

LAYERZERO · SECURITY REVIEW

LayerZero Canton DAML.

A security review of Canton DAML.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
LayerZero	Canton DAML	Roberto Reigada, Felipe Gomez	March 17 – April 3, 2026	April 3, 2026

§ 00

Contents.

103 pages: overview, confidence ranking, scope, 71 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
	Files in Scope	P. 06
§ 04	Findings	P. 09
	Main Review · 71 findings	P. 12
§ 05	Disclaimer	P. 102
§ 06	About Guardian	P. 103

§ 01

Overview.

Guardian reviewed Canton DAML across a single review round. The final report includes 1 High, 14 Medium, 36 Low, 20 Informational findings.

AUDIT FIRM	REPOSITORY
Guardian	https://github.com/GuardianOrg/canton-ver-team1-1773698451456 ,
CLIENT	https://github.com/LayerZero-Labs/canton-ver
PROJECT	METHODOLOGY
Canton DAML	Smart contract · Manual · Main Review
PREPARED BY	MAIN REVIEW COMMIT
Roberto Reigada, Felipe Gomez	826c190968e2
AUDIT WINDOW	
March 17 – April 3, 2026	
FINAL REPORT	
April 3, 2026	

1.1 Confidence Ranking

4 / 5

High Confidence

Guardian assigns a Confidence Ranking of 4 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	1	0	0	0	1
Medium	14	9	1	1	3
Low	36	33	2	0	1
Informational	20	12	8	0	0
All severities	71	54	11	1	5

§ 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	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 ASSIGNED	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 Canton DAML in

<https://github.com/GuardianOrg/canton-ver-team1-1773698451456>,

<https://github.com/LayerZero-Labs/canton-ver>. The scope includes 7,703 source lines across 98 files.

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.

Files in Scope 1/3.

FILE	NSLOC	TOTAL
daml/Layerzero/Common/daml/AppUID.daml	16	19
daml/Layerzero/Common/daml/MathUtils.daml	45	48
daml/Layerzero/Common/daml/MultiSig/MultiSig.daml	156	167
daml/Layerzero/Common/daml/MultiSig/MultiSigErrors.daml	41	53
daml/Layerzero/Common/daml/MultiSig/Utils.daml	74	85
daml/Layerzero/Common/daml/Serde/Serde.daml	133	153
daml/Layerzero/Common/daml/Serde/SerdeErrors.daml	60	80
daml/Layerzero/DiscoveryRegistry/daml/DiscoveryRegistry.daml	63	79
daml/Layerzero/DiscoveryRegistry/daml/DiscoveryRegistry/Errors.daml	9	11
daml/Layerzero/Executor/daml/Executor.daml	142	152
daml/Layerzero/FeeUtils/daml/FeeUtils/Constants.daml	16	22
daml/Layerzero/FeeUtils/daml/FeeUtils/FeeUtils.daml	328	361
daml/Layerzero/FeeUtils/daml/FeeUtils/FeeUtilsErrors.daml	44	58
daml/Layerzero/Governance/daml/Errors.daml	29	38
daml/Layerzero/Governance/daml/Governance.daml	330	365
daml/Layerzero/IAccessControl/daml/IAccessControl.daml	25	30
daml/Layerzero/IMinFee/daml/IMinFee.daml	22	26
daml/Layerzero/IOApp/daml/ILockedPayout.daml	38	46
daml/Layerzero/IOApp/daml/IOApp.daml	152	166
daml/Layerzero/IOApp/daml/Types.daml	26	29
daml/Layerzero/IRateLimiter/daml/IRateLimiterConfig.daml	69	78
daml/Layerzero/IRateLimiter/daml/IRateLimiterErrors.daml	3	4
daml/Layerzero/IRateLimiter/daml/IRateLimitState.daml	97	106
daml/Layerzero/IRateLimiter/daml/RateLimiterConfigTypes.daml	100	113
daml/Layerzero/IRequest/daml/Call.daml	149	173
daml/Layerzero/IRequest/daml/CallErrors.daml	45	65
daml/Layerzero/IRequest/daml/IPendingCallback.daml	34	39
daml/Layerzero/IRequest/daml/IRequest.daml	56	64
daml/Layerzero/IRequest/daml/IRequestCallback.daml	34	41
daml/Layerzero/IRequestFactory/daml/IRequestFactory.daml	74	82
daml/Layerzero/IRequestFactory/daml/Keys.daml	20	26
daml/Layerzero/IToken/daml/IAllowlist.daml	100	116
daml/Layerzero/IToken/daml/ILocalTransferPause.daml	41	49
daml/Layerzero/IToken/daml/ITransferRule.daml	43	51
daml/Layerzero/Oft/daml/Allocation/AllocationImpl.daml	44	52
daml/Layerzero/Oft/daml/Allocation/OftAllocation.daml	76	90

Files in Scope 2/3.

FILE	NSLOC	TOTAL
daml/Layerzero/Oft/daml/BurnMint/BurnMintImpl.daml	40	47
daml/Layerzero/Oft/daml/Common/ConfigUtils.daml	49	55
daml/Layerzero/Oft/daml/Common/Constants.daml	45	56
daml/Layerzero/Oft/daml/Common/Errors.daml	124	179
daml/Layerzero/Oft/daml/Common/Lock.daml	29	38
daml/Layerzero/Oft/daml/Common/RoleConstants.daml	22	32
daml/Layerzero/Oft/daml/Common/Utils.daml	176	200
daml/Layerzero/Oft/daml/OApp/Callback/Dispatcher.daml	130	143
daml/Layerzero/Oft/daml/OApp/Constants.daml	11	14
daml/Layerzero/Oft/daml/OApp/CostAsserts.daml	136	150
daml/Layerzero/Oft/daml/OApp/CostAssertsConstants.daml	34	43
daml/Layerzero/Oft/daml/OApp/CostAssertsErrors.daml	28	41
daml/Layerzero/Oft/daml/OApp/EnforcedOptions.daml	103	117
daml/Layerzero/Oft/daml/OApp/EnforcedOptionsConstants.daml	14	18
daml/Layerzero/Oft/daml/OApp/EnforcedOptionsErrors.daml	6	8
daml/Layerzero/Oft/daml/OApp/Keys.daml	5	6
daml/Layerzero/Oft/daml/OApp/LzReceive.daml	64	74
daml/Layerzero/Oft/daml/OApp/LzReceive/Call/ContextBuilder.daml	66	69
daml/Layerzero/Oft/daml/OApp/LzReceive/Call/Impl.daml	67	76
daml/Layerzero/Oft/daml/OApp/LzReceive/Call/Keys.daml	38	50
daml/Layerzero/Oft/daml/OApp/LzReceive/Callback/Accept.daml	34	38
daml/Layerzero/Oft/daml/OApp/LzReceive/Callback/ContextBuilder.daml	56	59
daml/Layerzero/Oft/daml/OApp/LzReceiveErrors.daml	8	10
daml/Layerzero/Oft/daml/OApp/LzSend/Call/ContextBuilder.daml	71	75
daml/Layerzero/Oft/daml/OApp/LzSend/Call/Impl.daml	263	308
daml/Layerzero/Oft/daml/OApp/LzSend/Call/Keys.daml	64	84
daml/Layerzero/Oft/daml/OApp/LzSend/Call/Parser.daml	114	127
daml/Layerzero/Oft/daml/OApp/LzSend/Callback/Accept.daml	83	94
daml/Layerzero/Oft/daml/OApp/LzSend/Callback/ContextBuilder.daml	73	77
daml/Layerzero/Oft/daml/OApp/LzSend/Callback/Parser.daml	42	48
daml/Layerzero/Oft/daml/OApp/LzSend/Callback/Reject.daml	34	38
daml/Layerzero/Oft/daml/OApp/LzSend/Callback/Shared.daml	53	62
daml/Layerzero/Oft/daml/OApp/LzSend/Errors.daml	34	49
daml/Layerzero/Oft/daml/OApp/MessageCodec.daml	133	157
daml/Layerzero/Oft/daml/OApp/MessageCodecErrors.daml	31	41
daml/Layerzero/Oft/daml/OApp/OAppImpl.daml	30	31

Files in Scope 3/3.

FILE	NSLOC	TOTAL
daml/Layerzero/Oft/daml/OApp/OptionsDecoder.daml	182	203
daml/Layerzero/Oft/daml/OApp/Payout.daml	63	71
daml/Layerzero/Oft/daml/Oft.daml	83	98
daml/Layerzero/Oft/daml/OftFactory.daml	317	388
daml/Layerzero/Oft/daml/OftFactoryConfig.daml	368	423
daml/Layerzero/Oft/daml/OftFactoryRbac.daml	49	60
daml/Layerzero/Oft/daml/Transfer/OftTransferOffer.daml	137	158
daml/Layerzero/Oft/daml/Transfer/OftTransferRule.daml	96	116
daml/Layerzero/Oft/daml/Transfer/TransferImpl.daml	75	87
daml/Layerzero/RateLimiter/daml/RateLimitErrors.daml	32	37
daml/Layerzero/RateLimiter/daml/RateLimiterState.daml	361	398
daml/Layerzero/RateLimiter/daml/RateLimiterStateHelpers.daml	77	85
daml/Layerzero/RateLimiter/daml/RateLimitTypes.daml	124	134
daml/Layerzero/Registry/daml/Registry.daml	97	119
daml/Layerzero/Registry/daml/Registry/Errors.daml	14	18
daml/Layerzero/Registry/daml/Registry/Utils.daml	15	17
daml/Layerzero/Request/daml/Errors.daml	32	42
daml/Layerzero/Request/daml/Events.daml	31	35
daml/Layerzero/Request/daml/PendingCallback.daml	74	85
daml/Layerzero/Request/daml/Request.daml	96	111
daml/Layerzero/RequestFactory/daml/RequestFactory.daml	135	154
daml/Layerzero/RequestFactory/daml/RequestFactoryConfig.daml	38	44
daml/Layerzero/StateTransition/daml/Committer/Committer.daml	42	48
daml/Layerzero/StateTransition/daml/Committer/CommitterErrors.daml	8	10
daml/Layerzero/StateTransition/daml/Committer/CommitterImpl.daml	95	109
daml/Layerzero/StateTransition/daml/StateCommitment/StateCommitment.daml	18	21

§ 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					
H-01	LzReceive overflow after request acceptance	Logical Error	HIGH	RESOLVED	p. 12
M-01	Rate-limiter config changes bypass checkpointing	Unexpected Behavior	MEDIUM	PARTIALLY RESOLVED	p. 14
M-02	One member can freeze governance	DoS	MEDIUM	RESOLVED	p. 16
M-03	Leader-held fee can freeze governance	Trust Assumptions	MEDIUM	PENDING	p. 17
M-04	Pending outflow does not freeze net accounting	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 19
M-05	Executor refunds leftover fees to Poor	Unexpected Behavior	MEDIUM	RESOLVED	p. 21
M-06	Dust-sized fees can wedge governance cleanup	DoS	MEDIUM	RESOLVED	p. 22
M-07	Net accounting overcredits opposite-side usage	Unexpected Behavior	MEDIUM	PENDING	p. 23
M-08	accepted OApp requests can run forged callbacks	Unexpected Behavior	MEDIUM	PENDING	p. 24
M-09	PendingCallback allows same-app callback swap	Validation	MEDIUM	PENDING	p. 26
M-10	Missing hint falls back to LZ recipient	Unexpected Behavior	MEDIUM	PENDING	p. 27
M-11	Odd composeMsg rewrites LZ payload on wire	Unexpected Behavior	MEDIUM	PENDING	p. 29
M-12	Fake TransferFactory can spoof payouts	Unexpected Behavior	MEDIUM	PENDING	p. 31
M-13	Dust governance fees brick cleanup	DoS	MEDIUM	PENDING	p. 33
M-14	Slash fee rounding bricks cleanup	Rounding	MEDIUM	PENDING	p. 35
L-01	Explicit treasury payout can undercut minFee	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 36
L-02	Non-ASCII instrument IDs break AppUID derivation	Validation	LOW	RESOLVED	p. 37
L-03	Payout expiry permanently locks tokens	Logical Error	LOW	ACKNOWLEDGED	p. 39

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-04	feeDeposit cannot send when issuer fee is set	Unexpected Behavior	LOW	PENDING	p. 40
L-05	OApp request validation ignores endpoint address	Validation	LOW	PENDING	p. 41
L-06	Zero minFee bricks registry-style writes	DoS	LOW	PENDING	p. 42
L-07	Missing validation on CostAsserts	Validation	LOW	PENDING	p. 43
L-08	DAML IzReceive lacks replay protection	Validation	LOW	PENDING	p. 44
L-09	DelegateLzReceive hands live fee to IOApp	Trust Assumptions	LOW	PENDING	p. 45
L-10	IzReceive accepts zero-amountSD messages	Validation	LOW	PENDING	p. 46
L-11	Unbounded receiver list in IzSend	Validation	LOW	PENDING	p. 47
L-12	issuerFeeBps=10000 bricks IzSend	Validation	LOW	PENDING	p. 48
L-13	Governance role separation not enforced	Validation	LOW	PENDING	p. 49
L-14	Enforced options bypass via composeMsg	Validation	LOW	PENDING	p. 50
L-15	PendingCallback finalizer is not bound	Trust Assumptions	LOW	PENDING	p. 51
L-16	Invalid accept callback can silently no-op	Unexpected Behavior	LOW	PENDING	p. 52
L-17	Malformed hints can brick registry resolution	Validation	LOW	PENDING	p. 53
L-18	AppUID hashes differ for extended-byte ids	Validation	LOW	PENDING	p. 54
L-19	Force-accept rechecks live local pause	Validation	LOW	PENDING	p. 55
L-20	Multi-registry lookup is first-match-wins	Unexpected Behavior	LOW	PENDING	p. 56
L-21	Deploy script defaults registry fee to zero	Warning	LOW	PENDING	p. 58
L-22	Cost asserts underprice executor fees	Validation	LOW	PENDING	p. 59
L-23	odd-length options fail at the ULN boundary	Unexpected Behavior	LOW	PENDING	p. 61
L-24	Cost assertions reject valid legacy ULN options	Validation	LOW	PENDING	p. 62
L-25	outboundLimit=0 blocks all outbound transfers	Logical Error	LOW	PENDING	p. 64
L-26	BurnMint has no supply conservation	Validation	LOW	PENDING	p. 65
L-27	High scaleDecimals disables rate limiter	Configuration	LOW	PENDING	p. 66
L-28	amuletContext steering wedges settlement	Validation	LOW	PENDING	p. 67
L-29	Transfer silently truncates amounts	Validation	LOW	PENDING	p. 68
L-30	Malformed enforcedOptions brick sends	Validation	LOW	PENDING	p. 69
L-31	RequestFactory accepts foreign-gateway IOApp	Validation	LOW	PENDING	p. 70
L-32	Request has no timeout or sender cancel	DoS	LOW	PENDING	p. 71

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-33	Unchecked RequestFactory Anchor	Validation	LOW	PENDING	p. 72
L-34	Treasury fee blocks governance cleanup	Access Control	LOW	PENDING	p. 73
L-35	Direct sender refunds revert below create fee	Unexpected Behavior	LOW	PENDING	p. 74
L-36	DiscoveryRegistry stores one unbounded URL map	Unexpected Behavior	LOW	PENDING	p. 75
I-01	Fingerprint-only identity is not party-unique	Unexpected Behavior	INFORMATIONAL	PENDING	p. 76
I-02	FinalizeRequest trusts settlement-time context	Trust Assumptions	INFORMATIONAL	ACKNOWLEDGED	p. 78
I-03	Settlement can use caller-chosen live instances	Trust Assumptions	INFORMATIONAL	ACKNOWLEDGED	p. 80
I-04	State-commitment chain is not canonical	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 81
I-05	Blacklist-after-creation bricks payouts	Logical Error	INFORMATIONAL	ACKNOWLEDGED	p. 82
I-06	stateRoot accepts any hex length	Validation	INFORMATIONAL	PENDING	p. 83
I-07	Registry SetPartyId accepts mismatched hints	Validation	INFORMATIONAL	ACKNOWLEDGED	p. 84
I-08	External payouts remain sender-cancellable	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 86
I-09	DEFAULT_ADMIN_ROLE exceeds documented RBAC	Warning	INFORMATIONAL	PENDING	p. 88
I-10	SEND_AND_CALL compose payload is ignored	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 89
I-11	Admin paths bypass holding lock invariants	Validation	INFORMATIONAL	ACKNOWLEDGED	p. 90
I-12	Batch commitments leak request linkage	Warning	INFORMATIONAL	PENDING	p. 92
I-13	State commitments are not isolated per committer	Warning	INFORMATIONAL	PENDING	p. 93
I-14	DiscoveryRegistry accepts foreign gateway IOApps	Validation	INFORMATIONAL	PENDING	p. 95
I-15	Uppercase OX options fail TS bytes schema	Validation	INFORMATIONAL	PENDING	p. 96
I-16	RequestFactory allows handler==treasury	Access Control	INFORMATIONAL	PENDING	p. 97
I-17	Governance member list validation gaps	Validation	INFORMATIONAL	PENDING	p. 98
I-18	SetTransferRule accepts unchecked ITransferRule	Validation	INFORMATIONAL	PENDING	p. 99
I-19	MINTER_ROLE and BURNER_ROLE are never enforced	Trust Assumptions	INFORMATIONAL	PENDING	p. 100
I-20	DEFAULT_ADMIN_ROLE need separate disclosure	Trust Assumptions	INFORMATIONAL	PENDING	p. 101

HIGH **H-01**

RESOLVED

LzReceive overflow after request acceptance

CATEGORY Logical Error
 LOCATION LzReceive.daml
 REVIEW Main Review

DESCRIPTION

Inbound receive amounts are accepted into the request flow before the protocol proves that they can be represented in local token units. The OFT balance model stores token amounts as signed `Int`, and cross-chain receive finalization converts the decoded shared-decimal amount into local decimals by multiplying it by $10^{(\text{localDecimals} - \text{sharedDecimals})}$. The receive request path validates that the encoded amount field is syntactically well formed and fits the signed `Int64` parser, but it does not validate that this later scaling step will also fit inside `Int64`.

