

UTEXO BRIDGE SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

AUGUST 12, 2026

1 EXECUTIVE SUMMARY

OXORIO

1.1 EXECUTIVE SUMMARY

This document presents the smart contracts security audit conducted by Oxorio for the UTEXO Bridge.

UTEXO Protocol develops a trust-minimized bridge connecting Bitcoin/RGB assets with EVM networks, with the stated design goal that even a compromise of the off-chain operators must not allow assets to be stolen from the bridge.

The UTEXO Bridge Solidity layer is the on-chain (EVM) side of this system: a set of contracts that custody the bridged pool, account for inflows and outflows, route releases through pluggable finality verifiers and settlement modules, govern configuration through a timelocked federation multisig, and integrate a LayerZero adapter for the USDT0 source flow.

The audit process involved a comprehensive approach, including manual code review, automated analysis, and extensive testing and simulations of the smart contracts to assess the project's security and functionality. The audit covered a total of 22 smart contracts, encompassing 4,589 total lines (2,435 lines of code). The codebase was thoroughly examined, with the audit team referencing the provided protocol specification to address any questions regarding the expected behavior. For an in-depth explanation of the smart contract security audit methodology, please refer to the Security Assessment Methodology section of this document.

Throughout the audit, a collaborative approach was maintained with UTEXO Protocol to address all concerns identified within the audit's scope. Each issue has been either resolved or formally acknowledged by UTEXO Protocol, contributing to the robustness of the project.

As a result, following a comprehensive review, our auditors have verified that the UTEXO Bridge, as of audited commits `dev@ea5de9b4` (bridge-smart-contracts) and `88fbc6a1` (usdt0-smart-contracts), has met the security and functionality requirements established for this audit, based on the code and documentation provided, and operates as intended within the defined scope.

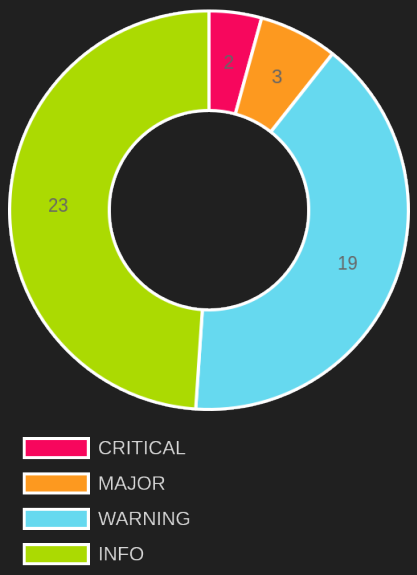
1.2 SUMMARY OF FINDINGS

The table below provides a comprehensive summary of the audit findings, categorizing each by status and severity level. For a detailed description of the severity levels and statuses of findings, see the Findings Classification Reference section.

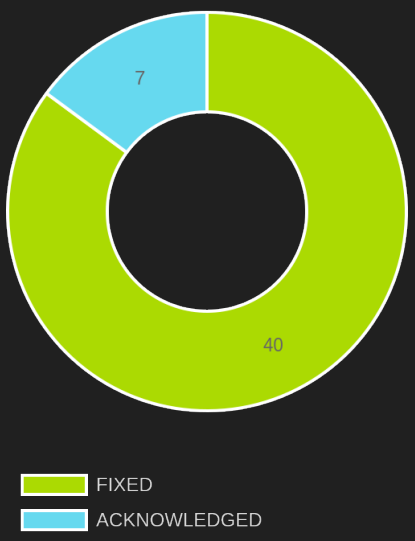
Detailed technical information on the audit findings, along with our recommendations for addressing them, is provided in the Findings Report section for further reference.

All identified issues have been addressed, with UTEXO Protocol fixing them or formally acknowledging their status.

Severity	TOTAL	NEW	FIXED	ACKNOWLEDGED	NO ISSUE
CRITICAL	**2**	0	1	1	0
MAJOR	**3**	0	2	1	0
WARNING	**19**	0	16	3	0
INFO	**23**	0	21	2	0
TOTAL	**47**	**0**	**40**	**7**	**0**



Issue distribution by severity



Issue distribution by status

2 AUDIT OVERVIEW

CONTENTS

1. EXECUTIVE SUMMARY 2

1.1. EXECUTIVE SUMMARY 3

1.2. SUMMARY OF FINDINGS 4

2. AUDIT OVERVIEW 5

2.1. DISCLAIMER 10

2.2. PROJECT BRIEF 11

2.3. PROJECT TIMELINE 12

2.4. AUDITED FILES 13

2.5. PROJECT OVERVIEW 15

2.6. CODEBASE QUALITY ASSESSMENT 16

2.7. FINDINGS BREAKDOWN BY FILE 18

2.8. CONCLUSION 19

3. FINDINGS REPORT 20

3.1. CRITICAL 21

C-01 The enclave-signed money path has no on-chain constraint: a quorum can drain the entire pool in one transaction in MultisigProxy.sol, Bridge.sol 21

C-02 The federation can replace the enclave signer set with no on-chain attestation, collapsing the two trust domains into one in MultisigProxy.sol 26

3.2. MAJOR 30

M-01 The on-chain finality proof is not bound to the release context; burnId is the sole replay guard in RGBVerifier.sol, Bridge.sol 30

M-02 Inbound lzCompose does not bind sourceChainId or entrypoint trust to the LayerZero transport srcEid in UtexoLZAdapter.sol 36

M-03 The TEE allowlist places no class restriction on selectors: the federation can grant the enclave un-timelocked control of privileged setters in MultisigProxy.sol 40

3.3. WARNING 43
W-01 Fee-on-transfer token makes the Bridge ledger overstate custody and bricks release of the recorded amount in Bridge.sol 43
W-02 Rotating the route registry makes existing fundsInRecords unreachable in Bridge.sol, RouteRegistry.sol 46
W-03 setGlobalDefaults and setCommissionRule accept configurations that cause netAmount underflow in CommissionManager.sol 49
W-04 CommissionManager accepts a NATIVE + FUNDS_OUT config that bricks the release path in CommissionManager.sol, Bridge.sol 51
W-05 Bridge.fundsOut is not gated by whenNotPaused in Bridge.sol, MultisigProxy.sol 53
W-06 RGBVerifier ignores the confirmations value returned by BtcRelay in RGBVerifier.sol 55
W-07 destinationChainId is not constrained to block.chainid on fundsOut in Bridge.sol 57
W-08 Predictable operationId allows blocking of victim transactions in Bridge.sol 59
W-09 Malformed lzCompose payload leaves USDT0 stuck without a _stuckFunds record in UtexoLZAdapter.sol 61
W-10 Issuer blocklist plus push delivery bricks a release to a frozen recipient in Bridge.sol, UtexoLZAdapter.sol 63
W-11 SetTimelockDuration has no lower bound; the federation can disable the governance delay entirely in MultisigProxy.sol 65
W-12 Signer-set thresholds have no on-chain floor: a 1-of-N enclave or federation is accepted in MultisigProxy.sol 67
W-13 DOMAIN_SEPARATOR is immutable and not recomputed on a chain-id change, enabling cross-fork signature replay in MultisigProxy.sol 69
W-14 No disjointness check between the enclave and federation signer sets in MultisigProxy.sol 72
W-15 AdminExecuteCommissionManager bypasses the commissionRecipient pin on commission withdrawals in MultisigProxy.sol, CommissionManager.sol 74
W-16 Outbound sendOut is not bound on-chain to its paired fundsOut; the link lives only in the TEE in UtexoLZAdapter.sol, MultisigProxy.sol, Bridge.sol 77
W-17 In-flight token commission inflates getContractBalance during the onFundsIn hook in Bridge.sol, BridgeBase.sol 80

W-18 Federation proposals use the live timelockDuration instead of a value snapshotted at creation in MultisigProxy.sol 82
W-19 An attacker can front-run lzCompose with an insufficient native value to park inbound deposits in the _stuckFunds map of UtexoLZAdapter.sol 84
3.4. INFO 86
I-01 Enclave execute / executeBatch deadlines have no upper bound in MultisigProxy.sol 86
I-02 Chainlink ETH/USD feed degradation reverts every native-commission route in CommissionManager.sol 88
I-03 Strict equality NativeValueMismatch does not tolerate Chainlink price drift in Bridge.sol 90
I-04 receiveTokenCommission balance-delta accounting absorbs unsolicited transfers in CommissionManager.sol 93
I-05 Unbounded settlementData storage write in the lzCompose catch branch in UtexoLZAdapter.sol 95
I-06 Federation signer set has quadratic validation cost and a 256-bit bitmap ceiling in MultisigProxy.sol 97
I-07 Zero-amount fundsIn succeeds and adds redundant entries to the event log in Bridge.sol ... 99
I-08 Floor-division precision loss allows dust amounts to bridge with zero commission in CommissionManager.sol 101
I-09 proposalNonce is shared between the regular and emergency proposal lanes in MultisigProxy.sol 104
I-10 RgbSettlementModule.beforeFundsOut exits the validation loop early in RgbSettlementModule.sol 107
I-11 High event-log gas cost from unbounded destinationAddress and sourceAddress in Bridge.sol 109
I-12 Unbounded proof length accepted by fundsOut in Bridge.sol 111
I-13 High cross-chain event-log gas cost from unbounded destinationAddress in UtexoLZAdapter.sol 113
I-14 Chainlink consumer omits the Arbitrum L2 sequencer-uptime check and a min/max circuit-breaker bound in CommissionManager.sol 115
I-15 Owned contracts use single-step Ownable; a mis-addressed transferOwnership orphans governance irreversibly in BridgeBase.sol, CommissionManager.sol, RouteRegistry.sol, MultisigProxy.sol 119

I-16 sendOut refunds native surplus to the transaction executor via tx.origin instead of a signed refund recipient in UtexoLZAdapter.sol 122
I-17 deposit refunds native surplus to msg.sender, which may be a contract that cannot receive ETH in UtexoSourceEntrypoint.sol 124
I-18 A proposal can be created with a deadline shorter than timelockDuration, leaving it unexecutable in MultisigProxy.sol 126
I-19 RgbSettlementModule.beforeFundsOut iterates an uncapped TEE-supplied fundsInIds[] in RgbSettlementModule.sol 128
I-20 Signer-set / threshold validation is deferred to execute-time in MultisigProxy.sol 130
I-21 setLZAdapter / UpdateLZAdapter lack a zero-check and a rotate-vs-disable event distinction in Bridge.sol 132
I-22 Inconsistent compiler pragma across sources in MultisigProxy.sol 134
I-23 Code-quality set: missing indexed event fields, PUSH0 target note, unused custom error in ICommissionManager.sol 135
4. APPENDIX 137
4.1. SECURITY ASSESSMENT METHODOLOGY 138
4.2. CODEBASE QUALITY ASSESSMENT REFERENCE 140
Rating Criteria 141
4.3. FINDINGS CLASSIFICATION REFERENCE 142
Severity Level Reference 142
Status Level Reference 142
4.4. ABOUT OXORIO 144

2.1 DISCLAIMER

At the request of the client, Oxorio consents to the public release of this audit report. The information contained herein is provided "as is" without any representations or warranties of any kind. Oxorio disclaims all liability for any damages arising from or related to the use of this audit report. Oxorio retains copyright over the contents of this report.

This report is based on the scope of materials and documentation provided to Oxorio for the security audit as detailed in the Executive Summary and Audited Files sections. The findings presented in this report may not encompass all potential vulnerabilities. Oxorio delivers this report and its findings on an as-is basis, and any reliance on this report is undertaken at the user's sole risk. It is important to recognize that blockchain technology remains in a developmental stage and is subject to inherent risks and flaws.

This audit does not extend beyond the programming language of smart contracts to include areas such as the compiler layer or other components that may introduce security risks. Consequently, this report should not be interpreted as an endorsement of any project or team, nor does it guarantee the security of the project under review.

THE CONTENT OF THIS REPORT, INCLUDING ITS ACCESS AND/OR USE, AS WELL AS ANY ASSOCIATED SERVICES OR MATERIALS, MUST NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE. Third parties should not rely on this report for making any decisions, including the purchase or sale of any product, service, or asset. Oxorio expressly disclaims any liability related to the report, its contents, and any associated services, including, but not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Oxorio does not warrant, endorse, or take responsibility for any product or service referenced or linked within this report.

For any decisions related to financial, legal, regulatory, or other professional advice, users are strongly encouraged to consult with qualified professionals.

2.2 PROJECT BRIEF

Title	Description
Client	UTEXO Protocol
Project name	UTEXO Bridge
Category	Cross-chain bridge (RGB ↔ EVM)
Website	https://utexo.com
Documentation	https://docs.utexo.com
Repository	bridge-smart-contracts usdt0-smart-contracts
Initial Commit	`80485109` `13560f5e`
Final Commit	`ea5de9b4` `88fbc6a1`
Platform	EVM
Network	Ethereum (with Bitcoin/RGB and LayerZero integration)
Languages	Solidity
Lead Auditor	Alexander Mazaletskiy - am@oxor.io
Project Manager	Ales Kuznetsova - ales@oxor.io

2.3 PROJECT TIMELINE

The key events and milestones of the project are outlined below.

Date	Event
May 25, 2026	The audit team initiated work on the project.
June 1, 2026	Submission of the preliminary audit report #1.
June 8, 2026	Submission of the preliminary audit report #2.
June 11, 2026	Submission of the preliminary audit report #3.
June 15, 2026	Submission of the preliminary audit report #4.
June 22, 2026	Submission of the audit report.
July 15, 2026	The team began the re-audit of the client's remediations.
July 21, 2026	Submission of the final audit report after the remediation re-audit.
August 11, 2026	Re-audit of the additional remediation PRs and their integration into `dev`.
August 12, 2026	Submission of the final audit report.

2.4 AUDITED FILES

The following table contains a list of the audited files. The scc tool was used to count the number of lines and assess complexity of the files.

	File	Lines	Blanks	Comments	Code	Complexity
1	ethereum/src/MultisigProxy.sol	1027	159	125	**743**	22%
2	ethereum/src/CommissionManager.sol	567	65	202	**300**	24%
3	ethereum/src/interfaces/IMultisigProxy.sol	420	42	125	**253**	0%
4	ethereum/src/Bridge.sol	326	37	89	**200**	17%
5	ethereum/src/interfaces/ICommissionManager.sol	234	47	38	**149**	0%
6	ethereum/src/RouteRegistry.sol	181	26	78	**77**	13%
7	ethereum/src/interfaces/IBridge.sol	161	18	78	**65**	0%
8	ethereum/src/settlement/RgbSettlementModule.sol	149	21	79	**49**	29%
9	ethereum/src/interfaces/IRouteRegistry.sol	115	15	68	**32**	0%
10	ethereum/src/BridgeBase.sol	88	17	37	**34**	12%
11	ethereum/src/interfaces/RouteTypes.sol	84	4	55	**25**	0%
12	ethereum/src/BaseBridge.sol	79	14	33	**32**	13%
13	ethereum/src/verifiers/RGBVerifier.sol	69	10	43	**16**	13%
14	ethereum/src/interfaces/ISettlementModule.sol	42	3	33	**6**	0%
15	ethereum/src/settlement/NullSettlementModule.sol	32	3	20	**9**	0%
16	ethereum/src/interfaces/IFinalityVerifier.sol	30	2	23	**5**	0%
17	ethereum/src/interfaces/IBtcRelayView.sol	30	5	14	**11**	0%
18	ethereum/src/verifiers/NullVerifier.sol	27	2	18	**7**	0%
19	ethereum/src/UtexoLZAdapter.sol	385	48	144	**193**	18%
20	ethereum/src/interfaces/IUtexoLZAdapter.sol	220	24	113	**83**	0%
21	ethereum/src/UtexoSourceEntrypoint.sol	213	24	75	**114**	13%
22	ethereum/src/interfaces/IUtexoSourceEntrypoint.sol	110	11	67	**32**	0%
	Total	**4589**	**597**	**1557**	**2435**	**15%**

Lines: The total number of lines in each file. This provides a quick overview of the file size and its contents.

Blanks: The count of blank lines in the file.

Comments: This column shows the number of lines that are comments.

Code: The count of lines that actually contain executable code. This metric is essential for understanding how much of the file is dedicated to operational elements rather than comments or whitespace.

Complexity: This column shows the file complexity per line of code. It is calculated by dividing the file's total complexity (an approximation of cyclomatic complexity that estimates logical depth and decision points like loops and conditional branches) by the

number of executable lines of code. A higher value suggests greater complexity per line, indicating areas with concentrated logic.

2.5 PROJECT OVERVIEW

The UTEXO Bridge moves value between the Bitcoin/RGB world and EVM networks. On the EVM side, `Bridge` custodies the pooled token and exposes the two value paths: `fundsIn` (inflow, user-funded and accounted by a settlement module) and `fundsOut` (outflow, an enclave-signed release). `BaseBridge` / `BridgeBase` hold the shared storage and pausing primitives.

Release routing is pluggable. `RouteRegistry` maps a `(sourceChainId, destChainId)` route key to a `FinalityVerifier` and a `SettlementModule`. `RGBVerifier` checks block inclusion through an external `BtcRelay` (based on `atomiqlabs/atomiq-contracts-evm`), while `RgbSettlementModule` records inflow/outflow accounting; `NullVerifier` / `NullSettlementModule` are no-op variants. `CommissionManager` computes inbound and outbound fees against a Chainlink ETH/USD feed.

Configuration and the privileged money path are governed by `MultisigProxy`, which combines a timelocked federation multisig (`proposeX → executeProposal`) with an enclave signer set that can call `Bridge.fundsOut` directly. The `usdt0-smart-contracts` repository adds the LayerZero leg: `UtexoSourceEntrypoint` accepts deposits on the source chain and `UtexoLZAdapter` composes them onto the bridge on the destination chain.

2.6 CODEBASE QUALITY ASSESSMENT

The Codebase Quality Assessment table offers a comprehensive assessment of various code metrics, as evaluated by our team during the audit, to gauge the overall quality and maturity of the project's codebase. By evaluating factors such as complexity, documentation and testing coverage to best practices, this table highlights areas where the project excels and identifies potential improvement opportunities. Each metric receives an individual rating, offering a clear snapshot of the project's current state, guiding prioritization for refactoring efforts, and providing insights into its maintainability, security, and scalability. For a detailed description of the categories and ratings, see the Codebase Quality Assessment Reference section.

Category	Assessment	Result
Access Control	The federation multisig and the enclave signer set remain the core authority model. Remediation added a selector-class restriction on the TEE allowlist (`M-03` fixed), blocked `AdminExecute` from calling the release and rebalance selectors (PR #137), added an immutable non-zero timelock floor (`W-11` fixed), and enforced a minimum signer count with a strict-majority threshold (`W-12` fixed). The enclave-signed release path is now bounded on-chain by TVL-relative per-chain and global outflow limits (`C-01` fixed). The residual is the trust model itself: the federation can still replace the enclave signer set with unattested keys after the timelock (`C-02`, acknowledged). Related: `C-01`, `C-02`, `M-03`, `W-11`, `W-12`.	**Good**
Arithmetic	Solidity 0.8 checked arithmetic is used throughout, and the commission setters now reject configurations that could underflow or brick the release path (`W-03`, `W-04` fixed). The dust and global-minimum commission handling that previously allowed zero-commission dust transfers is now fixed (`I-08` fixed). Related: `W-03`, `W-04`, `I-08`.	**Good**
Complexity	Most contracts are small and focused; `MultisigProxy` concentrates a large share of the logic in its proposal/execute-by-type machinery. Signer-set size is now capped before the quadratic validation and bitmap use (`I-06` fixed). Related: `I-06`.	**Good**

Category	Assessment	Result
Data Validation	Fee-on-transfer accounting is now based on the measured balance delta rather than the requested amount (`W-01` fixed). The remaining items are accepted operational assumptions rather than open code gaps: route/plugin rotation can still strand historical records (`W-02`, acknowledged) and the local-release `destinationChainId` is not bound to `block.chainid` (`W-07`, acknowledged). Related: `W-01`, `W-02`, `W-07`.	**Good**
Decentralization	A minimum signer count and strict-majority threshold are now enforced (`W-12` fixed) and releases now rest on an M-of-N quorum bounded by TVL-relative on-chain outflow caps (`C-01` fixed), but the signer set remains governable without on-chain attestation (`C-02`, acknowledged). Related: `C-01`, `C-02`, `W-12`.	**Fair**
Documentation	A protocol specification exists and is largely consistent with the code. The privileged release path now carries the on-chain outflow bound the spec implies (`C-01` fixed). The residual documentation/spec gap is the cross-domain OpId binding, which still requires an RGB consignment-schema change (`M-01`, acknowledged). Related: `C-01`, `M-01`.	**Good**
External Dependencies	The system relies on `BtcRelay`, a Chainlink price feed, LayerZero, and USDT0. LayerZero inbound is now bound to the trusted transport source (`M-02` fixed) and `BtcRelay` confirmation depth and relay freshness are enforced on-chain (`W-06` fixed). Chainlink feed degradation still fails native-fee routes closed by design (`I-02`, acknowledged) and the L2 sequencer-uptime / circuit-breaker check is now enforced (`I-14` fixed). Related: `M-02`, `W-06`, `I-02`, `I-14`.	**Good**
Error Handling	Custom errors are used consistently and revert paths are explicit and readable. Related: no findings map to this metric.	**Good**
Logging and Monitoring	Events are emitted on the key paths; the previously missing `indexed` fields were added and the unbounded event-payload sizes were capped (`I-11`, `I-13`, `I-23` fixed). Related: `I-11`, `I-13`, `I-23`.	**Good**
Low-Level Calls	Token movements use `SafeERC20`; there are no unjustified raw `call` / `delegatecall` constructs in scope. Related: no findings map to this metric.	**Good**
Testing and Verification	Foundry proofs of concept reproduce the key findings, and the remediation was re-verified against the fix commits; comprehensive adversarial coverage of the federation and release paths is still not fully evidenced. Related: no findings map to this metric.	**Good**

2.7 FINDINGS BREAKDOWN BY FILE

This table provides an overview of the findings across the audited files, categorized by severity level. It serves as a useful tool for identifying areas that may require attention, helping to prioritize remediation efforts, and provides a clear summary of the audit results.

