0
High	0	0	0	0	0
Medium	5	0	4	0	1
Low	23	0	15	0	8
Informational	4	0	3	0	1
All severities	32	0	22	10	10

50% OVERVIEWCONFIDENTIALP. 03 / 54

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence ASSIGNED	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed OneSig on Canton in <https://github.com/LayerZero-Labs/audit-external>.

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH

Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.

IMPACT · MEDIUM

Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.

IMPACT · LOW

Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.

LIKELIHOOD · HIGH

Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.

LIKELIHOOD · MEDIUM

Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.

LIKELIHOOD · LOW

Unlikely to ever occur in production.

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW					
M-01	CallContext collisions swap signed payloads	Signatures	MEDIUM	RESOLVED	p. 8
M-02	Admin bypasses the OneSig quorum	Access Control	MEDIUM	ACKNOWLEDGED	p. 11
L-01	Wrong-ID RBAC authorizes global updates	Access Control	LOW	RESOLVED	p. 12
L-02	Unsigned config replacement CIDs	Validation	LOW	ACKNOWLEDGED	p. 14
L-03	Unsigned fee config can lower request fees	Validation	LOW	ACKNOWLEDGED	p. 16
L-04	Rate-limiter state can be redirected	Validation	LOW	ACKNOWLEDGED	p. 17
L-05	Duplicate OApp config can receive updates	Validation	LOW	ACKNOWLEDGED	p. 19
L-06	Duplicate IOApp can receive OApp requests	Validation	LOW	ACKNOWLEDGED	p. 21
L-07	Duplicate RBAC can receive role updates	Validation	LOW	ACKNOWLEDGED	p. 22
L-08	Stale RBAC witness authorizes updates	Access Control	LOW	ACKNOWLEDGED	p. 23
L-09	Non-bytes32 proofs verify on Canton	Validation	LOW	RESOLVED	p. 24
L-10	New executors may miss the anchor	DoS	LOW	ACKNOWLEDGED	p. 25
L-11	Signer policy changes revive roots	Signatures	LOW	ACKNOWLEDGED	p. 26
L-12	Global mode drops EID usage	Logical Error	LOW	RESOLVED	p. 27
L-13	Noncanonical roots reject proofs	Validation	LOW	RESOLVED	p. 28
L-14	Proof siblings are not canonicalized	Validation	LOW	RESOLVED	p. 29
L-15	Mixed config updates use stale CIDs	DoS	LOW	RESOLVED	p. 30
L-16	Duplicate configs can replay roots	Signatures	LOW	ACKNOWLEDGED	p. 32
L-17	Malformed seed bricks all executes	Validation	LOW	RESOLVED	p. 34
L-18	Leaf lengthPrefix counts codepoints	Compatibility	LOW	ACKNOWLEDGED	p. 35
L-19	No upper cap on signers bricks config	DoS	LOW	ACKNOWLEDGED	p. 36
L-20	OneSig anchor lacks ensure on oneSigId	Validation	LOW	RESOLVED	p. 37
L-21	Canton rejects the shared EVM signature format	Signatures	LOW	ACKNOWLEDGED	p. 38
I-01	Ledger-time updates can hit duplicate configs	Validation	INFORMATIONAL	ACKNOWLEDGED	p. 40

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
REMEDATION REVIEW 1					
M-01	Cross-instance ledger-time auth bypass	Access Control	MEDIUM	ACKNOWLEDGED	p. 41
M-02	Old callbacks erase newer rate-limit usage	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 43
M-03	Old callbacks erase post-reset rate-limit usage	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 45
L-01	OpSetMinFee same-handler config substitution	Access Control	LOW	ACKNOWLEDGED	p. 47
L-02	No upper bound on validity period	Validation	LOW	ACKNOWLEDGED	p. 49
I-01	Misleading OAppldMismatch error comment	Documentation	INFORMATIONAL	ACKNOWLEDGED	p. 50
I-02	OneSigConfig observers see full config	Trust Assumptions	INFORMATIONAL	ACKNOWLEDGED	p. 51
REMEDATION REVIEW 3					
I-01	Signing zero fails a mandatory positive transfer	Compatibility	INFORMATIONAL	RESOLVED	p. 52

MEDIUM **M-01**

RESOLVED

CallContext collisions swap signed payloads

CATEGORY Signatures

LOCATION contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigEncoding.daml

REVIEW Main Review

DESCRIPTION

`encodeCallContext` does not uniquely encode a `CallContext`. It encodes each nested map entry as the encoded key followed immediately by the encoded child value. The key is length-prefixed, but the child subtree is not. The map also does not include an entry count. Since `CallContext` is a six-level nested map, two different trees can produce the same encoded text.

This matters because `OpCreateRequest` and `OpCreate0AppRequest` sign `lengthPrefix (encodeCallContext callContext)` and `lengthPrefix (encodeCallContext callbackContext)`. The outer length prefix only bounds the full encoded context. It does not prove that the encoded context has one unique structured interpretation. Therefore, a quorum can sign one request context while an executor submits a different `CallContext` value with the same encoding and the same OneSig leaf.

A minimal collision shows the problem. Assume two different function-argument maps inside the same surrounding `CallContext`:

```
```plain text
```

Context A:

```
function "1:a4:text" has no args
```

```
function "a" has arg "text" = AV_Text "a"
```

Context B:

```
function "1:a4:text" has arg "a" = AV_Text "text"
```

```
function "a" has no args
```

```
```\n`
```

These maps are not equal. However, both encode to the same text:

```
`plain text
```

```
9:1:a4:text1:a4:text4:text1:a
```

```
`\n`
```

The collision happens because the encoder does not mark where the empty child map for function "1:a4:text" ends or where the next sibling function entry begins. In Context A, the bytes after 9:1:a4:text start a new function entry named "a". In Context B, those same bytes are interpreted as an argument named "a" inside function "1:a4:text". The

M-01 CallContext collisions swap signed payloads

CONTINUED

signed bytes are identical, so the signature is still valid even though the submitted structured context is different.

The callback context is a realistic sink for this bug. The request factory validates `callContext` as a single request call, but it only size-checks `callbackContext`, stores it on the `Request` and later forwards the stored value into `PendingCallback`. Callback code such as `parseCallbackContextToFnArgs` then consumes the submitted Daml structure, not a re-decoded canonical byte string. A malicious executor can therefore get a quorum signature over one structured callback context while causing a different structured callback context with the same encoded bytes to be stored for downstream callback handling. This does not require forging signatures. The executor relies on `Context A != Context B` while `encodeCallContext Context A == encodeCallContext Context B`. The signatures verify because OneSig checks the encoded bytes. The downstream code then uses the submitted Daml value, which can contain different callback arguments than the signers intended.

— RECOMMENDATION

Make every nested container in `CallContext` self-delimiting. Prefix each map with its entry count and prefix each encoded child subtree before concatenating the next sibling entry. Apply this to `encodeTextMap`, `encodeAddressMap` and `encodeTextMapAnyValue`.