That omission matters because request creation, request acceptance, and callback finalization are split across separate transactions. A relayed message can therefore be admitted into the system, the handler can accept the request, the original request can be consumed and only then can callback finalization fail when the receive path performs the local-decimal conversion. At that point the receiver is not minted any tokens, but the accepted request has already been archived and the system is left with a live `PendingCallback` that will deterministically fail again on every normal retry because the overflowing message payload is unchanged.

The arithmetic itself is straightforward. If `sharedDecimals` is lower than `localDecimals`, the receive path computes:

```
``plain text
conversionRate = 10^(localDecimals - sharedDecimals)
amountLD = amountSD * conversionRate
`[
```

Any message with `amountSD > maxInt64 / conversionRate` is therefore guaranteed to revert during callback finalization. The danger is especially sharp under the repository's own default deployment assumptions, which set `localDecimals = 18` for OFT holdings and `sharedDecimals = 6` for the OFT adapter. Under that configuration the conversion rate is 10^{12} , so the largest shared-decimal amount that still fits is `9,223,372`. The next value, `9,223,373`, overflows. Expressed in whole-token terms, that boundary is only about `9.223372` tokens. In other words, a normal-looking inbound transfer from an external peer can be accepted into the workflow and still become impossible to finalize.

This behavior was confirmed with a standalone PoC test that exercises the real

H-01 LzReceive overflow after request acceptance

CONTINUED

`OApp_LzReceive -> IRequest_Accept -> IPendingCallback_FinalizeRequest` flow. The PoC proves the exact boundary condition: `amountSD = 9,223,372` finalizes successfully for an `18/6` deployment, while `amountSD = 9,223,373` is accepted into the request flow but fails only during callback finalization, leaving one live pending callback and zero minted holdings for the receiver.

The issue is not limited to the `18/6` combination, although that configuration makes it far more likely in practice. With `8/6`, the threshold rises to roughly `92.23` billion whole tokens, so the same bug becomes harder to trigger accidentally. With `6/6`, there is no scaling multiplication and this specific overflow path disappears. The protocol therefore has a configuration-sensitive liveness bug whose exploitability ranges from latent to immediate depending on decimals.

Because of this issue, a stuck inbound settlement after the request has already been accepted. For affected messages, the normal callback path cannot complete, the receiver never receives the minted balance, and operational recovery is pushed onto manual intervention or out-of-band remediation. In deployments that use the default `18/6` settings, this can be triggered by realistic cross-chain transfer sizes rather than by an obviously absurd payload.

— RECOMMENDATION

Add an explicit upper-bound check before the request is accepted, and ideally as early as request creation, so that any message whose scaled local-decimal amount would overflow is rejected before the protocol consumes the request and creates an irreversible liveness problem. The bound is simple:

```
``plain text
```

```
amountSD <= maxInt64 / 10^(localDecimals - sharedDecimals)
```

```
`[
```

Keep the same validation in the callback path as defense in depth so that malformed or legacy requests still fail with a clear domain-specific error instead of a generic arithmetic exception. If `18` local decimals must remain supported, the stronger long-term fix is to stop storing token balances in signed `Int64` and migrate the OFT amount model to a wider integer representation or another encoding that can safely represent standard ERC20-sized balances.

— RESOLUTION

Status: Resolved.

MEDIUM **M-01**

PARTIALLY RESOLVED

Rate-limiter config changes bypass checkpointing

CATEGORY	Unexpected Behavior
LOCATION	OfFactoryConfig.daml, IRateLimiterConfig.daml, RateLimiterState.daml, RateLimiterStateHelpers.daml
REVIEW	Main Review

— DESCRIPTION

The rate limiter was clearly designed around explicit checkpointing. `RateLimiterState` stores accumulated usage, while `IRateLimiterConfig` defines the limits, windows, and bucket-selection rules that tell the system how to interpret that usage. The codebase already includes dedicated checkpoint choices because recorded usage must be decayed under the old rules before a new configuration takes effect.

Despite that design, `OfFactoryConfig` still exposes direct config mutation paths that rewrite the live rate-limiter configuration without checkpointing the current state first. An authorized admin can change default limits, endpoint overrides, or `useGlobalState` in place while the previously accumulated usage remains untouched. From that moment on, the same stored usage is being read under rules that did not exist when that usage was recorded.

For example, imagine an endpoint bucket is almost exhausted under one override and the admin flips `useGlobalState` or replaces that override directly on the config contract. Future checks can start reading from a different bucket layout, or even from a fresh bucket at `eid = 0`, without ever migrating the old usage into that new layout. The reverse is just as problematic: tightening a window or lowering a limit in place can make historical traffic look more consumed than it should. In both directions, enforcement starts depending on how the config was edited rather than on the actual transfer history.

This is not an unprivileged attack because the caller still needs admin authority. The problem is that the admin interface bypasses the rate limiter's own accounting model. Once that happens, the limiter can grant extra headroom, strand legitimate capacity, or otherwise misstate current utilization.

— RECOMMENDATION

M-01 Rate-limiter config changes bypass checkpointing

CONTINUED

Any change that reinterprets historical usage should go through the checkpointed state transition path, not through a direct config edit. The direct `IRateLimiterConfig` choices should either be removed or restricted to fields that do not affect decay semantics or bucket selection. `useGlobalState` in particular should not change in place unless the protocol performs an explicit migration that rewrites the existing usage into the new bucket layout before the flag becomes active.

— RESOLUTION

Status: Partially resolved.

MEDIUM **M-02**

RESOLVED

One member can freeze governance

CATEGORY DoS
LOCATION Governance.daml
REVIEW Main Review

— DESCRIPTION

The governance module describes proposal changes as a committee process, but `Governance_Accept` and `Governance_Reject` are both controlled by the full `members` list. In practice that means proposal finalization is unanimous. At the same time, `Governance_Propose` only requires `expiry > now` and does not impose any upper bound on the proposal lifetime. A single member can therefore create a proposal with a far-future expiry and then refuse to co-sign either acceptance or rejection.

That proposal remains pending for as long as the chosen expiry allows. While it is pending, the contract refuses new proposals and also blocks `Governance_SelectLeader`. The result is that one committee member can lock the governance process for an arbitrarily long period without needing to break any authorization rule or compromise any other participant. If governance is supposed to be majority driven, the current design does not implement that property. Instead, it gives every member a unilateral veto plus the ability to turn that veto into a full governance freeze by choosing a distant expiry.

— RECOMMENDATION

Implement a real majority-based finalization rule for proposals and stop requiring unanimous participation for ordinary accept and reject flows. Proposal lifetime should also be bounded so that no submitter can create a practically permanent pending state. At minimum, add a majority-driven cancel or reject path that can clear a stalled proposal without requiring the cooperation of every member.

— RESOLUTION

Status: Resolved.

MEDIUM **M-03**

PENDING

Leader-held fee can freeze governance

CATEGORY Trust Assumptions

LOCATION Governance.daml

REVIEW Main Review

DESCRIPTION

`Governance_Propose` transfers the proposal fee into a normal `Amulet` owned by `leader` and stores only that contract id inside `ProposalData`. The later cleanup paths all depend on that exact live contract staying available. `Governance_Accept` fetches it to refund the proposer, `Governance_Reject` and `Governance_Expire` slash it through `slashProposalFee` and even the auto-slash path inside a later `Governance_Propose` depends on the same stored id.

This is especially problematic because the leader is not described as a special trusted escrow role. The contract says the leader must be a committee member and the module-level documentation says governance is supposed to be a committee process where a majority of vApps agrees on changes. In other words, the leader is presented as one participant inside a majority-driven governance system, not as an authority who is allowed to unilaterally break liveness by moving proposal funds.

That is unsafe because the fee is not escrowed under governance control. It is an ordinary owner-controlled `Amulet`, so the leader can consume it, merge it, or otherwise move it outside the governance workflow at any time after proposal creation. Once that happens, every later path that tries to fetch the stored `amuletCid` reverts. The proposal then remains pending, new proposals are blocked and `Governance_SelectLeader` is also blocked while `proposal != None`. A malicious or compromised leader can therefore freeze governance indefinitely while retaining custody of the fee.

Suppose Alice is the current leader and Bob proposes a new committee configuration.

`Governance_Propose` transfers Bob's fee into an `Amulet` owned by Alice and stores that exact contract id in the pending proposal. Before Bob and Charlie can finish governance, Alice performs an ordinary self-transfer of that fee amulet, which archives the stored contract and recreates the same value under a fresh contract id that only she controls. From that point forward, `Governance_Accept` fails because it tries to refund from the archived id, `Governance_Reject` fails because `slashProposalFee` fetches the same archived id and `Governance_Expire` fails for the same reason after the deadline. The proposal never clears, Bob cannot submit a replacement proposal and Bob plus Charlie cannot rotate Alice out because `Governance_SelectLeader` is blocked while a proposal is pending. The majority is therefore locked out by a single leader-controlled asset movement.

M-03 Leader-held fee can freeze governance

CONTINUED

Because of this, the leader-controlled amulet is enough to block a proposal resolution and leader rotation for the entire committee.

— RECOMMENDATION

Do not store the proposal fee in a leader-controlled `Amulet`. Move the fee into an escrow or lock whose spend paths are governed by the contract logic rather than by the current leader. If the fee must remain transferable, store enough state to reconstruct or recover it without depending on one leader-owned contract id remaining live for the whole proposal lifecycle.

— RESOLUTION

Status: Pending.

MEDIUM **M-04**

ACKNOWLEDGED

Pending outflow does not freeze net accounting

CATEGORY Unexpected Behavior
LOCATION RateLimiterState.daml
REVIEW Main Review

DESCRIPTION

`RecordOutflow` freezes `scaledAmount`, `resolvedEid` and `forwardRecorded`, but it does not freeze whether the later accept path should apply net accounting. `CommitOutflow` reads the live config again and decides on the spot whether inbound usage should be reduced based on the current `inboundEnabled` and `netAccountingEnabled` flags.

That means a pending send can be admitted under one policy and finalized under another. If net accounting is turned on after `RecordOutflow`, an old request can retroactively erase inbound usage and reopen capacity that did not exist when the send was created. If it is turned off before finalization, the opposite happens and the system keeps inbound usage that should have been offset.

Because of this, a rate-limit manager can record a large outbound transfer, then change the EID policy through the intended checkpoint flow before the callback is finalized. The checkpoint keeps decay math consistent, but `CommitOutflow` still re-reads the live `inboundEnabled` and `netAccountingEnabled` flags for that already-pending send. If the manager turns net accounting on, the old transfer now clears inbound usage and frees room for new traffic that was not available when the request was admitted. If the manager turns it off, the pending transfer no longer frees inbound capacity when it settles. Usage that should have been offset stays on the bucket, so later inbound transfers can be blocked even though that capacity would have been released under the earlier policy.

RECOMMENDATION

Freeze the net-accounting decision in `RecordOutflow` and carry it in the callback payload alongside `scaledAmount`, `resolvedEid` and `forwardRecorded`. `CommitOutflow` should use that frozen decision instead of rereading `inboundEnabled` and `netAccountingEnabled` from the live config. If config changes are only meant to affect new transfers, pending outflows must always settle under the policy that existed when they were recorded.

M-04 Pending outflow does not freeze net accounting

CONTINUED

— **RESOLUTION**

Status: Acknowledged.

MEDIUM **M-05**

RESOLVED

Executor refunds leftover fees to Poor

CATEGORY Unexpected Behavior

LOCATION Request.daml

REVIEW Main Review

DESCRIPTION

`Request` settlement sends leftover escrow to `this.sender`. That is fine for normal requests, but it is wrong for executor-backed inbound requests because the executor flow creates the request with `sender = poor` even though `rich` funded the fee.

`DelegateLzReceive` transfers `feeAmount` from `rich` to `poor` and calls `OApp_LzReceive` as `poor`. That value is carried into `Request_Create`, so the stored request sender is `poor`. Later, both `request_AcceptImpl` and `request_RejectImpl` call `distributeAmulet` with `this.sender` as the refund receiver. Any remainder after `minFee` and explicit payouts is therefore paid to `poor`, not back to `rich`. This breaks the Rich/Poor isolation model. `poor` is supposed to be a disposable identity with no funds, but accepted or rejected inbound requests can leave operator-funded Amulet on that party. If `poor` is compromised or handled with weaker controls because it was assumed valueless, the refunded balance can be lost.

RECOMMENDATION

Store the refund recipient separately from `sender`. Direct requests can keep `sender` and refund receiver equal, but `DelegateLzReceive` should create the request with `sender = poor` and refund receiver = `rich`. Both settlement paths should pass that refund receiver to `distributeAmulet` instead of `this.sender`.

RESOLUTION

Status: Resolved.

MEDIUM **M-06**

RESOLVED

Dust-sized fees can wedge governance cleanup

CATEGORY DoS
LOCATION Governance.daml
REVIEW Main Review

DESCRIPTION

`slashProposalFee` divides the escrowed proposal fee equally across all non-proposer members with raw `Decimal` division and then passes those payouts to `distributeAmulet`. That is unsafe for dust-sized but still valid fees.

The split helper assumes the fee can always be represented as an equal, strictly positive payout for every recipient. That is not guaranteed. `distributeAmulet` requires every payout to be `> 0.0` and also requires the total payouts to stay within the escrowed amulet amount. With Daml `Decimal` fixed at 10 decimal places, some legal fees are too small to satisfy both conditions at once. With three committee members, for example, a fee of `0.0000000001` must be split across two recipients, but no equal split can produce two strictly positive representable payouts without exceeding the input amount.

This is a liveness issue as the bad fee is accepted by `Governance_Propose` and only fails later during resolution. `Governance_Reject`, `Governance_Expire`, the expired auto-slash in `Governance_Propose`, and the cleanup path in `Governance_SetMembers` all depend on the same slashing helper. Before expiry the multisig can still avoid the slash path by accepting the proposal, but after expiry the normal governance cleanup flow can be wedged.

RECOMMENDATION

Do not compute slash payouts with raw `Decimal` division. Convert the escrowed fee into base units at `10-10` precision, compute the per-recipient share with integer division, require the share to be non-zero, and leave any remainder in the treasury. That guarantees every payout is representable and prevents the total from exceeding the available amount.