File	TOTAL	CRITICAL	MAJOR	WARNING	INFO
ethereum/src/MultisigProxy.sol	**19**	2	1	8	8
ethereum/src/Bridge.sol	**17**	1	1	9	6
ethereum/src/CommissionManager.sol	**8**	0	0	3	5
ethereum/src/UtexoLZAdapter.sol	**8**	0	1	4	3
ethereum/src/BridgeBase.sol	**2**	0	0	1	1
ethereum/src/RouteRegistry.sol	**2**	0	0	1	1
ethereum/src/settlement/RgbSettlementModule.sol	**2**	0	0	0	2
ethereum/src/verifiers/RGBVerifier.sol	**2**	0	1	1	0
ethereum/src/UtexoSourceEntrypoint.sol	**1**	0	0	0	1
ethereum/src/interfaces/ICommissionManager.sol	**1**	0	0	0	1
ethereum/src/interfaces/IMultisigProxy.sol	**1**	0	0	0	1
ethereum/src	**1**	0	0	0	1

2.8 CONCLUSION

A comprehensive audit was conducted on 22 smart contracts, initially revealing 2 critical and 3 major issues, along with numerous warnings and informational notes. The audit highlighted that the on-chain layer concentrates trust in the off-chain enclave and the federation multisig: the privileged enclave-signed release path lacked on-chain outflow bounds, and the federation can replace the enclave signer set with no on-chain attestation, collapsing the two intended trust domains into one. The major findings concern proof binding of releases, LayerZero source-chain binding, and the scope of calls the enclave can execute without a timelock.

Following our initial audit, UTEXO Protocol worked closely with our team to address the identified issues. The proposed changes focused on constraining the privileged money path on-chain (TVL-relative outflow limits, an outflow pause, and stronger release binding), binding LayerZero source-chain data to the trusted transport endpoint, restricting which calls the enclave can execute without a timelock, and hardening commission and accounting math against misconfiguration. Through multiple rounds of interaction, all identified issues have been successfully addressed or formally acknowledged.

As a result, the project has passed our audit. Our auditors have verified that the UTEXO Bridge, as of audited commits `dev@ea5de9b4` (bridge-smart-contracts) and `88fbc6a1` (usdt0-smart-contracts), operates as intended within the defined scope, based on the information and code provided at the time of evaluation. Of the 47 findings, 40 are verified fixed and 7 are acknowledged as accepted residual risks; none remain open. The robustness of the codebase has been significantly improved, meeting the necessary security and functionality requirements established for this audit.

Moreover, we advise increasing the test coverage of the codebase, particularly adversarial coverage of the federation and release paths. Comprehensive testing is essential to uncover edge cases and ensure that the smart contracts perform as expected under various conditions.

3 FINDINGS REPORT

OXORIO

3.1 CRITICAL

C-01	The enclave-signed money path has no on-chain constraint: a quorum can drain the entire pool in one transaction in `MultisigProxy.sol`, `Bridge.sol`
Severity	**CRITICAL**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `execute`	204
MultisigProxy.sol	contract `MultisigProxy` > function `executeBatch`	232
Bridge.sol	contract `Bridge` > function `fundsOut`	168

Description

In the functions `execute` / `executeBatch` of contract `MultisigProxy` and `fundsOut` of contract `Bridge`, the enclave releases funds — `execute` / `executeBatch` call `Bridge.fundsOut`. Three properties make this an instant, unlimited drain if the enclave quorum is faulty or compromised:

1. **No timelock.** `execute` / `executeBatch` only check the enclave signatures. The federation path (`proposeX → executeProposal`) is delayed by a `block.timestamp < p.proposedAt + timelockDuration` check that reverts `TimelockActive`, but the enclave path runs in the same transaction — no delay, no veto.
2. **Any recipient/amount, no freeze.** `Bridge.fundsOut` sends an arbitrary `recipient` and `amount` up to `balanceOf(this)`, and has no `whenNotPaused`, so the federation cannot freeze it during an incident.
3. **No finality binding.** `RGBVerifier.verify` only checks that a block exists in `BtcRelay` (an external relay based on `atomiqlabs/atomiq-contracts-evm`) and ignores `ctx`, so any real public block is a valid `proof` for any release (see the MAJOR proof-binding finding).

The deployment template sets the enclave threshold to 2, while signer arrays are operator-supplied; if deployed as 2-of-3, two keys authorise a release. With no delay, no pause, and

no proof binding, one signature set sends the **entire** pool to any address in a single transaction.

The spec puts three release checks on the contract itself:

> "the smart contract logic MUST check that the relevant header history is consistent with the chain accepted by BtcRelay; the smart contract logic MUST check that the EVM-side lock records (used for RGB mint) and locked funds exist for the same OpId; the smart contract logic MUST check that burnId has not already been consumed."

Only the third (`consumedBurnIds[burnId]`) is implemented. The "same OpId" lock-record check is absent, and the timelock the spec names as the mitigation for privileged power does not cover this path.

Proof of Concept
Amounts are token smallest-units; the result is decimal-independent (mock is 18-dp, production USDT0 is 6-dp).

```
uint256 pool = 100e18;
vm.prank(user);
bridge.fundsIn(pool, RGB_CHAIN_ID, 'rgb1quserdeposit', 1, ''); // step 1: users fund the pool

uint256 tBefore = block.timestamp;
bytes memory proof = abi.encode(BLOCK_HEIGHT, COMMITMENT_HASH); // any real public block is
a valid proof

// step 2: 2-of-3 enclave sigs -> proxy.execute -> Bridge.fundsOut, no proposal, no timelock.
// recipient `attacker` never deposited; fresh burnId; reused public block as proof
_executeFundsOut(attacker, pool, 0xDEADBEEF, RGB_CHAIN_ID, SOURCE_CHAIN_ID,
'rgb1qarbitrary', proof, abi.encode(_ids(1)));

// impact: the whole pool is gone to a non-depositor, in the SAME block — no delay, no pause
possible
assertEq(token.balanceOf(attacker), pool);
assertEq(token.balanceOf(address(bridge)), 0);
assertEq(block.timestamp, tBefore);
```

Recommendation

We recommend to add on-chain defence-in-depth so no single off-chain fault causes instant, unbounded, irreversible loss:

```
// 1. cap / delay large releases so the federation can observe and veto
if (amount > perEpochCap) revert ReleaseCapExceeded; // rate-limit the money path
// (or route large fundsOut through the same timelocked path as config changes)
```

```
// 2. give the federation a fast freeze on the release path
function fundsOut(...) external onlyOwner nonReentrant whenNotPaused {... }

// 3. bind the release to the settled burn (see MAJOR proof-binding finding):
//    verifier.verify must consume ctx, AND the proof semantics must be extended to
//    commit to the release context. Making verify(ctx, proof) read ctx is necessary
//    but not sufficient on its own, because today the proof carries no such binding.
verifier.verify(ctx, proof);
```

If the enclave is intentionally the sole authority on releases, state it explicitly as a CRITICAL centralization assumption: the full pool sits behind an M-of-N key set with no on-chain delay, cap, or veto.

Update

Client's response

Addressed with on-chain defence-in-depth across three related fixes:

1. **No timelock/cap on the enclave path** — fixed in PR #76: added isolated per-source-chain liquidity accounting (`lockedLiquidity`) and a two-tier outflow rate limiter (per-chain `chainBuckets` + aggregate `globalBucket`), so a single enclave-signed release can no longer draw more than the liquidity actually bridged for that chain, and outflow velocity is capped.

2. **No freeze during an incident** — fixed in PR #34: `Bridge.fundsOut` is now gated by a dedicated `whenOutflowNotPaused` modifier. `emergencyPauseAll()` now atomically freezes both the inflow and outflow paths together, closing the gap the audit flagged (an incident-response freeze now actually stops the enclave-signed release path). A separate, timelocked `pauseInflow` / `unpauseInflow` remains available for planned inflow-only freezes (e.g. a migration), independent of this emergency path.

3. **No finality binding** — partially addressed in PR #101: `burnId` is no longer a caller-supplied free number — it is now derived on-chain (`_deriveBurnId`) from the full release intent (recipient, amount, source/destination chain ids, proof, settlementData), so the replay guard is bound to the actual release parameters rather than an arbitrary value.

Partially fixed at: `3bd659f6`, `40e077c7`, `32e3f938`

Oxorio's response

Partially remediated.

Remaining issue: Configured liquidity and token-bucket limits can still equal TVL and permit a one-transaction full-pool drain.

Required remediation: Immutable TVL-relative limits or a timelocked/vetoed large-release lane.

Proof of Concept:

The local fixture configured both token buckets above TVL, deposited the full pool, built a canonical release, and supplied a valid 2-of-3 enclave quorum.

```
function test_R_C_01_fullPoolReleaseWhenLimitsCoverTVL() public {
    uint256 tvl = USER_DEPOSIT;

    vm.prank(user);
    bytes32 operationId =
        bridge.fundsIn(tvl, RGB_CHAIN_ID, "rgb:asset1qp0y3mq/utxo1abc",
abi.encode(RGB_OP_ID));

    assertGe(bridge.availableOutflow(RGB_CHAIN_ID), tvl, "chain bucket covers TVL");
    assertGe(bridge.availableGlobalOutflow(), tvl, "global bucket covers TVL");
    assertEq(token.balanceOf(address(bridge)), tvl, "pool funded");

    bytes32[] memory ids = new bytes32[](1);
    ids[0] = operationId;
    uint256[] memory amounts = new uint256[](1);
    amounts[0] = tvl;
    bytes memory settlementData = abi.encode(ids, amounts);
    bytes memory proof = abi.encode(BLOCK_HEIGHT, COMMITMENT_HASH, LATEST_HEIGHT,
LATEST_COMMIT);
    string memory sourceAddress = "rgb:sender/utxo1src";
    uint256 burnId =
        _deriveBurnId(recipient, tvl, RGB_CHAIN_ID, SOURCE_CHAIN_ID, sourceAddress, proof,
settlementData);
    IBridge.FundsOutParams memory params = IBridge.FundsOutParams(
        recipient,
        tvl,
        burnId,
        RGB_CHAIN_ID,
        SOURCE_CHAIN_ID,
        sourceAddress,
        proof,
        settlementData
    );

    uint256 nonce = proxy.teeNonce(RGB_CHAIN_ID);
    uint256 deadline = block.timestamp + 1 hours;
    bytes[] memory signatures = _signEnclave2of3(
```

```
            MultisigHelper.digestTeeFundsOut(domainSep, params, nonce, deadline)
        );
        proxy.fundsOutCall(params, nonce, deadline, 3, signatures);

        assertEq(token.balanceOf(address(bridge)), 0, "entire pool released");
        assertEq(token.balanceOf(recipient), tvl, "recipient received entire pool");
    }
```

Client's response

The outflow buckets now operate on TVL-relative shares instead of absolute token amounts. Per-chain and global limits are configured in basis points and automatically scale with the current reference liquidity.

Both bucket policies are bounded by compile-time constants:

```
MAX_CHAIN_OUTFLOW_BPS = 2_000;  // 20%
MAX_GLOBAL_OUTFLOW_BPS = 2_000; // 20%
```

The setters enforce that `burstBps + refillBpsPerWindow` does not exceed the corresponding ceiling. These constants cannot be changed by the federation or configured differently during deployment, so a bucket covering 100% of TVL is no longer representable.

An additional non-configurable rolling limiter caps both per-chain and aggregate physical outflow to 20% of reference liquidity over a rolling 24 hour window. The reference includes liquidity already released but still counted in the window, preventing the limit from being bypassed by splitting a drain across multiple transactions or bucket refills.

As a result, the PoC now reverts even with a valid enclave quorum, and neither a single transaction nor multiple transactions within the rolling window can drain the full pool.

Fixed at: `63aca5d8`

C-02	The federation can replace the enclave signer set with no on-chain attestation, collapsing the two trust domains into one in `MultisigProxy.sol`
Severity	**CRITICAL**
Status	• ACKNOWLEDGED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `proposeUpdateEnclaveSigners`	355
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType` (`UpdateEnclaveSigners`)	790

Description

In the function `proposeUpdateEnclaveSigners` of contract `MultisigProxy` (finalised through `_executeByType`), the design names two trust domains: the enclave/TEE that signs releases, and the federation that governs configuration. On-chain the second strictly dominates the first. `proposeUpdateEnclaveSigners` followed by `executeProposal` lets the federation M-of-N overwrite `_enclaveSigners` and `enclaveThreshold` with arbitrary addresses once `timelockDuration` elapses. The only validation in the `UpdateEnclaveSigners` branch is a non-empty set, `1 <= threshold <= n`, and no zero/duplicate addresses. There is **no on-chain attestation** that the new keys are genuine Nitro-enclave keys with the expected PCR measurements — any externally-owned account is accepted.

The federation can therefore install its own keys as "the enclave" and then sign `Bridge.fundsOut` to any recipient (see the CRITICAL enclave-drain finding above). The effective, sole root of trust is the federation quorum, which contradicts the design goal stated in the specification:

> "the design goal is to make it unnecessary to trust these operators, so that even in the event where they are compromised or hacked, assets cannot be stolen from the bridge."

This is a second, independent path to the same full-pool drain, reached through governance rather than direct enclave-key compromise. It is the weaker link whenever the federation set is easier to compromise than the enclave set, and it is exactly the class of incident seen at Ronin (5-of-9 validator keys captured by spear-phishing) and Harmony (a 2-of-5 multisig). The specification frames signer-set rotation as an accepted, timelock-observable trust assumption, but with no attestation and a removable timelock floor (see the

`SetTimelockDuration` finding) the observation window is the only barrier, and it does not prevent the rotation.

Proof of Concept
The drain chains on once the signer set has been swapped. The governance sequence runs against the audited sources:

```
// step 1: federation M-of-N proposes swapping the enclave set to attacker-controlled keys
bytes32 id = proxy.proposeUpdateEnclaveSigners(_arr(attacker), 1, nonce, deadline,
fedBitmap, fedSigs);

// step 2: wait out the timelock (0.. <30d; it can be set to 0 — see SetTimelockDuration
finding)
vm.warp(block.timestamp + proxy.timelockDuration);

// step 3: anyone finalises the swap; no attestation is required of the new "enclave" key
proxy.executeProposal(id, abi.encode(_arr(attacker), uint256(1)));

// step 4 / impact: the attacker now signs Bridge.fundsOut and drains the pool (enclave-
drain finding);
// emergencyPause does not stop fundsOut (separate WARNING)
```

Recommendation

We recommend requiring on-chain attestation evidence when rotating enclave signers: before accepting each new signer, verify a fresh Nitro attestation document that binds the key to the expected PCR measurements. Note that the attestation verifier shipped with `enclave-signer` is a Rust/off-chain component (`attestation-verify`), not a Solidity contract, so this path needs a dedicated on-chain attestation verifier contract that checks the COSE/certificate chain and PCRs and binds them to the proposed signer addresses. If full on-chain verification is impractical, at minimum publish the attestation evidence for each proposed key as process evidence and gate enclave-set rotation behind a longer, fixed-minimum timelock distinct from routine governance, so the federation cannot silently mint an enclave.

```
// in the UpdateEnclaveSigners handler, before assigning _enclaveSigners:
for (uint256 i; i < newSigners.length; i++) {
 requireFreshAttestation(newSigners[i], expectedPCRs); // bind each key to a genuine enclave
}
```

If this remains an accepted trust assumption, state plainly as a CRITICAL centralization assumption that the federation quorum can take over the enclave role, and with it the entire pool.

Update

Client's response

Accepted as a centralization/trust assumption: full on-chain attestation of enclave signer keys (verifying AWS Nitro PCR measurements against the enclave image) is not implemented, because it requires a dedicated on-chain attestation-verifier contract that does not exist in this codebase — the existing `attestation-verify` component is an off-chain Rust tool, not a Solidity contract. Building and auditing such a verifier is a substantial, separate effort out of scope for this remediation round.

That said, the contract does not leave enclave signer rotation completely unconstrained. Several on-chain guards narrow the blast radius and force any rotation to go through the same observable governance path as every other configuration change:

- **Timelocked, not instant.** `proposeUpdateEnclaveSigners` → `executeProposal` is gated by `timelockDuration`, with no bypass — there is no way for the federation to swap enclave signers in the same transaction as the proposal.
- **Non-zero timelock floor.** `SetTimelockDuration` is bounded by an immutable `MIN_TIMELOCK`, so the federation cannot shrink the observation window to zero to hide a rotation.
- **Disjoint signer sets, enforced on-chain.** A proposed enclave signer set is checked against the federation set *and* against every other source chain's enclave set (`_requireDisjointFromAllEnclaveSets`); an address cannot be installed as both a federation member and an enclave signer, and cannot be reused as "the enclave" across chains.
- **Strict-majority threshold floor.** Enclave (and federation) thresholds must satisfy `2 * threshold > n` — a 1-of-N or bare-majority-of-2 set can no longer be configured, on-chain, at propose time.
- **Per-source-chain segmentation.** Enclave signers are keyed by `sourceChainId`, so a federation-driven rotation for one source chain does not implicitly touch the enclave trust root for other chains — this limits (but does not eliminate) the blast radius of a single malicious rotation.
- **Propose-time validation.** Signer-set/threshold shape is validated when the proposal is created, not only at execution — a malformed proposal is rejected immediately rather than silently burning a timelock slot and failing later.

None of these substitute for genuine attestation — the federation quorum remains the effective root of trust for the enclave role, exactly as the finding states. They ensure that if this power is ever exercised, it happens through a timelocked, disjoint, threshold-bounded path with a guaranteed non-zero observation window, rather than instantly or silently.

Oxorio's response

The risk remains: full on-chain attestation of enclave signer keys is not implemented, so after the timelock a federation quorum can still install unattested EOAs as "the enclave" and,

through the release path, reach the same full-pool drain. The federation quorum therefore remains the effective root of trust for the enclave role, exactly as the finding states. Compensating controls now narrow the blast radius and force any rotation through the observable governance path: rotation is timelocked with no same-transaction bypass; the timelock cannot be shrunk below the immutable `MIN_TIMELOCK`; enclave/federation sets and every cross-chain enclave set are enforced disjoint; thresholds must be a strict majority of at least two; enclave signers are segmented per source chain; and signer-set shape is validated at propose time. Before production, the client should treat enclave-set rotation as a monitored, timelock-observable event and, ideally, add a dedicated on-chain Nitro attestation-verifier contract (or, at minimum, publish attestation evidence for each proposed key and gate rotation behind a longer fixed-minimum timelock).

3.2 MAJOR

M-01	The on-chain finality `proof` is not bound to the release context; `burnId` is the sole replay guard in `RGBVerifier.sol`, `Bridge.sol`
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Location

File	Location	Line
RGBVerifier.sol	contract `RGBVerifier` > function `verify`	58
Bridge.sol	contract `Bridge` > `consumedBurnIds[burnId]` replay guard inside `fundsOut`	186

Description

In the function `verify` of contract `RGBVerifier` and the function `fundsOut` of contract `Bridge`, on the RGB → EVM release path the only on-chain finality check is `RGBVerifier.verify(ctx, proof)`. It decodes `proof` into `(blockHeight, commitmentHash)` and asks `BtcRelay` (an external relay based on `atomiqlabs/atomiq-contracts-evm`) whether that block exists. It ignores the `ctx` argument completely, so the proof is **not** bound to `recipient`, `amount`, `burnId`, or `sourceAddress`, and the block reference is never consumed. A finalised Bitcoin block is public, so the same `proof` validates any number of `fundsOut` calls.

The only thing preventing a literal replay is the on-chain guard:

```
if (consumedBurnIds[burnId]) revert BurnIdAlreadyConsumed(burnId);
```

But `burnId` is a free number chosen by the off-chain caller (the operator/listener) and merely signed, not bound, by the enclave — it is not derived from the actual Bitcoin/RGB burn. So two `fundsOut` calls settling the **same** burn (same `recipient`, `amount`, `sourceAddress`, same `proof`) both succeed if the caller picks two different `burnId`s.

The settlement ledger does not bind the release to a burn either. `RgbSettlementModule` stores deposits as amount-only records — `fundsInRecords[operationId] = netAmount`, with **no depositor address** — and `beforeFundsOut` just consumes enough `fundsInIds` to cover the amount. Note that this `operationId` is the EVM-side, backend-assigned deposit key supplied at `fundsIn`, which is distinct from the RGB `OpId` the spec wants threaded across domains (below); the code never ties the two together. `Bridge` even hands the module `ctx.burnId` and `ctx.sourceAddress` (`RouteTypes` says they are "carried in the context so verifiers and settlement modules can reference it if they need to"), but the module uses neither. The binding data arrives on-chain and is discarded.

The spec requires this binding on the contract:

> "Bridge state transitions MUST be bound across domains using the RGB OpId. The same OpId MUST be used consistently across the RGB consignment, bridge metadata, EVM lock record, and EVM unlock authorization payload."
>
> "an EVM unlock payload not bound to the RGB OpId MUST NOT be accepted"

The unlock payload carries no RGB OpId at all:

```
Bridge.fundsOut(
    recipient, amount, burnId, sourceChainId,
    destChainId, sourceAddress, proof, settlementData
)
```

So this is not enforced on-chain. Double-spend prevention for a burn lives entirely in the off-chain enclave, and the money path (`proxy.execute`) has no timelock in which a duplicate could be caught.

Cross-user insolvency. A duplicate release does not just over-pay — it quietly moves the loss to another user. Because the consumed `fundsInIds` are fungible, the second release for one burn eats a **different, legitimate deposit record**. The identity `custody(TOKEN) == Σ fundsInRecords` still holds, so nothing reverts; but only one burn happened. When that record's real owner later redeems, `beforeFundsOut` reverts `FundsInNotFound` — their deposit is now unbacked and the shortfall has been pushed onto them.