For example:

```
```plain text
encodeTextMap encodeValue entries =
lengthPrefix (show (length (Map.toList entries)))
<> mconcat
(map
 \(key, value) ->
lengthPrefix key <> lengthPrefix (encodeValue value))
(Map.toList entries))
```

**M-01** CallContext collisions swap signed payloads

CONTINUED

```
encodeAddressMap encodeValue entries =
lengthPrefix (show (length (Map.toList entries)))
<> mconcat
(map
 \(\address, value) ->
 encodeAddress address <> lengthPrefix (encodeValue value))
(Map.toList entries))
``
```

Mirror the wire-format change in the off-chain signer SDK. Add regression tests proving that distinct nested `CallContext` values cannot collide at both `encodeCallContext` and `encodeLeaf`. Include a callback-specific regression where a two-entry `callbackContext` collision would otherwise change the arguments parsed by `parseCallbackContextToFnArgs``.

**— RESOLUTION**

Status: Resolved.

MEDIUM **M-02**

ACKNOWLEDGED

# Admin bypasses the OneSig quorum

CATEGORY Access Control

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml:66-131`

REVIEW Main Review

## DESCRIPTION

`OneSigConfig` holds the `secp256k1` signer set, quorum threshold and seed that OneSig layers as an M-of-N multisig on top of the single `admin` party. The config-mutating choices `OneSigConfig_SetThreshold`, `OneSigConfig_SetSigner` and `OneSigConfig_SetSeed` are `controller admin` (`OneSigConfig.daml:66`). A single admin bypasses the quorum to change the signers, quorum, and seed, and to exercise those choices, with no `secp256k1` signature.

This is a finding only because OneSig advertises the M-of-N property; otherwise it is trusted-admin behavior. Nothing on-ledger enforces decentralization. Severity depends on hosting: High if single-operator, Info if threshold-hosted.

## RECOMMENDATION

Route signer/quorum/seed mutation through a quorum-signed `OneSig_ExecuteTransaction` op rather than a direct `controller admin` choice, so the multisig governs its own configuration. If admin override is intended, document it explicitly as the trust model.

## RESOLUTION

Status: Acknowledged.

LOW **L-01**

RESOLVED

# Wrong-ID RBAC authorizes global updates

CATEGORY	Access Control
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigUtils.daml
REVIEW	Main Review

## DESCRIPTION

OneSig validates the signed `Scope` against the protocol config contract being updated, but it does not validate that the RBAC witness used for authorization belongs to the same full scope. For protocol-scoped `OpSetMinFee` and `OpSetProtocolLedgerTimeValidityPeriod`, the executor supplies `accessControlCid` through unsigned execution context. OneSig forwards that CID into `SetMinFee` or `SetLedgerTimeValidityPeriod` without first checking the witness scope.

This matters for global request-factory configuration. `RequestFactoryConfig` exposes its minimum-fee and ledger-time-validity interfaces with scope `category = _CATEGORY_REQUEST_FACTORY`, `id = ""` and `admin = handler`. The downstream RBAC check calls `assertRole` with `expectedId = None` and `assertRole` treats `None` as "skip the ID check". It only checks that the supplied `AccessControl` has the expected admin and category before checking the role assignment.

An executor can therefore submit a valid quorum-signed global config update while providing a same-admin, same-category `AccessControl` contract with an unrelated ID. If that wrong-ID RBAC contract grants the OneSig admin the required role, the global request-factory config update succeeds even though the RBAC contract for the signed empty-ID scope would not authorize it.

## RECOMMENDATION

Before forwarding an executor-supplied `accessControlCid` into RBAC-gated settlement-layer choices, OneSig should fetch the `AccessControl` interface view and require its full scope to match the signed target scope. The check should compare admin, category and ID.

The downstream global request-factory choices should also pass `Some ""` to `assertRole` instead of `None` so the empty global ID is checked.

Alternatively, change `assertRole` so callers cannot accidentally turn off ID validation. For example, represent "solo-admin mode" separately from "RBAC contract is present but ID should not be checked" or make the expected scope a full `Scope` value rather than separate admin, category and optional ID fields.

---

**L-01** Wrong-ID RBAC authorizes global updates

CONTINUED

---

— **RESOLUTION**

Status: Resolved.

LOW **L-02**

ACKNOWLEDGED

# Unsigned config replacement CIDs

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/I0AppConfig/Call.daml
REVIEW	Main Review

## DESCRIPTION

`OpSetTransferRule` and `OpSetRequestFactory` do not bind the replacement contract identity in the signed OneSig call. The executor supplies `newTransferRuleCid` and `newRequestFactoryCid` through unsigned `ConfigContext` and `apply0AppConfigCall` forwards those CIDs into the OApp config setter. The downstream checks only prove that the replacement has a compatible high-level identity. For transfer rules, `0ftFactoryConfig` requires the supplied `ITransferRule` to expose the same `instrumentId` and to have the instrument admin as a signatory. It does not prove that the supplied rule is the current singleton rule operators intended to install.

For request factories, OneSig calls `RequestFactory_PublicFetch` with the signed `expectedHandler`. The request factory implementation checks that the handler matches, but OneSig does not bind or compare the returned `treasury` or `permanentObservers`. Consequently, a same-handler factory with different economics or observers can satisfy the signed check.

This does not bypass the signer quorum for the replacement operation. The quorum still authorized changing the transfer rule or request factory for the signed OApp. The gap is narrower: if an alternate same-instrument transfer rule or same-handler request factory already exists and is disclosed, the executor can choose that alternate dependency even if signers expected a different current contract.

## RECOMMENDATION

Bind a stable replacement identity in the signed call and verify it before exercising the config setter.

For `OpSetTransferRule`, sign and check a stable current-rule identifier or implementation hash in addition to the `instrumentId`. For `OpSetRequestFactory`, sign and check the relevant factory view fields, including `handler`, `treasury` and a stable representation of `permanentObservers`.

If contract IDs are too volatile to sign directly, use a registry-backed current pointer or deterministic metadata hash that the on-ledger code can fetch and compare. The important property is that a signed replacement operation identifies the exact dependency, not just a compatible contract family.

---

**L-02** Unsigned config replacement CIDs

CONTINUED

---

— **RESOLUTION**

Status: Acknowledged.

LOW **L-03**

ACKNOWLEDGED

# Unsigned fee config can lower request fees

CATEGORY Validation

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigTypes.daml`

REVIEW Main Review

## DESCRIPTION

`RequestContext.ownerConfig` and `OAppRequestContext.requestFactoryConfig` are unsigned execution-context fields. They are not included in the OneSig Merkle leaf, but they decide which `RequestFactoryConfig` is used when `Request_Create` checks `minFee`. `executeCreateRequest` forwards `ownerConfig` directly into `Request_Create`. The OApp request code forwards `requestFactoryConfig` into `OApp_CreateRequest`, which later passes it to `Request_Create` as `ownerConfig`. The production `RequestFactory` then checks only that the supplied config is controlled by the same handler. It does not prove that the supplied config is the current fee policy for the factory. Consequently, if a stale or duplicate same-handler `RequestFactoryConfig` exists, the executor can choose the lower-fee config while submitting the same signed request. The request factory will validate the signed `feeAmount` against that lower `minFee`, then store that lower `minFee` on the created `Request`. This does not let the executor spend admin funds. The executor still pays the signed `feeAmount`. The issue is narrower: the OneSig signature does not bind the fee policy used by the request factory, so request creation can use a stale or duplicate config instead of the intended current config.

## RECOMMENDATION

Bind request creation to the current `RequestFactoryConfig`, not only to the request factory handler. If contract IDs cannot be signed directly, add a stable config identifier or registry-backed current-config pointer and verify it before calling `Request_Create`.