As a secondary guard, reject `minFee` values that are too small for the current committee size. Regression tests should cover dust fees, expiry cleanup, auto-slash during repropose, and `Governance_SetMembers` with an active proposal.

RESOLUTION

Status: Resolved.

MEDIUM **M-07**

PENDING

Net accounting overcredits opposite-side usage

CATEGORY Unexpected Behavior
LOCATION RateLimiterState.daml
REVIEW Main Review

DESCRIPTION

`applyRateLimit` first rounds the transfer amount up into `scaledAmount`, then uses that same rounded-up value not only to charge the forward direction but also to reduce usage in the opposite direction when net accounting is enabled. `commitOutflow` applies the same pattern again for accepted outflows. This means the credit side is not conservative: a very small transfer can erase one full scaled unit of opposite-side usage.

This becomes exploitable in the OFT flow because transfers are already quantized to the shared-decimal boundary by `removeDust`, while the limiter can be configured with a coarser `scaleDecimals`. If `scaleDecimals` is larger than `localDecimals - sharedDecimals`, a minimum nonzero transfer still rounds up to 1 scaled unit and can cancel a full unit of opposite-direction usage. An attacker can then use dust-sized balancing transfers to keep clearing inbound or outbound usage far faster than the real opposite-side flow would justify.

RECOMMENDATION

Do not use the same rounded-up `scaledAmount` for opposite-direction credits. Keep forward charging in scaled units if needed, but track net-accounting credits with exact raw amounts or with a separate remainder accumulator so small transfers only offset the usage they actually represent. At minimum, reject configurations where `netAccountingEnabled = True` and `scaleDecimals > localDecimals - sharedDecimals`.

RESOLUTION

Status: Pending.

MEDIUM **M-08**

PENDING

accepted OApp requests can run forged callbacks

CATEGORY Unexpected Behavior

LOCATION OftFactory.daml

REVIEW Main Review

DESCRIPTION

`OApp_CreateRequest` lets a `DEFAULT_ADMIN_ROLE` holder submit arbitrary `callContext` and arbitrary `callbackContext` and the request path never proves that the callback payload matches the accepted request. `RequestFactory` validates only the request-side `callContext`, stores the supplied `callbackContext` unchanged, and `Request` later copies it into `PendingCallback`. At finalization time, the OFT dispatcher reads that stored callback payload and routes on it as if it were canonical callback provenance.

That means request acceptance and callback execution are only loosely coupled. A delegated admin can get one request accepted by the validator, but make finalization execute a different callback family than the accepted request ever created on-chain. The sharpest case is a forged `LzReceive_callback`: the normal `OApp_LzReceive` entypoint checks `guid`, `origin` and peer consistency before a request is created, but the callback-side `handleLzReceiveAccept` path does not replay those checks. It just takes `lzMessage` from `callbackContext`, takes `receiverHint` from `executeContext` and calls `lzReceive`, which mints new OFT directly.

A realistic trigger under the current repository is an admin request that uses a harmless accepted `callContext`, but a malicious callback payload. The validator currently executes only the request `callContext` and builds write context with `ZERO_BYTES32_ADDRESS` sender data, so a delegated admin can submit `OApp_CreateRequest` with `callContext = EndpointV2.lzReceiveAlert(...)` and `callbackContext = LzReceive_callback(...)` carrying an attacker-chosen `lzMessage`. `lzReceiveAlert` is a benign endpoint write that only emits an event, so the request can be accepted without touching token state. Once accepted, `PendingCallback` is created with the forged callback payload. The attacker then finalizes it with a chosen `receiverHint` and the dispatcher runs the forged receive callback, minting OFT for the attacker-controlled receiver even though no canonical `OApp_LzReceive` request ever passed the normal peer and origin checks.

M-08 accepted OApp requests can run forged callbacks

CONTINUED

— RECOMMENDATION

Do not let `OApp_CreateRequest` inject free-form callback payloads into the OApp callback path. The smallest safe fix is to forbid non-empty `callbackContext` in `OApp_CreateRequest` and reserve callback creation for canonical flows like `OApp_LzSend` and `OApp_LzReceive`.

If generic admin requests must keep callback support, bind callback provenance on-ledger.

`RequestFactory` should validate that the supplied `callbackContext` is the canonical callback derived from the accepted `callContext`, or store a typed callback discriminator and callback data generated inside trusted code instead of accepting arbitrary nested maps from the caller.

— RESOLUTION

Status: Pending.

MEDIUM **M-09**

PENDING

PendingCallback allows same-app callback swap

CATEGORY Validation
LOCATION PendingCallback.daml
REVIEW Main Review

— DESCRIPTION

`PendingCallback` does not store which callback contract was approved when the request was accepted or rejected. At finalize time the caller supplies `callbackCid` and the implementation only checks that the chosen contract belongs to the same app and the same admin.

That does not preserve callback provenance. A visible finalizer can point settlement at a different

`IRequestCallback` contract for the same app/admin pair, even if the original request carried no callback intent.

For example, imagine a bridge app has two callback contracts under the same admin: one releases tokens to the intended receiver and another sends accepted transfers into a manual-review queue. A request is accepted and everyone expects the release callback to run. Under the current design, the finalizer can still supply the manual-review callback at finalize time. The request is still treated as accepted, but the follow-up action is no longer the one that was originally approved.

The result is that request authority is decided in two steps: the request is accepted first, then the actual callback target is picked later by whoever finalizes first. That is enough to redirect settlement side effects inside the same app boundary.

— RECOMMENDATION

Bind the callback target before the request becomes finalizable. Store the exact callback contract, or store an explicit marker that no callback is allowed and reject any finalize call that does not match that stored value.

If requests without callback intent are supported, they should not create a `PendingCallback` that can later be pointed at an arbitrary same-app callback.

— RESOLUTION

Status: Pending.

MEDIUM **M-10**

PENDING

Missing hint falls back to LZ recipient

CATEGORY Unexpected Behavior
LOCATION OftFactory.daml, executor-sdk.ts
REVIEW Main Review

DESCRIPTION

Before 0517a22, `ExecutorSdk.getReceiverHint` failed closed. A missing `Registry` contract or a missing `fingerprint -> hint` entry stopped callback preparation immediately and surfaced a clear off-chain error to the operator. This commit changes that behavior. `getReceiverHint` now scans all visible registries for the `appId` and returns `undefined` when it does not find a match.

That `undefined` value is not rejected later in the SDK. `prepareFinalizeCallback` passes it through and the execute-context builder only inserts `receiverHint` when the value is truthy. The standard wrapper flow in `scripts/src/test-sdk-wrappers.ts` now does exactly that: it calls `getReceiverHint`, logs the possibly missing value and still submits `prepareFinalizeCallback`.

Once the finalize request reaches the ledger without a `receiverHint`, the OFT callback dispatcher falls back to `_DEFAULT_RECEIVER_HINT = "LZ"`. `LzReceive` then reconstructs the actual settlement target as `LZ::1220<fingerprint>`. A normal registry miss is therefore no longer treated as a safe lookup failure. It becomes a settlement attempt against a synthetic or sibling party id, or a late finalize-time revert if that party id is not hosted.

This is a real regression introduced by commit 0517a22, not just a restatement of the older finalize-time hint-trust issue. The older bug was that on-ledger settlement trusted the supplied hint too much. This commit widens the reachability of that dangerous path by removing the off-chain fail-closed guard that used to stop finalization when registry resolution failed. This SDK change directly turns ordinary lookup misses into misrouting candidates.

RECOMMENDATION

Restore the fail-closed behavior in the SDK. `getReceiverHint` should reject missing or ambiguous registry matches and `prepareFinalizeCallback` should require an explicit `receiverHint` for `LzReceive` callbacks instead of silently omitting the field. The finalize wrapper should abort rather than submit a callback with `receiverHint = undefined`.

M-10 Missing hint falls back to LZ recipient

CONTINUED

— RESOLUTION

Status: Pending.

MEDIUM **M-11**

PENDING

Odd composeMsg rewrites LZ payload on wire

CATEGORY Unexpected Behavior

LOCATION MessageCodec.daml

REVIEW Main Review

DESCRIPTION

`encodeMessage` accepts any `composeMsg` that passes `isValidHex`. That helper only checks characters. It does not require an even number of hex digits. When `composeMsg` has odd length, `encodeMessage` still appends it directly to the packed `lzMessage`.

That is not a harmless formatting quirk. The off-chain transport converts the full `lzMessage` string to bytes with `hexToBytes`, and that helper left-pads the whole string when the hex length is odd. The extra nibble is inserted at the start of the message, not at the start of `composeMsg`. As a result, every fixed-width field shifts: the receiver fingerprint, the amount, the sender fingerprint, and the compose payload.

The effect is visible with a 1-nibble payload. For `composeMsg = "a"`, the encoded message length is 145 hex chars. The transport normalizes it to 146 chars by prepending `0`. The first 64 chars no longer match the original `sendTo`, and the next 16 chars become `8000000000000003e` instead of `0000000000000003e8`. Under the common `18 -> 6` decimal setup, that mutated amount is already above the signed `Int64` limit for `amountSD`, so the receive path fails. Under lower conversion rates, the same bug can settle a different amount than the source side encoded.

The impact is on the transfer itself, not only on the compose leg. In the common `18 -> 6` deployment, the mutated amount usually overflows during inbound validation, so the destination leg becomes undeliverable before minting. If the shifted fields still decode cleanly under a different decimal setup, the destination can instead mint the wrong amount to the wrong fingerprint-derived recipient.

For example, imagine an integrator enables `SEND_AND_CALL` and submits a transfer with a tiny odd-length `composeMsg` such as `"a"`. The source side builds a valid-looking `lzMessage` and the cross-chain packet is sent. When the off-chain transport converts that message to bytes, it prepends a nibble to the front of the whole payload. The destination then decodes a different receiver fingerprint and a different amount than the source side encoded. Under the repo's normal `18 -> 6` settings that mutated amount already exceeds signed `Int64`, so the receive request fails and the packet is effectively stuck. Under a lower conversion rate, the same malformed packet can stay decodable and settle a different mint than the sender intended. This is distinct from the already-known "compose payload is ignored" issue. Here the

M-11 Odd composeMsg rewrites LZ payload on wire

CONTINUED

transfer itself is rewritten in transit before the destination ever decides what to do with the compose payload.

— RECOMMENDATION

Reject odd-length `composeMsg` values before packing the message. The smallest safe fix is to normalize the compose payload the same way as `to` and then require byte-aligned hex before concatenation.

— RESOLUTION

Status: Pending.

<https://github.com/LayerZero-Labs/canton-ver/commit/010c0f3c5c19fac818c8e770089cd0002807dc76>

MEDIUM **M-12**

PENDING

Fake TransferFactory can spoof payouts

CATEGORY Unexpected Behavior

LOCATION FeeUtils.daml

REVIEW Main Review

DESCRIPTION

`validateExternalPartyAmuletRules` does not bind `external-party-amulet-rules` to the real `ExternalPartyAmuletRules` template or to a trusted factory contract id. It takes a caller-supplied contract id out of `ChoiceContext`, coerces it to `ContractId ExternalPartyAmuletRules`, converts it to the generic `TransferFactory` interface and then treats a successful `TransferFactory_PublicFetch` as proof that the factory is safe to use.

That is not a real trust check. A malicious vetted package can implement `TransferFactory_PublicFetch` and `TransferFactory_Transfer` with the expected surface behavior while skipping real settlement. `distributeAmulet` then accepts a `Completed` result without verifying that any transfer instruction or recipient holding was actually created.

This is most meaningful in the governance cleanup path, where the caller is not supposed to control payout settlement. An ordinary member creates a malicious `TransferFactory`, waits for a proposal to expire and calls `Governance_Expire` with `external-party-amulet-rules` replaced by that contract. The proposal is cleared, the fake factory reports two successful slash payouts, no real `TransferInstruction` is created for either recipient and the treasury keeps the full fee amulet.

For example, Bob submits a governance proposal with a 1.0 Amulet fee and it later expires. Alice, another ordinary member, deploys a vetted `TransferFactory` that only records payouts and always returns `Completed`. She then calls `Governance_Expire` with `external-party-amulet-rules` replaced by that contract. `slashProposalFee` splits the treasury-held fee into two 0.5 Amulet payouts, but no real payout artifact is created for Alice or Charlie. Governance still clears the proposal as if the slash succeeded. The same root cause also reaches other `distributeAmulet` consumers that trust `ChoiceContext` for `external-party-amulet-rules`, including `Request` settlement and `Executor.NativeDrop`. Those paths are weaker impact arguments because the handler, treasury, or executor already has broad control over payout execution. Governance is the strongest case because an ordinary member can use the bug to clear an expired proposal without delivering the slash payouts that the rest of the committee expects.

M-12 Fake TransferFactory can spoof payouts

CONTINUED

— RECOMMENDATION

Do not trust a caller-supplied generic `TransferFactory` contract id here. Bind the external payout factory to a trusted source controlled by the protocol, such as a stored config field or a DSO-owned contract reference and use that exact contract for payout creation.

If the factory must still come from context, add an explicit trust-root check before using it. In practice that means validating more than `expectedAdmin`: prove that the contract is the concrete `ExternalPartyAmuletRules` instance you intended to use, or reject the call. Also make `distributeAmulet` verify a real payout artifact was created instead of treating any `Completed` response as settlement.

— RESOLUTION

Status: Pending.

MEDIUM **M-13**

PENDING

Dust governance fees brick cleanup

CATEGORY DoS
LOCATION FeeUtils/FeeUtils.daml:271-359
REVIEW Main Review

— DESCRIPTION

`distributeAmulet` routes all non-treasury payouts through `TransferFactory_Transfer`, creating `TransferInstructions` with a fixed 24-hour lock. When a payout amount is below approximately 0.00274 CC ($\text{holdingFeeRate} * \text{transferInstructionExpiry} / \text{tickDuration}$), the amulet's holding-fee-adjusted lifetime is shorter than the lock period. Splice rejects with "lock expires before amulet" and the entire transaction reverts.

This affects every caller of `distributeAmulet`. On the governance path, `slashProposalFee` divides the escrowed fee equally among recipients. The existing `sharePerRecipient > 0.0` guard handles zero-share truncation but not small positive shares below the viability threshold, so governance reject, expire, and propose-with-auto-slash all fail permanently. On the request settlement path, `IRequest_Accept` and `IRequest_Reject` both call `distributeAmulet` directly with handler-supplied payouts. A handler who includes an external party with a dust amount bricks the accept. More critically, when `refundAddress` is set (all executor-backed inbound requests), any fee remainder below the threshold is automatically routed through the `TransferInstruction` path. A handler who sets a reasonable-looking treasury payout that leaves a small positive remainder — for example, 0.999 out of a 1.0 fee — triggers the failure without intending to. The handler cannot avoid this without knowing the exact amulet amount in advance. Recovery on the request path requires the handler to retry with a payout that avoids the external TI route entirely, such as setting the treasury payout to the full fee amount. For the `refundAddress` variant this is non-trivial because the handler must match the payout sum to the exact amulet balance.

— RECOMMENDATION

Add a minimum-viable-amount guard inside `distributeAmulet` before creating each `TransferInstruction`. Compute the threshold from the current round's holding fee rate and skip the TI path for amounts below it — either accumulate dust into the provider payout or use a direct `transferAmulet` call which has no TI lock constraint.

M-13 Dust governance fees brick cleanup

CONTINUED

— **RESOLUTION**

Status: Pending.

MEDIUM **M-14**

PENDING

Slash fee rounding bricks cleanup

CATEGORY Rounding
LOCATION Governance.daml#L364
REVIEW Main Review

DESCRIPTION

`slashProposalFee` computes `sharePerRecipient = amulet.amount.initialAmount / divisor` using raw DAML `Decimal` division, then pays every recipient that full fixed-precision share. DAML `Decimal` keeps 10 decimal places, so repeating-decimal splits such as `1.0 / 7 = 0.142857142857...` are rounded to `0.1428571429`. Reusing that rounded share for all 7 recipients produces `1.0000000003`, which is slightly more than the original `1.0` fee amulet amount.