Proof of Concept

```
bytes memory proof = abi.encode(BLOCK_HEIGHT, COMMITMENT_HASH); // one real public block

// release #1 — burnId A, draws record 1
_executeFundsOut(recipient, RELEASE, 0xA11, RGB_CHAIN_ID, SOURCE_CHAIN_ID, 'rgb1qburner',
proof, abi.encode(_ids(1)));
```

```
// release #2 — SAME proof / recipient / amount, only burnId differs -> also succeeds,
// and it silently consumes a DIFFERENT user's record (2) for the same single burn
_executeFundsOut(recipient, RELEASE, 0xB22, RGB_CHAIN_ID, SOURCE_CHAIN_ID, 'rgb1qburner',
proof, abi.encode(_ids(2)));
assertEq(token.balanceOf(recipient), 2 * RELEASE); // impact: paid twice for one Bitcoin burn

// control — reusing burnId A reverts, proving burnId is the ONLY on-chain guard
vm.expectRevert(abi.encodeWithSignature('BurnIdAlreadyConsumed(uint256)', uint256(0xA11)));
proxy.execute(callData /* burnId 0xA11 again */, nonce, deadline, 3, sigs);
```

Recommendation

We recommend to derive the replay key from the actual settled burn — the RGB OpId / burn-transition identity — instead of a free-form `burnId`, to carry that OpId in the unlock payload (`fundsOut` / `FundsOutContext`), and to make the verifier consume `ctx`:

```
// replay key tied to the RGB OpId of the settled burn, not a caller-supplied number
bytes32 burnKey = keccak256(abi.encode(rgbOpId));
if (consumedBurnKeys[burnKey]) revert BurnAlreadyConsumed(burnKey);

// verifier must consume ctx (today it is ignored) and check the proof against the release
context
verifier.verify(ctx, proof);
```

Bind the `RgbSettlementModule` records to the same OpId — the module already receives `ctx.burnId` / `ctx.sourceAddress` but discards them — so a single burn cannot be settled by more than one `fundsOut`. Note that additionally binding the EVM `recipient` and `sourceAddress` to the proof requires committing those EVM addresses on the RGB side first: the current consignment and `proof` carry neither (see the enclave recipient-binding and cross-chain-metadata findings). Until that schema-level commitment exists, the on-chain guard can enforce OpId-uniqueness and settlement-record binding but cannot prove the destination/source from the proof alone. If this stays an accepted reliance on the enclave, state plainly that the RGB-release path has no on-chain double-spend guard beyond `burnId` and no timelock, so enclave correctness and key custody are the only barrier.

Update

Client's response

Partially addressed. PR #101 replaces the free-form replay key with a deterministic burnId derived from the Bridge address, current chain ID, token, recipient, amount, source and destination chain IDs, and hashes of sourceAddress, proof, and settlementData. On a

successful fundsOut, Bridge validates the supplied value against that derivation and records it as consumed. PR #79 requires every referenced settlement record to exist and match its recorded amount.

Why the proof itself remains unbound to the release context. `RGBVerifier.verify` still does not consume `ctx` — it only proves that a given Bitcoin block/commitment exists in the chain accepted by `BtcRelay`, at sufficient depth. This is a structural limit, not an oversight we chose to leave: Bitcoin/RGB and the EVM side are independent systems with no native way for one to reference specifics of the other. A Bitcoin SPV-style proof can only attest to what is actually committed on the Bitcoin side — it cannot retroactively prove facts (recipient, amount, burnId) that were never encoded into that commitment in the first place. Binding the proof itself to the EVM release context would require the RGB consignment schema to commit to the destination EVM address and amount *at the point the burn is created on the Bitcoin/RGB side* — a cross-system protocol change to how consignments are constructed, not something an EVM contract change alone can retrofit. This is also what the audit's own recommendation notes: "additionally binding the EVM recipient and sourceAddress to the proof requires committing those EVM addresses on the RGB side first... Until that schema-level commitment exists, the on-chain guard can enforce OpId-uniqueness and settlement-record binding but cannot prove the destination/source from the proof alone."

Given that constraint, our fix focuses on what the EVM side can bind unilaterally: the replay key (burnId, now tied to the full release parameters) and the settlement ledger (existence + exact-amount check per operation id). Together these close the "same burn settled twice" and "burnId is a free number" gaps, but the underlying proof remains "any real Bitcoin block" — a full close of this finding is contingent on a future RGB consignment schema change that commits destination/amount on the Bitcoin side, which is outside the scope of this remediation round.

Oxorio's response

We recommend the client record this finding as ACKNOWLEDGED rather than FIXED. Deriving `burnId` on-chain from the full EVM release intent (PR #101) closes the free-form replay key, and requiring each settlement record to exist and match its recorded amount (PR #79) hardens the ledger — but neither establishes a unique, on-chain identity for the source burn. `RGBVerifier.verify` still ignores `FundsOutContext` and only attests that a Bitcoin block/commitment exists at sufficient depth, and the derived `burnId` hashes the EVM-side `proof` and `settlementData` rather than the actual RGB burn. As a result the same burn re-presented as a distinct intent (for example a different recipient over the same public proof) yields a different `burnId` and can settle again, silently consuming a second, unrelated deposit record. The PoC below reproduces this on the remediated code.

Fully binding the finality proof to a canonical RGB source-burn identity requires an RGB consignment-schema change that commits the destination EVM address and amount on the Bitcoin side, which is out of scope for this remediation round. Until then, double-spend

prevention for a burn relies on enclave correctness and the aggregate pool balance, which is the accepted residual risk — hence the ACKNOWLEDGED status.

Recommended follow-up (implementable now, independent of the RGB schema): in `RgbSettlementModule.beforeFundsOut`, reject an empty `operationIds` array, reject duplicate ids, require the referenced record amounts to sum exactly to `ctx.amount`, and consume (delete) each record it draws, so a single deposit record cannot back more than one release. A separate canonical `sourceBurnId` verified against RGB evidence and consumed exactly once would fully close the finding once the schema change lands.

Proof of Concept:

The test reused the same proof, source address, amount, and settlement record while changing only the recipient, producing a distinct intent-derived `burnId`; both releases succeeded.

```
function test_R_M_01_sameSourceEvidenceDifferentIntentGetsFreshBurnId() public {
    vm.prank(user);
    bytes32 reusedRecord = bridge.fundsIn(AMOUNT, RGB_CHAIN_ID, DST_ADDR, _rgbData(1));
    vm.prank(user);
    bytes32 otherUsersRecord = bridge.fundsIn(AMOUNT, RGB_CHAIN_ID, DST_ADDR, _rgbData(2));

    bytes memory proof = _proof();
    bytes memory settlementData = _settlement(_ids(reusedRecord));
    address recipientB = makeAddr("recipientB");
    uint256 burnIdA =
        _deriveBurnId(recipient, AMOUNT, RGB_CHAIN_ID, SOURCE_CHAIN_ID, SRC_ADDR, proof,
settlementData);
    uint256 burnIdB =
        _deriveBurnId(recipientB, AMOUNT, RGB_CHAIN_ID, SOURCE_CHAIN_ID, SRC_ADDR, proof,
settlementData);
    assertNotEq(burnIdA, burnIdB, "intent change derives a fresh replay key");

    vm.prank(multisig);
    _fundsOutWithBurnId(
        recipient, AMOUNT, burnIdA, RGB_CHAIN_ID, SOURCE_CHAIN_ID, SRC_ADDR, proof,
settlementData
    );
    vm.prank(multisig);
    _fundsOutWithBurnId(
        recipientB, AMOUNT, burnIdB, RGB_CHAIN_ID, SOURCE_CHAIN_ID, SRC_ADDR, proof,
settlementData
    );

    assertEq(usdt0.balanceOf(recipient), AMOUNT);
```

```
    assertEq(usdt0.balanceOf(recipientB), AMOUNT);
    assertEq(rgbModule.fundsInRecords(reusedRecord), AMOUNT, "same settlement identity
remains reusable");
    assertEq(rgbModule.fundsInRecords(otherUsersRecord), AMOUNT, "other record funded the
second release");
}
```

M-02	Inbound `lzCompose` does not bind `sourceChainId` or entrypoint trust to the LayerZero transport `srcEid` in `UtexoLZAdapter.sol`
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `lzCompose` (entry)	150
UtexoLZAdapter.sol	`trustedEntrypoints` address-only trust check (no `srcEid` bind)	167
UtexoLZAdapter.sol	payload `abi.decode` (routing/commission fields)	183

Description

In the function `lzCompose` of contract `UtexoLZAdapter` the inbound trust decision is made, and the gap described below arises there.

Inbound topology (where this leg sits). The EVM→RGB deposit leg runs `UtexoSourceEntrypoint.deposit` (source chain) → LayerZero OFT → `UtexoLZAdapter.lzCompose` (Arbitrum) → `Bridge.fundsIn`. On this leg the asset is *issued on the RGB/Bitcoin side*, so the Arbitrum route is registered with `NullSettlementModule` + `NullVerifier`: `onFundsIn` is a no-op and no source-side proof is verified. That is by design — there is no Arbitrum-side liquidity to account for on a deposit — but it means the only on-chain artefact of a deposit is the emitted `ComposeFundsIn` / `BridgeFundsIn` event, and the binding of that deposit to its true origin rests on the honesty of the trusted entrypoint plus the event (the reverse RGB→EVM release leg is the one that uses `RGBVerifier` + `RgbSettlementModule`).

The gap. Every compose message carries the LayerZero transport source id, readable as `OFTComposeMsgCodec.srcEid(_message)`. `lzCompose` never reads it. Trust is decided purely by the *raw caller address*:

```
bytes32 composeFrom_ = OFTComposeMsgCodec.composeFrom(_message);
if (!trustedEntrypoints[composeFrom_]) revert UntrustedComposeSource(composeFrom_);
```

and the business fields — `sourceChainId`, `destinationChainId`, `destinationAddress`, `operationId` — are then taken verbatim from the decoded payload (L183–L189) and used to select the route (settlement module + verifier) and the commission rule. The in-code comment overstates this:

```
// `sourceChainId` is the EVM chain id captured by `UtexoSourceEntrypoint`
// from `block.chainid` at deposit time — non-spoofable.
```

It is non-spoofable *only given an honest trusted entrypoint*. The adapter performs no cross-check that `srcEid` (what LayerZero says about the real origin) is consistent with the self-declared `sourceChainId`, and it does not bind a trusted entrypoint to a specific `srcEid`. So a buggy or compromised trusted entrypoint can declare any `sourceChainId`/route/`destinationAddress`/`operationId` for a real, minted deposit, and the chain accepts it. Because the inbound route uses `NullSettlementModule`, there is also no on-chain `operationId` de-duplication on this leg (see the predictable-`operationId` finding) — the metadata that drives the off-chain RGB mint is whatever the payload says.

A second facet is that trust is keyed by address rather than by `(srcEid → expected entrypoint)`. On EVM chains *without* EIP-6780 this exposes the classic metamorphic risk: a self-destructible entrypoint that is allow-listed could be destroyed and re-deployed with different code at the same `CREATE2` address, inheriting its trusted status. On post-Cancun chains (Arbitrum and the relevant source chains) EIP-6780 neutralises that specific resurrection vector, but the weaker point stands: trust survives entrypoint replacement and is never pinned to the transport source.

The `amountLD` itself cannot be forged — it is the USDT0 actually minted to the adapter by `lzReceive` — so under the trusted-entrypoint assumption this is a metadata-integrity / defence-in-depth gap rather than a direct theft path. It is raised to MAJOR under the zero-trust lens because the design leans entirely on entrypoint honesty for a value (`srcEid`) it already has in hand.

Proof of Concept

This PoC configures the declared route with `RgbSettlementModule` to make the route selection visible in storage. In the intended `NullSettlementModule` inbound route, the same mismatch appears in emitted metadata rather than in `fundsInRecords`.

```
// step 1: route enabled for a *declared* source id 777 -> RGB; payload declares
sourceChainId = 777
bytes memory innerPayload = abi.encode(
 uint256(777), // sourceChainId (self-declared)
 RGB_CHAIN_ID, // destinationChainId
 string('rgb:dest/utxo'), // destinationAddress
```

```
 uint256(555), // operationId
 bytes('') // settlementData
);
// step 2: deliver a message whose transport srcEid (30110) is unrelated to the declared 777
bytes memory msg_ = OFTComposeMsgCodec.encode(
 uint64(1), uint32(30110) /* transport srcEid != 777 */, amountLD,
 abi.encodePacked(trustedEntry, innerPayload)
);

vm.prank(lzEndpoint);
adapter.lzCompose(lzOft, guid, msg_, address(0), ''); // step 3: succeeds despite the
mismatch

// impact: route/metadata selected under the self-declared 777; transport srcEid is never
consulted
assertEq(rgbModule.fundsInRecords(555), amountLD, 'recorded under declared sourceChainId
777');
```

Recommendation

We recommend binding the inbound trust decision to the transport source and validating the declared `sourceChainId` against it:

```
uint32 srcEid = OFTComposeMsgCodec.srcEid(_message);
// trust keyed per (srcEid, entrypoint) rather than by raw address alone
if (!trustedEntrypoints[srcEid][composeFrom_]) revert UntrustedComposeSource(composeFrom_);
// the self-declared sourceChainId must match the transport origin
if (eidToChainId[srcEid] != sourceChainId) revert SourceChainIdMismatch(srcEid,
sourceChainId);
```

This costs one mapping lookup and turns `sourceChainId` from a self-asserted field into one corroborated by the LayerZero transport. Note that `srcEid` is a LayerZero *endpoint id*, not an EVM `chainid`, so `eidToChainId` must be a governed mapping maintained by the federation and kept in sync with LayerZero's endpoint assignments; likewise `trustedEntrypoints` should be keyed per `(srcEid, entrypoint)` rather than by raw address alone.

Update

Client's response

UtexoLZAdapter obtains srcEid from the LayerZero transport envelope, requires composeFrom to match the registered entrypoint for that srcEid, and rejects a payload whose declared sourceChainId differs from eidToChainId[srcEid].

Oxorio's response

The adapter now reads the LayerZero transport `srcEid` from the compose envelope, requires `composeFrom` to match the entrypoint registered for that `srcEid`, and reverts when the self-declared `sourceChainId` does not equal `eidToChainId[srcEid]`. Inbound trust and the downstream route/commission selection are therefore corroborated by the transport origin rather than taken solely on the entrypoint's word, closing the metadata-integrity gap under the trusted-entrypoint model.

Fixed at: `5ac5fe6f`

M-03	The TEE allowlist places no class restriction on selectors: the federation can grant the enclave un-timelocked control of privileged setters in `MultisigProxy.sol`
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `executeBatch`	232
MultisigProxy.sol	contract `MultisigProxy` > function `proposeSetTeeAllowedCall`	435
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType` (`SetTeeAllowedCall`)	822

Description

In the functions `executeBatch` and `proposeSetTeeAllowedCall` of contract `MultisigProxy` (and the `SetTeeAllowedCall` handler in `_executeByType`), the design splits authority into two paths: the enclave path (`execute` / `executeBatch`) runs in one transaction with no delay, while every governed configuration change runs through `proposeX → executeProposal` and is gated by a `block.timestamp < p.proposedAt + timelockDuration` check that reverts `TimelockActive`. The `teeAllowedCalls` mapping is the gate that is supposed to keep the no-timelock enclave path narrow.

That gate has no on-chain notion of *which* selectors are safe to put on it. `executeBatch` dispatches **any** allowlisted `(target, selector)` to an arbitrary target:

```
if (!teeAllowedCalls[targets[i]][sel]) revert CallNotAllowed(...);
targets[i].call{ value: values[i] }(callDatas[i]);
```

The only validation in the `SetTeeAllowedCall` handler is `if (target == address(0)) revert ZeroTarget;`. The selector is never constrained, and the target is not restricted to `Bridge`. The constructor seeds only `Bridge.fundsOut`, and the in-code comment frames later additions as operational (e.g. the LayerZero adapter's `sendOut`), but nothing on-chain enforces that intent.

Consequently, a single federation `proposeSetTeeAllowedCall` (one timelocked action) can allow-list a privileged/configuration selector — for example `CommissionManager.setCommissionRule` / `setGlobalDefaults`, `Bridge.setRouteRegistry` / `setLZAdapter` / `pause` / `unpause`, `RouteRegistry.setRoute`, or even `Ownable.transferOwnership` — after which the **enclave** can invoke that setter through `executeBatch` instantly, with no delay and no federation veto. This collapses the timelock the design relies on for configuration changes: any setter routed onto the TEE allowlist stops being observable before it takes effect.

The escalation is one-directional but real. The enclave cannot widen its own allowlist (`proposeSetTeeAllowedCall` is federation-gated), so the precondition is a federation governance action. But there is no on-chain guard that distinguishes the operational money/transport selectors the path is meant for (`fundsOut`, `sendOut`) from privileged setters, so an honest-but-careless federation can silently hand the enclave un-timelocked config power, and a compromised one gains an instant-config path that bypasses the very delay meant to make its actions observable. This contradicts the design intent of a timelocked governance path for configuration changes; the contract treats the no-delay enclave path and the privileged-setter surface as freely composable.

Rated MAJOR: it does not by itself raise the worst-case loss above the enclave-drain CRITICAL, and it requires a federation enabling step, but it defeats the timelock guarantee for whatever is allow-listed and is reachable without any code change to the contracts.

Recommendation

We recommend to constrain what can ever reach the no-timelock enclave path. Restrict the allowlist target to the operational contracts (`Bridge`, and the LZ adapter for `sendOut`) and pin the permitted selectors to an explicit operational set, rejecting privileged/configuration selectors so configuration changes can only flow through the timelocked `AdminExecute` path:

```
// in the SetTeeAllowedCall handler, before writing teeAllowedCalls[target][sel]:
// restrict targets to the bridge and the bridge-configured LZ adapter
if (target != bridge && target != IBridge(bridge).lzAdapter()) revert
TargetNotAllowlistable(target);
// only operational selectors may ever bypass the timelock; setters must use the governed
path
if (!_isOperationalSelector(target, sel)) revert SelectorNotAllowlistable(target, sel);
```

Alternatively, keep an explicit on-chain whitelist of `(target, selector)` pairs that the federation may toggle, seeded with the operational money/transport surface (`Bridge.fundsOut`, and `UtexoLZAdapter.sendOut` for the LayerZero leg), and make every other selector unreachable from `execute` / `executeBatch`. Note that `sendOut` is the on-chain selector the LZ flow expects to be allowlisted; the enclave API at the audited commit

signs single-call `fundsOut`-shaped operations, so treat `sendOut`/`executeBatch` as an allowlistable on-chain surface rather than a call the enclave constructs today.

Update

Client's response

The generic enclave execute/executeBatch dispatchers, mutable teeAllowedCalls mapping, and federation operation that could add arbitrary selectors were removed entirely. The enclave release surface is now two purpose-built functions — fundsOutCall and lzFundsOutCall — with hardcoded call targets (Bridge.fundsOut, and the LZ adapter's sendOut for the second), not a generic (target, selector) dispatch. Enclave signatures cannot select an arbitrary privileged setter because there is no longer a mechanism for the enclave to call anything other than these two functions. Planned administration and configuration remain on federation-signed proposal paths subject to the proposal timelock. The separate federation-signed emergencyPause and emergencyUnpause paths remain immediate and are not covered by that timelock (by design — this is the incident-response freeze, not a configuration change).

Oxorio's response

The generic enclave `execute`/`executeBatch` dispatchers and the mutable `teeAllowedCalls` allowlist (and the federation operation that could extend it with arbitrary selectors) were removed. The enclave release surface is now two purpose-built entrypoints with hardcoded targets — `fundsOutCall` (`Bridge.fundsOut`) and `lzFundsOutCall` (the adapter's `sendOut`) — so an enclave signature can no longer reach an arbitrary privileged setter, and every configuration change stays on the timelocked federation proposal path. The finding is fully closed.

Fixed at: `316168fd`

3.3 WARNING

W-01	Fee-on-transfer token makes the `Bridge` ledger overstate custody and bricks release of the recorded amount in `Bridge.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `_fundsIn`	284
Bridge.sol	contract `Bridge` > function `fundsOut`	181

Description

In the function `_fundsIn` of contract `Bridge` the deposit is pulled with `safeTransferFrom(from, address(this), amount)` and the settlement module records the net amount derived from the nominal `amount`. If the bridge token charges a fee on transfer, the contract receives strictly less than `amount`, while the ledger records the full nominal value. The custody therefore drifts below the recorded liability. A later `fundsOut` for the recorded amount reverts at the bridge-pool guard (`amount > IERC20(TOKEN).balanceOf(address(this))` reverts `AmountExceedBridgePool`), so the recorded amount becomes unreleasable.

The deployed Tether `USDT` contract on Ethereum (`0xdAC17F958D2ee523a2206206994597C13D831ec7`) defines `basisPointsRate` and `maximumFee`, which allow the issuer to enable a fee on transfer. They default to zero on the current deployment, but the issuer retains the on-chain ability to set them at any time, and the USDT0 wrapper inherits the same legacy interface. The risk is therefore not purely theoretical.