At minimum, sign and verify stable fee-policy fields such as `handler`, `minFee` and `ledgerTimeValidityPeriod`. Apply the same check to both `RequestContext.ownerConfig` and `OAppRequestContext.requestFactoryConfig`.

## RESOLUTION

Status: Acknowledged.

LOW **L-04**

ACKNOWLEDGED

# Rate-limiter state can be redirected

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/I0AppConfig/Call.daml
REVIEW	Main Review

## DESCRIPTION

`applyOAppConfigCall` takes `rateLimitStateCid` from the unsigned `ConfigContext` and forwards it into signed rate-limiter operations. This affects `OpRateLimiterSetDefault`, `OpRateLimiterSetEidOverride`, `OpRateLimiterRemoveEidOverride` and `OpRateLimiterSetState`.

The signed OneSig leaf binds the `OAppId`, the operation and the new limit or state values. It does not bind the concrete `RateLimiterState` that should receive the checkpoint or overwrite. Downstream validation only checks that the supplied state is signed by the same OApp admin and exposes the same `OAppId`. A second active `RateLimiterState` for the same OApp therefore passes the check.

Consequently, an executor with disclosure of the alternate state can submit a valid quorum-signed rate-limiter update while choosing which same-OApp state contract receives it. For checkpointed config changes, the config is still updated, but the intended state is not checkpointed under the old decay parameters before the new limits take effect. For `OpRateLimiterSetState`, the signed overwrite can be applied to the alternate state while the intended state remains unchanged.

This does not bypass OneSig quorum checks. The issue is that the quorum authorizes a rate-limiter state transition for an OApp, but the executor-controlled context selects which same-OApp state object receives the transition.

## RECOMMENDATION

Bind OneSig rate-limiter operations to the current state identity, not only to `OAppId`. If contract IDs cannot be signed because state contracts rotate, introduce a stable `rateLimiterStateId` or registry-backed current pointer per OApp. The config implementation should reject any `rateLimitStateCid` that is not the registered current state.

Alternatively, enforce one active `RateLimiterState` per `OAppId` and have

`requireMatchingRateLimitState` reject non-current duplicates. Batch threading can still use the freshly returned replacement CID after the first validated state operation in a batch.

---

**L-04** Rate-limiter state can be redirected

CONTINUED

---

— **RESOLUTION**

Status: Acknowledged.

LOW **L-05**

ACKNOWLEDGED

# Duplicate OApp config can receive updates

CATEGORY Validation

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/I0AppConfig/Fetch.daml`

REVIEW Main Review

## DESCRIPTION

`fetchAndValidateOAppConfig` binds a signed OApp-config operation to an `OAppId`, but not to a specific current config contract. The executor supplies `ConfigContext.configCid` through unsigned execution context. The code coerces that CID to `I0AppConfig`, fetches it and checks that the interface view exposes the signed `OAppId`.

That check is only sufficient if each `OAppId` has one current config. If a duplicate same-OApp config exists, both contracts can pass validation because the signed `OAppId` is unchanged. An executor with disclosure of the duplicate CID can therefore submit a valid quorum-signed config update while choosing which same-OApp config receives the update.

Consequently, the OneSig nonce can be consumed and the signed update can be applied to the duplicate config while the intended live config remains unchanged. This does not bypass the signer quorum. The narrower issue is that the quorum authorizes a config change for an OApp identity, but unsigned execution context selects which compatible same-OApp config receives it.

The duplicate-config prerequisite limits impact. For canonical OApp configs, the duplicate normally has to be created or left active by the trusted OApp admin or deployment process. If deployment guarantees exactly one current config per `OAppId`, this becomes a hardening issue around stale or duplicate config disclosure.

## RECOMMENDATION

Bind OApp config operations to the current config identity, not only to `OAppId`. If CIDs are too volatile to sign directly, add a stable config identifier or registry-backed current-config pointer and verify it before exercising any `I0AppConfig` choice.

Alternatively, make duplicate active configs for the same `OAppId` impossible. Apply the same current-config check to the interface-specific helpers in this module so allowlist, pause, rate-limiter, ledger-time and generic config updates all enforce the same target-binding rule.

---

**L-05** Duplicate OApp config can receive updates

CONTINUED

---

— **RESOLUTION**

Status: Acknowledged.

LOW **L-06**

ACKNOWLEDGED

## Duplicate IOApp can receive OApp requests

CATEGORY Validation

LOCATION contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/IOApp/Call.daml

REVIEW Main Review

### — DESCRIPTION

`executeOAppCreateRequest` binds the signed operation to an `OAppId`, but not to a specific `IOApp` contract or config contract. The executor supplies both CIDs through `OAppRequestContext`. The code fetches them, checks that each one exposes the signed `OAppId` and then exercises `OApp_CreateRequest` on the supplied `IOApp` with the supplied config.

That check is only sufficient if each `OAppId` has one current `IOApp` and one current config. If a duplicate same-`OAppId` pair exists, the executor can submit a valid quorum-signed `OpCreateOAppRequest` using the duplicate pair instead of the intended one. The signed payload still verifies because the signed `OAppId` is unchanged.

Consequently, the created `Request` can be handled by a different gateway, request factory, treasury or observer set than operators expected.

### — RECOMMENDATION

Bind `OpCreateOAppRequest` to the current `IOApp` and config identity, not only to `OAppId`. If CIDs are too volatile to sign directly, use a registry-backed current pointer or stable target identifier and verify it before exercising `OApp_CreateRequest`.

At minimum, include stable target fields in the signed call and compare them on-ledger. Useful fields include the expected `oappGateway`, request-factory handler, request-factory treasury and a stable config identifier.

### — RESOLUTION

Status: Acknowledged.

LOW **L-07**

ACKNOWLEDGED

# Duplicate RBAC can receive role updates

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/IAccessControl/Validate.daml
REVIEW	Main Review

## DESCRIPTION

`fetchAndValidateAccessControl` binds the signed operation to a `Scope`, but not to a specific `AccessControl` contract. The executor supplies the target CID through `AccessControlContext.targetContractId`. The code fetches that contract, checks that its interface view exposes the signed `Scope` and then allows the role update to continue. That check is only sufficient if each RBAC scope has one current `AccessControl` contract. If a duplicate same-scope contract exists, the executor can submit a valid quorum-signed grant or revoke operation using the duplicate CID instead of the intended one. The signed payload still verifies because the signed `Scope` is unchanged.

Consequently, a valid role update can be consumed without changing the RBAC contract that operators expected to update. For example, a signed `OpRevokeRoles` can be applied to an alternate same-scope contract while the intended contract keeps the revoked role active.

## RECOMMENDATION

Bind access-control operations to the current RBAC contract identity, not only to `{category, id, admin}`. If CIDs are too volatile to sign directly, add a stable `accessControlId` or registry-backed current pointer to the signed call. OneSig should verify that identity before exercising `AccessControl_GrantRoles` or `AccessControl_RevokeRoles`.

Alternatively, reject or make unrepresentable multiple active `AccessControl` contracts for the same scope. Batch threading can still use the freshly produced replacement CID after the first validated role update.

## RESOLUTION

Status: Acknowledged.

LOW **L-08**

ACKNOWLEDGED

## Stale RBAC witness authorizes updates

CATEGORY	Access Control
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigUtils.daml
REVIEW	Main Review