When that happens, `distributeAmulet` rejects the payout map with `ERR_FEE_UTILS_PAYOUTS_EXCEED_AMOUNT`. Governance cleanup then rolls back, so active proposals cannot be rejected or expired even with normal-sized fees. With 7 slash recipients, a standard `1.0` fee already fails this way.

Impacts:

- `Governance_SetMembers` calls `slashProposalFee` when a proposal is active → `**SetMembers reverts**`
- `Governance_Propose` auto-slashes expired proposals before accepting new ones → `**new Propose reverts**` after the bricked proposal expires
- Combined with the original Reject/Expire brick, `**all four governance mutation paths are frozen**`. The only escape is `Governance_Accept`, which requires multisig cooperation. If the proposer is uncooperative with the multisig, governance is permanently stuck.

RECOMMENDATION

Do not split the fee using raw rounded `Decimal` shares. Compute slash payouts in a way that guarantees `sum(payouts) <= fee`, and keep any remainder with the treasury instead of assigning the rounded full share to every recipient.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/GovernanceSlashRoundingTest.daml>

RESOLUTION

Status: Pending.

LOW **L-01**

ACKNOWLEDGED

Explicit treasury payout can undercut minFee

CATEGORY Unexpected Behavior
LOCATION Request.daml, FeeUtils.daml
REVIEW Main Review

DESCRIPTION

`minFee` is not really enforced as a minimum. When a request is created, the code suggests that treasury should receive at least `minFee`. But during settlement, the payout logic only uses `minFee` if treasury is missing from the `payouts` map. If treasury is listed explicitly, that explicit amount wins, even if it is smaller than `minFee`. So if `minFee` is `100`, and the handler settles with `payouts = {treasury -> 10, ...}`, treasury gets `10`, not `100`. In other words, `minFee` behaves like a default value, not like a guaranteed floor. The impact is limited because the handler already controls the payout map. The main problem is that the code and the intended rule do not match. Someone reading the system would reasonably think treasury can never receive less than `minFee`, but that is not actually true.

RECOMMENDATION

The code should make one clear choice and enforce it consistently. If `minFee` is supposed to be a real minimum, then settlement should reject any payout map that gives treasury less than `minFee`, even when treasury is included explicitly. If `minFee` is only meant to be a fallback when treasury is omitted, then the naming and documentation should be updated to state that clearly.

RESOLUTION

Status: Acknowledged.

LOW **L-02**

RESOLVED

Non-ASCII instrument IDs break AppUID derivation

CATEGORY	Validation
LOCATION	OftFactory.daml, AppUID.daml
REVIEW	Main Review

— DESCRIPTION

The OFT setup path only requires `instrumentId.id` to be non-empty. Later, the protocol derives the `AppUID` for that same instrument by sending the text through `textToHex`, which rejects characters outside the supported byte range. The result is a split validation boundary: an instrument identifier can be accepted when the token is configured and then fail later when the protocol tries to use it in request creation, callback routing, or discovery.

This is not a case where an invalid identifier is rejected up front. The system accepts it first and only trips over it when a cross-chain flow reaches code that depends on `AppUID` derivation. For example, imagine an admin creates an OFT with `instrumentId.id = "Bond-EUR-€"` or `"□□"`. Creation succeeds because the factory only checks that the string is non-empty. Later, when the protocol builds the `LzSend` call context, it tries to derive `instrumentIdToAppUID`, which calls `textToHex` on that identifier. At that point the flow fails because those characters cannot be encoded by the current helper. The token exists on-ledger, but any path that depends on `appUID` derivation can now break at runtime.

In practice this turns a bad identifier into a latent configuration bug. The failure shows up later, in a real token flow, instead of at the point where the configuration was introduced.

— RECOMMENDATION

Validate the identifier at the point it is introduced, using the same character-set rules that `AppUID` derivation expects. If wider Unicode support is intended, then `AppUID` derivation needs to move to a canonical encoding that can represent the full accepted input space. Either way, the creation path and the consumption path need to enforce the same contract.

— RESOLUTION

Status: Resolved.

L-02 Non-ASCII instrument IDs break AppUID derivation

CONTINUED

<https://github.com/GuardianOrg/canton-ver-team1-1773698451456/commit/def9e52e75283b34339ffa3724804057e42b1168>

LOW **L-03**

ACKNOWLEDGED

Payout expiry permanently locks tokens

CATEGORY Logical Error
LOCATION OftTransferRule.daml:93, Common/Utils.daml:182
REVIEW Main Review

— DESCRIPTION

`lzSend` creates payout offers with `executeBefore = now + offerExpiryDelayPeriod`. During callback finalization, `ILockedPayout_ForceAccept` calls `finalizeTransfer` → `assertTransferTimeConstraints`, which enforces `executeBefore > now`.
If finalization is delayed past `executeBefore`, `force-accept` fails with `ERR_OFT_TRANSFER_EXECUTE_BEFORE_PAST`. Because DAML transactions are atomic, the entire accept callback reverts — including the `archive` (burn) of the cross-chain holding. No partial success is possible. The reject callback (`forceRejectImpl`) does NOT enforce time constraints, it calls `unlockAndFail` directly, bypassing `finalizeTransfer`. So the reject path always succeeds as recovery.
Net impact: delayed finalization forces the reject path (sender gets tokens back, cross-chain message fails), not permanent fund loss. However, `TransferRule_ExecuteAsAdmin` enforcing `executeBefore` on admin-forced settlement is unnecessary and causes avoidable reject-path fallbacks.

— RECOMMENDATION

Remove the `executeBefore` check from `TransferRule_ExecuteAsAdmin` / `ILockedPayout_ForceAccept`. The admin already has full authority, so the time gate adds no security value on the force path and causes unnecessary accept-callback failures.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/AttackVectorTest.daml#L154>

— RESOLUTION

Status: Acknowledged.

LOW **L-04**

PENDING

feeDeposit cannot send when issuer fee is set

CATEGORY	Unexpected Behavior
LOCATION	Impl.daml, Payout.daml, OftTransferOffer.daml
REVIEW	Main Review

— DESCRIPTION

`lzSend` prepends `feeDeposit` to the payout receiver list whenever `issuerFeeAmount > 0`. If the sender is also the configured `feeDeposit` party, the payout path later tries to create an `OftTransferOffer` with `transfer.sender == transfer.receiver`.

That shape is forbidden by `OftTransferOffer`. The send therefore reverts even though the sender is only trying to route the issuer fee back to itself. The code already hints at this failure mode in the `lzSend` implementation, but there is no validation on `feeDeposit` and no special handling for the self-fee case. Because of this, any party configured as `feeDeposit` is unable to originate sends while a positive issuer fee is active.

— RECOMMENDATION

Handle the self-fee case explicitly. When `sender == feeDeposit`, do not create a self-directed `OftTransferOffer`. Instead, keep the issuer-fee amount on the sender side as unlocked change or fold it into the refund path while still reducing the cross-chain amount by the fee.

— RESOLUTION

Status: Pending.

LOW **L-05**

PENDING

OApp request validation ignores endpoint address

CATEGORY Validation
LOCATION RequestFactory.daml
REVIEW Main Review

DESCRIPTION

`validateCallContextIdentity` checks that an OApp request uses `callerName = "OApp"`, `callerAddr = appUID` and `targetName = "EndpointV2"`, but it never checks `targetAddr`. The OFT builders and TS SDK do set a concrete endpoint address, so this field is meant to carry real routing identity, not just metadata. That means a privileged caller using `OApp_CreateRequest` can submit a request with a forged endpoint address that still passes on-ledger identity validation. Off-chain code preserves that address and uses it for transaction routing. This does not let the caller spoof an arbitrary contract, because the container still checks that the address belongs to an `EndpointV2` instance, but it can route to the wrong live endpoint or create requests that only fail later during execution. The issue is low severity because the affected path is already admin or delegated-admin only, and the standard `OApp_LzSend` and `OApp_LzReceive` flows build the correct call context internally.

RECOMMENDATION

For OApp requests, bind `targetAddr` to trusted config. `Request_Create` should compare it against the configured `endpointAddress`, or the OApp side should derive the endpoint address internally instead of accepting it from caller-supplied `callContext`.

RESOLUTION

Status: Pending.

LOW **L-06**

PENDING

Zero minFee bricks registry-style writes

CATEGORY	DoS
LOCATION	Registry.daml
REVIEW	Main Review

— DESCRIPTION

`Registry` allows `minFee >= 0`, but `SetPartyId` always charges exactly `minFee`. If an admin sets `minFee = 0`, the choice still calls `transferAmulet` with `amount = 0`.

That does not create a free registration mode. With the bundled Splice Amulet rules, a zero output is rejected during transfer processing, so `SetPartyId` becomes uncallable. The same pattern exists in `DiscoveryRegistry.SetUrl`, which also allows `minFee = 0` and then always calls `transferAmulet` with `amount = minFee`.

Therefore, an admin can put the contract into a state that looks like "free mode" in configuration but actually bricks the public write path.

— RECOMMENDATION

Do not allow `minFee = 0` on write paths that always charge exactly `minFee`. The simplest fix is to require `minFee > 0` in both `Registry` and `DiscoveryRegistry`.

If zero-fee mode is intentional, handle it explicitly by skipping the fee transfer when `minFee = 0` and document that spam protection is disabled in that mode.

— RESOLUTION

Status: Pending.

LOW **L-07**

PENDING

Missing validation on CostAsserts

CATEGORY Validation
LOCATION OApp/CostAsserts.daml:143
REVIEW Main Review

DESCRIPTION

The `isValidCostAsserts` checks `nativeDecimalShift - gasPriceScale <= 10` but does not check the opposite direction, `gasPriceScale - nativeDecimalShift <= 10`. The comment in `CostAssertsConstants.daml:35` says the scale ratio should be capped at `10^10`, but the current validation only enforces one side of that bound.

As a result, a `_COST_ASSERTS_ADMIN_ROLE` holder can store a config where `gasPriceScale` greatly exceeds `nativeDecimalShift`. Such values are unlikely in a well-configured production deployment, but the validation is broad enough that extreme or mistaken configurations are still accepted.

For example, `totalGas=3,000,000`, `maxGasPrice=5000`, `gasPriceScale=21`, and `nativeDecimalShift=3` are accepted today and make `computeOptionsCost` overflow DAML `Decimal` arithmetic. In those cases, `lzSend` aborts for the affected destination until the config is removed or corrected.

RECOMMENDATION

As the primary fix, add `&& config.gasPriceScale - config.nativeDecimalShift <= _MAX_SCALE_DELTA` to `isValidCostAsserts`. As defense-in-depth, also cap `maxGasPrice` or add a runtime guard in `computeOptionsCost` before the multiplication so extreme-but-valid configs cannot still overflow `Decimal`.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/CostAssertsOverflowTest.daml#L291>

RESOLUTION

Status: Pending.

<https://github.com/LayerZero-Labs/canton-ver/commit/6bbf5203dcd7b5e53ceab94fa99bf0ca6c5f358d>

LOW **L-08**

PENDING

DAML LzReceive lacks replay protection

CATEGORY Validation

LOCATION OApp/LzReceive/Call/Impl.daml:50-53, IOApp.daml:79

REVIEW Main Review

DESCRIPTION

The DAML `LzReceive` path does not enforce idempotency for inbound messages. `Impl.daml:50-53` validates `guid` format (bytes32 hex) and `origin.nonce > 0`, but neither value is recorded or checked for uniqueness. `OApp_LzReceive` is a nonconsuming choice on `IOApp` (`IOApp.daml:79`), and `RequestFactory` has no idempotency key. Each call creates a fresh `Request` regardless of whether the same `(srcEid, sender, nonce)` tuple was already processed.

The off-ledger TypeScript endpoint (`packages/endpoint/src/messaging-channel/index.ts`) provides replay protection through `verify` and `clearPayload` with strict nonce sequencing (`nonce === previous + 1`). However, this protection does not extend to the DAML contracts that ultimately mint tokens. The executor — a permissioned backend vApp service authenticated via Canton JWT — calls `DelegateLzReceive` on the `Executor` contract, which calls `OApp_LzReceive` on the DAML `IOApp`. Supply safety therefore depends entirely on the off-ledger executor service behaving correctly; the DAML layer has no independent idempotency guard.

This is a defense-in-depth concern, not a permissionless attack: the executor is a trusted role. The finding is that this trust assumption is implicit and undocumented — the DAML contracts accept any well-formed inbound message without checking whether it was already processed.

RECOMMENDATION

Either add DAML-side replay protection (e.g., atomically create a unique processed-message contract keyed by `(srcEid, sender, nonce)` or `guid` — DAML contract uniqueness would reject duplicates), or explicitly document the executor service as a trust assumption for supply safety.

RESOLUTION

Status: Pending.

LOW **L-09**

PENDING

DelegateLzReceive hands live fee to IOApp

CATEGORY Trust Assumptions

LOCATION Executor.daml

REVIEW Main Review

DESCRIPTION

`DelegateLzReceive` transfers `feeAmount` from `rich` to `poor` and then passes the resulting `poorAmuletCid` into arbitrary `IOApp.OApp_LzReceive` code before any `Request` exists. At that point the untrusted IOApp has both the live fee asset and `poor`'s authorization.

```
``plain text
```

```
poorAmuletCid <- transferAmulet ... rich poor ...
exercise ioAppCid OApp_LzReceive with
sender = poor
amuletCid = poorAmuletCid
`
```

That lets a malicious IOApp spend the fee and abort the flow before request creation. The executor's "poor has no value" safety story does not hold during this step because the bridge itself has just funded `poor`.

RECOMMENDATION

Do not pass a spendable fee amulet into untrusted IOApp code. Escrow the fee in trusted code before the IOApp call, or create a wrapper that `poor` can only forward into the trusted request path.

If the IOApp still needs fee metadata, pass a non-spendable reference instead of a live amulet CID.

RESOLUTION

Status: Pending.

LOW **L-10**

PENDING

IzReceive accepts zero-amountSD messages

CATEGORY	Validation
LOCATION	0App/LzReceive/Call/Impl.daml:46
REVIEW	Main Review

— DESCRIPTION

The LayerZero reference OFT on EVM allows zero-amount transfers when `minAmountLD == 0`: source-side send succeeds with `(amountSentLD=0, amountReceivedLD=0)`, destination `IzReceive` credits `0`, and composed messages still enqueue `IzCompose` with amount `0`. DAML diverges from that behavior. `createLzReceiveRequest` accepts `amountSD = 0` because the guard is `amountSD >= 0`. The handler can accept the request and consume the fee. But finalization later calls `IzReceive`, computes `amountLD = 0`, and tries to `create OfT` with `amount = 0`. The `OfT` template requires `amount > 0`, so finalization reverts and the `PendingCallback` remains stuck. Result: a zero-value OFT message that is valid in upstream LayerZero semantics is accepted into the DAML request lifecycle, but DAML rejects it only at the final mint step, after fee consumption.

— RECOMMENDATION

Reject zero amount at request creation time by requiring `amountSD > 0` in `createLzReceiveRequest`.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/ZeroAmountReceiveTest.daml#L70>

— RESOLUTION

Status: Pending.

LOW **L-11**

PENDING

Unbounded receiver list in lzSend

CATEGORY	Validation
LOCATION	Parser.daml:89
REVIEW	Main Review

DESCRIPTION

`parseLzSendArgs` validates `receiverCount >= 0` but has no upper bound. Each receiver creates one locked `Oft` holding and one locked `OftTransferOffer` during `lzSend`, and callback finalization must process all payouts in a single transaction. If the receiver list is large enough, finalization can exceed chain limits and revert.

In that state, the accepted callback remains stuck: the cross-chain holding stays locked, the user has no self-service recovery path because the payout offers are admin-locked, and the request fee has already been consumed.

The admin can likely recover funds manually by unlocking the main holding and force-rejecting payouts in smaller batches, but recovery is operationally painful and not part of the normal application flow.

RECOMMENDATION

Limit `receiverCount <= MAX_RECEIVERS` in `parseLzSendArgs`

RESOLUTION

Status: Pending.