Proof of Concept
The proof of concept redeploys the full bridge stack against a fee-on-transfer token configured with a 1 % fee, on a route with zero commission so the shortfall equals the transfer fee.

```
// step 1: user deposits 40 ether; with zero commission the ledger records the full nominal
40 ether
vm.prank(user);
bridge.fundsIn(40 ether, RGB_CHAIN_ID, 'rgb:in', 1, '');

// step 2: the bridge actually holds 39.6 ether — the 1% fee was skimmed inside
safeTransferFrom
uint256 held = token.balanceOf(address(bridge));
assertLt(held, 40 ether, 'bridge custodies less than the recorded amount');

// step 3 / impact: a signed fundsOut for the recorded 40 ether reverts; the record is
unreleasable
vm.expectRevert; // AmountExceedBridgePool
proxy.execute(fundsOutCalldataFor(40 ether), nonce, deadline, 3, sigs);
```

Recommendation

We recommend to measure the actually received amount with balance-delta accounting on the ingress side and to record that value, so the ledger never exceeds custody:

```
uint256 balanceBefore = IERC20(TOKEN).balanceOf(address(this));
IERC20(TOKEN).safeTransferFrom(from, address(this), amount);
uint256 received = IERC20(TOKEN).balanceOf(address(this)) - balanceBefore;
// account `received` instead of `amount`
```

The same delta-based treatment should be applied to the commission forwarding and to `CommissionManager.withdrawAllTokenCommission`, so the recorded pool counter is updated from the observed balance difference rather than from the requested amount.

Update

Client's response

`Bridge._fundsIn` measures the token balance before and after **safeTransferFrom** and derives netAmount, the settlement record, and lockedLiquidity from the amount actually received rather than the nominal request. Commission forwarding separately measures the balance increase at CommissionManager, and the manager credits that measured value while checking that its balance covers the recorded pool.

Oxorio's response

`Bridge._fundsIn` now measures the token balance before and after `safeTransferFrom` and derives the net amount, the settlement record, and `lockedLiquidity` from the amount actually received rather than the nominal request; commission forwarding

independently credits the measured balance increase at the CommissionManager. The ledger can therefore no longer overstate custody for a fee-on-transfer token, so the recorded amount always remains releasable and the finding is closed.

Fixed at: `dd1379a0`

W-02	Rotating the route registry makes existing `fundsInRecords` unreachable in `Bridge.sol`, `RouteRegistry.sol`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `setRouteRegistry`	113
RouteRegistry.sol	contract `RouteRegistry` > function `setRoute`	95

Description

The storage of inflow records `RgbSettlementModule.fundsInRecords` is per-module. After `setRouteRegistry` re-points the bridge to a new registry that exposes a different `RgbSettlementModule`, the prior records become unreachable. Any subsequent `fundsOut(operationId)` that references a pre-rotation identifier reverts with `FundsInNotFound(operationId)`, which means that pre-rotation deposits cannot be released.

The same orphaning is reachable through a **second, lower-effort path**: `RouteRegistry.setRoute` overwrites the `RouteConfig` for a single route key in place, so swapping just the `settlementModule` of a live route — without redeploying the registry or touching `Bridge.routeRegistry` — strands every record held in the previous module exactly the same way. This is the more natural operation (re-point one route's plugin vs. rotate the entire registry) and therefore the more likely to be performed inadvertently. Note that `setRoute` can likewise swap the `finalityVerifier` (for example `RGBVerifier` → `NullVerifier`), which retroactively changes the finality trust of a route — a federation power covered by the trust model.

More fundamentally, **a route's release plugins are not pinned in code at all**. `Bridge.fundsOut` resolves the verifier and settlement module purely from the route key `(sourceChainId, destChainId)` via `RouteRegistry.beforeFundsOut`, and the registry only requires the slots to be non-zero — it does not require the release route to use `RGBVerifier` / `RgbSettlementModule`. The deploy script (`DeployAll.s.sol`) wires no routes at all and defers every `(src, dst)` registration to post-deploy federation `proposeSetRoute`. Consequently there is no on-chain guarantee that a release route is

RGB-gated: registering (or later swapping) a release route to `NullVerifier` + `NullSettlementModule` makes `fundsOut` perform **no on-chain finality check and no settlement accounting** (`verify` and `beforeFundsOut` become no-ops), so the release relies entirely on enclave correctness. The same applies to the outbound LayerZero leg, whose paired `fundsOut` is gated only by whatever plugins its route happens to hold.

Proof of Concept

```
// whole-registry rotation: a record was captured in the original module, then a fresh
// RouteRegistry + RgbSettlementModule is deployed and the bridge re-pointed to it
vm.prank(address(proxy));
bridge.setRouteRegistry(address(newRegistry));

// impact: the pre-rotation record is now unreachable (the old module still holds it,
unmigrated)
vm.expectRevert; // FundsInNotFound(operationId) from the new module
proxy.execute(callData, nonce, deadline, 3, sigs);
assertEq(rgbModule.fundsInRecords(operationId), 50 ether, 'old module still holds it');
```

The same orphaning happens via a single in-place `setRoute` module swap (same registry, one call):

```
// swap just the settlement module of a live route — no registry redeploy
RgbSettlementModule newModule = new RgbSettlementModule(address(routeRegistry));
vm.prank(deployer); // registry owner = MultisigProxy in production
routeRegistry.setRoute(RGB_CHAIN_ID, SOURCE_CHAIN_ID, true, address(rgbVerifier),
address(newModule));

// impact: same orphaning — the freshly-wired module has no prior record
vm.expectRevert; // FundsInNotFound(operationId) from the freshly-wired module
proxy.execute(callData, nonce, deadline, 3, sigs);
assertEq(rgbModule.fundsInRecords(operationId), 50 ether, 'old module still holds it');
```

Recommendation

We recommend to require, as an operational invariant, that the federation process all outstanding inflow records by issuing the corresponding `fundsOut` operations before queuing either a `proposeUpdateRouteRegistry` **or** a `setRoute` that changes a live route's `settlementModule`. As a contract-level alternative, add a `migrateFundsInRecords(uint256[] ids)` helper to the settlement module (it does not exist today) so records can be moved into the new module instead of being stranded. Apply the same caution to the `finalityVerifier` slot: swapping a live route's verifier (for

example `RGBVerifier` → `NullVerifier`) silently removes the finality check, so release routes should pin or assert their expected `(verifier, settlementModule)` pair rather than accepting any non-zero plugins.

Update

Client's response

Accepted as an operational trust assumption, not an on-chain guarantee.

Route/settlement-module rotation (or trusted entrypoint / commission feed / lzAdapter rotation) is a deliberate, infrequent federation governance action — it goes through the timelocked propose → execute path, not something that can happen accidentally or instantly. We rely on the federation to treat any such rotation as an operational migration: verify that no live `fundsInRecords` would be stranded under the settlement module being replaced (or migrate/settle them first) before proposing the change.

We considered adding an on-chain migration mechanism (e.g., in MultisigProxy or RouteRegistry) that would move or re-validate `fundsInRecords` on rotation, and decided against it for this remediation round: the bridge currently supports a single settlement path (RGB via `RgbSettlementModule`), so there is exactly one module whose records could ever need migrating. Building a generic on-chain migration mechanism now — before there is a second settlement module to migrate *between* — would be speculative complexity sized for a problem we don't have yet. Worse, coupling `MultisigProxy` to a specific migration procedure for today's RGB module would likely require redeploying or upgrading `MultisigProxy` itself whenever a future route/settlement module is added, which is disproportionate: adding a new chain/route should only require deploying its own verifier and settlement module, not touching the governance contract.

Oxorio's response

The risk remains: route/settlement-module rotation (via `setRouteRegistry` or an in-place `setRoute` swap) can still orphan `fundsInRecords`, and a release route's plugins are not pinned in code. The client accepts this as an operational trust assumption rather than an on-chain guarantee, noting that any such rotation is an infrequent action gated by the timelocked propose→execute path, and that the federation is relied on to settle or migrate outstanding records before rotating. A generic on-chain migration mechanism was intentionally not added because the bridge currently supports a single settlement path. Before production, the client should keep the operational invariant that no live records are stranded under a module being replaced, and should pin or assert the expected `(verifier, settlementModule)` pair for release routes so a verifier downgrade cannot silently remove the finality check.

W-03	`setGlobalDefaults` and `setCommissionRule` accept configurations that cause `netAmount` underflow in `CommissionManager.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `setGlobalDefaults`	287
CommissionManager.sol	contract `CommissionManager` > function `setCommissionRule`	331

Description

In the functions `setGlobalDefaults` and `setCommissionRule` of contract `CommissionManager` only the individual ranges of `multiplier` and `stablePercent` are validated. The relationship between the two values is not constrained. The fee formula computes `stableFee` as `amount * stablePercent / multiplier / multiplier`. A configuration such as `(stablePercent = 9000, multiplier = 1)` produces a fee of `9000 * amount`, and the subsequent computation `netAmount = amount - tokenCommission` underflows and reverts with Panic(0x11). Every subsequent `fundsIn` on a token-currency route is blocked until the federation applies a corrective configuration through the timelock.

Proof of Concept

```
// step 1: federation stores stablePercent = 9000, multiplier = 1 through the timelock
_proposeAndExecuteCmAdminCall(abi.encodeWithSelector(
 CommissionManager.setGlobalDefaults.selector,
 uint256(9000), uint8(1), CommissionSide.FUNDS_IN, CommissionCurrency.TOKEN));

// step 2 / impact: every fundsIn on the token route now reverts on the netAmount underflow,
blocking the route
vm.prank(user);
vm.expectRevert(stdError.arithmeticError);
bridge.fundsIn(100 ether, RGB_CHAIN_ID, 'rgb:asset/utxo1', 1, '');
```

Recommendation

We recommend enforcing the joint invariant at the setter level, so an accepted configuration can never make the calculated fee larger than the deposited amount:

```
// reject any config where stableFee can exceed the deposit (stablePercent must be <=
multiplier^2)
if (newConfig.stablePercent > newConfig.multiplier * newConfig.multiplier) {
 revert InvalidFeeShape(newConfig.stablePercent, newConfig.multiplier);
}
```

Update

Client's response

Both global and route-specific setters now require stablePercent to be no greater than multiplier squared, using widened uint256 arithmetic before multiplication. A stored configuration can therefore no longer calculate a fee above the bridged amount and trigger the reported netAmount underflow. Equality remains allowed and can produce a zero net amount, which is a separate configuration-policy concern rather than this underflow.

Oxorio's response

Both `setGlobalDefaults` and `setCommissionRule` now require `stablePercent <= multiplier * multiplier`, computed with widened `uint256` arithmetic before multiplication. A stored configuration can no longer produce a fee larger than the deposited amount, so the reported `netAmount` underflow is unreachable. Equality is still permitted and can yield a zero net amount, which is a separate configuration-policy observation rather than this underflow; the finding is closed.

Fixed at: `11d66e1c`

W-04	`CommissionManager` accepts a `NATIVE` + `FUNDS_OUT` config that bricks the release path in `CommissionManager.sol`, `Bridge.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `setCommissionRule`	331
CommissionManager.sol	contract `CommissionManager` > function `setGlobalDefaults`	287
Bridge.sol	contract `Bridge` > function `fundsOut`	201

Description

The setters `setCommissionRule` and `setGlobalDefaults` of contract `CommissionManager` validate only `multiplier != 0` and `stablePercent <= 9000`; they never check the `side`/`currency` combination. A configuration with `side = FUNDS_OUT`, `currency = NATIVE`, and `stablePercent > 0` is therefore stored without complaint. On the release path, `Bridge.fundsOut` quotes `calculateFundsOutCommission`, which returns a non-zero `nativeCommission`, and then hard-rejects it — reverting `NativeCommissionNotAllowedOnFundsOut` whenever `nativeCommission != 0` — because there is no user to fund a native payment on a federation-driven release. As a result, every redemption on that route (or every redemption, if the bad combination is set as the global default) reverts at the commission gate, before the verifier and settlement plugins run, and stays frozen until the federation reconfigures through the timelock. The contract already encodes that `NATIVE` is invalid on `FUNDS_OUT`; it merely enforces it at execution time on the critical money path instead of at configuration time.

Proof of Concept

```
// step 1: setter accepts a (FUNDS_OUT, NATIVE, stablePercent = 100) rule on the release
route
_proposeAndExecuteCmAdminCall(abi.encodeWithSelector(
 ICommissionManager.setCommissionRule.selector,
 RGB_CHAIN_ID, SOURCE_CHAIN_ID, address(token),
 CommissionConfig({ stablePercent: 100, multiplier: 100,
 side: CommissionSide.FUNDS_OUT, currency: CommissionCurrency.NATIVE, isSet: true })));
```

```
// step 2 / impact: calculateFundsOutCommission returns a non-zero nativeCommission, so
every release
// on the route reverts at the commission gate (before verifier/settlement run) and stays
frozen
vm.expectRevert(IBridge.NativeCommissionNotAllowedOnFundsOut.selector);
proxy.execute(callData, nonce, deadline, 3, sigs); // fundsOut(...) on the route
```

Recommendation

We recommend to reject the invalid combination at the setter level, in both `setCommissionRule` and `setGlobalDefaults`:

```
if (currency == CommissionCurrency.NATIVE && side == CommissionSide.FUNDS_OUT) {
 revert NativeCommissionNotAllowedOnFundsOut;
}
```

This makes the bad shape unrepresentable and keeps the failure off the critical release path, mirroring the joint-range check recommended for the `netAmount` underflow finding.

Update

Client's response

setGlobalDefaults and setCommissionRule now reject the unrepresentable NATIVE plus FUNDS_OUT combination with NativeCommissionNotAllowedOnFundsOut. The invalid shape is stopped at configuration time instead of surfacing as a release-path revert.

Oxorio's response

`setGlobalDefaults` and `setCommissionRule` now reject the `NATIVE` + `FUNDS_OUT` combination with `NativeCommissionNotAllowedOnFundsOut` at configuration time, so the unrepresentable shape can no longer be stored and can no longer surface as a revert on the critical release path. The finding is closed.

Fixed at: `9d785da9`

W-05	`Bridge.fundsOut` is not gated by `whenNotPaused` in `Bridge.sol`, `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `fundsOut`	168
MultisigProxy.sol	contract `MultisigProxy` > function `emergencyPause`	285

Description

In the function `fundsOut` of contract `Bridge` only the modifiers `external override onlyOwner nonReentrant` are applied:

```
function fundsOut(...) external override onlyOwner nonReentrant {
 // releases TOKEN to recipient
}
```

The inherited `whenNotPaused` modifier is missing. When the federation calls `emergencyPause`, the `Pausable` flag is set, but only the inflow path (`fundsIn`) honours it. The outflow path continues to release funds. During an incident, the federation has no rapid on-chain mechanism to halt outflows.

Proof of Concept

```
// step 1: federation pauses the bridge
proxy.emergencyPause(nonce, deadline, 3, sigs);
assertTrue(bridge.paused, 'bridge paused');

// step 2: the inflow path honours the pause
vm.prank(user);
vm.expectRevert; // fundsIn is gated
bridge.fundsIn(10 ether, RGB_CHAIN_ID, 'rgb:in', 8, '');

// step 3 / impact: a signed fundsOut still succeeds while paused — the outflow path is not
gated
```

```
_executeFundsOut(recipient, 40 ether, 99, RGB_CHAIN_ID, SOURCE_CHAIN_ID, 'rgb:src', proof,
settlementData);
assertEq(token.balanceOf(recipient), 40 ether, 'outflow not blocked by pause');
```

Recommendation

We recommend to add the `whenNotPaused` modifier to `Bridge.fundsOut`:

```
function fundsOut(...) external override onlyOwner nonReentrant whenNotPaused {
 //...
}
```

Alternatively, a dedicated freeze flag can be introduced in `MultisigProxy` to gate the enclave money path (`execute` / `executeBatch`) — but not `executeProposal`, so the federation governance path stays operational and can still respond during an incident — so that signed releases cannot land during an active emergency.

Update

Client's response

Bridge.fundsOut is gated by whenOutflowNotPaused, and MultisigProxy.emergencyPause calls Bridge.emergencyPauseAll, which sets both the inflow and outflow pause flags. When Bridge ownership is assigned to that proxy, the federation-signed emergency path halts enclave releases. A separate proposal path calls pauseInflow without setting the outflow flag.

Oxorio's response

`Bridge.fundsOut` is now gated by `whenOutflowNotPaused`, and `MultisigProxy.emergencyPause` calls `Bridge.emergencyPauseAll`, which atomically sets both the inflow and outflow pause flags. With Bridge ownership assigned to the proxy, the federation's emergency freeze now halts enclave-signed releases; a separate, timelocked `pauseInflow` remains available for planned inflow-only freezes. The federation now has the rapid on-chain mechanism to halt outflows that the finding required, so it is closed.

Fixed at: `40e077c7`

W-06	`RGBVerifier` ignores the `confirmations` value returned by `BtcRelay` in `RGBVerifier.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
RGBVerifier.sol	contract `RGBVerifier` > function `verify` (discards relay return)	67

Description

In the contract `RGBVerifier` the function `verify` invokes `BtcRelay.verifyBlockheaderHash` (`BtcRelay` is an external relay based on `atomiqlabs/atomiq-contracts-evm`), whose `IBtcRelayView` signature returns a single `uint256 confirmations` value (`tip - height + 1`) and reverts if the block is unknown. The call therefore validates that the block exists at the given height, but the verifier discards the returned `confirmations` value rather than comparing it against a minimum threshold. The source code documents this decision: the TEE callers gate on confirmation depth off-chain before they sign, so the authors treat an on-chain check as a redundant policy duplicated in two places. There is therefore no on-chain minimum-confirmation floor, and the safety of shallow-block acceptance rests entirely on the off-chain signer. This is an accepted design tradeoff rather than a contract defect, recorded here so that the off-chain depth check is recognised as a security-critical invariant. The Bitcoin convention for high-value settlements is to require at least six confirmations.

Proof of Concept

```
// step 1: the referenced block has only one confirmation
btcRelay.setBlock(BLOCK_HEIGHT, COMMITMENT_HASH, 1); // single confirmation
//...
// step 2 / impact: a signed fundsOut against that shallow block still releases — no on-
chain confirmation floor
_executeFundsOut(recipient, 40 ether, 71_000, RGB_CHAIN_ID, SOURCE_CHAIN_ID, 'rgb:sender/
utxo1tip', proof, settlementData);
assertEq(token.balanceOf(recipient), 40 ether, 'release at 1 confirmation');
```

Recommendation

We recommend, as an optional defence-in-depth measure, to also enforce a minimum confirmation depth at the verifier level so that the guarantee does not rest on a single off-chain component:

```
uint256 internal constant MIN_BTC_CONFIRMATIONS = 6; // Bitcoin convention for high-value settlements

uint256 confirmations = btcRelay.verifyBlockheaderHash(blockHeight, commitmentHash);
if (confirmations < MIN_BTC_CONFIRMATIONS) { // enforce a floor instead of discarding the value
 revert InsufficientConfirmations(confirmations, MIN_BTC_CONFIRMATIONS);
}
```

Update

Client's response

RGBVerifier now consumes confirmation depths for both a source block and a relay-head block and applies its configured minimum source depth, maximum latest-block depth, and minimum confirmation gap. Deployment scripts default these immutable values to 6 / 1 / 5.

Oxorio's response

`RGBVerifier.verify` now consumes the confirmation depths returned for both a source block and a relay-head block and enforces its configured minimum source depth, maximum latest-block depth, and minimum confirmation gap on-chain (deployment defaults 6/1/5), instead of discarding the relay's returned value. Shallow-block acceptance no longer rests solely on the off-chain signer, closing the finding. As a deployment note, real depth safety rests on `minSourceConfirmations` (the relay-tip term cancels in the gap check), so a low `minSourceConfirmations` should be avoided, and modest `maxLatestConfirmations` headroom is advisable to avoid unnecessary `RelayNotFresh` re-signing under delayed inclusion.

Fixed at: `b18fcf04`

W-07	`destinationChainId` is not constrained to `block.chainid` on `fundsOut` in `Bridge.sol`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `fundsOut` (`sourceChainId` / `destinationChainId` only checked `!= 0`)	179-180

Description

In the function `fundsOut` of contract `Bridge` the validation is limited to `destinationChainId != 0` (L180), `sourceChainId != 0` (L179) and `recipient != address(0)` (L178). A federation-signed `fundsOut(..., destinationChainId = 99999,...)` is therefore accepted on-chain even when the value does not match `block.chainid`. The correctness of the routing check is delegated entirely to the off-chain backend.

Proof of Concept

```
// route registered for destChain 99999 while block.chainid is 31337; record seeded on the canonical route.
// impact: a signed fundsOut with the mismatched destination chain id still releases the tokens
_executeFundsOut(recipient, 40 ether, 777, RGB_CHAIN_ID, /*destChain*/ 99999, 'rgb:sender/utxo1mismatch', proof, settlementData);
assertEq(token.balanceOf(recipient), 40 ether, 'release on mismatched destChain');
```

Recommendation

We recommend enforcing the binding on-chain as an additional safeguard, but in a route-aware way rather than a blanket `== block.chainid`. For a local `Bridge.fundsOut` release (funds delivered on this chain), bind `destinationChainId` to the current bridge chain / configured route:

```
if (destinationChainId != block.chainid) revert DestinationChainMismatch; // local release:
bind to this chain
```

For releases that are forwarded onward (for example the LayerZero leg), `destinationChainId` is the final business chain and may legitimately differ from the bridge chain, so it should be validated against the route configuration or the paired `dstEid` mapping instead of `block.chainid`.

Update

Client's response

A blanket `destinationChainId == block.chainid` check would be incorrect: the LayerZero fundsOut flow legitimately releases with a destinationChainId different from the local chain, since the release is forwarded onward via the adapter rather than delivered locally. destinationChainId correctness for local releases currently relies on the off-chain backend / federation signing process.

Oxorio's response

The risk remains and is accepted: `Bridge.fundsOut` still validates only `destinationChainId != 0`, so a local release is not bound on-chain to `block.chainid`. A blanket equality check was intentionally not added because the LayerZero flow legitimately releases with a `destinationChainId` different from the local chain (the release is forwarded onward via the adapter rather than delivered locally), so correctness of local-release routing relies on the off-chain backend/federation signing process. Before production, the client should consider a route-aware binding that ties local releases to the bridge chain while validating forwarded releases against the route/ `dstEid` configuration.