### — DESCRIPTION

OneSig signs the requested operation and its logical target, but it does not sign the `AccessControl` contract that downstream settlement-layer choices use as their delegated-role witness. In `executeCall`, the executor can supply an optional `accessControlCid` through unsigned execution context. That supplied CID is then preferred for OApp request creation, OApp config updates, min-fee updates and ledger-time validity updates.

The downstream choices still validate the signed OApp or protocol scope against the target object. However, delegated authorization is checked against the supplied `IAccessControl` contract. `assertRole` accepts an access-control contract when its view matches the expected admin, category and id and when its role map grants the required role to the caller.

Consequently, the signed payload does not bind the current RBAC state that should authorize the operation. If the intended access-control contract no longer grants the OneSig admin a delegated role, but a stale same-scope access-control contract still grants it, the executor can submit the signed operation with the stale RBAC CID. The operation then executes against the intended live target even though the intended RBAC witness would reject the caller.

### — RECOMMENDATION

Bind every RBAC-gated OneSig operation to the current access-control state for the signed scope. A direct fix is to maintain a singleton or registry-backed current `AccessControl` pointer per scope and reject any executor-supplied `accessControlCid` that is not that current pointer before forwarding it into downstream choices.

If access-control contracts must rotate, sign and verify a stable access-control identity or version for the operation. Batch threading can still use the replacement CID produced by a prior RBAC update in the same batch, but only after the first RBAC witness has been proven to be the current state.

### — RESOLUTION

Status: Acknowledged.

LOW **L-09**

RESOLVED

# Non-bytes32 proofs verify on Canton

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/MerkleProof.daml
REVIEW	Main Review

## DESCRIPTION

`verifyMerkleProof` accepts proof siblings as arbitrary `Text`. `commutativeKeccak` sorts the two text values and hashes their concatenation without first requiring both values to be canonical bytes32 hashes. `computeMerkleRoot` then folds every submitted sibling through that helper.

This lets Canton accept a quorum-signed Merkle root that depends on a non-bytes32 proof node. For example, a normal `encodeLeaf` output can be paired with a one-byte sibling such as `"aa"`. If the signer quorum signs `commutativeKeccak leaf "aa"`, `OneSig_ExecuteTransaction` accepts `proof = ["aa"]` and consumes the nonce.

That root is outside the shared OneSig fixed-width proof format used by the other chain implementations and off-chain proof tooling. The Canton verifier therefore accepts a proof shape that cannot be represented as a normal list of bytes32 Merkle siblings.

## RECOMMENDATION

Reject every proof sibling that is not exactly one canonical bytes32 value before sorting or hashing. Apply the same canonicalization rule to the leaf and expected root or normalize them once and use the normalized values consistently throughout signature verification and proof verification.

The pair hash should always be computed over two fixed-width 32-byte values, equivalent to `keccak256(bytes32 || bytes32)`.

## RESOLUTION

Status: Resolved.

LOW **L-10**

ACKNOWLEDGED

# New executors may miss the anchor

CATEGORY	DoS
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig.daml
REVIEW	Main Review

## — DESCRIPTION

`OneSig` keeps its observer list on the stable anchor. The anchor is not archived or recreated during normal execution, so this observer list does not change when a signed batch updates `OneSigConfig.executors`. A quorum-signed rotation can therefore remove the current executor and add a new one in `OneSigConfig` while leaving the new executor outside `OneSig.observers`. The recreated config is visible to the new executor because `OneSigConfig` observes `executors`. The anchor remains visible only to its original stakeholders unless it is explicitly disclosed.

Consequently, the new executor can be authorized by `isAuthorizedExecutor` but still be unable to submit `OneSig_ExecuteTransaction`, because the submitter cannot exercise an invisible anchor contract. The old executor still observes the anchor, but it no longer satisfies the executor allowlist after the rotation.

## — RECOMMENDATION

Ensure any party added to `OneSigConfig.executors` can also observe the `OneSig` anchor before the rotation takes effect. If the anchor remains immutable, reject executor additions unless the new executor is already an anchor observer or require operational explicit disclosure of the anchor before that executor is expected to submit.

A stronger fix is to add a controlled anchor-observer update mechanism or to move executor visibility to a durable delegation contract that can be observed independently of the immutable anchor.

## — RESOLUTION

Status: Acknowledged.

LOW **L-11**

ACKNOWLEDGED

# Signer policy changes revive roots

CATEGORY	Signatures
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml
REVIEW	Main Review

## — DESCRIPTION

`OneSigConfig_SetSigner` and `OneSigConfig_SetThreshold` recreate `OneSigConfig` with a changed signer set or quorum, but they preserve the existing `nonce` and `seed`. `OneSig_ExecuteTransaction` later verifies Merkle-root signatures against the current config before checking the leaf for the current nonce. Consequently, signer-policy maintenance does not revoke outstanding roots. A root that was rejected under the previous policy can become valid after the policy changes, as long as the root still targets the active `oneSigId`, `admin`, `nonce`, `seed`, expiry and call bytes. For example, a root with one valid signature is rejected while the config requires a 2-of-2 quorum. If the trusted admin later lowers the threshold to 1 without rotating `seed`, the same root and signature can execute. Similarly, a root signed only by an address that was not in `signers` can execute after that address is added, even if the signature existed before the signer was authorized.

## — RECOMMENDATION

Revoke outstanding roots whenever signer-policy maintenance is meant to change who can authorize pending work. The smallest on-ledger fix is to rotate `seed` or another signed-domain version, inside `OneSigConfig_SetSigner` and `OneSigConfig_SetThreshold`. If preserving outstanding roots across signer-policy changes is intentional, document it as an explicit invariant. Operational runbooks should require a `seed` rotation before or during any signer-set or threshold update that is intended to invalidate pending signatures.

## — RESOLUTION

Status: Acknowledged.

LOW **L-12**

RESOLVED

## Global mode drops EID usage

CATEGORY	Logical Error
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/I0AppConfig/Cal1.daml
REVIEW	Main Review

### — DESCRIPTION

`OpRateLimiterSetGlobalFlags` updates the rate-limiter config directly and returns no updated rate-limiter state. The OneSig call does not require `ConfigContext.rateLimitStateCid`, does not checkpoint the current state and does not migrate existing per-EID buckets when `newUseGlobalState` changes. This is unsafe when the config switches from per-EID state to global state. Existing usage is stored under each destination endpoint ID. After `useGlobalState` becomes `True`, the rate-limiter state code resolves every endpoint to bucket `0`. If bucket `0` does not already contain the aggregate usage, later transfers read a fresh bucket and recover full capacity even though the old per-EID buckets still contain active usage. For example, assume endpoint `eid = 101` has `900` outbound usage against a `1000` limit. A signed `OpRateLimiterSetGlobalFlags` switches `useGlobalState` to `True` without touching the state contract. The next read for endpoint `101` resolves to bucket `0`, sees no prior usage and reports `1000` available capacity instead of carrying over the existing `900` usage.

### — RECOMMENDATION

Treat changes to `useGlobalState` as state-affecting rate-limiter updates. Require a current `rateLimitStateCid` when `OpRateLimiterSetGlobalFlags` changes `useGlobalState` and update the state atomically with the config.