LOW **L-12**

PENDING

issuerFeeBps=10000 bricks lzSend

CATEGORY Validation
LOCATION OftFactoryConfig.daml:135
REVIEW Main Review

DESCRIPTION

The ensure clause allows `issuerFeeBps` up to 10000 (`feeBps <= _BPS_DENOMINATOR`). Setting 10000 (100%) passes config validation but bricks all `lzSend` for that eid: `safeMulDivCeil(amountLD, 10000, 10000) = amountLD`, so `totalPayouts >= amountLD` always reverts. No error at config time.

RECOMMENDATION

Use `feeBps < _BPS_DENOMINATOR` (strict inequality) in the ensure clause.

RESOLUTION

Status: Pending.

LOW **L-13**

PENDING

Governance role separation not enforced

CATEGORY Validation
LOCATION Governance.daml:104-110 (ensure clause)
REVIEW Main Review

— DESCRIPTION

The 3e2c36d refactor split governance into `multisig` (decision authority) and `treasury` (fee custody), but the ensure clause does not enforce `treasury /= multisig`. A deployment can collapse both roles into one party, letting it unilaterally accept/reject proposals and control slashed fees without member co-signing. More broadly, the contract never proves `multisig` represents the declared `members` — authority is a deployment assumption, not an on-ledger invariant.

— RECOMMENDATION

Add `treasury /= multisig` to the Governance ensure clause.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/GovernanceMultisigTest.daml>

— RESOLUTION

Status: Pending.

LOW **L-14**

PENDING

Enforced options bypass via composeMsg

CATEGORY Validation

LOCATION OApp/EnforcedOptions.daml:64, OApp/LzSend/Call/Impl.daml:102

REVIEW Main Review

DESCRIPTION

`combineOptions` looks up enforced options by `(eid, msgType)`. The `msgType` is derived from the caller-supplied `composeMsg`: empty means SEND (`msgType=1`), non-empty means SEND_AND_CALL (`msgType=2`). If the admin only configures enforced options for SEND, a caller can set `composeMsg` to any non-empty hex (e.g., "ff") to select SEND_AND_CALL and skip the enforced options lookup entirely. `lookupEnforcedOption` returns "" and `combineOptions` passes the caller's raw `extraOptions` through without prepending the admin's enforced TLVs.

Because the receive side does not execute the compose payload, the caller ends up with a plain token transfer whose cross-chain options contain no admin-enforced gas parameters. Enforced options typically encode a minimum `lzReceiveGas` for reliable destination execution; bypassing them lets the caller underfund delivery gas. `costAsserts` provide partial mitigation when configured, but enforced options and cost asserts are independent configuration mechanisms.

RECOMMENDATION

When no enforced options are configured for SEND_AND_CALL on a given eid, fall back to the SEND enforced options for that eid. Alternatively, auto-replicate SEND enforced options to SEND_AND_CALL unless explicitly overridden.

RESOLUTION

Status: Pending.

LOW **L-15**

PENDING

PendingCallback finalizer is not bound

CATEGORY Trust Assumptions
LOCATION IPendingCallback.daml
REVIEW Main Review

— DESCRIPTION

`IPendingCallback_FinalizeRequest` is controlled by a caller-supplied `actor`. The implementation in `daml/Layerzero/Request/daml/PendingCallback.daml` validates the callback contract, but it never checks that `actor` is the sender, the handler, the instrument admin, or any other role that was bound when the request was created.

That makes visibility enough to finalize. Any party that can see both the `PendingCallback` and the callback contract can submit the choice first. This turns observers into active operators. It also widens every finalize-time binding issue, because the first visible actor can win the race and choose the settlement inputs.

— RECOMMENDATION

Store an explicit finalizer, or an explicit allowlist of finalizers, on `PendingCallback` and reject any `actor` outside that set.

If broad liveness support is intentional, keep that separate from callback authority. The code should still bind who is allowed to finalize before the request reaches the pending-callback stage.

— RESOLUTION

Status: Pending.

LOW **L-16**

PENDING

Invalid accept callback can silently no-op

CATEGORY Unexpected Behavior
LOCATION Dispatcher.daml
REVIEW Main Review

— DESCRIPTION

On the `OfFactory` accept path, `dispatchAccept` only handles the expected `lzSend` and `lzReceive` callback families. If the stored `callbackContext` is empty or names some other function, the dispatcher does not reject it. It falls through to `pure ()`.

That would be less dangerous if finalization stayed retryable, but it does not. `PendingCallback` still archives after `OnRequestAccepted` returns, so the request can look fully finalized on-ledger even though no callback-side action ran.

In practice, an accepted request can consume its only finalization handle without running the callback logic that should mint, unlock, or settle value.

```
` `plain text
if this.isAccept
then exercise callbackCid OnRequestAccepted ...
...
archive pendingCid
` `
```

— RECOMMENDATION

Reject invalid accept callback contexts instead of treating them as a no-op. `dispatchAccept` should abort when `callbackContext` is empty or does not match a supported callback family.

Also validate `callbackContext` when the request is created so malformed accept callbacks are rejected before a request can ever reach the pending-callback stage.

— RESOLUTION

Status: Pending.

LOW **L-17**

PENDING

Malformed hints can brick registry resolution

CATEGORY	Validation
LOCATION	Registry.daml
REVIEW	Main Review

— DESCRIPTION

`Registry.SetPartyId` stores the caller-supplied `hint` without checking that it round-trips into a valid party id. `GetPartyId` later rebuilds `hint::1220<fingerprint>` and aborts if `partyFromText` rejects it. That lets a caller write a malformed hint and make its own fingerprint unresolvable. Once stored, every lookup for that fingerprint fails until the entry is replaced with a valid hint.

— RECOMMENDATION

Reject non-canonical hints at registration time, or stop using caller-provided hints as identity material. The safest fix is to derive canonical identity from `caller` and keep any user-facing label separate from the value used for resolution.

— RESOLUTION

Status: Pending.

LOW **L-18**

PENDING

AppUID hashes differ for extended-byte ids

CATEGORY	Validation
LOCATION	AppUID.daml
REVIEW	Main Review

— DESCRIPTION

`instrumentIdToAppUID` hashes the hex encoding of `partyToText admin <> ":" <> id`. That only works if every caller turns the text into bytes the same way. This repo does not.

On the DAML side, `textToHex` uses a one-byte-per-code-point encoding for values below `256`. On the TypeScript side, `computeAppUID` UTF-8 encodes the same string before hashing. Extended-byte ids that are still accepted on-ledger therefore hash to different `appUID` values off-ledger.

That breaks any flow that relies on a stable `appUID`. Request creation, callback finalization and discovery entries can all fail once the SDK and the ledger disagree about the same instrument id.

— RECOMMENDATION

Choose one byte encoding and enforce it everywhere. The safest fix is to reject extended-byte instrument ids at the OFT boundary.

If extended bytes must stay supported, make the TypeScript helper use the same byte encoding as DAML and lock that down with cross-language test vectors.

— RESOLUTION

Status: Pending.

LOW **L-19**

PENDING

Force-accept rechecks live local pause

CATEGORY Validation
LOCATION `OftTransferRule.daml`
REVIEW Main Review

— DESCRIPTION

`ILockedPayout_ForceAccept` uses the admin settlement path, but that path still calls `finalizeTransfer`, which re-runs `assertLocalTransfersNotPaused`.

That lets an operator pause local transfers after a payout was already locked and make a later force-settlement fail. The problem is not limited to the `LzSend` callback flow. Any locked payout that relies on `ILockedPayout_ForceAccept` inherits the same live pause revalidation.

A forced settlement path is supposed to finish a payout that is already in its locked finalization stage. Reusing the ordinary local-transfer pause gate on that path means a later config toggle can brick settlement for work that was already committed.

— RECOMMENDATION

Do not run the normal local-transfer pause check on `TransferRule_ExecuteAsAdmin`. If callback settlement needs its own emergency stop, model it as a dedicated payout-settlement pause.

— RESOLUTION

Status: Pending.

LOW L-20

PENDING

Multi-registry lookup is first-match-wins

CATEGORY Unexpected Behavior
LOCATION OftFactory.daml, executor-sdk.ts
REVIEW Main Review

DESCRIPTION

Commit `0517a22` changes `ExecutorSdk.getReceiverHint` from a single-registry lookup into a scan across every visible `Registry` for the same `appUid`. The loop stops on the first matching `fingerprint -> hint` entry and returns it.

That only works if one app can never have two live registries with different entries for the same fingerprint, or if there is a clear rule for which registry wins. Neither is true here. `Registry.SetPartyId` still only enforces uniqueness inside one contract. It does not stop the same fingerprint from being present in two active registries for the same app with different hints. This same commit also adds `maxEntries`, which makes multi-registry setups an intended mode rather than an odd corner case.

The result is an ambiguous receiver resolution. If two registries both contain the same fingerprint, the SDK will silently take whichever contract happens to appear first in the active-contract query. There is no tie-break by version, shard, creation time, or anything else that would make the result predictable.

This matters because the hint is later fed into OFT callback finalization. `OApp.LzReceive` rebuilds the mint recipient from `hint <> "::1220" <> fingerprint`. Because of this, a stale or conflicting registry entry can send an inbound receive to the wrong same-fingerprint party, or make finalization fail if the chosen hint no longer forms a valid hosted party id.

RECOMMENDATION

Do not resolve receiver hints by scanning all live registries and taking the first hit. Finalization should be tied to one authoritative registry, one shard, or an explicit registry contract id carried through the callback path. If multiple registries must coexist, the SDK should collect all matches and reject anything except one unambiguous result. The registry layer should also enforce uniqueness across active registries for the same `appUid`.

L-20 Multi-registry lookup is first-match-wins

CONTINUED

— **RESOLUTION**

Status: Pending.

LOW **L-21**

PENDING

Deploy script defaults registry fee to zero

CATEGORY Warning
LOCATION discovery-registry.ts
REVIEW Main Review

— DESCRIPTION

This commit adds a new `buildDiscoveryRegistry` deployment helper and makes `minFee` default to `0.0` when the caller does not supply one.

That is unsafe for `DiscoveryRegistry`. The on-ledger contract still routes `minFee` directly into `transferAmulet` during `SetUrl`, and the current fee path does not support a zero-valued transfer output. The result is that the new script can deploy a registry that looks valid but rejects every registration until the handler repairs the fee config.

`scripts/src/deploy.ts` now calls `buildDiscoveryRegistry` without a `minFee` argument, so the broken default is used by the new deployment flow.

— RECOMMENDATION

Do not default `DiscoveryRegistry.minFee` to `0.0` in deployment code while the contract still treats zero as an invalid transfer amount. The smallest safe fix is to require an explicit positive `minFee` here or to default to the same positive floor the project already uses for tests and other live registration flows.

If zero-fee mode is meant to be supported long term, the on-ledger contract must first gain an explicit `minFee = 0` branch that skips fee transfer safely. Until then, deployment code should reject or avoid zero.

— RESOLUTION

Status: Pending.

LOW L-22

PENDING

Cost asserts underprice executor fees

CATEGORY	Validation
LOCATION	CostAsserts.daml
REVIEW	Main Review

— DESCRIPTION

`computeOptionsCost` only prices the raw gas and value decoded from the type-3 options blob. It adds `lzReceiveGas` and `lzComposeGas`, checks `nativeCap` and multiplies that result by the configured gas price and price ratio. The offchain executor charges more than that. In `packages/endpoint/src/executor/fee-lib.ts`, `decodeExecutorOptions` adds `lzReceiveBaseGas`, adds `lzComposeBaseGas * lzComposeCount` and applies the `orderedExecution` 2% premium before the final fee is quoted. The fee path also runs through `estimateFeeByEid`, which charges for `callDataSize * gasPerByte`, but the DAML cost-assert path has no calldata-size input and cannot model that term at all. There is also a Canton-specific mismatch between what cost assertion validates and what execution actually receives. `assertOptionsCost` compares the computed cost against the gross `feeAmount`, but request execution later reconstructs `msgValue` as the escrowed amulet amount minus `RequestFactoryConfig.minFee`. That reserved `minFee` stays with treasury on accept and reject, so it is not available to satisfy `EndpointV2`'s fee sufficiency check. On deployments with non-zero `minFee`, the spendable execution budget is therefore smaller than the amount cost assertion approved even before accounting for the missing executor base-gas, premium and calldata terms. That combined gap means a sender can supply a fee that passes the DAML cost assertion but is still too small for the executor. The repo's own guide presents cost assertion as the anti-spam check that stops underfunded sends before they consume the send flow, but the current formula cannot model the executor's real gas floor and also validates the wrong fee base when `minFee > 0`. The `daml-props` harness reproduced the executor-formula mismatch three ways: a plain `lzReceive` underpriced by missing base gas, `orderedExecution` underpriced because the 2% premium is ignored and a `lzCompose` send underpriced because the per-compose base gas is ignored. The impact is operational rather than fund-stealing. When the handler rejects the underfunded request, the holding is unlocked and outbound usage is reversed, but the request still reaches the offchain path that cost assertions were meant to filter out. On deployments with low `minFee`, this gives users a cheaper way to churn requests and rejection work even when `costAsserts` are configured.

L-22 Cost asserts underprice executor fees**CONTINUED**

— RECOMMENDATION

Make the on-ledger estimate match the executor fee model closely enough to be a real prefilter and compare it against the net fee budget that execution can actually spend. At minimum, extend the DAML config and formula so `computeOptionsCost` accounts for `lzReceiveBaseGas`, `lzComposeBaseGas`, the `orderedExecution` premium and the calldata-size term used by the price feed. If

`RequestFactoryConfig.minFee` remains a separate treasury reserve, subtract it before the cost-assert comparison or otherwise ensure callers provide an execution fee that is explicitly exclusive of `minFee`. If exact parity with the executor's multiplier and floor-margin settings is required, add those terms too or reject option combinations that the DAML config cannot price safely.

— RESOLUTION

Status: Pending.

LOW **L-23**

PENDING

odd-length options fail at the ULN boundary

CATEGORY Unexpected Behavior
LOCATION EnforcedOptions.daml
REVIEW Main Review

— DESCRIPTION

`combineOptions` only checks that `extraOptions` starts with the type-3 prefix `0003`. It does not require byte-aligned hex, so a caller can supply a one-nibble-truncated type-3 blob and `prepareSend` will still return it as the final `options`. The enforced-options config validator has the same gap in `isValidEnforcedOptions`, because it only checks `isValidHex` and `isOptionsType3`. That string does not survive the TypeScript `bytes` boundary. The offchain codec converts hex to bytes with `hexToBytes`, which left-pads odd-length hex at the front. For the `daml-props` witness `0003010011016d13193f3fbd9392ff414794a43b54b`, the transport rewrites the prefix from `0003` to `0000` before the ULN parser sees it. `OftFactory_BuildQuote` uses `prepareSend` directly, so it can hand the offchain layer a quote that is already invalid. `lzSend` only catches the same malformed options early when `costAsserts` is configured for that route; otherwise the request can be created on-ledger and then fail later in the offchain send path.

This is a liveness and griefing issue. A user can submit malformed odd-length `extraOptions` to create requests that are guaranteed to be rejected later and an admin can brick a route by storing odd-length enforced options.

— RECOMMENDATION

Reject odd-length hex anywhere options are accepted. The smallest safe fix is to add an explicit even-length check in `combineOptions`, `isValidEnforcedOptions` and any send-param validation path that accepts raw `extraOptions`.

— RESOLUTION

Status: Pending.

LOW **L-24**

PENDING

Cost assertions reject valid legacy ULN options

CATEGORY	Validation
LOCATION	OptionsDecoder.daml
REVIEW	Main Review

— DESCRIPTION

`decodeExecutorOptions` only understands type-3 executor worker options. If the blob does not start with `0003`, it falls straight into the worker-TLV parser at offset `0`. That means valid legacy ULN options such as type 1 (`0001 + u256 gas`) and type 2 (`0002 + u256 gas + u256 amount + bytes receiver`) are treated as malformed worker headers and rejected by the DAML cost-assert path.