W-08	Predictable `operationId` allows blocking of victim transactions in `Bridge.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `_fundsIn`	284

Description

In the function `_fundsIn` of contract `Bridge` the `operationId` is not bound on-chain to the tuple `(sender, amount, route, destinationAddress)`. If the off-chain backend issues monotonically increasing identifiers, an attacker can predict the next identifier, submit `fundsIn(amount = 1, operationId = N + 1,...)`, and execute its transaction before the victim's. The victim's transaction then reverts with the settlement module's duplicate-record error (`DuplicateOperationId`, raised in `RgbSettlementModule.onFundsIn`, not in `Bridge._fundsIn`). The pre-emption therefore applies on routes whose settlement module de-duplicates `operationId` — the RGB-settlement route — whereas the deposit route backed by `NullSettlementModule` performs no on-chain `operationId` de-duplication. The cost to the attacker is bounded by one wei, gas, and the native commission.

Proof of Concept

```
// step 1: attacker deposits first with the victim's predicted identifier (12345)
vm.prank(attacker);
bridge.fundsIn(1 ether, RGB_CHAIN_ID, 'rgb:attacker/utxo1', 12345, '');

// step 2 / impact: the victim's deposit with the same id reverts — pre-empted for ~1 wei
vm.prank(user);
vm.expectRevert; // reverts DuplicateOperationId (RgbSettlementModule.onFundsIn)
bridge.fundsIn(75 ether, RGB_CHAIN_ID, 'rgb:victim/utxo1', 12345, '');
```

Recommendation

We recommend to apply the fix off-chain: the backend should issue unpredictable 128-bit random identifiers from a CSPRNG rather than monotonically increasing ones, and the user interface should catch the revert and retry with a new identifier obtained from the backend. As optional on-chain hardening, the `operationId` can be bound to the depositor — for example by deriving/deduplicating it from `(sender, amount, route, destinationAddress)` or requiring an EIP-712 user commitment — so that a third party cannot pre-empt a victim's identifier.

Update

Client's response

Callers no longer supply the canonical operationId; Bridge derives it from the authenticated source sender, a per-sender nonce, token, amount, route, destination, settlement data, contract address, and chain ID. The LZ entrypoint stamps its actual caller as sourceSender, so another sender cannot preempt the same sender/nonce domain.
On the usdt0 side (both Ethereum and Tron UtexoSourceEntrypoint, and UtexoLZAdapter), the caller-supplied operationId/opId parameter was removed entirely from deposit/compose payloads. The adapter now only surfaces the operationId Bridge itself returns from fundsIn (in its own ComposeFundsIn event, for backend visibility) — it never computes or accepts one as input.

Oxorio's response

Callers no longer supply the canonical `operationId`; `Bridge` derives it on-chain from the authenticated source sender, a per-sender nonce, token, amount, route, destination, settlement data, contract address, and chain id, and the LZ entrypoint stamps its real caller as `sourceSender`. On the USDT0 side (both the Ethereum and Tron `UtexoSourceEntrypoint` and the adapter) the caller-supplied identifier was removed entirely. A third party can therefore no longer predict and pre-empt a victim's identifier, so the finding is closed.

Fixed at: `7391a9af`, `1cee5bb1`, `59fb06b1`

W-09	Malformed `lzCompose` payload leaves USDT0 stuck without a `_stuckFunds` record in `UtexoLZAdapter.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `lzCompose` (payload `abi.decode` outside `try/catch`)	183-189

Description

In the function `lzCompose` of contract `UtexoLZAdapter` the `abi.decode` of the payload sits outside the `try / catch` block. A malformed payload therefore reverts before the catch branch has the opportunity to record `_stuckFunds[guid]`. By that point the OFT has already minted the USDT0 to the adapter, so the tokens are held with no recovery anchor and every LayerZero retry reverts identically. The trigger requires a compromise of `trustedEntrypoints`, which is gated by the federation.

Proof of Concept

```
// the adapter is credited with USDT0 (as a real lzReceive does); a truncated payload is delivered
bytes memory innerPayload = abi.encode(uint256(0xdeadbeef)); // truncated, not the full five-field struct
//...
vm.prank(lzEndpoint);
vm.expectRevert; // abi.decode fails, outside try/catch
adapter.lzCompose(lzOft, guid, msg_, address(0), '');

// impact: no _stuckFunds slot is written, so the minted USDT0 is stuck with no recovery anchor
assertEq(adapter.getStuckFunds(guid).amountLD, 0, 'no stuck-funds slot populated');
```

Recommendation

We recommend pre-checking the length and shape of the payload before decoding — Solidity `try / catch` cannot wrap a bare `abi.decode`, since it only catches reverts from external calls — or moving the decode into an external decoder-helper call that is wrapped in `try / catch`. Either way, a malformed payload should also land in `_stuckFunds[guid]` with a recovery anchor instead of reverting before the catch branch.

Update

Client's response

The business payload is now decoded through an external pure helper wrapped in try/catch. If decoding fails, the adapter records a minimal recoverable _stuckFunds entry containing the OFT-credited amountLD and forwarded native value, and refundStuckFunds can release both assets.

Oxorio's response

The business payload is now decoded through an external pure helper wrapped in `try/catch`; on a decode failure the adapter records a minimal recoverable `_stuckFunds` entry holding the OFT-credited `amountLD` and forwarded native value, and `refundStuckFunds` can release both. Malformed compose payloads are parked as recoverable records rather than stranding the minted USDT0, closing the finding. Deep review additionally confirmed there is no guid-replay via front-run: `lzCompose` is endpoint-gated, authenticates the origin, and the LayerZero compose queue is one-shot.

Fixed at: `ca2a9861`

W-10	Issuer blocklist plus push delivery bricks a release to a frozen recipient in `Bridge.sol`, `UtexoLZAdapter.sol`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `fundsOut`	227
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `refundStuckFunds`	353

Description

The bridge asset is USDT0, an OFT wrapper over Tether liquidity, and is therefore exposed to the issuer's blocklist (`addBlackList` / `destroyBlackFunds` on canonical USDT; the OFT inherits the same control surface). The release path delivers funds with a **push** `IERC20(TOKEN).safeTransfer(recipient, netAmount)` in `Bridge.fundsOut`. If the issuer freezes `recipient`, that transfer reverts and the **entire** `fundsOut` rolls back — including the `consumedBurnIds[burnId]` mark — so there is no partial credit and no per-recipient escape hatch. The signed payload fixes a single `recipient`, so the federation cannot redirect that release without a fresh TEE signature for a different address. The same shape applies to the push transfers in `UtexoLZAdapter.refundStuckFunds` and `sendOut`: a blocked recipient bricks recovery / the outbound leg.

This is the classic push-vs-pull bridge failure mode. Under the trusted-issuer assumption (Tether will not freeze legitimate bridge users) the impact is operational rather than a loss of custody (INFO); under the zero-trust lens (untrusted issuer) it is rated WARNING, because the money path has no pull-based fallback.

Proof of Concept
The proof of concept re-deploys the stack against a `BlocklistERC20` (USDT-style freeze) on a Null `RGB ↔ SOURCE` route.

```
// pool funded by a normal deposit; the issuer then blocks `recipient`
blToken.setBlocked(recipient, true);
// impact: the whole fundsOut rolls back on the push transfer — the release is stranded,
funds stay pooled
```

```
vm.expectRevert; // AddressBlocked -> whole fundsOut rolls back
proxy.execute(callData /* recipient frozen */, nonce, deadline, 3, sigs);
assertEq(token.balanceOf(address(bridge)), deposit, 'funds still trapped in the pool');
```

Recommendation

We recommend a pull-based withdrawal (credit a per-recipient balance and let the recipient claim), or a federation-gated `redeliverFundsOut(burnId, newRecipient)` escape hatch so a release frozen at one address can be re-pointed without minting a second `burnId`. At minimum, document that a frozen recipient strands that release.

Update

Client's response

If the issuer freezes a release recipient, the push transfer reverts and the whole fundsOut rolls back atomically (burnId is not consumed) — funds stay safely pooled in the Bridge rather than lost or stranded mid-flight. Recovery is operational, not on-chain: the federation/enclave can simply sign a fresh release for a different, unblocked recipient using the same burn.

Oxorio's response

The risk remains and is accepted under the trusted-issuer assumption: the release path still uses a push `safeTransfer`, so an issuer freeze on the recipient reverts the whole `fundsOut` (and likewise `refundStuckFunds` / `sendOut`), with no pull-based fallback. The compensating control is that the revert is atomic — `burnId` is not consumed and the funds stay pooled in the Bridge rather than being lost or stranded mid-flight — and recovery is operational: the enclave/federation signs a fresh release to a different, unblocked recipient using the same burn. Before production, a pull-based withdrawal or a federation-gated re-delivery hatch remains the more robust option if the issuer is treated as untrusted.

W-11	`SetTimelockDuration` has no lower bound; the federation can disable the governance delay entirely in `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType` (`SetTimelockDuration`)	828
MultisigProxy.sol	contract `MultisigProxy` > `executeProposal` (timelock gate)	694

Description

In the `SetTimelockDuration` handler of `_executeByType` and the constructor of contract `MultisigProxy`, the federation timelock (`timelockDuration`) is set. It is the delay between proposal creation and execution, enforced in `executeProposal` by a `block.timestamp < p.proposedAt + timelockDuration` check that reverts `TimelockActive`. It is mutable via the `SetTimelockDuration` governance op, but the only bound applied — in both the constructor and `_executeByType` — is the **upper** one: it reverts `TimelockTooLong` when `newDuration >= MAX_PROPOSAL_LIFETIME`. A value of `0` is accepted.

Once `timelockDuration` is set to `0`, the gate `block.timestamp < proposedAt + 0` is always false, so every subsequent federation proposal can be **proposed and executed in the same block**. The observation/veto window that the governance model relies on to make privileged changes (signer-set rotation, registry/adapter swaps, withdrawals via `AdminExecute`) externally visible before they take effect is removed, with no on-chain floor to prevent it. The change to `0` itself is still gated by the *current* timelock, so it is observable once — but everything after it is instant.

Proof of Concept

```
// step 1: a proposal sets timelockDuration = 0 (still gated once by the current 1h timelock)
assertEq(proxy.timelockDuration, 0, 'timelock disabled (no floor enforced)');
// step 2 / impact: a fresh AdminExecute is now proposed and executed in the SAME block, no
vm.warp
bytes32 pid2 = proxy.proposeAdminExecute(cd, nonce2, deadline2, 3, sigs2);
```

```
proxy.executeProposal(pid2, cd); // no delay, same block
assertEq(bridge.lzAdapter, adapter, 'proposal executed instantly with timelock = 0');
```

Recommendation

We recommend to enforce a non-zero lower bound (`MIN_TIMELOCK`, e.g. 1 hour) in both the constructor and the `SetTimelockDuration` handler:

```
uint256 public constant MIN_TIMELOCK = 1 hours; // non-zero lower bound, enforced in both
places
// constructor and _executeByType(SetTimelockDuration):
if (newDuration < MIN_TIMELOCK || newDuration >= MAX_PROPOSAL_LIFETIME) revert
InvalidTimelockDuration;
```

Update

Client's response

MultisigProxy now receives a nonzero minTimelock at deployment, stores it as immutable MIN_TIMELOCK, rejects an initial duration below that floor, and rejects every later SetTimelockDuration value below it. Governance cannot reduce the observation window to zero.

Oxorio's response

`MultisigProxy` now receives a non-zero `minTimelock` at deployment, stores it as immutable `MIN_TIMELOCK`, and rejects both an initial and any later `SetTimelockDuration` value below that floor. Governance can no longer reduce the observation window to zero, so the finding is closed.

Fixed at: `1035c3df`

W-12	Signer-set thresholds have no on-chain floor: a 1-of-N enclave or federation is accepted in `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > constructor	163-165
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType` (`UpdateEnclaveSigners` / `UpdateFederationSigners`)	790-806

Description

In the constructor and the `_executeByType` signer-update handlers of contract `MultisigProxy`, the only constraint on a threshold reverts `InvalidThreshold` when `threshold == 0 || threshold > signers.length`. There is no minimum and no majority requirement, so a `1-of-N` enclave or federation is accepted. A single key would then authorise a release (enclave) or any governance action including rotating the enclave set (federation, see the trust-domain-collapse finding). The deployment template sets threshold defaults to 2, while signer arrays are operator-supplied; nothing on-chain prevents a weaker configuration, and the federation can lower its own threshold or shrink its set through `proposeUpdateFederationSigners`. Combined with the removable timelock floor (see the `SetTimelockDuration` finding), the practical cost of capturing the money path can fall to a single key with no observation window.

Recommendation

We recommend to enforce an on-chain minimum threshold — ideally a strict majority — for both signer sets, in the constructor and in both signer-update handlers:

```
// reject any sub-majority quorum (also rejects the 1-of-N degenerate case)
if (threshold == 0 || threshold > n || 2 * threshold <= n) revert InvalidThreshold;
```

Update

Client's response

A shared validator is applied to enclave and federation domains at construction and on every signer-set update. It requires at least two signers, a threshold of at least two, a threshold no greater than the set size, and a strict majority. Consequently, 1-of-N, 1-of-1, and other sub-majority configurations are no longer representable.

Oxorio's response

A shared validator now enforces, at construction and on every signer-set update for both domains, at least two signers, a threshold of at least two, a threshold no greater than the set size, and a strict majority (`2 * threshold > n`). 1-of-N, 1-of-1, and other sub-majority quorums are no longer representable on-chain, closing the finding.

Fixed at: `dc26e901`

W-13	`DOMAIN_SEPARATOR` is immutable and not recomputed on a chain-id change, enabling cross-fork signature replay in `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > constructor	190
MultisigProxy.sol	contract `MultisigProxy` > function `_hashTypedData`	918

Description

In the constructor and the function `_hashTypedData` of contract `MultisigProxy`, the EIP-712 domain separator is computed once over `block.chainid` and stored, then reused for every digest:

```
DOMAIN_SEPARATOR = keccak256(abi.encode(
 _DOMAIN_TYPEHASH, keccak256('MultisigProxy'), keccak256('1'), block.chainid, address(this)
));
//...
function _hashTypedData(bytes32 structHash) private view returns (bytes32) {
 return keccak256(abi.encodePacked('\x19\x01', DOMAIN_SEPARATOR, structHash));
}
```

The value is never recomputed. If the settlement chain undergoes a contentious hard fork, both forks keep the original `chainId` baked into the separator, so every signature (enclave `execute` / `executeBatch` and federation `propose*`) is valid on **both** chains. Because contract state — including the per-selector `nonces`, `batchNonce`, and `proposalNonce` — is duplicated across the fork, a single signed operation can be replayed on the sibling chain against the same nonce. The released asset (USDT0) and pooled balances likewise exist on both forks, so a release signed on one chain drains the mirror pool on the other.

This is the well-known cached-domain-separator pitfall that OpenZeppelin's `EIP712` base contract guards against by caching the separator together with the chain id and rebuilding it whenever `block.chainid` differs from the cached value:

```
// OpenZeppelin EIP712 (abridged): cache the separator AND the chain id at construction,
// then return the cached value only while the chain id is unchanged, else rebuild it.
bytes32 private immutable _cachedDomainSeparator;
uint256 private immutable _cachedChainId;
address private immutable _cachedThis;
bytes32 private immutable _hashedName;
bytes32 private immutable _hashedVersion;

function _domainSeparatorV4 internal view returns (bytes32) {
 if (address(this) == _cachedThis && block.chainid == _cachedChainId) {
 return _cachedDomainSeparator; // fast path: chain id unchanged
 } else {
 return _buildDomainSeparator; // chain forked / proxy delegate: rebuild
 }
}

function _buildDomainSeparator private view returns (bytes32) {
 return keccak256(abi.encode(_TYPE_HASH, _hashedName, _hashedVersion, block.chainid,
address(this)));
}
```

Recommendation

We recommend inheriting OpenZeppelin's `EIP712` base contract directly and replacing the hand-rolled `DOMAIN_SEPARATOR` / `_hashTypedData` with `_domainSeparatorV4()` / `_hashTypedDataV4()`, so the separator is rebuilt automatically after a chain-id change and pre-fork signatures cannot be replayed on the forked chain.

Update

Client's response

MultisigProxy caches the deployment chain ID and domain separator, uses the cached separator only while block.chainid is unchanged, and rebuilds it from the current chain ID after a change. Every typed-data digest uses this accessor, so a signature from the old domain is invalid in the changed domain.

Oxorio's response

`MultisigProxy` now caches the deployment chain id alongside the domain separator, uses the cached separator only while `block.chainid` is unchanged, and rebuilds it from the current chain id after a change; every typed-data digest uses this accessor. A signature from the pre-fork domain is therefore invalid on a forked chain, closing the cross-fork replay vector.

Fixed at: `451ecac6`

W-14	No disjointness check between the enclave and federation signer sets in `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `_validateSigners`	992
MultisigProxy.sol	contract `MultisigProxy` > constructor	169

Description

In the function `_validateSigners` of contract `MultisigProxy`, zero addresses and duplicate entries are rejected, and it is run on both sets at deploy and on every rotation. The check is, however, applied to each set **in isolation**: there is no constraint that the enclave set and the federation set be disjoint. The same key (or an overlapping subset) may sit in both `_enclaveSigners` and `_federationSigners`.

The design intends two separate trust domains — the enclave authorises releases, the federation governs configuration and can rotate the enclave set (see the trust-domain-collapse CRITICAL). If the two sets overlap, that separation is partially fictional on-chain: a single compromised key counts toward quorum in both domains, lowering the real cost of capturing the money path below what the two nominal thresholds suggest. In the extreme — identical sets with identical thresholds — there is effectively one M-of-N committee wearing two hats, and the second domain provides no independent control.

Recommendation

We recommend to enforce disjointness whenever either set is assigned (constructor and both signer-update handlers): reject any address present in the other set. A straightforward check is to revert if any candidate signer is already a member of the counterpart set.

```
// when setting/rotating one set, reject any address that is also in the other set
for (uint256 i; i < newSigners.length; i++) {
 if (_isMember(counterpartSet, newSigners[i])) revert SignerSetsOverlap(newSigners[i]);
}
```

If overlap is an accepted operational choice, document it explicitly together with the reduced effective security of the two-domain model.

Update

Client's response

Construction rejects federation/enclave overlap, enclave rotations are checked against the federation and every other enclave domain, and federation rotations are checked against all registered enclave sets. Any overlap reverts, so one address can no longer count toward quorum in both intended trust domains.

Oxorio's response

Construction rejects federation/enclave overlap, enclave rotations are checked against the federation set and every other enclave domain, and federation rotations are checked against all registered enclave sets, with any overlap reverting. A single key can no longer count toward quorum in both trust domains, so the finding is closed.

Fixed at: `d96f4ed9`

W-15	`AdminExecuteCommissionManager` bypasses the `commissionRecipient` pin on commission withdrawals in `MultisigProxy.sol`, `CommissionManager.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType(AdminExecuteCommissionManager)`	834
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType(WithdrawTokenCommissionCM)`	839
CommissionManager.sol	contract `CommissionManager` > function `withdrawTokenCommission`	498

Description

In the functions `withdrawTokenCommission`, `withdrawNativeCommission`, `withdrawAllTokenCommission`, and `withdrawAllNativeCommission` of contract `CommissionManager`, the recipient `to` is taken as a free parameter, validated only as `to != address(0)`. Their owner is the `MultisigProxy`, which exposes two governance routes to them:

- the dedicated ops `WithdrawTokenCommissionCM` / `WithdrawNativeCommissionCM` force the recipient to the governance-configured `commissionRecipient` (`address recipient = commissionRecipient;`), so revenue can only leave to the agreed address;
- the generic `AdminExecuteCommissionManager` performs a raw `commissionManager.call(opData)` from the proxy (the owner), so a proposal may encode `withdrawTokenCommission(token, anyAddress, amount)` / `withdrawAllTokenCommission(token, anyAddress)` and send commission to an arbitrary recipient.

The `commissionRecipient` pin is therefore an invariant of one path only; an equally federation-gated parallel path ignores it. This does not grant authority beyond what the federation already holds as owner (it is part of the accepted cold-admin trust assumption from the admin-federation model), but it makes the pinned-recipient control illusory at the

asset layer: a mistaken or malicious proposal — or any off-chain monitoring keyed on "commission only ever moves to `commissionRecipient`" — is not actually constrained. The same raw-call shape lets `AdminExecute` reach `Bridge.fundsOut` with an arbitrary recipient, but that is already covered by the pool-control findings.

Recommendation

We recommend to make the recipient pin an asset-layer invariant rather than a property of one proxy path. Either remove the free `to` parameter from the `CommissionManager` withdrawal functions and always pay a recipient stored in `CommissionManager` itself, or reject the `withdraw*` selectors inside `AdminExecuteCommissionManager` so commission can only ever exit through the dedicated pinned operations:

```
// in AdminExecuteCommissionManager, before commissionManager.call(opData):
bytes4 sel = bytes4(opData[:4]);
if (
 sel == CommissionManager.withdrawTokenCommission.selector ||
 sel == CommissionManager.withdrawNativeCommission.selector ||
 sel == CommissionManager.withdrawAllTokenCommission.selector ||
 sel == CommissionManager.withdrawAllNativeCommission.selector
) revert UseDedicatedWithdrawOp;
```

Update

Client's response

The generic AdminExecuteCommissionManager path rejects all four free-recipient withdrawal selectors at proposal creation and again at execution. Token and native withdrawals must use dedicated operations that resolve the destination from the proxy's stored commissionRecipient, so generic governance calls can no longer bypass the recipient pin.

Partially fixed at: `7a013cb4`

Oxorio's response

Remaining issue: The `commissionRecipient` pin is still not an asset-layer invariant. Blocking the four free-recipient withdrawal selectors on the `AdminExecuteCommissionManager` path closes the direct bypass, but generic ownership migration remains available: because `CommissionManager` withdrawal functions still take a free `to` parameter validated only as non-zero, a new CommissionManager owner that accepts ownership can call them with an arbitrary recipient, escaping the pin.