When switching from per-EID mode to global mode, aggregate the active per-EID buckets into bucket `0` using the old config before creating the new config. When switching away from global mode, define an explicit policy for splitting, preserving or resetting global usage and make that policy part of the signed operation.

### — RESOLUTION

Status: Resolved.

LOW **L-13**

RESOLVED

# Noncanonical roots reject proofs

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig.daml
REVIEW	Main Review

## DESCRIPTION

`OneSig_ExecuteTransaction` uses the submitted `merkleRoot` in two different forms. The signature payload is built with `buildSignMerkleRootPayload`, which treats the root as a bytes32 value and accepts noncanonical text forms such as an optional `0x` prefix or uppercase hex. The Merkle proof check then passes the raw submitted `merkleRoot` into `verifyMerkleProof`, which compares it directly against the reconstructed root text.

Consequently, two text values that represent the same bytes32 root can be treated differently in the same execution. For example, `r` and `"0x" <> r` produce the same EIP-712 signing payload, so the same signatures verify for both. The proof check still fails for `"0x" <> r`, because `computeMerkleRoot` reconstructs the canonical root text `r` and compares it against the raw prefixed value.

Uppercase roots have the same problem. They are accepted by the signing payload builder as the same bytes, but the proof equality check rejects them when the reconstructed root is lowercase.

## RECOMMENDATION

Canonicalize `merkleRoot` once in `OneSig_ExecuteTransaction` and use that canonical value for both signature verification and Merkle proof verification.

If OneSig's intended wire format is strict, reject noncanonical roots before building the signing payload. A safe rule is to require exactly 64 lowercase hex characters with no `0x` prefix for submitted roots, leaves and proof nodes.

## RESOLUTION

Status: Resolved.

LOW **L-14**

RESOLVED

# Proof siblings are not canonicalized

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/MerkleProof.daml
REVIEW	Main Review

## DESCRIPTION

`commutativeKeccak` treats each Merkle node as raw `Text`. It compares the two text values to choose their order and then concatenates the same raw strings before hashing. The proof verifier never first converts the leaf and proof siblings into one canonical bytes32 representation.

Consequently, two text values that denote the same 32-byte sibling can be handled differently. A sibling supplied with a `0x` or `0X` prefix introduces non-hex characters into the hash input and aborts verification. An uppercase sibling can also fail even though it denotes the same bytes as the lowercase value, because raw text ordering can put the uppercase value on the opposite side of the pair before hashing.

## RECOMMENDATION

Normalize every Merkle node before sorting and hashing it. Require the leaf, each proof sibling and the expected root to be exactly one canonical bytes32 value or strip an optional prefix and lowercase the value once before comparing or concatenating it. Use the normalized value consistently for signature payload construction and Merkle proof verification.

## RESOLUTION

Status: Resolved.

LOW **L-15**

RESOLVED

## Mixed config updates use stale CIDs

CATEGORY	DoS
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigUtils.daml
REVIEW	Main Review

### — DESCRIPTION

`executeCall` threads replacement config CIDs in separate maps for each logical operation group. OApp config updates use `configContracts`, min-fee updates use `minFeeContracts` and protocol ledger-time updates use `ledgerTimeValidityContracts`.

That works when every consuming update stays inside one operation group. It breaks when the same underlying config contract implements more than one update interface. For example,

`RequestFactoryConfig`, `RegistryConfig`, `DiscoveryRegistry` and `Governance` implement both `IMinFee` and `ILedgerTimeValidityPeriod`. Their interface choices consume the current contract and create a replacement of the same underlying template.

Consequently, a valid signer-approved batch can fail even though each call is valid on its own. If the batch first updates `minFee` on a `RequestFactoryConfig`, OneSig records the replacement only in `minFeeContracts`. A later ledger-time update for the same `Scope` does not look there. It falls back to the original `ledgerTimeValidityContractId`, which was archived by the first call, so the whole batch aborts. The reverse order fails in the same way.

The same stale-CID behavior exists between OApp config updates and protocol-scoped ledger-time updates for OApp configs such as `OftFactoryConfig` and `AdapterConfig`. An OApp config ledger-time update records the replacement only in `configContracts`, while the protocol-scoped ledger-time branch reads only `ledgerTimeValidityContracts`.

### — RECOMMENDATION

Track replacements by the exact consumed contract ID, not only by logical scope or operation group. After any consuming config update, record a mapping from the old underlying CID to the newly created underlying CID. Before a later branch uses an executor-supplied config CID, rewrite it only when it exactly matches a CID consumed earlier in the same batch.

Do not thread replacements only by `Scope` or `OAppId`. Separate contracts can legitimately share those logical identifiers. Exact CID rewriting preserves that separation while allowing mixed-interface updates on the same consumed config contract to compose.

---

**L-15** Mixed config updates use stale CIDs

CONTINUED

---

— **RESOLUTION**

Status: Resolved.

LOW **L-16**

ACKNOWLEDGED

## Duplicate configs can replay roots

CATEGORY Signatures

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig.daml`

REVIEW Main Review

### — DESCRIPTION

`OneSig_ExecuteTransaction` takes the `OneSigConfig` to use from the executor-supplied `configCid`. The helper `fetchOwnedConfig` only checks that the fetched config has the same `admin` and `oneSigId` as the anchor. It does not prove that the supplied config is the single current config for that OneSig instance. The nonce is also local to the supplied config. Execution builds the signed leaf from `config.oneSigId`, `config.admin`, `config.nonce` and the signed calls, verifies the root using `config.seed`, then archives and recreates only that supplied config with `nonce + 1`.

This means replay protection depends on there being exactly one active `OneSigConfig` for each `(admin, oneSigId)`. If two active configs exist with the same `admin`, `oneSigId`, `nonce`, `seed`, signer set, quorum and executor policy, the same signed root can execute once against each config. The first execution advances the nonce on the first config, but the second config still has the old nonce and still satisfies the ownership check, signature check and Merkle proof check.

For example, assume a OneSig instance has two active configs `A` and `B`, both at nonce `0` and both using the same seed and 2-of-2 signer set. The quorum signs a root for a self-management operation at nonce `0`. An executor can submit that root with `configCid = A`, which advances only `A` to nonce `1`. The executor can then submit the exact same root, expiry, signatures and transaction with `configCid = B`, because `B` is still at nonce `0`. The signed operation is consumed twice even though the leaf was intended to be single-use for the OneSig instance.

### — RECOMMENDATION

**L-16** Duplicate configs can replay roots

CONTINUED

Make the current config unique for each OneSig anchor. Prefer a ledger-enforced singleton keyed by `(admin, oneSigId)` if contract keys are acceptable for this package and deployment.

If contract keys are not suitable, store or derive a stable current-config pointer for each anchor and have `OneSig_ExecuteTransaction` reject any supplied config that is not the current one. The pointer should be updated atomically when execution recreates the config, so nonce advancement is global to the OneSig instance rather than local to an arbitrary matching config CID.

**— RESOLUTION**

Status: Acknowledged.

LOW **L-17**

RESOLVED

# Malformed seed bricks all executes

CATEGORY	Validation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml:44-56
REVIEW	Main Review

## DESCRIPTION

Neither the `OneSigConfig` `ensure` nor `OneSigConfig_SetSeed` (`OneSigConfig.daml:117`) checks that `seed` is a 64-character hex string. The shape is asserted only later, inside `buildSignMerkleRootPayload` (`assertHexLength`), at execute time.