That does not match the shipped offchain endpoint. `packages/endpoint/src/message-library/uln/uln-302/uln-options.ts` explicitly accepts legacy types 1 and 2 and normalizes them into executor options before fee handling. The repo's TS tests cover both successful legacy decode cases. The `daml-props` harness reproduced the mismatch on seed `1` with a valid type-1 witness, and fixed-witness scripts confirmed that the real DAML decoder rejects both `0001` and `0002` forms even though the TS decoder accepts them.

The result is a route-dependent interoperability failure. When `costAsserts` are configured, a sender can provide options that the endpoint would accept and execute, but the ledger rejects them first during `lzSend`. When `costAsserts` are not configured, the same legacy options still pass through `prepareSend` and can be consumed offchain. `OfFactory_BuildQuote` also returns the legacy options unchanged, so the quote path does not fail by itself, but it no longer matches the stricter send behavior on cost-asserted routes. This does not let an attacker steal funds, but it can break sends for integrations that still use legacy ULN options and it makes the DAML precheck disagree with the real endpoint behavior.

— RECOMMENDATION

Make the DAML cost-assert decoder follow the same option semantics as the endpoint. The smallest safe fix is to recognize legacy ULN types 1 and 2 in `decodeExecutorOptions` and translate them into the same aggregate executor fields that the type-3 path produces. If legacy options are intentionally unsupported on-ledger, reject them earlier in send and quote validation too, so the whole stack enforces one rule instead of only the cost-assert path doing so.

L-24 Cost assertions reject valid legacy ULN options

CONTINUED

— RESOLUTION

Status: Pending.

LOW L-25

PENDING

outboundLimit=0 blocks all outbound transfers

CATEGORY	Logical Error
LOCATION	RateLimiterConfigTypes.daml:93
REVIEW	Main Review

DESCRIPTION

`isValidConfigEntry` validates `outboundLimit >= 0`, allowing `outboundLimit=0`. When rate limiting is active (`outboundEnabled=True`, `isGloballyDisabled=False`), `computeDecay` always returns `available=0`:

```
`plain text
decay = safeMulDiv(limit=0, elapsedSecs, window) = 0
usage = max(0, amountInFlight - 0) = amountInFlight
available = max(0, 0 - usage) = 0
`
```

Any `scaledAmount > 0` fails with `ERR_RATE_LIMIT_EXCEEDED`. This is permanent – no time-based recovery exists because `limit=0` means zero decay rate. The entire outbound path is bricked until admin reconfigures.

The footgun is subtle: `_EMPTY_RATE_LIMITER_CONFIG` uses `outboundLimit=0` with `isGloballyDisabled=True` as its default. If an admin copies this template and enables rate limiting (`isGloballyDisabled=False`) without updating limits, all outbound traffic is permanently blocked with no error at config time.

RECOMMENDATION

Either require `outboundLimit > 0` when `outboundEnabled=True` in `isValidConfigEntry`, or document that `outboundLimit=0` with rate limiting enabled permanently blocks outbound.

RESOLUTION

Status: Pending.

LOW **L-26**

PENDING

BurnMint has no supply conservation

CATEGORY Validation
LOCATION BurnMintImpl.daml:22-45
REVIEW Main Review

DESCRIPTION

`executeBurnMint` burns inputs and mints outputs independently with no assertion that `sum(inputs) == sum(outputs)`. Admin can call `BurnMintFactory_BurnMint` with empty `inputHoldingCids` and non-empty `outputs` to mint tokens from nothing, or non-empty inputs with empty outputs to destroy tokens. This violates the cross-chain supply conservation invariant — admin can arbitrarily inflate or deflate total supply, which is a stronger primitive than any other admin capability (e.g., `RecoverFunds` preserves supply).

RECOMMENDATION

Add `assertMsg \"$ sumInputs == sumOutputs` inside `executeBurnMint`, where `sumInputs` is computed from fetched input holdings and `sumOutputs` from the outputs list.

RESOLUTION

Status: Pending.

LOW **L-27**

PENDING

High scaleDecimals disables rate limiter

CATEGORY Configuration
LOCATION RateLimitTypes.daml:121-124
REVIEW Main Review

DESCRIPTION

`downscale` uses ceiling division: $(\text{amount} - 1) / \text{scaleFactor} + 1$. Config validation (`RateLimiterConfigTypes.daml:101`) allows `scaleDecimals` up to 18. When `scaleDecimals` greatly exceeds `localDecimals` (e.g., 18 vs 6), `scaleFactor = 1018` and all practical token amounts (up to $\sim 10^{15}$ for a 6-decimal token) map to 1 unit. A rate limiter with `outboundLimit=1000` would allow 1000 transfers of 1 billion tokens each, making rate limiting meaningless.

RECOMMENDATION

Validate `scaleDecimals <= localDecimals` in `OfFactoryConfig` ensure clause.

RESOLUTION

Status: Pending.

LOW **L-28**

PENDING

amuletContext steering wedges settlement

CATEGORY	Validation
LOCATION	FeeUtils/FeeUtils.daml:79-97
REVIEW	Main Review

— DESCRIPTION

`IRequest_Accept` and `IRequest_Reject` take a caller-supplied `amuletContext` at settlement time. `validateContextWithExternalAmulet` only checks that the supplied `AmuletRules`, `OpenMiningRound`, and `ExternalPartyAmuletRules` belong to the expected DSO — it does not bind settlement to the canonical fee infrastructure. A handler can supply a different live same-DSO `OpenMiningRound` with higher `createFee`, causing `distributeAmulet` to fail with `ITR_InsufficientFunds`. The `Request` remains live (wedged). A fix for config/state selection does not fix this unless it also binds `amuletContext` itself.

— RECOMMENDATION

Store canonical fee-infra CIDs at request creation and require them at settlement, or derive the canonical live round internally instead of trusting settlement-time CIDs.

— RESOLUTION

Status: Pending.

LOW **L-29**

PENDING

Transfer silently truncates amounts

CATEGORY Validation
LOCATION `OftTransferOffer.daml:51-53`
REVIEW Main Review

— DESCRIPTION

`OftTransferOffer.ensure` allows `transfer.amount > 0.0` (fractional accepted). `OftTransferRule` (line 102) silently truncates via `truncate transfer.amount`, so a transfer offer for 1.5 tokens delivers only 1. `BurnMintImpl` (line 36) correctly rejects fractional amounts with `assertMsg` `errOftBurnMint_AmountNotWholeNumber`. The same protocol has inconsistent validation for the same property — transfers silently lose precision while burn/mint correctly validates.

— RECOMMENDATION

Add `transfer.amount == intToDecimal (truncate transfer.amount)` to `OftTransferOffer` ensure clause.

— RESOLUTION

Status: Pending.

LOW **L-30**

PENDING

Malformed enforcedOptions brick sends

CATEGORY	Validation
LOCATION	Oft/OftFactoryConfig.daml, OApp/CostAsserts.daml
REVIEW	Main Review

— DESCRIPTION

The SetEnforcedOptions choice validates enforced options only as valid hex plus starts with 0003 (type-3 prefix). It does not validate the internal worker-option TLV structure within the type-3 envelope.

An admin can store malformed type-3 enforced options that look valid at config-update time. The ensure clause accepts them and the config is written without error.

Later, once cost assertions are enabled for the same destination endpoint ID, every IzSend call triggers assertOptionsCost, which decodes the combined options. The malformed worker option causes decodeExecutorOptions to abort with ERR_COST_ASSERTS_UNSUPPORTED_OPTION_TYPE.

A syntactically accepted config silently bricks all sends for the affected endpoint. The failure only appears on live send traffic, not when the config is written.

— RECOMMENDATION

Validate the full type-3 worker-option structure in SetEnforcedOptions and in the OftFactoryConfig ensure clause, not just hex shape and prefix. Run the same decodeExecutorOptions logic at config-write time so malformed entries are rejected immediately rather than deferring failure to the send path.

— RESOLUTION

Status: Pending.

LOW **L-31**

PENDING

RequestFactory accepts foreign-gateway IOApp

CATEGORY	Validation
LOCATION	RequestFactory/RequestFactory.daml:request_CreateImpl
REVIEW	Main Review

— DESCRIPTION

Request_Create fetches the caller-supplied IOApp and checks only that its instrumentId.admin is a signatory and matches the caller-supplied instrumentAdmin. It never checks that the IOApp actually belongs to the factory's handler — specifically, it does not verify `IOAppView.oappGateway == RequestFactory.handler`.

Because of this, a handler-A RequestFactory accepts an IOApp whose own view says it belongs to gateway-B. Handler-A can then accept that request and create a PendingCallback carrying the foreign app's appUID.

Impact: request creation is not bound to the handler's own app trust anchor. A foreign-gateway IOApp can be injected into another handler's live Request and PendingCallback flow. This breaks the intended isolation that each handler only processes requests for its own registered applications.

— RECOMMENDATION

After fetching the IOApp view in Request_Create, add a check that the IOApp's oappGateway matches the RequestFactory's handler: `require (view ioApp).oappGateway == handler`. This ensures each handler can only create requests for IOApps that are registered under its own gateway.

— RESOLUTION

Status: Pending.

LOW **L-32**

PENDING

Request has no timeout or sender cancel

CATEGORY	DoS
LOCATION	Request/Request.daml, IRequest/IRequest.daml
REVIEW	Main Review

— DESCRIPTION

The Request template defines exactly two choices: `IRequest_Accept` and `IRequest_Reject`, both controlled exclusively by the handler party. The sender — who pays the fee — is a signatory but has zero choices to exercise. There is no expiry field, no timeout mechanism, no sender-cancel choice, and no admin override. If the handler party goes offline permanently (participant failure, key loss, operational shutdown), all pending Requests for that handler are permanently stuck. The fee amulets (owned by treasury, referenced by `Request.amuletCid`) are locked with no recovery mechanism.

Every other contract in the protocol's 3-phase lifecycle has a fallback path for stuck states:

- `PendingCallback`: any party with visibility can call `FinalizeRequest` (controller is caller-supplied actor)
- `OffTransferOffer`: sender can call `TransferInstruction_Withdraw` at any time
- `Governance`: any member can call `Governance_Expire` after the proposal deadline passes

`Request` is the only contract with zero fallback paths. The handler is the sole exit.

In the `DelegateLzReceive` flow, the fee payer (the executor) is not even a signatory or observer on the `Request`. They pay via `refundAddress` but have no control if the handler abandons the `Request`.

— RECOMMENDATION

Add a sender-controlled timeout choice to the `IRequest` interface, such as `IRequest_Cancel` with controller sender, gated by a time check (e.g., `now > createdAt + timeoutPeriod`). On cancellation, the fee amulet should be returned to the sender via the same `distributeAmulet` mechanism with empty payouts.

Alternatively, add a `createdAt` and `expiry` field to the `Request` template and allow any signatory to cancel after expiry.

— RESOLUTION

Status: Pending.

LOW **L-33**

PENDING

Unchecked RequestFactory Anchor

CATEGORY	Validation
LOCATION	OftFactoryConfig.daml:252
REVIEW	Main Review

DESCRIPTION

`OftFactoryConfig_SetRequestFactory` lets the OFT admin install any `IRequestFactory` CID without fetching or validating it. Unlike `OftFactoryConfig_SetTransferRule`, it does not verify any identity fields before rebinding the config.

`OftFactory.oApp_createRequestImpl` later trusts `config.requestFactoryCid` directly and exercises `Request_Create` on whatever factory is currently stored there. Because of that, a same-handler sibling `RequestFactory` with a different `treasury` or different `permanentObservers` silently becomes the lifecycle instance for all future requests created through that OFT.

This means OFT config does not bind request creation to a canonical request factory. An admin-side wrong-instance swap can silently reroute fee escrow and request visibility to a different lifecycle contract without changing the OFT instance itself.

RECOMMENDATION

Validate `newRequestFactoryCid` before storing it. At minimum, fetch the target and require the expected handler and DSO to match the OFT's trust boundary; ideally bind a canonical request factory per OFT and reject sibling substitutions.

RESOLUTION

Status: Pending.

LOW **L-34**

PENDING

Treasury fee blocks governance cleanup

CATEGORY	Access Control
LOCATION	Governance.daml:288-301 (Governance_Expire), Governance.daml:352-367 (slashProposalFee)
REVIEW	Main Review

— DESCRIPTION

M-03 reported that the leader-held proposal fee could freeze governance. The team added `treasury /= multisig` to the ensure clause, correctly separating fee custody from decision authority. However, `Governance_Expire` still depends on the treasury-owned `amuletCid` disclosure. `Governance_Expire` is `controller submitter` (any single member), but calls `slashProposalFee` which does `fetch p.amuletCid` where `amuletCid` is owned by `treasury`. In Canton's privacy model, the submitting participant must have disclosure of the amulet contract. Even though DAML authorization succeeds (the Governance contract has `signatory multisig, treasury`), the exercising member's participant cannot submit the transaction without first obtaining the amulet disclosure from treasury's participant.

`Governance_Expire` is documented as individually exercisable by any member, but in practice requires treasury cooperation to obtain the amulet disclosure. If the treasury party's participant is offline or uncooperative, no member can expire a stale proposal. The same dependency applies to `Governance_Reject`, `Governance_Accept`, auto-slash in `Governance_Propose`, and `Governance_SetMembers`.

— RECOMMENDATION

Either make the fee amulet visible to all members at proposal creation time (e.g., add members as observers on the amulet or store a disclosure reference in the proposal) so any member can independently exercise `Governance_Expire` without treasury cooperation, or update the documentation to reflect that `Governance_Expire` requires treasury cooperation and is not unilaterally exercisable by any single member.

— RESOLUTION

Status: Pending.

LOW **L-35**

PENDING

Direct sender refunds revert below create fee

CATEGORY Unexpected Behavior

LOCATION FeeUtils.daml

REVIEW Main Review

— DESCRIPTION

`distributeAmulet` refunds a positive remainder to `sender` by calling `transferAmuletWithFeeRatio` with `receiverFeeRatio = 1.0`. That makes the refund output pay its full output fee from the refunded amount itself.

This is unsafe when the remainder is positive but smaller than the current amulet `createFee`. The underlying Splice transfer rules reject receiver-paid outputs unless `receiverOutputFee < amount`, while native sender change handles sub-fee dust by consuming it and returning no change. Consequently, `Request` settlement can fail on a small leftover amount even though the same dust could have been absorbed safely by the provider. The request is left live and must be retried or handled operationally.

— RECOMMENDATION

Do not attempt a direct sender refund unless the remainder can cover the receiver-paid output fee. If it cannot, skip the direct refund and leave the dust with the provider, matching Splice's native sender-change behavior.

— RESOLUTION

Status: Pending.

LOW **L-36**

PENDING

DiscoveryRegistry stores one unbounded URL map

CATEGORY Unexpected Behavior
LOCATION DiscoveryRegistry.daml
REVIEW Main Review

— DESCRIPTION

`DiscoveryRegistry` keeps every `AppUID -> URL` entry in one shared `urlMap`. `SetUrl` updates that map by recreating the whole contract with `TextMap.insert hash url urlMap`. It does not limit how many entries the map may hold, and it does not limit the size of `url`.

Consequently, successful registrations grow one shared contract forever. Every new writer must consume and recreate the same state object. In Canton, that is an operational scalability problem as the registry gets larger over time, update payloads get heavier and concurrent writers contend on the same singleton.

— RECOMMENDATION

If `DiscoveryRegistry` remains a singleton, add a maximum URL length and a maximum number of entries. Consider also to stop rewriting one shared `TextMap` and store each app's discovery record separately, for example behind a keyed per-app contract.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-01

PENDING

Fingerprint-only identity is not party-unique

CATEGORY	Unexpected Behavior
LOCATION	Serde.daml, OftFactoryConfig.daml, Registry.daml
REVIEW	Main Review

DESCRIPTION

A `Party` is the ledger identity that owns contracts and authorizes actions. A `template` is an on-ledger contract type and a `choice` is the operation a party exercises on an existing contract. `IAAllowlist` is a DAML interface that decides whether a given `Party` may use restricted token flows. `Registry` is the contract that records how inbound cross-chain transfers should be resolved to a local Canton party. `Oft` is the token holding itself and its `owner` field is a `Party`. That means this issue is not about labels or metadata. If the protocol resolves the wrong `Party`, it creates or authorizes the wrong legal owner on the ledger.