Required remediation: Make the recipient pin a property of the asset layer rather than of one proxy path — remove the free `to` parameter from the CommissionManager

withdrawal functions and always pay a recipient stored in `CommissionManager` itself, so that no owner (current or migrated) can direct commission to an arbitrary address. Alternatively, constrain or disable ownership transfer of `CommissionManager` so it cannot be used to bypass the pin.

Reproduction: Through the governed path, transfer `CommissionManager` ownership to an attacker-controlled address; once that address calls `acceptOwnership` it invokes `withdrawTokenCommission` / `withdrawAllTokenCommission` (whose `to` is validated only as non-zero) and sends accrued commission to an arbitrary recipient, even though the dedicated pinned withdrawal operations still resolve the destination from the stored `commissionRecipient`.

Client's response

`commissionRecipient` is now stored as a non-zero immutable value directly in `CommissionManager`. The free `to` parameter was removed from all four withdrawal functions, and every token or native withdrawal always transfers funds to this immutable recipient.

The recipient restriction is therefore enforced at the commission contract layer and no longer depends on a particular `MultisigProxy` execution path. Even if ownership of `CommissionManager` is transferred, the new owner cannot redirect accrued commissions to another address.

Fixed at: `4ac4a515`

W-16	Outbound `sendOut` is not bound on-chain to its paired `fundsOut`; the link lives only in the TEE in `UtexoLZAdapter.sol`, `MultisigProxy.sol`, `Bridge.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `sendOut`	243
MultisigProxy.sol	contract `MultisigProxy` > function `executeBatch`	232
Bridge.sol	contract `Bridge` > function `fundsOut`	168

Description

In the function `sendOut` of contract `UtexoLZAdapter`, the outbound RGB → non-Arbitrum release is composed by the TEE as one `executeBatch` carrying two calls: `Bridge.fundsOut` (which releases USDT0 to the adapter, gated by the route's `RGBVerifier` + `RgbSettlementModule`) followed by `UtexoLZAdapter.sendOut` (which forwards that USDT0 onward over LayerZero). The release leg is plugin-gated, but `sendOut` carries no verifier or settlement module of its own — it is a transport primitive guarded only by `onlyMultisigProxy`:

```
function sendOut(uint32 dstEid, bytes32 recipient, uint256 amount, uint256 minAmountLD,
bytes calldata extraOptions)
 external payable override onlyMultisigProxy nonReentrant
{
 if (amount == 0) revert ZeroAmount;
 if (recipient == bytes32(0)) revert InvalidRecipient;
 //...forwards `amount` of USDT0 to `recipient` on `dstEid` via the OFT...
}
```

Nothing on-chain ties `sendOut` to the `fundsOut` it is supposed to settle. `executeBatch` dispatches the two calls independently — there is no check that `sendOut`'s `amount` / `recipient` / `dstEid` match the paired `fundsOut`, that a `sendOut` is preceded by a

corresponding `fundsOut`, or that the two are even in the same batch. The atomic pairing exists solely because the TEE chose to batch them; the contracts treat `fundsOut` and `sendOut` as unrelated allowlisted operations. Consequently the binding between "what left the pool" and "what was forwarded over LayerZero, to whom, on which chain" is entirely an off-chain TEE responsibility.

This is the same unbound-release class as the MAJOR `proof`-not-bound finding, extended to the outbound transport: under a faulty or compromised enclave quorum, `sendOut` can move any USDT0 the adapter holds to any `recipient` on any `dstEid`, decoupled from any release accounting. It does not raise the worst-case impact above the enclave-drain CRITICAL (the enclave already controls the money path), but it removes one more cross-domain binding that a zero-trust design would enforce on-chain.

Recommendation

We recommend to bind the onward transport to the release on-chain rather than trusting the TEE to pair them. Options, in order of preference:

- merge the release-and-forward into a single privileged entry point (e.g. `fundsOutAndSend`) that performs the `fundsOut` accounting and the LayerZero send atomically over shared `(amount, recipient, dstEid)` parameters, so the two cannot diverge; or
- extend `fundsOut` (and the batch handler) to record a release key bound to the forwarding destination and require `sendOut` to present and consume that exact key. Note that `Bridge.fundsOut` currently takes no `dstEid`, so this first requires adding the destination (`dstEid` / forwarding recipient) to the `fundsOut` / context parameters before such a key can be computed — `keccak256(burnId, recipient, amount, dstEid)` is not derivable from today's `fundsOut` signature as-is.

If the pairing remains a TEE responsibility, state explicitly that outbound delivery (amount/recipient/destination) is not bound on-chain to the pool release and relies on enclave correctness.

Update

Client's response

Generic enclave execute/executeBatch were replaced by typed entrypoints. lzFundsOutCall commits the release and LayerZero forwarding parameters in one enclave-signed call, forces the Bridge recipient to the configured adapter, measures the adapter's actual token balance increase from Bridge.fundsOut, and then calls sendOut with that delivered amount in the same transaction. The enclave path can no longer invoke an independent or differently parameterized onward leg.

Oxorio's response

The generic dispatch was replaced by the typed `lzFundsOutCall`, which commits the release and LayerZero-forwarding parameters in one enclave-signed call, forces the Bridge recipient to the configured adapter, measures the adapter's actual token-balance increase from `Bridge.fundsOut`, and forwards exactly that delivered amount via `sendOut` in the same transaction. The onward transport is now bound on-chain to the release it settles, so the enclave path can no longer invoke an independent or differently parameterized onward leg; the finding is closed.

Fixed at: `316168fd`

W-17	In-flight token commission inflates `getContractBalance` during the `onFundsIn` hook in `Bridge.sol`, `BridgeBase.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `_fundsIn`	287
BridgeBase.sol	contract `BridgeBase` > function `getContractBalance`	78

Description

In the function `_fundsIn` of contract `Bridge`, funds move in three ordered steps — the full gross `amount` is pulled in, the per-route settlement hook is invoked, and only afterwards is the token commission forwarded to the `CommissionManager`:

```
// 1) pull the full gross amount (commission included) into the bridge
IERC20(TOKEN).safeTransferFrom(from, address(this), amount);

// 2) per-route inbound hook runs while the commission is still on the balance
IRouteRegistry(routeRegistry).onFundsIn(/*... */);

// 3) only now is the commission moved out
if (tokenCommission != 0) {
 IERC20(TOKEN).safeTransfer(address(commissionManager), tokenCommission);
 commissionManager.receiveTokenCommission(TOKEN);
}
```

During the `onFundsIn` callback the contract therefore still holds the commission portion, so the `getContractBalance` view defined in `BridgeBase` reports a transiently inflated balance. A settlement module that re-enters — or any third-party contract that reads `getContractBalance` synchronously from within that hook — observes a balance that includes commission about to leave. This is a read-only-reentrancy surface: it moves no bridge funds itself, but an integrating contract that prices off `getContractBalance` can be fed a stale value.

Recommendation

We recommend to forward the token commission to the `CommissionManager` before invoking the `onFundsIn` hook, so that `getContractBalance` reflects only the funds the bridge will actually retain at the moment any external callback runs.

Update

Client's response

`Bridge._fundsIn` now derives the received and net amounts, forwards token commission to CommissionManager, and only then invokes the settlement hook.

Oxorio's response

`Bridge._fundsIn` now forwards the token commission to the CommissionManager before invoking the `onFundsIn` settlement hook, so `getContractBalance` reflects only the funds the bridge will retain at the moment any external callback runs. The read-only-reentrancy surface is eliminated, closing the finding.

Fixed at: `d1854750`

W-18	Federation proposals use the live `timelockDuration` instead of a value snapshotted at creation in `MultisigProxy.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `executeProposal`	694

Description

In the function `executeProposal` of contract `MultisigProxy`, the timelock is enforced against the current `timelockDuration`:

```
if (block.timestamp < p.proposedAt + timelockDuration) revert TimelockActive;
```

The proposal stores its creation time `proposedAt` but not the timelock in force at that moment, and `timelockDuration` is itself mutable through a `SetTimelockDuration` proposal. Reducing `timelockDuration` therefore retroactively shortens the delay of every already-pending proposal, and lowering it can make several queued proposals become executable at once. Where those proposals are order-sensitive (for example, set the bridge address to `A` and later to `B`), the final state then depends on execution ordering rather than on the schedule intended when each was created. This compounds the missing lower bound covered by the `SetTimelockDuration` finding.

Recommendation

We recommend to snapshot the effective delay into the proposal at creation time and enforce it in `executeProposal`, so a later change to `timelockDuration` cannot alter the unlock schedule of proposals that already exist:

```
// at proposal creation:
p.timelockAtProposal = timelockDuration;
```

```
// in executeProposal:
if (block.timestamp < p.proposedAt + p.timelockAtProposal) revert TimelockActive;
```

Update

Client's response

Every proposal stores the timelockDuration in force at creation as timelockSnapshot, and executeProposal uses that stored value rather than the current mutable duration. Later governance changes cannot accelerate or postpone already pending proposals.

Oxorio's response

Every proposal now stores the `timelockDuration` in force at creation as `timelockSnapshot`, and `executeProposal` enforces that stored value rather than the current mutable duration. A later change to the timelock can no longer accelerate or postpone already-pending proposals, closing the finding.

Fixed at: `1e325b14`

W-19	An attacker can front-run `lzCompose` with an insufficient native value to park inbound deposits in the `_stuckFunds` map of `UtexoLZAdapter.sol`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `lzCompose`	150

Description

In the function `lzCompose` of contract `UtexoLZAdapter`, the inbound `Bridge.fundsIn` is invoked inside a `try / catch`, forwarding the native value the compose call received:

```
try IBridge(bridge).fundsIn{ value: msg.value }(
 amountLD, sourceChainId, destinationChainId,
 destinationAddress, operationId, settlementData
) {
 //...emit ComposeFundsIn...
} catch (bytes memory reason) {
 IERC20(token).forceApprove(bridge, 0);
 _stuckFunds[_guid] = StuckFunds({ amountLD: amountLD, nativeValue: msg.value, /*... */ });
 //...emit ComposeFundsInFailed...
}
```

If the forwarded `msg.value` is below the native commission that `_fundsIn` requires, `fundsIn` reverts (`NativeValueMismatch`), the call lands in the `catch` branch, the approval is reset, and the minted USDT0 is parked in `_stuckFunds[_guid]` instead of completing the deposit. The recipient does not receive their funds on the destination route, and the inbound transfer hangs until an operator manually calls `refundStuckFunds`.

The forwarded native value is not fixed on-chain by the route — it is supplied by whoever triggers the LayerZero compose delivery, and the adapter does not pre-validate that it covers the route's required commission before attempting `fundsIn`. This gives an attacker a concrete griefing motive rather than describing a merely accidental under-quote: by front-running a pending delivery and executing the compose with an insufficient native value, an

attacker can deliberately push other users' inbound transfers into the stuck-funds buffer, stalling deliveries and forcing repeated operator intervention. The parked funds remain recoverable, so this is a liveness / griefing vector rather than a direct loss, and it is distinct from the malformed-payload and out-of-gas cases that leave no `_stuckFunds` record at all.

Recommendation

We recommend to compute the required native commission up front via `CommissionManager.calculateFundsInCommission` (the same value `Bridge._fundsIn` enforces) after decoding the payload, and to revert `lzCompose` with an explicit error when `msg.value` cannot cover it, so an under-funded compose cannot be used to park a victim's deposit in `_stuckFunds` and LayerZero can instead retry the message with the correct native value:

```
(, uint256 nativeCommission, ) = commissionManager.calculateFundsInCommission(sourceChainId,
destinationChainId, token, amountLD);
if (msg.value < nativeCommission) revert InsufficientNativeValue(msg.value,
nativeCommission);
// forward exactly nativeCommission to Bridge.fundsIn — _fundsIn requires msg.value ==
nativeCommission
```

Update

Client's response

The depositor's expected native drop is now pinned at deposit time (UtexoSourceEntrypoint.deposit takes expectedComposeValue as part of DepositParams) and carried through the composeMsg. On delivery, UtexoLZAdapter checks msg.value against this pinned value before calling Bridge.fundsIn — a mismatch reverts (ComposeValueMismatch) instead of parking the deposit, so an attacker delivering the compose with an insufficient value can no longer grief a victim's deposit into _stuckFunds; LayerZero retries with the correct value instead.

Oxorio's response

The depositor's expected native drop is now pinned at deposit time (`expectedComposeValue` in `DepositParams`) and carried through the compose message; on delivery the adapter checks `msg.value` against the pinned value before calling `Bridge.fundsIn` and reverts with `ComposeValueMismatch` on a mismatch instead of parking the deposit. An under-funded compose can therefore no longer grief a victim's deposit into `_stuckFunds` — LayerZero retries with the correct value — so the finding is closed.

Fixed at: `32ad0905`, `4368b0dc`

3.4 INFO

I-01	Enclave `execute` / `executeBatch` deadlines have no upper bound in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `execute`	211
MultisigProxy.sol	contract `MultisigProxy` > function `executeBatch`	241

Description

In the functions `execute` / `executeBatch` of contract `MultisigProxy`, the deadline is checked only for expiry. Federation proposals bound their deadline (reverting `DeadlineTooFar` when `deadline > block.timestamp + MAX_PROPOSAL_LIFETIME`), but `execute` / `executeBatch` check only `if (block.timestamp > deadline) revert Expired;`. An enclave signature with a far-future deadline therefore stays valid until the per-selector `nonce` (or `batchNonce`) advances. A leaked or pre-signed `fundsOut` payload is effectively a bearer cheque executable at any later time until some other call consumes that nonce. Rated INFO because it requires a signature to leak; it widens the window for misuse of a pre-signed operation rather than introducing a new authority.

Recommendation

We recommend bounding the enclave deadline the same way proposals are bounded. This keeps enclave signatures short-lived and limits the time window in which a leaked pre-signed payload can be submitted:

```
if (deadline > block.timestamp + MAX_PROPOSAL_LIFETIME) revert DeadlineTooFar;
```

Update

Client's response

The former generic enclave execution entrypoints have since been replaced by the typed fundsOutCall and lzFundsOutCall flows, and both pass through _checkTeeTiming. This guard rejects expired authorizations and deadlines more than MAX_TEE_DEADLINE — currently one day — in the future, limiting the useful lifetime of leaked or pre-signed enclave payloads.

Oxorio's response

The generic enclave entrypoints were replaced by the typed `fundsOutCall`/`lzFundsOutCall` flows, both of which pass through `_checkTeeTiming`; this guard rejects expired authorizations and any deadline more than `MAX_TEE_DEADLINE` (currently one day) in the future. Enclave authorizations are now short-lived, bounding the window in which a leaked or pre-signed payload can be submitted, so the finding is closed.

Fixed at: `1503fa63`, `316168fd`

I-02	Chainlink ETH/USD feed degradation reverts every native-commission route in `CommissionManager.sol`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `convertTokenFeeToNative`	245

Description

In the function `convertTokenFeeToNative` of contract `CommissionManager` the price conversion can revert through several branches: `EthUsdFeedNotSet`, `StalePrice` (`updatedAt + heartbeat < now`), `InvalidPrice` (`answer <= 0`), `TokenDecimalsTooLarge`, and `TokenDecimalsUnavailable`. The architecture is fail-safe: reverting is preferred over pricing on degraded data, but every `fundsIn` on routes that pay the commission in native currency is blocked while the feed is unhealthy.

Proof of Concept
The proof of concept exercises each failure mode.

```
// native-currency commission rule wired; the test walks each failure mode (no feed, stale,
zero/negative, healthy)
_enableEthUsdFeed(2_000e8, 1 hours);
_wireNativeRule;
vm.warp(block.timestamp + 1 hours + 1);
// impact: once the heartbeat elapses, every fundsIn on a native-commission route reverts
until the feed recovers
vm.expectRevert; // StalePrice
cm.calculateFundsInCommission(SOURCE_CHAIN_ID, RGB_CHAIN_ID, address(token), USER_DEPOSIT);
```

Recommendation

We recommend to monitor the feed off-chain and to raise an alert ten minutes before the staleness threshold (`feed.updatedAt + heartbeat - 600s`). A fallback Chainlink feed should be pre-deployed so that a federation proposal `proposeSetEthUsdFeed` can be executed promptly.

Update

Client's response

The revert-over-mispricing design is deliberate: proceeding with stale or invalid feed data would risk mispriced commission, which is worse than a temporary block on native-commission routes. Chainlink feed degradation on a major pair (ETH/USD) is a low-probability event, and there is no comparably reliable oracle alternative readily available to fall back to automatically.

If the feed does degrade, the federation is not without recourse: setCommissionRule lets governance reconfigure any affected route's commission currency to TOKEN (bypassing the feed entirely), or setEthUsdFeed lets it point at a replacement feed — both via the standard timelocked proposal path.

Oxorio's response

The risk remains and is accepted as a deliberate fail-closed design: `convertTokenFeeToNative` reverts rather than pricing on stale or invalid feed data, so native-commission routes are blocked while the ETH/USD feed is unhealthy. The client judges mispriced commission worse than a temporary block and notes there is no comparably reliable automatic fallback oracle. Recourse remains through the timelocked path — governance can reconfigure an affected route's commission currency to `TOKEN` or point `setEthUsdFeed` at a replacement feed. Before production, the client should monitor the feed (alerting ahead of the staleness threshold) and pre-stage a fallback so the reconfiguration can be executed promptly.

I-03	Strict equality `NativeValueMismatch` does not tolerate Chainlink price drift in `Bridge.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `_fundsIn`	281

Description

In the function `_fundsIn` of contract `Bridge` the native value check is exact:

```
if (msg.value != nativeCommission) revert NativeValueMismatch;
```

`nativeCommission` is derived from `CommissionManager.convertTokenFeeToNative`, which reads the Chainlink ETH/USD feed at the quoting step. `msg.value` is fixed at the moment the user submits the transaction. A movement of one to two per cent in the ETH/USD price between the quote and submission causes every `fundsIn` on routes that pay the commission in native currency to revert.

There is also **no take-and-refund path**: the contract cannot accept the oracle-quoted `nativeCommission` and return the surplus to the user. `_fundsIn` forwards exactly `nativeCommission` to the `CommissionManager` with a raw `call{ value: nativeCommission }('')` (L308–L311) and reverts on any `msg.value` that is not exactly equal (L281). So a user cannot send a small buffer to absorb the drift — overpayment reverts just like underpayment, and the contract never refunds the difference. Note this is the native-commission (ETH/BNB) leg only; the bridged asset itself is the USDT0 token (the `TOKEN`-currency commission path deducts from the token amount and involves no `msg.value`).

Proof of Concept

```
// step 1: user reads the quote at $2,000/ETH
(, uint256 nativeQuoteOld,) = cm.calculateFundsInCommission(...);
// step 2: the feed moves to $1,000/ETH before the tx mines, doubling the fee in wei
ethUsdFeed.setAnswer(1_000e8);
```

```
ethUsdFeed.setUpdatedAt(block.timestamp);

// step 3 / impact: the tx carrying the stale quote reverts; overpaying would revert the
same way
vm.prank(user);
vm.expectRevert; // NativeValueMismatch — stale quote insufficient
bridge.fundsIn{ value: nativeQuoteOld }(USER_DEPOSIT, RGB_CHAIN_ID, 'rgb:in', 2, '');
```

Recommendation

We recommend to accept a bounded surplus and return the difference to the caller:

```
uint256 maxAccepted = (nativeCommission * 102) / 100; // accept up to 2% headroom for price
drift
if (msg.value < nativeCommission || msg.value > maxAccepted) revert NativeValueMismatch;
if (msg.value > nativeCommission) {
 (bool ok, ) = msg.sender.call{value: msg.value - nativeCommission}(""); // refund the
surplus
 if (!ok) revert NativeRefundFailed;
}
```

Update

Client's response

Partially addressed. The strict-equality check was replaced with a lower-bound check for routes whose calculated native commission is non-zero: Bridge now requires msg.value >= nativeCommission, forwards the complete supplied value to CommissionManager, and records it as native commission. On those NATIVE-commission routes, underpayment still reverts while overpayment is accepted and retained.

Partially fixed at: `cb8ce809`

Oxorio's response

Remaining issue: The native-commission amount check still does not tolerate price drift symmetrically. Replacing strict equality with a lower bound (`msg.value >= nativeCommission`) tolerates favorable movement, but underpayment still reverts and any overpayment is retained as commission rather than refunded. The invariant "a user who overpays the oracle-quoted native commission is charged only the quote and refunded the surplus" is therefore still violated.

Required remediation: Accept a bounded surplus and refund the difference — for example require `nativeCommission <= msg.value <= nativeCommission * (1 + tolerance)` and return

`msg.value - nativeCommission` to the caller — so both small adverse and favorable drift are tolerated without overcharging, and forward exactly `nativeCommission` onward.

Reproduction: On a NATIVE-commission route, quote `nativeCommission` at the current ETH/USD price, then submit `fundsIn` after the price moves. Sending slightly less than the newly required commission still reverts, and sending more than required succeeds but the surplus is kept as commission with no refund — the behavior preserved by the repository test that asserts a native overpayment is collected as surplus.

Client's response

A compile-time 5% tolerance is now applied around the fresh destination-side native commission quote.

For direct `fundsIn`, the caller may provide the fresh quote plus up to 5% buffer. The Bridge collects exactly the fresh quote and refunds the entire surplus, so moderate adverse drift can be absorbed without overcharging the user, while favorable drift produces a larger refund.

For LayerZero `fundsIn`, a cross-domain refund is not possible because the original payer is on the source chain. The authenticated source-agreed value must therefore be within a symmetric 95%–105% range of the fresh destination quote and is collected in full. Values outside this bounded range revert.

Token-commission routes continue to require `msg.value == 0`.

Fixed at: `05f34bc3`

I-04	`receiveTokenCommission` balance-delta accounting absorbs unsolicited transfers in `CommissionManager.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `receiveTokenCommission`	468

Description

In the function `receiveTokenCommission` of contract `CommissionManager` the credit is computed as the difference between the current token balance and the recorded pool counter. Any direct ERC-20 `transfer` issued by a third party to `CommissionManager` is therefore added to the next legitimate credit. Although this does not constitute theft, the federation cannot distinguish revenue from accidental donations and the accounting is opaque.