A quorum or admin that sets a malformed seed bricks every future `OneSig_ExecuteTransaction`, because each execute must first build the sign payload and fails that assertion — including the execute needed to fix the seed. The failure is fail-closed (no forgery), so the harm is a self-inflicted permanent denial of service on the instance.

## RECOMMENDATION

Assert the seed is well-formed (64 hex chars) in the `OneSigConfig` `ensure` and in `OneSigConfig_SetSeed`, so a bad seed is rejected at set time instead of bricking the instance.

## RESOLUTION

Status: Resolved.

LOW **L-18**

ACKNOWLEDGED

# Leaf lengthPrefix counts codepoints

CATEGORY	Compatibility
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigEncoding.daml:31-32
REVIEW	Main Review

## — DESCRIPTION

`lengthPrefix` prefixes each text field with `show (T.length value)`, a Unicode codepoint count (`OneSigEncoding.daml:31`). The encoding contract requires the off-chain Canton SDK to reproduce the signed leaf byte-for-byte.

For any signed field carrying non-ASCII content, an SDK measuring length in UTF-8 bytes or UTF-16 units produces a different length prefix, and therefore a different leaf. On the ledger side `textToHex` (`Serde.daml:176`) rejects any non-ASCII code point before hashing, so the divergence never yields a valid-but-wrong leaf — there is no forgery. The residual harm is cross-implementation canonicity: a signer whose SDK admits non-ASCII content can produce a root the ledger will not accept (a signed-but-unexecutable batch), and byte-for-byte agreement with the off-chain encoder is otherwise unproven for the codepoint-vs-byte case.

## — RECOMMENDATION

Length-prefix by UTF-8 byte length (or fix a single canonical measure and document it), and add a cross-implementation test vector with non-ASCII content to guarantee on-/off-ledger agreement.

## — RESOLUTION

Status: Acknowledged.

LOW **L-19**

ACKNOWLEDGED

# No upper cap on signers bricks config

CATEGORY	DoS
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml:44-81
REVIEW	Main Review

## — DESCRIPTION

The `OneSigConfig` ensure caps only `quorum <= length signers`, never a maximum list length (`OneSigConfig.daml:44`). Every `OneSig_ExecuteTransaction` folds the whole signer list through `verifySignatures`, and every setter recreates the full config record.

An admin growing `signers` or `executors` one at a time via `SetSigner` / `SetExecutor` eventually pushes the recreated record past the Canton transaction-size limit, at which point both execution and any setter that would shrink the list fail. The instance is bricked with no on-ledger recovery.

## — RECOMMENDATION

Add a sane maximum length for `signers` and `executors` to the `ensure` clause so the config can never grow past what a single transaction can carry.

## — RESOLUTION

Status: Acknowledged.

LOW L-20

RESOLVED

# OneSig anchor lacks ensure on oneSigId

CATEGORY Validation

LOCATION contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig.daml:31-38

REVIEW Main Review

## DESCRIPTION

The `OneSig` anchor template has no `ensure` clause ( `OneSig.daml:31` ), so it can be created with a negative `oneSigId`. `OneSigConfig.ensure` requires `oneSigId >= 0`, so no matching config can ever be created, and `fetchOwnedConfig`'s `config.oneSigId == oneSigId` assertion in every execute reverts. An anchor created with a negative `oneSigId` is permanently unusable. The harm is self-inflicted, since the admin is the sole creator of the anchor, but the asymmetry between the anchor and config `ensure` constraints has no guard.

## RECOMMENDATION

Add an `ensure oneSigId >= 0` (matching `OneSigConfig`) to the `OneSig` anchor template.

## RESOLUTION

Status: Resolved.

LOW **L-21**

ACKNOWLEDGED

# Canton rejects the shared EVM signature format

CATEGORY	Signatures
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig.daml
REVIEW	Main Review

## — DESCRIPTION

`OneSig_ExecuteTransaction` accepts signer approvals as `[SignerSignature]` and passes them directly to `verifySignatures`. That is not the shared OneSig signature object produced by `onesig-core`. The shared library emits normal EVM-style recoverable signatures: 65 bytes encoded as `r || s || v`.

The Canton verifier expects a different representation. `SignerSignature` carries a hex SPKI-DER public key and a hex DER ECDSA signature. `verifySignatures` then hashes the EIP-712 payload, verifies the DER signature against the supplied public key, derives the signer's Ethereum address from that public key and checks that the derived address is in the authorized signer set.

The signed digest can still be correct. The problem is the submitted signature format. A valid `onesig-core` approval for the right root, seed and expiry is rejected by Canton unless the same `(r,s)` values are re-encoded as DER and paired with the public key before submission.

Consequently, the package does not accept the signature object produced by the documented shared OneSig signing library. A Canton executor using canonical `onesig-core` output cannot execute otherwise valid OneSig batches unless an additional, canonical conversion layer exists outside the package. This can block material protocol administration on Canton in the cross-chain OneSig signing model.

## — RECOMMENDATION

Make the Canton OneSig execution ABI accept the documented shared signature format or provide a canonical Canton adapter and make it part of the required signing interface.

If the Daml verifier must keep using `secp256k1WithEcdsaOnly`, change the submitted signature representation to carry the public key plus the 65-byte recoverable signature, then convert `r` and `s` into DER before verification. Alternatively, have the official Canton SDK transform a `onesig-core` `Signature` into `[SignerSignature]` deterministically and document that transformed value as the required Canton submission format.

---

**L-21** Canton rejects the shared EVM signature format

CONTINUED

---

— **RESOLUTION**

Status: Acknowledged.

## INFORMATIONAL I-01

ACKNOWLEDGED

# Ledger-time updates can hit duplicate configs

CATEGORY Validation

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/ILedgerTimeValidityPeriod/Validate.daml`

REVIEW Main Review

## DESCRIPTION

`OpSetProtocolLedgerTimeValidityPeriod` signs a `Scope`, but the target `ledgerTimeValidityContractId` comes from unsigned execution context. `fetchAndValidateLedgerTimeValidity` fetches that `ILedgerTimeValidityPeriod` contract and checks that its interface view exposes the signed scope with the expected admin signatory. It does not prove that the supplied contract is the current ledger-time config for that scope.

This lets a disclosed duplicate config absorb a valid signed update. For example, two active `RequestFactoryConfig` contracts for the same handler both expose the request-factory scope `{category = _CATEGORY_REQUEST_FACTORY, id = "", admin = handler}`. An executor can submit the signed ledger-time update with the alternate config CID. The alternate contract is consumed and recreated with the new `ledgerTimeValidityPeriod`, while the intended live config keeps its old timestamp-validity rule.

## RECOMMENDATION

Bind protocol-scoped ledger-time updates to the current config identity, not only to `{category, id, admin}`. If contract IDs are too volatile to sign directly, add a stable config identifier or registry-backed current pointer and verify it before exercising `SetLedgerTimeValidityPeriod`.

Also make multiple active ledger-time configs for the same scope unrepresentable or rejected. Batch threading can still use the newly returned CID after a validated update, but the first target in the batch should be proven to be the current config.

## RESOLUTION

Status: Acknowledged.

MEDIUM **M-01**

ACKNOWLEDGED

# Cross-instance ledger-time auth bypass

CATEGORY	Access Control
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/ILedgerTimeValidityPeriod/Validate.daml:19-27
REVIEW	Remediation Review 1