The relevant Canton detail is that a full party ID is text of the form `<hint>::1220<64 hex chars>`. The left side is a human-readable identifier, often called the hint. The right side is the cryptographic namespace-sized suffix that this codebase calls a fingerprint. The helper `extractFingerprint` discards the hint and keeps only that 64-character suffix. In other words, the protocol takes a full DAML `Party`, throws away the human-readable identifier and then treats the remaining suffix as if it were a unique address for that party. That assumption conflicts with the official Canton identity model. Canton documentation describes a party identifier as an identifier within a namespace in the topology docs

(<https://docs.digitalasset.com/overview/3.4/explanations/canton/topology.html>)

(<https://docs.digitalasset.com/overview/3.4/explanations/canton/topology.html>) and separately notes that several unique identifiers can share the same namespace in the getting started docs

(https://docs.digitalasset.com/operate/3.4/tutorials/getting_started.html)

(https://docs.digitalasset.com/operate/3.4/tutorials/getting_started.html). The DAML SDK documentation also advises applications to treat party IDs as opaque identifiers rather than parsing them and making format assumptions.

The rest of the protocol is built around that reduced value. `IAAllowlist` stores whitelist and blacklist entries as extracted fingerprints, so the `IsAllowlisted` check compares only the suffix extracted from each party ID. In DAML terms, `IsAllowlisted` is a `nonconsuming choice`, which means it performs a permission check without rewriting the contract. `MessageCodec` also serializes the cross-chain recipient as a 32-byte fingerprint rather

I-01 Fingerprint-only identity is not party-unique

CONTINUED

than as a full Canton `Party`. `Registry.SetPartyId` is a state-changing choice controlled by the registering user and it stores a single `fingerprint -> hint` mapping. `LzReceive` later decodes the incoming message, reads the recipient fingerprint, combines it with the stored hint, reconstructs a full party ID as `hint::1220<fingerprint>` and mints a new `Oft` holding to that reconstructed `Party`.

This design is only safe if the extracted fingerprint uniquely identifies exactly one party. The official Canton documentation shows that this is not true, because a party is identified by the full pair of identifier and namespace, not by the namespace alone. Once the protocol has discarded the identifier, it can no longer distinguish sibling parties that share the same namespace. An allowlist entry intended for one party becomes an allowlist entry for every party that shares that suffix. A registry entry intended to help resolve one party becomes a single mutable slot for that entire suffix. The sibling party that most recently updates the registry controls which full party ID will be reconstructed during inbound settlement.

For example, imagine a bank that operates both `TreasuryDesk::N` and `SandboxTrader::N`. The bank wants only the treasury desk to use a restricted OFT, so the treasury party is added through the allowlist admin flow. However, `IAllowlist` stores only `N`, not the full party ID, so `SandboxTrader::N` also passes `IsAllowlisted`. Later, the sandbox trader exercises `Registry.SetPartyId`, which is the normal DAML mechanism for a party to register or refresh its own receive hint and overwrites the `N -> hint` mapping with `SandboxTrader`. When a remote chain sends tokens to the treasury's cross-chain address, the message still contains only `N`. `LzReceive` then reconstructs `SandboxTrader::N` from the latest registry hint and creates the new `Oft` holding with `owner = SandboxTrader::N`. The impact is direct ledger misdelivery and broken party-scoped access control: the wrong DAML party becomes token owner and controls meant for one party silently expand to all parties sharing that suffix.

— RECOMMENDATION

Stop using the extracted fingerprint alone as the canonical address anywhere the protocol needs party-specific behavior. The safest fix is to carry and validate a party-unique identifier end to end in the allowlist, registry and cross-chain receive path, so that resolution never depends on a mutable `fingerprint -> hint` slot shared by sibling parties. If the design intentionally wants namespace-scoped identity, then that constraint must be enforced operationally by allowing only one party per namespace and by documenting clearly that allowlists and receive resolution apply to namespaces, not to individual DAML parties.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-02

ACKNOWLEDGED

FinalizeRequest trusts settlement-time context

CATEGORY	Trust Assumptions
LOCATION	PendingCallback.daml
REVIEW	Main Review

— DESCRIPTION

`PendingCallback_FinalizeRequest` does not settle against a fully bound request. Instead, it accepts an `executeContext` at finalization time and forwards that runtime payload into the OFT callback path. The interface intentionally allows any visible party to finalize, so the first actor who sees the pending callback can also choose the context that settlement will use. The dispatcher then reads `configCid`, `rateLimiterStateCid`, and, for inbound receives, `receiverHint` directly from that finalizer-supplied context.

That means important settlement inputs are still live when the callback is finalized. The code rejects obviously unrelated instruments, but it does not prove that the chosen config or rate-limiter state is the exact instance that was supposed to govern the request. If multiple compatible instances exist, the finalizer can pick the one it prefers. On the receive side the problem is sharper: `LzReceive` reconstructs the recipient party from the message fingerprint plus the `receiverHint` supplied during finalization, instead of resolving the recipient from authoritative on-ledger state at the moment of settlement.

For example, imagine an inbound transfer is waiting in `PendingCallback` and both the intended recipient and another visible party can see that callback. If the other party finalizes first, it can supply a different `receiverHint` for the same namespace or choose a different compatible config and rate-limiter state. If that hint points to an existing sibling party under the same namespace, the transfer settles to that sibling. If the hint is simply wrong, finalization fails and successful settlement depends on off-chain retry behavior. In both cases, the outcome is being shaped at settlement time by whoever finalized first, not by data that was fixed when the request was accepted.

A callback settlement path should not depend on mutable runtime context like this. By the time finalization starts, the recipient and the security-relevant policy objects should already be fixed.

— RECOMMENDATION

Bind the settlement context before the callback becomes finalizable. The pending callback, or a signed payload attached to it, should contain the exact config, state, and receive

I-02 FinalizeRequest trusts settlement-time context

CONTINUED

identity that were approved earlier, and finalization should verify those bindings rather than trust fresh `executeContext` fields. On the receive path in particular, stop taking `receiverHint` from off-chain context and resolve the final local party from authoritative on-ledger state during settlement.

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-03

ACKNOWLEDGED

Settlement can use caller-chosen live instances

CATEGORY	Trust Assumptions
LOCATION	RequestFactory.daml, ConfigUtils.daml, OftFactoryConfig.daml, PendingCallback.daml, Dispatcher.daml
REVIEW	Main Review

— DESCRIPTION

Some settlement paths take config or state contract ids directly from the caller. The code then checks only that the chosen contract matches the same admin or instrument. It does not prove that this is the one live instance the protocol actually meant to use.

That matters if duplicate compatible instances exist on-ledger. In that case, the caller can steer execution toward the instance with the policy they want. There is no canonical lookup, singleton guard, or parent-held reference that pins settlement to one authoritative config or one authoritative rate-limiter state.

For example, imagine an operator leaves two live `OftFactoryConfig` contracts behind after a migration. One keeps transfers paused and enforces the stricter allowlist. The other is older and more permissive. A user who creates the request, or later finalizes the callback, can point the flow at the permissive config and still pass validation because the contract is for the same admin and instrument. The transaction then settles under a valid but unintended policy.

This turns duplicate-instance mistakes into real policy bypasses. Fees, pause state, allowlist checks, peer settings, and rate-limit behavior can all end up coming from the caller's chosen instance instead of from the intended live one.

— RECOMMENDATION

Do not let callers choose security-relevant config or state instances directly. Settlement should resolve those references from one canonical on-ledger source, such as a parent-held pointer or a registry that guarantees a single active instance. The current same-admin and same-instrument checks can stay, but only as a sanity check on a contract the protocol already selected for itself.

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-04

ACKNOWLEDGED

State-commitment chain is not canonical

CATEGORY	Unexpected Behavior
LOCATION	Committer.daml, CommitterImpl.daml, StateCommitment.daml
REVIEW	Main Review

— DESCRIPTION

The code records a predecessor pointer in `prevStateCommitmentCid`, but it does not enforce that there is only one live head. `CreateStateCommitment` and `CommitBatch` are both nonconsuming choices on `Committer` and `createStateCommitmentImpl` only archives the predecessor if the caller explicitly supplies one. If the caller passes `None`, the new `StateCommitment` is created and every existing one stays live. That makes `prevStateCommitmentCid` just a backlink, not a canonical chain rule. The ledger can end up with multiple active `StateCommitment` contracts for the same owner at the same time and nothing on-ledger tells you which one is the current head.

For example, an operator could use the shipped `CantonChainClient.buildCommitBatch` helper twice in a row. That helper currently sends `prevStateCommitmentCid = undefined`. The first batch creates one active state commitment. The second batch creates another one without archiving the first. Now both commitments are live, both look current, and any off-ledger component that expects a single canonical root has to guess which one to follow.

This turns the state-commitment chain into an ambiguous set of live roots instead of one enforced history.

— RECOMMENDATION

The protocol should track one explicit chain head on-ledger and require every new commitment to advance that head. `None` should only be valid for genesis. If a new commitment is created, the previous head should be consumed or otherwise proven to be the unique current one in the same transaction. The client helper should also stop defaulting `prevStateCommitmentCid` to `undefined` and fetch the current head before building a batch.

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-05

ACKNOWLEDGED

Blacklist-after-creation bricks payouts

CATEGORY	Logical Error
LOCATION	OftTransferRule.daml:87, Common/ConfigUtils.daml:39-40
REVIEW	Main Review

DESCRIPTION

`IzSend` validates sender and receiver against the allowlist at offer creation time. But `ILockedPayout_ForceAccept` re-validates via `finalizeTransfer` → `assertAllowlisted` at execution time. If the admin blacklists the receiver between creation and callback finalization, `force-accept` fails with `ERR_OFT_NOT_ALLOWLISTED`. Because DAML transactions are atomic, the entire accept callback reverts — including the burn. No partial success is possible. The reject path (`forceRejectImpl`) does NOT check the allowlist, so it always succeeds as recovery. Net impact: a post-creation blacklist forces the reject path (sender gets tokens back), not permanent fund loss. But the accept callback fails unnecessarily — the admin is already authorizing the transfer via `force-accept`, so re-checking the allowlist adds no security value.

RECOMMENDATION

Skip allowlist checks on `ILockedPayout_ForceAccept`. The admin is already authorizing the transfer by exercising the force choice. Alternatively, snapshot the allowlist decision at offer creation time so `force-accept` is not affected by later policy changes.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/AttackVectorTest.daml#L226>

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-06

PENDING

stateRoot accepts any hex length

CATEGORY	Validation
LOCATION	StateCommitment/StateCommitment.daml:18
REVIEW	Main Review

DESCRIPTION

`StateCommitment.ensure` only validates `isValidHex stateRoot`. It does not enforce a fixed length. A 2-char or 128-char hex string both pass validation.

The codebase already has `isBytes32Hex` (used for peer addresses in `OfFactoryConfig.ensure`) but does not apply it to `stateRoot`. If downstream verifiers or bridge proofs assume a 32-byte (64-char) root, this creates an integration mismatch.

Impact is limited because only the committer (`owner`, the trusted sequencer) can create `StateCommitment` contracts. This is a defense-in-depth gap, not a direct exploit.

RECOMMENDATION

Add `isBytes32Hex stateRoot` to the `ensure` clause (or `T.length (strip0x stateRoot) == 64`) if bytes32 is the intended format. The existing `isBytes32Hex` helper from `DA.Crypto.Text` can be reused directly.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/AttackVectorTest.daml#L301>

RESOLUTION

Status: Pending.

INFORMATIONAL I-07

ACKNOWLEDGED

Registry SetPartyId accepts mismatched hints

CATEGORY Validation
 LOCATION Registry.daml:93-112
 REVIEW Main Review

DESCRIPTION

`SetPartyId` accepts any `Text` as the `hint` parameter (line 95) without validating it matches the caller's actual party hint. The fingerprint is correctly extracted from the caller (line 108), but the hint is free-form user input. If a caller passes a hint that differs from their real party prefix — whether by typo, mistake, or intent — the Registry stores the wrong mapping.

Later, `GetPartyId` reconstructs the party ID as `hint <> "::1220" <> fingerprint`. With a mismatched hint, this produces a different party than the caller. The fee is charged, the wrong hint is stored permanently, and all future lookups for this fingerprint silently return the wrong party.

If an off-chain executor uses the Registry to resolve the `receiverHint` for `lzReceive`, the reconstructed party would differ from the intended receiver, and tokens would be minted to the wrong party.

Example: Caller is `Alice::1220abcd...`. They call `SetPartyId` with `hint = "WrongHint"`. `GetPartyId` now reconstructs `WrongHint::1220abcd...` — a different party. No error, no revert.

RECOMMENDATION

Derive the hint from the caller's party ID instead of accepting it as user input. The pattern already exists in the codebase (`getPartyPrefix` in `TestUtils.daml`): `partyToText caller` → split on `"::"` → take the first part. Additionally, validate that the extracted hint is non-empty (length `> 0`). This eliminates the mismatch entirely.

```
```plain text
choice SetPartyId : ContractId Registry
with
caller : Party
amuletCid : ContractId Amulet
amuletContext : ChoiceContext
controller caller
do
```

**I-07** Registry SetPartyId accepts mismatched hints

CONTINUED

```
let partyStr = partyToText caller
let parts = T.splitOn ":" partyStr
let hint = case parts of
(prefix :: _) | prefix /= "" -> prefix
_ -> error "ERR_REGISTRY_EMPTY_HINT"
let fingerprint = extractFingerprint caller
...
```
```

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/RegistrySecurityTest.daml#L163>

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-08

ACKNOWLEDGED

External payouts remain sender-cancellable

CATEGORY Unexpected Behavior

LOCATION FeeUtils.daml

REVIEW Main Review

DESCRIPTION

`distributeAmulet` does not actually finish an external payout. If the recipient does not have a matching `TransferPreapproval`, it only creates an `AmuletTransferInstruction` backed by a locked amulet and then returns. The calling flow still behaves as if the money was paid: `Request` is archived, governance slashing moves on and `Executor.NativeDrop` reports success.

That instruction is still pending and the sender side can later unwind it through the normal `TransferInstruction_Withdraw` and lock-expiry path. So these payouts are not final when the parent contract records them as done. Because `distributeAmulet` does not persist the created instruction ids or keep the parent workflow open, the protocol can record a completed payout even though the transfer can still be reversed later.

For example, let's imagine a rejected governance proposal. The proposal fee is supposed to be split among committee members, but if the leader is the payout provider and some members are external recipients, those members do not receive final amulets. They only get pending transfer instructions. The protocol still treats the slash as paid out, yet the leader remains on the sender side of a flow that can later be unwound through the expiry path. In practice, the slash distribution becomes a soft promise instead of a final transfer. The same problem affects request payouts and native drops whenever an external recipient is expected to rely on immediate delivery.

RECOMMENDATION

Do not mark an external payout as complete unless it has actually settled. The cleanest fix is to require `TransferPreapproval` for payouts that must be final in the same transaction. If the two-step transfer model has to stay, `distributeAmulet` should return and persist the created instruction ids and the parent flow should stay pending until those instructions are accepted or otherwise reach a state that cannot be cancelled by the sender side anymore.

I-08 External payouts remain sender-cancellable

CONTINUED

— **RESOLUTION**

Status: Acknowledged.

INFORMATIONAL I-09

PENDING

DEFAULT_ADMIN_ROLE exceeds documented RBAC

| | |
|----------|-----------------------|
| CATEGORY | Warning |
| LOCATION | 0ftFactoryConfig.daml |
| REVIEW | Main Review |

DESCRIPTION

The docs describe separate operational roles like `oappAdmin` and `allowlistAdmin`, which makes it sound like peer setup, observer updates and allowlist mode can be delegated separately. The code does not actually enforce that split. Those actions all sit behind `DEFAULT_ADMIN_ROLE` and the same role can also change `feeDeposit`, create arbitrary OApp requests and call `RecoverFunds`.