Proof of Concept

```
// step 1: with a 2% inbound rule enabled, an attacker donates 50 ether directly; the pool counter stays 0
vm.prank(attacker);
token.transfer(address(cm), 50 ether);
assertEq(cm.tokenCommissionPool(address(token)), 0);

// step 2: a legitimate fundsIn produces a 2 ether commission and calls receiveTokenCommission
vm.prank(user);
bridge.fundsIn(100 ether, RGB_CHAIN_ID, 'rgb:asset/in', 1, '');
// impact: the donation is classified as commission (pool jumps to 52 ether), withdrawable by the federation
assertEq(cm.tokenCommissionPool(address(token)), 52 ether, 'inflated pool');
```

Recommendation

We recommend to switch to an explicit-amount signature and to verify the observed delta against the expected value:

```
function receiveTokenCommission(address token_, uint256 expectedAmount) external onlyBridge {
 uint256 newBalance = IERC20(token_).balanceOf(address(this));
 uint256 delta = newBalance - tokenCommissionPool[token_];
 if (delta != expectedAmount) revert UnsolicitedDelta(expectedAmount, delta); // reject
donated balance
 tokenCommissionPool[token_] = newBalance;
 emit TokenCommissionReceived(token_, expectedAmount);
}
```

The `Bridge` contract must then pass the actual fee value as the `expectedAmount` argument.

Update

Client's response

Bridge now measures CommissionManager's token balance immediately before and after its own commission transfer and passes the actual credited delta to receiveTokenCommission. The manager adds only that value to its accounting pool and verifies that the recorded pool does not exceed its real balance. Pre-existing direct transfers therefore no longer become classified as protocol commission.

Oxorio's response

`Bridge` now measures the CommissionManager's token balance immediately before and after its own commission transfer and passes the exact credited delta to `receiveTokenCommission`, which adds only that value and verifies the recorded pool does not exceed its real balance. Pre-existing or unsolicited direct transfers are no longer absorbed as protocol commission, closing the finding. Deep review confirmed the paired `BalanceBelowRecordedPool` guard cannot trip on the normal path for the supported non-rebasing asset.

Fixed at: `f5e7adc6`

I-05	Unbounded `settlementData` storage write in the `lzCompose` catch branch in `UtexoLZAdapter.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > `lzCompose` catch branch `_stuckFunds[_guid]` write	221-228

Description

In the catch branch of the function `lzCompose` of contract `UtexoLZAdapter` the `_stuckFunds[_guid] = StuckFunds({ ... })` write stores a caller-controlled byte array (the `settlementData` field) to storage at the price of roughly 22 100 gas per 32-byte word. Under a LayerZero executor that has been quoted with insufficient native drop, the `lzCompose` invocation runs out of gas, the message dies, and the adapter holds USDT0 with no corresponding `_stuckFunds[guid]` record.

Proof of Concept
The proof of concept drives `lzCompose` into the catch branch with an unregistered destination route.

```
// an unregistered destination route forces the catch branch, which stores the full
settlementData
uint256 gas0 = _runCompose(0);
uint256 gas1k = _runCompose(1024);
uint256 gas4k = _runCompose(4 * 1024);
// impact: gas grows ~linearly with payload length, and can exhaust the executor budget
assertGt(gas1k, gas0); // 231 650 -> 329 264 gas
assertGt(gas4k, gas1k); // 329 264 -> 601 688 gas
```

Recommendation

We recommend to introduce an explicit length cap:

```
uint256 internal constant MAX_SETTLEMENT_DATA_BYTES = 1024;
if (settlementData.length > MAX_SETTLEMENT_DATA_BYTES) revert
SettlementDataTooLarge(settlementData.length);
```

The cap should be enforced both on the source-chain entrypoint (`UtexoSourceEntrypoint.deposit`) and on the destination-chain adapter (`UtexoLZAdapter.lzCompose`).

Update

Client's response

Both source entrypoint and destination adapter reject settlementData longer than 1024 bytes. The destination check occurs before the catch-path _stuckFunds write, bounding its storage cost and preventing an arbitrarily large payload from exhausting the executor gas budget during record creation.

Oxorio's response

Both the source entrypoint and the destination adapter now reject `settlementData` longer than 1024 bytes, and the destination check runs before the catch-path `_stuckFunds` write, so an oversized payload can no longer inflate that storage write or exhaust the executor gas budget during record creation. The cap is mirrored on the Tron entrypoint, closing the finding.

Fixed at: `75339eb9`, `ef68feba`

I-06	Federation signer set has quadratic validation cost and a 256-bit bitmap ceiling in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `_validateSigners`	992

Description

In the contract `MultisigProxy` the function `_validateSigners` performs an O(N²) duplicate check and O(N) storage writes. Additionally, `_verifySignatures` indexes signers through a `uint256` bitmap. The guard `if (bitmap >> signersLen != 0) revert BitmapOutOfRange;` has no effect for `signersLen >= 256`, because Solidity defines `x >> 256` to equal zero. Signers with indices beyond 255 are therefore unreachable, and a configured set larger than 256 locks those signers out of governance.

Proof of Concept

```
// deploy with 20, 50, 100 signers; gas grows super-linearly (quadratic validation tail)
assertGt(g50 - g20, 600_000); // 20 -> 50 signers
assertGt(g100 - g50, 1_000_000); // 50 -> 100 signers (quadratic tail)

// impact: the bitmap guard is a no-op at index 256, so signers beyond 255 are locked out
uint256 bitmap = type(uint256).max;
assertEq(bitmap >> 256, 0, 'shift by width = 0 -> guard is a no-op at 256');
```

Recommendation

We recommend to add an explicit upper bound on the signer count:

```
uint256 internal constant MAX_SIGNERS = 256; // bound the set to the bitmap width
if (newSigners.length > MAX_SIGNERS) revert TooManySigners(newSigners.length);
```

A tighter cap, such as 32, is appropriate for operational gas efficiency.

Update

Client's response

Both federation and enclave signer sets are capped at MAX_SIGNERS = 20. The cap is enforced before quadratic duplicate checks and keeps every signer index well within the 256-bit signature bitmap, bounding configuration gas and preventing unreachable signers.

Oxorio's response

Both signer sets are now capped at `MAX_SIGNERS = 20`, enforced before the quadratic duplicate check, which bounds configuration gas and keeps every signer index well within the 256-bit signature bitmap. The quadratic-cost and bitmap-ceiling exposure is removed, so the finding is closed.

Fixed at: `762a445a`

I-07	Zero-amount `fundsIn` succeeds and adds redundant entries to the event log in `Bridge.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `_fundsIn` (validator block has no `amount == 0` guard)	256-268

Description

In the function `_fundsIn` of contract `Bridge` there is no guard against `amount == 0`. A zero-amount deposit passes all validators and emits a full `BridgeFundsIn` event. The earlier concern that this could block an `operationId` was disproved: `RgbSettlementModule.onFundsIn` writes `fundsInRecords[operationId] = 0`, the duplicate guard `!= 0` does not trigger, and a subsequent legitimate non-zero deposit with the same `operationId` succeeds. The real impact is therefore limited to extra log entries.

Proof of Concept

```
// step 1: a zero-amount fundsIn succeeds and writes a zero record
vm.prank(user);
bridge.fundsIn(0, RGB_CHAIN_ID, 'rgb:asset/utxo1zero', 7, '');
assertEq(rgbModule.fundsInRecords(7), 0);

// step 2 / impact: a later non-zero fundsIn with the same operationId still lands — no pre-
emption, only log noise
vm.prank(user);
bridge.fundsIn(100 ether, RGB_CHAIN_ID, 'rgb:asset/utxo1real', 7, '');
assertEq(rgbModule.fundsInRecords(7), 100 ether, 'real record landed');
```

Recommendation

We recommend to reject zero-amount calls explicitly in both `_fundsIn` and `fundsOut`:

```
if (amount == 0) revert ZeroAmount; // reject empty deposits in both _fundsIn and fundsOut
```

Update

Client's response

A non-zero minFundsInAmount is mandatory at deployment and through governance updates, causing every zero-value inbound call to revert before token movement or event emission. The outbound path separately rejects amount == 0 with ZeroAmount, closing the redundant-event behavior in both Bridge directions.

Oxorio's response

A non-zero `minFundsInAmount` is now mandatory at deployment and on governance updates, so every zero-value inbound call reverts before any token movement or event emission, and the outbound path rejects `amount == 0` with `ZeroAmount`. The redundant zero-amount event behavior is closed in both directions.

Fixed at: `7fc82213`

I-08	Floor-division precision loss allows dust amounts to bridge with zero commission in `CommissionManager.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `calculateStableFee`	235

Description

In the function `calculateStableFee` of contract `CommissionManager`, under a configured rule (for example `setGlobalDefaults(stablePercent = 400, multiplier = 100, ...)`, a 4 percent fee; the contract's initial defaults are `globalStablePercent = 0` and `globalMultiplier = 100`, i.e. no commission), the formula rounds the fee to zero for any deposit with `amount < 25` smallest units. The bridge accepts such deposits without charging commission. The impact is limited to extra entries in the event graph.

Proof of Concept

```
// step 1: the fee rounds to zero for amount = 24 (one for amount = 25) — the dust ceiling
(uint256 fee0,, ) = cm.calculateFundsInCommission(SOURCE_CHAIN_ID, RGB_CHAIN_ID,
address(token), 24);
assertEq(fee0, 0, 'amount 24: fee rounds to 0');

// step 2 / impact: ten sub-threshold deposits all bridge while the commission pool never
moves
for (uint256 i; i < 10; i++) bridge.fundsIn(24, RGB_CHAIN_ID, 'rgb:dust', 1000 + i, '');
assertEq(cm.tokenCommissionPool(address(token)), poolBefore, 'no commission from dust
deposits');
```

Recommendation

We recommend optionally enforcing a minimum bridgeable amount. The minimum should be high enough that the configured commission cannot round to zero for accepted deposits:

```
if (amount < MIN_BRIDGEABLE_AMOUNT) revert AmountTooSmall; // optional floor to stop free dust bridging
```

Update

Client's response

`Bridge._fundsIn` rejects amounts below the global minFundsInAmount, and for enabled routes with a non-zero fee governance can choose a floor high enough that accepted deposits produce a non-zero commission.

Partially fixed at: `7fc82213`

Oxorio's response

Partially remediated.

Remaining issue: The global minimum amount is not coupled to active inbound/outbound fee rules, allowing zero-net deposits under valid configuration.

Required remediation: Validate the minimum amount against every active fee rule and reject `netAmount == 0`.

Proof of Concept:

```
function test_R_I_08_validHundredPercentFeeProducesZeroNetSettlement() public {
    uint256 amount = AMOUNT;
    vm.prank(deployer);
    cm.setCommissionRule(
        SOURCE_CHAIN_ID,
        RGB_CHAIN_ID,
        address(usdt0),
        CommissionConfig({
            stablePercent: 8_100,
            multiplier: 90,
            side: CommissionSide.FUNDS_IN,
            currency: CommissionCurrency.TOKEN,
            isSet: true
        })
    );

    vm.prank(user);
    bytes32 operationId = bridge.fundsIn(amount, RGB_CHAIN_ID, DST_ADDR, _rgbData());

    assertEq(usdt0.balanceOf(address(bridge)), 0, "bridge retains zero");
```

```
    assertEq(cm.tokenCommissionPool(address(usdt0)), amount, "fee receives full amount");
    assertEq(rgbModule.fundsInRecords(operationId), 0, "zero-net settlement uses zero
sentinel");
    assertEq(bridge.lockedLiquidity(RGB_CHAIN_ID), 0, "zero net liquidity credited");
}
```

Client's response

The fix enforces the invariant at three levels:

1. Commission configuration rejects fee ratios greater than or equal to 100%, preventing a valid rule from consuming the entire bridged amount.
2. When a non-zero fee policy is active, both `calculateFundsInCommission` and `calculateFundsOutCommission` revert if integer division rounds the commission to zero or if the resulting net amount would be zero.
3. `Bridge._fundsIn` also checks the actual token balance received and reverts if transfer fees plus commission would leave no net liquidity.

Fixed at: `09b1555b`

I-09	`proposalNonce` is shared between the regular and emergency proposal lanes in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `_propose` / `emergencyPause`	—

Description

In the contract `MultisigProxy` a single counter `proposalNonce` is incremented across `_propose`, `cancelProposal`, `emergencyPause`, and `emergencyUnpause`. A pending propose signed at `nonce = N` becomes stale if `emergencyPause` is executed first at the same nonce, and the federation must re-sign the propose. The behaviour does not compromise security because each lane has its own EIP-712 typehash, but it complicates the operational flow.

Proof of Concept

```
// step 1: a propose is pre-signed at the current nonce but not submitted; an emergencyPause runs at the same nonce
proxy.emergencyPause(nonceBefore, pauseDeadline, 3, pauseSigs);
assertEq(proxy.proposalNonce, nonceBefore + 1, 'pause advanced shared nonce');

// step 2 / impact: the pre-signed propose is now stale and must be re-signed (no security loss, just friction)
vm.expectRevert; // InvalidNonce
proxy.proposeAdminExecuteCommissionManager(cmCallData, nonceBefore, deadline, 3, pendingSigs);
```

Recommendation

We recommend maintaining two independent counters, so emergency operations cannot invalidate already signed regular proposals:

```
uint256 public proposalNonce; // propose / cancel / execute lane
uint256 public emergencyNonce; // emergencyPause / emergencyUnpause lane
```

Update

Client's response

Emergency pause and unpause operations now use an independent emergencyNonce, while ordinary federation proposals and cancellations retain proposalNonce.

Partially fixed at: `017cd996`

Oxorio's response

Partially remediated.

Remaining issue: Production emergency scripts sign with the regular nonce instead of the emergency-lane nonce.

Required remediation: Derive and sign the exact nonce for each proposal lane; add script/contract known-answer tests.

Proof of Concept:

The test forces the proposal and emergency nonce lanes to diverge, replays the exact `EmergencyPause.s.sol` nonce derivation (`uint256 nonce = proxy.proposalNonce();`) against the live `MultisigProxy`, observes the emergency submission revert with `InvalidNonce`, and confirms as a positive control that signing with the correct emergency-lane nonce freezes the bridge.

```
function test_R_I_09_emergencyScriptNonceFromProposalLaneRevertsInvalidNonce() public {
    (uint256[] memory fedPks, uint256 fedBitmap) = _fedSigSet2of3();

    // 1) Force the two lanes to diverge: a single ordinary federation
    //    proposal advances `proposalNonce` while `emergencyNonce` stays at 0.
    address newBridge = makeAddr("riNine_newBridge");
    uint256 proposeNonce = proxy.proposalNonce();
    uint256 proposeDeadline = block.timestamp + 1 days;
    bytes32 proposeDigest =
        MultisigHelper.digestProposeUpdateBridge(domainSep, newBridge, proposeNonce,
proposeDeadline);
    proxy.proposeUpdateBridge(
        newBridge, proposeNonce, proposeDeadline, fedBitmap, MultisigHelper.signAll(vm,
proposeDigest, fedPks)
    );
```

```
    uint256 proposalLaneNonce = proxy.proposalNonce();
    uint256 emergencyLaneNonce = proxy.emergencyNonce();
    assertTrue(proposalLaneNonce != emergencyLaneNonce, "the two nonce lanes have diverged");

    // 2) Emulate `EmergencyPause.s.sol` exactly: it reads the wrong lane
    //    (`uint256 nonce = proxy.proposalNonce();`) for an emergency op and
    //    signs the EmergencyPause digest with it.
    uint256 scriptNonce = proxy.proposalNonce();
    uint256 deadline = block.timestamp + 1 hours;
    bytes32 emergencyDigest = MultisigHelper.digestEmergencyPause(domainSep, scriptNonce,
deadline);
    bytes[] memory sigs = MultisigHelper.signAll(vm, emergencyDigest, fedPks);

    // 3) The contract checks `nonce == emergencyNonce`, so the script's
    //    submission reverts InvalidNonce — the freeze never fires.
    vm.expectRevert(IMultisigProxy.InvalidNonce.selector);
    proxy.emergencyPause(scriptNonce, deadline, fedBitmap, sigs);

    // 4) Impact confirmation: the emergency pause had no effect.
    assertFalse(bridge.paused(), "bridge NOT frozen - the emergency action reverted");

    // 5) Control: signing with the correct emergency-lane nonce succeeds,
    //    proving the revert above is caused solely by the wrong-lane nonce.
    bytes32 correctDigest = MultisigHelper.digestEmergencyPause(domainSep,
emergencyLaneNonce, deadline);
    bytes[] memory correctSigs = MultisigHelper.signAll(vm, correctDigest, fedPks);
    proxy.emergencyPause(emergencyLaneNonce, deadline, fedBitmap, correctSigs);
    assertTrue(bridge.paused(), "correct emergency-lane nonce freezes the bridge");
}
```

Client's response

The emergency scripts now read `proxy.emergencyNonce()` and include that exact nonce in the EIP-712 digest submitted to `emergencyPause` or `emergencyUnpause`. Regular timelocked proposals continue to use the independent `proposalNonce` lane.

Fixed at: `5fe27298`

I-10	`RgbSettlementModule.beforeFundsOut` exits the validation loop early in `RgbSettlementModule.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
RgbSettlementModule.sol	contract `RgbSettlementModule` > function `beforeFundsOut`	—

Description

In the function `beforeFundsOut` of contract `RgbSettlementModule` the iteration over `fundsInIds` exits as soon as the accumulated amount reaches `netAmount`. The trailing tail of `fundsInIds[i + 1..]` is not range-checked. A malformed off-chain construction that includes extra identifiers in the tail is therefore not detected on-chain.

Proof of Concept

```
// record 1 fully covers the release; ids 999/1000 are appended but do not exist
uint256[] memory ids = new uint256[](3);
ids[0] = 1; ids[1] = 999; ids[2] = 1000; // 999, 1000 do not exist
// impact: the release succeeds and the trailing ids are never validated (loop breaks once
covered)
_executeFundsOut(recipient, 50 ether, 333, RGB_CHAIN_ID, SOURCE_CHAIN_ID, 'rgb:src', proof,
abi.encode(ids));
assertEq(rgbModule.fundsInRecords(999), 0, 'fake id NOT validated');
```

Recommendation

We recommend making the intended semantics explicit and enforcing them. If `settlementData` / `fundsInIds[]` is meant to represent the exact settlement set, validate the full array and reject any trailing identifiers that are not consumed (no unread tail). If partial consumption up to `netAmount` is intentional, document the early-break behaviour explicitly and ensure the enclave/backend canonicalizes `fundsInIds[]` (no unused trailing entries) so an unread tail cannot carry meaning or smuggle data through the signed calldata.

Update

Client's response

beforeFundsOut now decodes parallel operation-ID and amount arrays, requires equal lengths, and iterates over every supplied element. Each record must exist and match its exact recorded amount, and there is no early exit that can leave a trailing identifier tail unchecked.

Oxorio's response

`beforeFundsOut` now decodes parallel operation-id and amount arrays, requires equal lengths, and iterates every supplied element, requiring each record to exist and match its exact recorded amount. There is no early exit, so a trailing identifier tail can no longer pass unvalidated through the signed calldata; the finding is closed.

Fixed at: `05d67c2f`

I-11	High event-log gas cost from unbounded `destinationAddress` and `sourceAddress` in `Bridge.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	`emit BridgeFundsIn` (full `destinationAddress`) in `_fundsIn`	314-324
Bridge.sol	`emit BridgeFundsOut` (full `sourceAddress`) in `fundsOut`	229-238

Description

The events `BridgeFundsIn` and `BridgeFundsOut` of contract `Bridge` emit the full byte sequences of `destinationAddress` and `sourceAddress` at roughly twenty-five gas per non-indexed byte, with no length cap. The attacker pays the cost, but log storage and indexer cost grow linearly.

Proof of Concept

```
// fundsIn gas measured for destinationAddress lengths of 32 B / 1 KB / 4 KB / 16 KB (same
for sourceAddress on fundsOut)
uint256 g32 = _measureFundsIn(32, 1); // ~60 619 gas
uint256 g16k = _measureFundsIn(16384, 4); // ~478 932 gas
// impact: event-log gas grows ~linearly with the unbounded address bytes
assertGt(g16k, g32, 'gas grows with destinationAddress length');
```

Recommendation

We recommend to enforce a length cap on `destinationAddress` and on `sourceAddress` (256 bytes covers every reasonable representation).

Update

Client's response

Bridge rejects inbound destinationAddress and outbound sourceAddress values longer than 512 bytes before token processing or event emission. The code therefore imposes a finite bound on these address fields; whether 512 bytes covers every future address format is not established by the contract.

Oxorio's response

`Bridge` now rejects inbound `destinationAddress` and outbound `sourceAddress` values longer than 512 bytes before any token processing or event emission, imposing a finite bound on these fields and closing the unbounded event-log gas cost.

Fixed at: `85b224a2`

I-12	Unbounded `proof` length accepted by `fundsOut` in `Bridge.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	contract `Bridge` > function `fundsOut`	168

Description

In the function `fundsOut` of contract `Bridge` the `proof` parameter is forwarded to `verifier.verify` without a length cap. A real verifier that reads the buffer amplifies the cost on top of the calldata-passthrough baseline.

Proof of Concept
The proof of concept is measured on a minimal verifier route to isolate the proof-length dimension.

```
// fundsOut gas measured for proof lengths of 0 B / 1 KB / 4 KB / 16 KB
uint256 g0 = _measure(0, 201); // ~74 579 gas
uint256 g16k = _measure(16384, 204); // ~533 226 gas
// impact: gas grows ~linearly with the uncapped proof blob forwarded to the verifier
assertGt(g16k, g0, 'proof gas grows with length');
```

Recommendation

We recommend enforcing a length cap on `proof` at the entry of `fundsOut`. The cap should match the largest valid proof format expected from the selected verifier and reject oversized calldata before it is forwarded.