## — DESCRIPTION

The remediation makes `OneSigConfig` implement the shared `ILedgerTimeValidityPeriod` interface, exposing view scope `{category = _CATEGORY_ONESIG, id = show oneSigId, admin}` (`OneSigConfig.daml:70-81`). The protocol op `OpSetProtocolLedgerTimeValidityPeriod` validates its executor-supplied target only via `fetchAndValidateLedgerTimeValidity -\> assertSignedScope` (`OneSig/ILedgerTimeValidityPeriod/Validate.daml:19-27`): it checks the signed scope equals the target's view scope and that `admin` is a signatory, plus `category /= 0App`. It never checks that `scope.id` equals the `oneSigId` of the `OneSig` anchor executing the batch — the leaf is bound to the anchor's `oneSigId` (`encodeLeaf, OneSig.daml:88`), but that binding is never compared to the target scope. When one admin operates more than one `OneSig` instance (distinct `oneSigId`, each with its own EIP-712 signer set/quorum, sharing a Canton `admin`), an executor of instance A can place a call scoped to instance B inside instance A's quorum-signed batch. It routes onto B's `OneSigConfig` and mutates B's governed `ledgerTimeValidityPeriod` — even though B's own quorum never approved it. A's quorum thus governs B's fee-staleness window: widen it to accept stale fee timestamps, or narrow it to DoS B's fee path. The attacker is instance A's distinct signer quorum, NOT the admin: this is independent of the accepted "admin bypasses quorum" finding. The admin's authority merely propagates through A's anchor (signatory admin), so a compromise of A's quorum plus an authorized A executor reaches B without B's keys or the admin's Canton key. It requires no duplicate/singleton violation — it works on a correct multi-instance deployment.

Confirmed end-to-end with an executable Daml Script PoC

(`tests/daml/Watchtower/PoC_6a4f6739_e2e.daml`): a batch signed only by instance A's quorum keys (leaf `oneSigId=1`), run through the real `OneSig_ExecuteTransaction`, changed instance B's (`oneSigId=2`, own distinct quorum) `ledgerTimeValidityPeriod` from 3600 to 99999 with B's keys never used. The different-admin case fails on Daml signatory authority (`PoC_6a4f6739.daml`).

## — RECOMMENDATION

On the protocol ledger-time path, bind the signed scope to the executing instance: require `scope.id == show oneSigId` of the `OneSig` anchor being exercised, and reject otherwise.

**M-01** Cross-instance ledger-time auth bypass

CONTINUED

A `OneSigConfig` should be updated only through its dedicated self-management op (`OpSetOneSigLedgerTimeValidityPeriod`), never through the settlement-config protocol path. Equivalently, reject the `_CATEGORY_ONESIG` scope category on the protocol ledger-time path. This fix is independent of the admin-quorum-bypass finding (removing the admin's direct setter would not close this cross-instance path, and vice versa).

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-02**

ACKNOWLEDGED

# Old callbacks erase newer rate-limit usage

CATEGORY	Unexpected Behavior
LOCATION	contracts/protocol/canton/contracts/Layerzero/RateLimiter/daml/RateLimiterState.daml
REVIEW	Remediation Review 1

## DESCRIPTION

`OpRateLimiterSetDefault` correctly checkpoints aggregate usage under the old rate-limit parameters before installing the signed replacement. The callback state for an already pending outflow is not checkpointed or versioned. It keeps only `scaledAmount`, `resolvedEid`, `forwardRecorded` and `recordedAt`.

When that outflow later settles, `commitOutflow` and `reverseOutflow` fetch the current config.

`remainingContribution` then decays the old amount from its original `recordedAt` using the current limit and window. This applies the new leak rate retroactively to time that elapsed before the OneSig update, even though the aggregate bucket was already checkpointed under the old rate.

For example, A records 100 under a 100-second window. Fifty seconds later a signed OneSig update checkpoints A to 50 and changes the window to 1,000 seconds. B then records 50, so the bucket holds 100. Rejecting A recomputes its remaining contribution as 95 under the new window and subtracts it from the bucket. Only 5 remains instead of B's 50. A normal 95-unit outflow can then pass even though the correct remaining capacity is 50.

The full proof uses the canonical lock/unlock adapter, its real request and callback lifecycle, the single intended config and state contracts and a signed `OpRateLimiterSetDefault` batch. The gateway rejects A, but an ordinary executor finalizes A after the update and after B records newer usage. No alternate state, duplicate OApp, callback substitution, malicious package or forged signature is involved.

This issue requires an in-flight outflow across a legitimate parameter update and favorable finalization order. The resulting accounting error can erase other users' usage and allow cross-chain traffic above the governed limit. Changing to a faster decay rate creates the opposite failure and can leave phantom usage that unnecessarily blocks traffic.

## RECOMMENDATION

Do not calculate an old pending contribution from `recordedAt` with whichever config is active at settlement. Track decay across configuration epochs. One robust design is a cumulative per-bucket leak index: checkpoint the index whenever the limit or window changes, store the index in `RecordOutflowResult` and subtract only `max(0, scaledAmount -`

**M-02** Old callbacks erase newer rate-limit usage

CONTINUED

(currentIndex - recordedIndex))` when the callback settles.

Alternatively, store pending reservations explicitly and checkpoint their remaining contributions during config changes or prevent parameter activation until pending reservations from the previous epoch settle. Merely storing the old limit and window and applying them forever after the update is not sufficient if the new policy is meant to govern decay after activation.

**— RESOLUTION**

Status: Acknowledged.

MEDIUM **M-03**

ACKNOWLEDGED

# Old callbacks erase post-reset rate-limit usage

CATEGORY	Unexpected Behavior
LOCATION	contracts/protocol/canton/contracts/Layerzero/RateLimiter/daml/RateLimiterState.daml
REVIEW	Remediation Review 1

## DESCRIPTION

`IRateLimitState_SetState` replaces an EID bucket without invalidating receipts returned by earlier `IRateLimitState_RecordOutflow` calls. Those receipts contain the recorded amount, EID, direction flag and timestamp. They do not identify the state generation in which the amount was recorded. Consequently, `IRateLimitState_ReverseOutflow` can settle a pre-reset receipt against usage recorded after the reset. It subtracts the old receipt's remaining contribution from the current bucket at `RateLimiterState.daml:375-405`, even though the reset already removed that contribution. The canonical adapter proof first records A's 100-unit outflow and leaves its rejected request pending. A valid quorum then executes signed `OpRateLimiterSetState` values that reset the bucket to zero. B records 50 units in the new state generation. Any party that can see the pending callback may finalize A's rejection. This changes the bucket from 50 to zero, erasing all of B's usage. A new 100-unit outflow then succeeds even though only 50 units of capacity should be available. The quorum authorizes the exact reset values, but it does not authorize a later executor to remove post-reset usage. The impact is a rate-limit bypass after a legitimate administrative state correction. Exploitation requires an unresolved pre-reset outflow, a signed reset of the same bucket, newer usage and favorable callback ordering.

## RECOMMENDATION

Give each rate-limit bucket a monotonic state generation. Store that generation in `RecordOutflowResult` and its callback context. Increment it whenever `SetState` destructively replaces the bucket. `CommitOutflow` and `ReverseOutflow` should modify the bucket only when the receipt generation matches the current generation; an older receipt has already been superseded by the administrative reset and must not be applied again. Alternatively, track every pending reservation and either settle or explicitly invalidate them before a raw overwrite.

---

**M-03** Old callbacks erase post-reset rate-limit usage

CONTINUED

---

— RESOLUTION

Status: Acknowledged.

LOW **L-01**

ACKNOWLEDGED

# OpSetMinFee same-handler config substitution

CATEGORY	Access Control
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/IMinFee/Validate.daml:14-20
REVIEW	Remediation Review 1