So in practice `DEFAULT_ADMIN_ROLE` is not a narrow config role. It is a broad operator role with both configuration and economic power. A team can hand out what they think is routine OApp admin access and accidentally also give that party the ability to redirect fee flows or recover funds from non-allowlisted holders.

RECOMMENDATION

Split `DEFAULT_ADMIN_ROLE` into narrower roles that match the documented permission model. At minimum, keep OApp configuration, allowlist mode, fee recipient changes, request creation and fund recovery separate. If the broad scope is intentional, the docs and runbooks should say clearly that

`DEFAULT_ADMIN_ROLE` is a super-admin role and should not be delegated as if it were just `oappAdmin`.

RESOLUTION

Status: Pending.

INFORMATIONAL I-10

ACKNOWLEDGED

SEND_AND_CALL compose payload is ignored

| | |
|----------|---------------------|
| CATEGORY | Unexpected Behavior |
| LOCATION | LzReceive.daml |
| REVIEW | Main Review |

DESCRIPTION

The send side treats a non-empty `composeMsg` as a real `SEND_AND_CALL` message. It switches message type, encodes the extra payload and includes compose-related options. But on receive, `LzReceive` only decodes the message, resolves the recipient, applies rate limiting and mints tokens. The compose payload is never used.

So the protocol accepts a message that looks like "send tokens and then run follow-up logic", but settles it as a plain token transfer. It does not fail closed. The transfer can succeed and the compose leg is simply dropped.

For example, an integrator can send tokens to a remote chain expecting the compose payload to deposit them into a vault or trigger some follow-up action on receipt. The receive side mints the tokens to the user, but the compose action never runs. From the sender's point of view the `SEND_AND_CALL` transfer succeeded, but the destination-side workflow was silently skipped.

RECOMMENDATION

Reject any non-empty `composeMsg` until the receive side actually supports it. If compose support is intended, the receive path needs to carry the decoded payload through settlement and dispatch the follow-up action instead of silently minting and stopping there.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-11

ACKNOWLEDGED

Admin paths bypass holding lock invariants

CATEGORY Validation

LOCATION `OftFactory.daml:378` (`RecoverFunds`), `BurnMintImpl.daml:30` (`BurnMint`), `OftTransferRule.daml:48` (`ExecuteAsAdmin`)

REVIEW Main Review

DESCRIPTION

The lock model (`Lock` with `lockers` list) is meant to prevent holdings from being consumed without the lock holder's consent. Three admin paths bypass this invariant in different ways:

1. `RecoverFunds` (`OftFactory.daml:378`): Archives locked holdings without any lock check. Callable by admin or `DEFAULT_ADMIN_ROLE` holder.
2. `BurnMint` (`BurnMintImpl.daml:30`): Archives locked holdings without any lock check. Callable by admin.
3. `TransferRule_ExecuteAsAdmin` (`OftTransferRule.daml:48`): Calls `applyHoldingAction(UnlockMergeSplitTransfer)` which checks that the holding IS locked (`mustBeLocked`) but does not verify the caller is one of the `lockers` . The admin can unlock and transfer a holding locked by a different lock holder.

Every other holding-consuming operation checks lock state via `mustBeLocked` / `mustBeUnlocked` . These three admin paths are the only ones that skip or weaken the check.

Impact (paths 1-2): If a holding is locked by a pending `lzSend` callback, `RecoverFunds` or `BurnMint` can archive it. The holding CID in `PendingCallback.callbackContext` becomes stale, so neither accept nor reject finalization succeeds. The `PendingCallback` is permanently orphaned and the sender loses tokens with no recovery path.

Impact (path 3): `TransferRule_ExecuteAsAdmin` is the path exercised by `ILockedPayout_ForceAccept` during callback settlement. The lock is supposed to ensure only the designated lock holder can settle the payout. But `applyHoldingAction` only calls `mustBeLocked` — it does not verify the caller is one of the `lockers` . In the current single-admin-per-instrument design this is benign, but if the lock model is ever extended to support third-party lock holders (e.g., escrow agents, DvP coordinators), any admin could settle payouts locked by a different party.

RECOMMENDATION

1. `RecoverFunds` and `BurnMint` (paths 1-2): Add `mustBeUnlocked oft.lock` before `archive oftCid` . This prevents these functions from consuming holdings locked by pending callbacks.

I-11 Admin paths bypass holding lock invariants

CONTINUED

2. `TransferRule_ExecuteAsAdmin` (path 3): Verify that the exercising admin is one of the current `lockers` before proceeding with `UnlockMergeSplitTransfer`. If the protocol needs a special admin-settleable lock type, mark it explicitly and restrict the admin path to that lock type only.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/RecoverFundsSecurityTest.daml>

— RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-12

PENDING

Batch commitments leak request linkage

| | |
|----------|--|
| CATEGORY | Warning |
| LOCATION | CommitterImpl.daml, StateCommitment.daml |
| REVIEW | Main Review |

DESCRIPTION

`Committer.CommitterImpl.commitBatchImpl` fetches each request in the batch, collects every request's `observers`, unions them into `allViewers` and then creates a single `StateCommitment` that stores the full `requestCids` array. Because `StateCommitment` is visible to all of those `viewers`, any observer of one request becomes able to see the contract identifiers of every other request that was included in the same batch and to learn that they were processed together.

This leaks cross-request linkage that did not exist at the individual request layer. An observer that was only meant to track request A can infer the existence of requests B, C and D and correlate them as part of one settlement batch. The leak does not expose the full payloads of the other requests, but it does reveal metadata that Canton normally tries to minimize across unrelated visibility domains. In the standard OFT send and receive flows the request observers are usually internal, so the practical impact depends on whether a deployment batches requests that have external observers. Where that does happen, batching creates an avoidable privacy downgrade.

RECOMMENDATION

Do not expose the raw full-batch `requestCids` list to the union of all request observers. If observers from different visibility domains need proof that their request was committed, publish per-request inclusion artifacts instead, or store only a commitment such as a batch hash or Merkle root and provide request-specific inclusion proofs off-ledger. If the current full-list design must remain, restrict batching so that only requests with the same intended visibility set are committed together.

RESOLUTION

Status: Pending.

INFORMATIONAL I-13

PENDING

State commitments are not isolated per committer

| | |
|----------|--------------------|
| CATEGORY | Warning |
| LOCATION | CommitterImpl.daml |
| REVIEW | Main Review |

— DESCRIPTION

`createStateCommitmentImpl` accepts any visible `prevStateCommitmentCid`, archives it and creates a successor without checking which `Committer` originally created that predecessor. `StateCommitment` itself does not record a committer identity either, so commitments are keyed only by owner.

That means two committers owned by the same party do not have isolated histories. If one owner deploys `committerA` and `committerB`, `committerB` can take a live predecessor that was created by `committerA`, archive it and continue the chain as if it belonged to the same lineage.

This is best understood as a design constraint that is not enforced explicitly. The current implementation behaves like "one active commitment chain per owner", not "one chain per committer". That can surprise upgrades, redeployments, or multi-committer setups if operators assume each `Committer` instance has its own independent head. The off-ledger SDK follows the same owner-scoped model because it resolves the latest commitment by owner rather than by committer identity.

That lets two committers owned by the same party operate on the same chain head. If one owner deploys `committerA` and `committerB`, `committerB` can take a live predecessor that was created by `committerA`, archive it and continue the chain as if it had produced that earlier state itself.

This breaks committer isolation. A redeployed or misconfigured committer can take over another committer's head and the off-ledger SDK makes the problem easier to hit because it also discovers the latest commitment by owner rather than by committer identity.

— RECOMMENDATION

If one active chain per owner is the intended design, document that constraint clearly and avoid running multiple committers for the same owner.

If per-committer isolation is intended, bind every `StateCommitment` to a stable chain identity and reject predecessors from any other chain. The SDK should resolve the latest commitment by that same identity instead of scanning every active commitment for the owner.

I-13 State commitments are not isolated per committer

CONTINUED

— **RESOLUTION**

Status: Pending.

INFORMATIONAL I-14

PENDING

DiscoveryRegistry accepts foreign gateway IOApps

| | |
|----------|--|
| CATEGORY | Validation |
| LOCATION | DiscoveryRegistry.daml:58-79 (SetUrl choice) |
| REVIEW | Main Review |

DESCRIPTION

SetUrl validates that the caller is the IOApp's instrumentId.admin and a signatory, but never checks that `oappGateway == handler`. A handler-A registry can store discovery URLs for `IOApps` belonging to a different gateway. No on-chain economic impact — off-chain trust-anchor integrity issue only.

RECOMMENDATION

Add `oappGateway == handler` check in SetUrl after fetching the IOApp view.

PROOF OF CONCEPT <https://github.com/GuardianOrg/canton-ver-team1-1773698451456/blob/guardian/daml/tests/daml/Audit/DiscoveryRegistryTrustTest.daml>

RESOLUTION

Status: Pending.

INFORMATIONAL I-15

PENDING

Uppercase 0X options fail TS bytes schema

| | |
|----------|----------------------|
| CATEGORY | Validation |
| LOCATION | EnforcedOptions.daml |
| REVIEW | Main Review |

DESCRIPTION

When no enforced options are configured for a route, `combineOptions` returns `extraOptions` unchanged. `prepareSend` forwards that text into both `OfFactory_BuildQuote` and the outbound request path. That means a caller can supply `extraOptions = "0X000301..."` and DAML will treat it as acceptable outbound data.

The offchain side does not accept the same syntax. The `bytes` codec used by the validator container only allows lowercase `0x` or no prefix at all. Its schema is `/^(0x)?[a-fA-F0-9]*\$/`, so `0X...` is rejected before any byte decoding happens. The result is a boundary mismatch: quote/request creation succeeds on-ledger, but the validator-side send path rejects the payload later as malformed input.

RECOMMENDATION

Canonicalize raw option hex before it leaves DAML. The smallest safe fix is to strip both `0x` and `0X` at the send boundary and emit one canonical form or reject uppercase-prefixed input explicitly in send-param validation.

RESOLUTION

Status: Pending.

INFORMATIONAL I-16

PENDING

RequestFactory allows handler==treasury

| | |
|----------|---------------------------|
| CATEGORY | Access Control |
| LOCATION | RequestFactory.daml:37-38 |
| REVIEW | Main Review |

DESCRIPTION

`RequestFactory` has `signatory handler, treasury` with no `ensure` clause enforcing `handler /= treasury`. When both are the same party, a single party controls request acceptance AND fee custody — it can accept requests and route all fees to itself with no second-party oversight. `OfftTransferOffer` enforces `sender /= receiver` (line 54), but `RequestFactory` omits this check.

RECOMMENDATION

Add `ensure handler /= treasury` to `RequestFactory`.

RESOLUTION

Status: Pending.

INFORMATIONAL I-17

PENDING

Governance member list validation gaps

| | |
|----------|------------------------------------|
| CATEGORY | Validation |
| LOCATION | Governance/Governance.daml:108-110 |
| REVIEW | Main Review |

DESCRIPTION

The Governance template ensure clause only checks `not (null members) && minFee > 0.0`. Three validations are missing:

1. No duplicate members check: `members = [alice, alice, alice]` is accepted. The apparent 3-member committee is actually 1-member. The `slashProposalFee` helper deduplicates internally via `dedup \ $ sort \ $ filter`, so fee distribution sees only 1 unique member while observers see 3 entries. This misrepresents committee diversity.
2. No multisig-not-in-members check: the multisig party can appear in the members list. Since multisig is both a member (passes the `elem members` check in `Governance_Propose`) and the controller for `Governance_Accept`, a single party can propose and accept its own governance proposals without any other member's involvement.
3. Single-member slash silently burns fee: when the proposer is the only member, `filter (/= proposer) [proposer]` yields an empty list, `divisor = 0`, and the `if divisor > 0` branch is skipped. The fee stays in the treasury with no distribution and no error. The proposer loses their fee and no one receives it.

RECOMMENDATION

Add validation to the Governance ensure clause: `length (dedup members) == length members` to reject duplicates, and `multisig \ notElem \ members \` to prevent the multisig party from appearing in the member list.

RESOLUTION

Status: Pending.

INFORMATIONAL I-18

PENDING

SetTransferRule accepts unchecked ITransferRule

| | |
|----------|-----------------------------------|
| CATEGORY | Validation |
| LOCATION | Oft/OftFactoryConfig.daml:260-269 |
| REVIEW | Main Review |

DESCRIPTION

`OftFactoryConfig_SetTransferRule` lets the OFT admin install any `ITransferRule` whose view reports the same `instrumentId`. It does not verify that the rule is the intended canonical transfer-settlement backend for the OFT.

`TransferFactory_Transfer` later stores `config.transferRuleCid` on the created `OftTransferOffer`, and `TransferInstruction_Accept` then trusts that stored rule to execute settlement. Because of that, an admin-side wrong-instance or custom-rule swap can redirect accepted transfers away from the intended receiver while still satisfying the current setter checks.

This is an admin-controlled trust-anchor bug, not a user-triggerable path. The impact is that OFT transfer settlement is not bound to a canonical rule implementation.

RECOMMENDATION

Validate `newTransferRuleCid` against a stronger trust boundary than `instrumentId` alone. At minimum, require the expected admin/signatory and canonical implementation family; ideally bind one canonical transfer rule per OFT and reject custom or sibling substitutions.

RESOLUTION

Status: Pending.

INFORMATIONAL I-19

PENDING

MINTER_ROLE and BURNER_ROLE are never enforced

| | |
|----------|---------------------|
| CATEGORY | Trust Assumptions |
| LOCATION | OftFactoryRbac.daml |
| REVIEW | Main Review |

DESCRIPTION

`validRoleNames` accepts `MINTER_ROLE` and `BURNER_ROLE`, so the admin can grant and revoke those roles through `IAccessControl`. The problem is that no production choice checks either role. `OftFactoryConfig` uses the other role constants throughout its delegated admin choices, but there is no corresponding `MINTER_ROLE` or `BURNER_ROLE` authorization gate in that file. The burn/mint implementation also runs without consulting `roles`.

Consequently, operators can assign these roles and believe mint or burn authority has been delegated when it has not. This is especially misleading because the test suite grants both roles in `daml/tests/daml/Oft/RbacTest.daml:182-206`, and the TypeScript config serializer also treats both names as valid input in `scripts/src/contracts/oft-adapter/oft-factory-config.ts:196-219`. Granting either role also makes the assignee an observer of the full config through `roleParties`, so the dead roles widen disclosure without enabling any real capability.

RECOMMENDATION

Remove `MINTER_ROLE` and `BURNER_ROLE` from the accepted role namespace until they are wired to concrete authorization checks. If delegated mint or burn authority is intended, add explicit gates for those roles in the relevant production choices.

RESOLUTION

Status: Pending.

INFORMATIONAL I-20

PENDING

DEFAULT_ADMIN_ROLE need separate disclosure

| | |
|----------|-------------------|
| CATEGORY | Trust Assumptions |
| LOCATION | OftFactory.daml |
| REVIEW | Main Review |

DESCRIPTION

`OftFactory` trusts `DEFAULT_ADMIN_ROLE` holders in several implementation checks, including `oApp_createRequestImpl`, `oApp_setPeerImpl`, and `OftFactory_RecoverFunds`. However, the template only declares `oappGateway` as an observer. Role assignment happens on `OftFactoryConfig`, where role holders become observers of the config contract, not of the factory contract or the holdings they may need to fetch.

Consequently, granting `DEFAULT_ADMIN_ROLE` does not by itself let the grantee execute every delegated admin action. The grantee still needs a stakeholder to disclose `OftFactory` and, for some operations, related holdings. The tests make this explicit. `testRoleHolderCanRecoverFunds` first obtains disclosures for the factory, config, and holding before the role holder can exercise `OftFactory_RecoverFunds` in `daml/tests/daml/Oft/RbacTest.daml:827-839`. If operational runbooks assume that role grant alone is enough, time-sensitive actions can fail until a separate disclosure step happens.

RECOMMENDATION

Treat visibility as part of role assignment. Either make the relevant delegated operators observers of `OftFactory` and any other contracts they must fetch, or document and automate the disclosure workflow whenever `DEFAULT_ADMIN_ROLE` is granted.

RESOLUTION

Status: Pending.

§ 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.