## DESCRIPTION

OpSetMinFee validates its executor-supplied `minFeeContractId` only via `fetchAndValidateMinFee` -\> `assertSignedScope ( OneSig/IMinFee/Validate.daml:14-20 )`: it checks `signedScope == (view minFee).scope` and that the target admin is a signatory. For the GLOBAL-scoped IMinFee targets — `RequestFactoryConfig ( RequestFactoryConfig.daml:112-117 )` and `DiscoveryRegistry ( DiscoveryRegistry.daml:78-81 )` — the view scope is `{category, _GLOBAL_SCOPE_ID, handler}`; the id is ALWAYS `GLOBAL`, never per-instance. The tuple is thus non-injective over multiple same-`{category, GLOBAL, handler}` config contracts, so an executor supplying a signed `OpSetMinFee` could point it at a different same-handler config than intended, setting `minFee` (fee floor at `FeeUtils/Validators.daml:77`) on the wrong one -\> fee bypass (lower) or request DoS (raise). CORRECTION (re-rated Medium -\> Low). This is NOT the cross-instance escape of `6a4f6739`: that target (`OneSigConfig`) has a PER-INSTANCE scope `id = show oneSigId`, so one instance's quorum reaches another's DISTINCT-scoped config with no duplicate. Here the scope id is fixed GLOBAL, so the attack REQUIRES two same-`{category, GLOBAL, handler}` config contracts to coexist — and both templates are `signatory handler`, so ONLY the handler can create the duplicate; an executor cannot. That makes this the same operationally-gated duplicate-substitution family as H-01 (`6a461062`, the same `RequestFactoryConfig minFee` via the request-creation path) and L-81 (`6a461534`), which are accepted at LOW. The `scope.id == show oneSigId` fix does not apply (`GLOBAL != oneSigId`).

## RECOMMENDATION

Family-level defense-in-depth (shared with H-01 `6a461062` and L-81 `6a461534`): prevent two same-scope config contracts from coexisting — a uniqueness key on the config Scope if Canton supports it, and/or make config rotation archive the old contract atomically with creating the new one so no substitutable duplicate exists, and/or bind the exact target CID into the signed `OpSetMinFee` leaf. Do NOT apply `scope.id == show oneSigId` here (the GLOBAL min-fee scope is not per-instance). This is a facet of the accepted substitution family reached via the `OpSetMinFee` governance path; consider folding into `6a461062` if the team prefers a single record.

---

**L-01** OpSetMinFee same-handler config substitution

---

CONTINUED

---

**— RESOLUTION**

Status: Acknowledged.

LOW **L-02**

ACKNOWLEDGED

# No upper bound on validity period

CATEGORY Validation

LOCATION `contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml:64`

REVIEW Remediation Review 1

## DESCRIPTION

(Re-created — replaces 6a4f673e8d2fb776f9783163, which was accidentally set to Invalidated and cannot be reverted. Content unchanged.)

The new governed `ledgerTimeValidityPeriod` field on `OneSigConfig` is constrained by the template `ensure` only to be greater than zero ( `OneSigConfig.daml:64` ); it has no upper bound. This field caps the maximum permitted age of the caller-supplied timestamp on the executor-funded fee transfer — its purpose is to bound staleness.

With no ceiling, a governance action can set the period arbitrarily large, silently neutralizing the staleness protection the field is meant to enforce and allowing fee transfers carrying very old timestamps to be accepted. `SetLedgerTimeValidityPeriod` performs the same lower-bound-only check, so an unbounded value also survives every later update.

Note: the value is set by a quorum-signed governance action (an executor cannot widen it unilaterally), so impact is a governance-configuration hazard rather than an executor exploit — reasonable to treat as Low or Info.

## RECOMMENDATION

Add an upper-bound check to the `OneSigConfig` `ensure` (e.g. `ledgerTimeValidityPeriod <= <sane maximum>` ) and enforce the same ceiling in `SetLedgerTimeValidityPeriod` , so the staleness window cannot be set beyond a defensible maximum.

## RESOLUTION

Status: Acknowledged.

## INFORMATIONAL I-01

ACKNOWLEDGED

# Misleading OAppIdMismatch error comment

CATEGORY	Documentation
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigErrors.daml:84-90
REVIEW	Remediation Review 1

## DESCRIPTION

The doc comment on `errOneSig_OAppIdMismatch` states the OApp identity is read "from the `IAllowlist` view" and that "the signed allowlist change cannot be applied to it," implying the error is allowlist-specific. In fact the error is raised by `assertConfigMatchesOApp` (`IOAppConfig/Utls.daml:20`), which is invoked by six validators: allowlist, rate-limiter, local-transfer-pause, cross-chain-pause, `IOAppConfig`, and the IOApp create-request path (confirmed across `IOApp/Validate.daml`, `IOAppConfig/Utls.daml`, `IOAppConfig/Fetch.daml`). The allowlist-specific wording is stale for a general-purpose OApp-identity guard, and a maintainer reading it could wrongly assume the check only protects allowlist changes.

## RECOMMENDATION

Generalize the comment on `errOneSig_OAppIdMismatch` to state that the error covers any OApp-scoped config or request whose view OApp identity does not match the signed `oappId` — not just allowlist changes — so it accurately reflects the six call sites that raise it.

## RESOLUTION

Status: Acknowledged.

## INFORMATIONAL I-02

ACKNOWLEDGED

# OneSigConfig observers see full config

CATEGORY	Trust Assumptions
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSigConfig.daml:41-50
REVIEW	Remediation Review 1

## DESCRIPTION

`OneSigConfig` declares `observer` executors, `observers`, where `observers` is a free-form `[Party]` list set at create time. Every party in that list can read the entire contract payload, including `signers`, `quorum`, `seed`, `executors`, and `nonce`.

The list is fixed at creation — there is no `OneSigConfig_SetObservers` choice — and it is preserved across every self-management recreate, so the disclosure is permanent for the life of the instance.

This is the admin's own instance data, and disclosing `seed` does not enable forgery (that requires the `secp256k1` signer private keys). However, a party added only for monitoring/observability gains full visibility of the multisig security parameters — the signer set, the `quorum` threshold, and the revocation `seed` — which a deployment may not intend to expose to observers.

## RECOMMENDATION

If observation is only meant for liveness/monitoring, separate observation from full disclosure: expose a minimal view (e.g. instance id and status) to monitoring parties rather than adding them to the

`OneSigConfig` `observers` set. Otherwise, document explicitly that any party placed in `observers` is trusted with the full signer set, `quorum`, and revocation `seed`.

## RESOLUTION

Status: Acknowledged.

## INFORMATIONAL I-01

RESOLVED

# Signing zero fails a mandatory positive transfer

CATEGORY	Compatibility
LOCATION	contracts/protocol/canton/contracts/Layerzero/OneSig/daml/OneSig/I0App/Call.daml:219-228; contracts/protocol/canton/contracts/Layerzero/FeeUtils/daml/FeeUtils.daml:43-45
REVIEW	Remediation Review 3

## DESCRIPTION

`DiscoveryRegistry` supports registries whose `minFee` is zero: its `SetUrl` choice performs fee validation and transfer only when `minFee > 0`. The direct `OApp_SetDiscoveryUrl` route can therefore update such a registry with no holdings supplied for the fee.

The new OneSig route does not preserve that behavior. Before it calls `OApp_SetDiscoveryUrl`, `executeOAppDiscovery` always calls `transferAmulet` with the signed `feeAmount`. `transferAmulet` requires its amount to be strictly positive. Consequently, when the quorum signs the exact fee required by a registry whose fee is zero—`feeAmount = 0`—execution aborts at the preliminary transfer from executor to admin before the registry can process the URL update.

The failed Daml transaction leaves the discovery URL unchanged. The quorum cannot execute the supported update through OneSig with the correct zero value. A native administrator can still use the direct route. This makes the defect a narrow compatibility and operational liveness issue rather than an asset loss issue.

## RECOMMENDATION

Handle the zero fee case before calling `transferAmulet`. Fetch and validate the selected registry. When its current `minFee` is zero, skip the preliminary transfer and pass an empty holdings list to `OApp_SetDiscoveryUrl`. Preserve the positive transfer path only when a positive registry fee is actually due.

## RESOLUTION

Status: Resolved.

## § 05

# Disclaimer.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian's position is that each company and individual are responsible for their own due diligence and continuous security. Guardian's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract's safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

§ 06

## About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

**\$45B+**DIGITAL ASSETS UNDER  
REVIEW**200+**

ENGAGEMENTS COMPLETED

**500+**CRITICAL & HIGH FINDINGS  
REPORTED

Reach us at [audits@guardianaudits.com](mailto:audits@guardianaudits.com).