Update

Client's response

Bridge.fundsOut rejects proof payloads larger than 1024 bytes before invoking the configured verifier.

Oxorio's response

`Bridge.fundsOut` now rejects `proof` payloads larger than 1024 bytes before forwarding them to the configured verifier, bounding the calldata-passthrough and verifier gas cost, so the finding is closed.

Fixed at: `9e53e78e`

I-13	High cross-chain event-log gas cost from unbounded `destinationAddress` in `UtexoLZAdapter.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > `lzCompose` forwards `destinationAddress` into `fundsIn` / `ComposeFundsIn`	204-215
Bridge.sol	`BridgeFundsIn` emits the full `destinationAddress`	314-324

Description

A source-chain `deposit` forwards a caller-supplied `destinationAddress` through the LayerZero `composeMsg`. The destination chain re-emits the full byte sequence in `BridgeFundsIn` through `lzCompose`. A single source-chain user therefore charges gas on two chains, and the destination-side emission cost is borne by the executor budget.

Proof of Concept

Scenario:

1. `lzCompose` gas is measured for `destinationAddress` lengths of 32 B, 1 KB, 4 KB and 16 KB after a warm-up call.
2. Gas grows monotonically and roughly linearly with the payload length.

```
uint256 g0 = _measureCompose(32, 1001); // ~68 944 gas
uint256 g16k = _measureCompose(16384, 1004); // ~481 050 gas
assertGt(g16k, g0, 'lzCompose gas grows with destinationAddress length');
```

Recommendation

We recommend to apply length caps on `destinationAddress` and `settlementData` at both the source-chain entrypoint and the destination-chain adapter.

Update

Client's response

The source entrypoint rejects destination addresses longer than 512 bytes before sending the LayerZero message. The destination adapter performs the same check after transport delivery and payload decoding but before calling Bridge, writing a stuck-funds record, or emitting the compose event; settlement data is independently limited to 1024 bytes. Bridge also enforces its own 512-byte address cap.

Oxorio's response

The source entrypoint now rejects destination addresses longer than 512 bytes before sending the LayerZero message, and the destination adapter performs the same check after delivery and decoding but before calling Bridge, writing a stuck-funds record, or emitting the compose event; `settlementData` is independently capped at 1024 bytes and Bridge enforces its own 512-byte address cap. The cross-chain event-log cost is bounded on both ends, closing the finding.

Fixed at: `f3402cdd`, `61987d25`

I-14 Chainlink consumer omits the Arbitrum L2 sequencer-uptime check and a min/max circuit-breaker bound in `CommissionManager.sol`

Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
CommissionManager.sol	contract `CommissionManager` > function `convertTokenFeeToNative`	256

Description

In the function `convertTokenFeeToNative` of contract `CommissionManager`, the ETH/USD feed is read with `latestRoundData` and gated only on `answer > 0` (`InvalidPrice`) and `block.timestamp - updatedAt <= ethUsdHeartbeat` (`StalePrice`). The bridge settles on Arbitrum (an L2), and two robustness checks that recur across audit databases (Solodit / Code4rena / Sherlock) for L2 Chainlink consumers are absent:

1. **L2 sequencer-uptime feed.** Chainlink's recommended pattern for Arbitrum/Optimism is to query the Sequencer Uptime feed and reject prices both during downtime *and during the grace period right after the sequencer restarts*. Without it, a price whose `updatedAt` is fresh relative to the (operator-chosen, often multi-hour) heartbeat is still accepted in the window immediately after a sequencer recovery, when off-chain markets have moved but the L2-published price has not yet caught up. The code references no sequencer feed at all — the heartbeat is the sole liveness gate.
2. **No min/max (circuit-breaker) bound.** Chainlink aggregators clamp to `minAnswer` / `maxAnswer`; during an extreme move the feed returns the floored/capped value, which `answer > 0` does not catch. A configurable sanity band would reject obviously broken prices.
3. **(Minor)** `answeredInRound` / `roundId` completeness is not checked. This is deprecated guidance on modern feeds and the heartbeat check largely subsumes it; noted only for completeness.

Scope of impact is bounded: the feed prices the **NATIVE commission only** (token→ETH fee conversion), never principal, collateral, or release amounts. During a sequencer outage the L2 is not processing user transactions anyway; the realistic exposure is the short grace window after recovery, where a `fundsIn` paying NATIVE commission could be quoted on a slightly stale price. Combined with the strict `NativeValueMismatch` equality (separate

finding), the user-visible effect is a mis-quoted or reverting deposit, not a loss of custody. The NATIVE path can also be disabled outright via `setEthUsdFeed(address(0))`. Rated INFO and framed as an Arbitrum operational-hardening recommendation.

Proof of Concept
The proof of concept enables the ETH/USD feed with a realistic 1-day heartbeat, then sets the feed's `updatedAt` to ~23h in the past (a window that could span a sequencer outage) and shows the quote still succeeds:

```
_enableEthUsdFeed(2_000e8, 1 days);
_wireNativeRule;
ethUsdFeed.setUpdatedAt(block.timestamp - 23 hours); // old, but within heartbeat
(, uint256 nativeQuote,) =
 cm.calculateFundsInCommission(SOURCE_CHAIN_ID, RGB_CHAIN_ID, address(token), USER_DEPOSIT);
assertEq(nativeQuote, 1e15, 'a ~23h-old price is consumed: heartbeat is the sole liveness
gate');
```

A sequencer-aware consumer would reject the same read during the grace period after an Arbitrum sequencer restart; this contract has no such concept.

Recommendation

We recommend, for an Arbitrum deployment, to add a Sequencer Uptime feed check (with the standard grace period, e.g. 3600s) before consuming `latestRoundData`, and to add a configurable `[minPrice, maxPrice]` sanity band. Both are pure additions to `convertTokenFeeToNative` and preserve the existing fail-closed behaviour. Reference implementation:

```
// Chainlink Arbitrum sequencer uptime feed (e.g.
0xFdB631F5EE196F0ed6FAa767959853A9F217697D).
AggregatorV3Interface public sequencerUptimeFeed;
uint256 public constant SEQUENCER_GRACE_PERIOD = 3600;

uint256 public ethUsdMinPrice; // e.g. 100e8 ($100)
uint256 public ethUsdMaxPrice; // e.g. 100_000e8 ($100k)

error SequencerDown;
error GracePeriodNotOver;
error PriceOutOfBounds;

function _requireSequencerUp internal view {
 (, int256 up, uint256 startedAt,,) = sequencerUptimeFeed.latestRoundData;
 // up == 0 => sequencer is up; up == 1 => down.
 if (up != 0) revert SequencerDown;
```

```
 if (block.timestamp - startedAt <= SEQUENCER_GRACE_PERIOD) revert GracePeriodNotOver;
}

// inside convertTokenFeeToNative, before/after reading latestRoundData:
_requireSequencerUp;
//... existing answer > 0 and heartbeat staleness checks...
if (uint256(answer) < ethUsdMinPrice || uint256(answer) > ethUsdMaxPrice) revert
PriceOutOfBounds;
```

Update

Client's response

convertTokenFeeToNative rejects quotes while a configured L2 sequencer feed reports downtime or remains within the one-hour recovery grace period, and rejects ETH/USD answers outside configured non-zero minimum/maximum bounds.

Partially fixed at: `ec7f7245`

Oxorio's response

Remaining issue: The L2/oracle hardening exists in code but is not enforced by default in production. The sequencer-uptime and min/max price-band guards are opt-in and `REQUIRE_CHAINLINK_HARDENING` defaults to false, and `_requireSequencerUp` does not explicitly reject an uninitialized round (`startedAt == 0`). A default deployment therefore still consumes ETH/USD prices without the L2 sequencer/grace and circuit-breaker protections the finding calls for.

Required remediation: Make the sequencer-uptime and min/max price-band checks mandatory for production (default the hardening flag on, or remove the opt-out for mainnet deployments), and explicitly reject `startedAt == 0` in `_requireSequencerUp` in addition to the downtime and grace-period checks.

Reproduction: Deploy with the default configuration (`REQUIRE_CHAINLINK_HARDENING = false`). `convertTokenFeeToNative` then prices native commission using only the `answer > 0` and heartbeat checks, so a price whose `updatedAt` is fresh relative to the multi-hour heartbeat is accepted in the grace window immediately after a sequencer recovery, and an uninitialized sequencer round (`startedAt == 0`) is not rejected — the repository test consumes a ~23h-old price within a 1-day heartbeat to demonstrate the heartbeat is the sole liveness gate.

Client's response

Chainlink hardening is now mandatory for every non-zero NATIVE commission quote. `convertTokenFeeToNative` fails closed with `ChainlinkHardeningNotConfigured` unless both the sequencer uptime feed and valid non-zero ETH/USD price bounds are configured.

Fixed at: `1192658f`

I-15	Owned contracts use single-step `Ownable`; a mis-addressed `transferOwnership` orphans governance irreversibly in `BridgeBase.sol`, `CommissionManager.sol`, `RouteRegistry.sol`, `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
BridgeBase.sol	contract `BridgeBase is Ownable, Pausable`	16
CommissionManager.sol	contract `CommissionManager is Ownable,...`	43
RouteRegistry.sol	contract `RouteRegistry is IRouteRegistry, Ownable`	36
MultisigProxy.sol	contract `MultisigProxy` > `_executeByType` (`AdminExecute` → `bridge.call(opData)`)	787

Description

In the contracts `Bridge` (via `BridgeBase`), `CommissionManager`, and `RouteRegistry`, OpenZeppelin's single-step `Ownable` is used, and each overrides `renounceOwnership` to revert (`RenounceOwnershipBlocked`). In production the owner of all three is the `MultisigProxy`. The federation can invoke any function on an owned contract through the `AdminExecute` / `AdminExecuteCommissionManager` governance op, which performs a raw `target.call(opData)` — this includes `Ownable.transferOwnership(address)`.

Because `transferOwnership` is single-step, a federation `AdminExecute` carrying a wrong `newOwner` (a typo, an EOA, or any contract that cannot drive the propose/timelock flow) transfers ownership **immediately and irreversibly**:

- the `MultisigProxy` is no longer the owner, so every owner-gated entry (`setLZAdapter`, `setRouteRegistry`, `setRoute`, `setCommissionRule`, `setEthUsdFeed`, withdrawals, `pause` / `unpause`, ...) becomes uncallable through governance;
- the stray new owner cannot operate the federation propose/execute machinery; and
- `renounceOwnership` is blocked, so there is no fallback to reset and re-assign.

OpenZeppelin's `Ownable` already rejects `transferOwnership(address(0))`, so only the **wrong-non-zero-address** case is unguarded — which is exactly the case `Ownable2Step`

neutralises: the transfer only sets a `pendingOwner`, and a wrong address that never calls `acceptOwnership` leaves ownership untouched.

Proof of Concept
The proof of concept drives a real federation flow (propose → timelock → `executeProposal`):

1. The proxy governs the Bridge (an owner-only `setLZAdapter` succeeds).
2. Federation `AdminExecute` calls `transferOwnership(strayEOA)`; ownership moves in one step (`bridge.owner == strayEOA`).
3. A fresh `AdminExecute` setter proposal then **reverts on execution** (`OwnableUnauthorizedAccount(proxy)`), and ownership stays stranded — governance is orphaned.

```
_proposeAndExecuteBridgeAdminCall(abi.encodeWithSelector(TRANSFER_OWNERSHIP_SELECTOR,
stray));
assertEq(bridge.owner, stray, 'ownership transferred in one step');
// later: a new AdminExecute setter proposal reverts on executeProposal
vm.expectRevert; // OwnableUnauthorizedAccount(proxy)
proxy.executeProposal(pid, cd);
```

Recommendation

We recommend to switch the three owned contracts to `Ownable2Step` so ownership only changes after the recipient explicitly accepts. This is compatible with the multisig model: a replacement `MultisigProxy` accepts ownership via its own `AdminExecute` calling `acceptOwnership`.

```
import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
import { Ownable2Step } from "@openzeppelin/contracts/access/Ownable2Step.sol";

abstract contract BridgeBase is Ownable2Step, Pausable {
 constructor(address token_) Ownable(msg.sender) { /*... */ }
 // renounceOwnership stays overridden to revert.
}
// Likewise: contract CommissionManager is Ownable2Step,...
// contract RouteRegistry is Ownable2Step,...
```

Note that `Ownable2Step` overrides `transferOwnership`/`_transferOwnership`, so the `renounceOwnership` revert override is preserved as-is.

Update

Client's response

Bridge, CommissionManager, and RouteRegistry use Ownable2Step, so transferOwnership only nominates a pending owner and the current owner remains in place until the nominated address calls acceptOwnership.

Oxorio's response

Bridge, CommissionManager, and RouteRegistry now use `Ownable2Step`, so `transferOwnership` only nominates a pending owner and the current owner remains in place until the nominee calls `acceptOwnership`. A mis-addressed transfer to a wrong non-zero address can no longer orphan governance irreversibly, closing the finding.

Fixed at: `007eda59`

I-16	`sendOut` refunds native surplus to the transaction executor via `tx.origin` instead of a signed refund recipient in `UtexoLZAdapter.sol`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
UtexoLZAdapter.sol	contract `UtexoLZAdapter` > function `sendOut`	286

Description

In the function `sendOut` of contract `UtexoLZAdapter`, both the OFT `refundAddress` and the native-surplus refund are directed to `tx.origin`:

```
(MessagingReceipt memory receipt, ) = IOFT(oft).send{ value: fee.nativeFee }(
 sp, fee, tx.origin /* refundAddress */
);

uint256 excess = msg.value - fee.nativeFee;
if (excess != 0) {
 (bool ok, ) = tx.origin.call{ value: excess }('');
 if (!ok) revert NativeRefundFailed;
}
```

`sendOut` is `onlyMultisigProxy` and is reached through `MultisigProxy.execute` / `executeBatch`, which are permissionless — any account can submit the enclave-signed batch and thereby become `tx.origin`. The signed payload does not fix who receives the refund. Whoever executes the signed batch receives any refund for the native value they supplied; this is mainly an executor-accounting / off-chain reimbursement ambiguity, not direct theft of another submitter's ETH.

Recommendation

We recommend to pass the intended refund recipient explicitly into `sendOut` so it becomes part of the enclave-signed calldata, instead of relying on `tx.origin`; execution and reimbursement can then be reconciled against the signed intent.

Update

Client's response

sendOut is reached only through protocol backend calling MultisigProxy.execute/ executeBatch with a known, backend-controlled EOA.
The audit itself frames this as an executor-accounting / off-chain reimbursement ambiguity rather than theft of bridge funds — whoever funds the call gets their own excess back, since msg.value is fronted by the submitter. Because the submitter is always a plain EOA (never a smart contract or smart wallet) in our setup, tx.origin also never introduces a functional risk (e.g. a receive-less contract bricking the refund).

Oxorio's response

The risk remains and is accepted under an operational assumption: `sendOut` still refunds native surplus (and sets the OFT refund address) to `tx.origin` rather than a signed refund recipient, so whoever submits the enclave-signed batch receives the refund for the value they supplied. The client accepts this because `sendOut` is only ever reached through the protocol backend calling `MultisigProxy.execute` / `executeBatch` with a known, backend-controlled plain EOA, which also avoids a receive-less-contract refund failure. As the original finding notes, this is an executor-accounting / off-chain reimbursement ambiguity rather than theft of bridge funds. Passing an explicit signed refund recipient into `sendOut` remains the more robust option should the executor assumption ever change.

I-17	`deposit` refunds native surplus to `msg.sender`, which may be a contract that cannot receive ETH in `UtexoSourceEntrypoint.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
UtexoSourceEntrypoint.sol	contract `UtexoSourceEntrypoint` > function `deposit`	157

Description

In the function `deposit` of contract `UtexoSourceEntrypoint`, the OFT `refundAddress` and the native-surplus refund are both directed to `msg.sender`:

```
(MessagingReceipt memory receipt, ) =
 IOFT(oft).send{ value: fee.nativeFee }(sp, fee, msg.sender);

uint256 excess = msg.value - fee.nativeFee;
if (excess != 0) {
 (bool ok, ) = msg.sender.call{ value: excess }('');
 if (!ok) revert NativeRefundFailed;
}
```

When the caller is a contract without a payable fallback or ETH-withdrawal path, the surplus refund reverts the whole deposit (`NativeRefundFailed`), so a smart-contract integrator that overpays the native fee cannot complete the call. The bridged token amount itself is unaffected; the impact is on contract callers that send a native buffer.

Recommendation

We recommend to let the caller pass an explicit refund recipient to `deposit`, defaulting to `msg.sender` only when none is supplied, so contract callers can route the native refund to an address able to receive ETH.

Update

Client's response

Depositors can specify an explicit refundTo address used for both the OFT refund address and the entrypoint's native-surplus refund. A zero value preserves msg.sender as the default, while smart-contract callers can choose an address capable of receiving native currency.

Oxorio's response

`deposit` now accepts an explicit `refundTo` address, used for both the OFT refund address and the native-surplus refund, defaulting to `msg.sender` only when zero. A smart-contract caller that overpays the native fee can route the refund to an address able to receive native currency, so an overpaying contract caller can no longer be bricked; the fix is mirrored on the Tron entrypoint and the finding is closed.

Fixed at: `e582b0c1`, `08732322`

I-18	A proposal can be created with a `deadline` shorter than `timelockDuration`, leaving it unexecutable in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	contract `MultisigProxy` > function `_propose`	755

Description

In the function `_propose` of contract `MultisigProxy`, the only bound on `deadline` is the upper one:

```
if (deadline > block.timestamp + MAX_PROPOSAL_LIFETIME) revert DeadlineTooFar;
```

Nothing requires `deadline` to be at least `proposedAt + timelockDuration`. A proposal whose `deadline < block.timestamp + timelockDuration` can never execute under the current timelock: `executeProposal` first rejects it with `TimelockActive` while the timelock is running, and once the timelock elapses the `deadline` has already passed. Such a proposal consumes a nonce and is dead on arrival, wasting a governance round.

Recommendation

We recommend to validate, in `_propose`, that the deadline leaves room for the timelock, and revert otherwise, so a proposal cannot be created in an inherently unexecutable state:

```
if (deadline < block.timestamp + timelockDuration) revert DeadlineBeforeTimelock;
```

Update

Client's response

Proposal creation now reverts with DeadlineBeforeTimelock when its deadline precedes the earliest timelock execution time. An inherently unexecutable proposal fails before nonce consumption or proposal storage.

Oxorio's response

`_propose` now reverts with `DeadlineBeforeTimelock` when a proposal's deadline precedes the earliest possible timelock-execution time, so an inherently unexecutable proposal fails before consuming a nonce or proposal storage. The finding is closed.

Fixed at: `0001b2dc`

I-19	`RgbSettlementModule.beforeFundsOut` iterates an uncapped TEE-supplied `fundsInIds[]` in `RgbSettlementModule.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
RgbSettlementModule.sol	`beforeFundsOut`	126-147

Description

In the contract `RgbSettlementModule` the function `beforeFundsOut` decodes `fundsInIds` from `settlementData` with no length cap. The loop does break early once `remaining == 0`, and `fundsOut` is `onlyOwner` (the TEE quorum), so this is not an external DoS. But a redemption that must span a very large number of small deposit records can exceed the block gas limit and revert `fundsOut`; such redemptions must then be split into multiple calls, discovered only via out-of-gas. This is the gas/availability angle and is distinct from the malformed ABI-tail finding reported in the companion Enclave report.

```
// RgbSettlementModule.sol
uint256[] memory fundsInIds = abi.decode(settlementData, (uint256[])); // no length cap
for (uint256 i = 0; i < fundsInIds.length; i++) { // unbounded loop
 if (fundsInRecords[fundsInIds[i]] == 0) revert FundsInNotFound(fundsInIds[i]);
...
}
```

Recommendation

We recommend adding a `MAX_FUNDS_IN_IDS` constant and reverting `if (fundsInIds.length > MAX_FUNDS_IN_IDS)` right after the `abi.decode` (`RgbSettlementModule.sol`), and mirroring the same cap in the backend so it splits large redemptions across multiple `fundsOut` calls.

Update

Client's response

Bridge caps the complete settlementData payload at 4096 bytes before route dispatch, and RouteRegistry accepts dispatch only from Bridge.

Oxorio's response

`Bridge` now caps the complete `settlementData` payload at 4096 bytes before route dispatch, and `RouteRegistry` accepts dispatch only from `Bridge`. This bounds the decoded `fundsInIds[]` array to roughly 62 records, preventing an over-long redemption from exceeding the block gas limit, so the finding is closed.

Fixed at: `04b425d2`

I-20	Signer-set / threshold validation is deferred to execute-time in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	`proposeUpdate*Signers` (validated only inside `_executeByType`)	355-394

Description

In the contract `MultisigProxy` the `proposeUpdate*Signers` functions validate `newSigners` / `newThreshold` only inside `_executeByType`, i.e. at execution rather than at propose time. A malformed proposal therefore consumes a `proposalNonce` and a timelock slot before reverting at execution — wasted governance bandwidth and a confusing failure mode during a rotation.

```
// MultisigProxy.sol — propose: no validation, just stores the proposal
return _propose(OperationType.UpdateEnclaveSigners, abi.encode(newSigners,
newThreshold),...);

// MultisigProxy.sol — _executeByType: the only place it is checked
if (newSigners.length == 0) revert NoSigners;
if (newThreshold == 0 || newThreshold > newSigners.length) revert InvalidThreshold;
_validateSigners(newSigners);
```

Recommendation

We recommend adding the execute-time guards (`NoSigners` / `InvalidThreshold` / `_validateSigners`) at propose time too — in `proposeUpdateEnclaveSigners` (`MultisigProxy.sol`) and `proposeUpdateFederationSigners` — so a malformed rotation fails before it burns a nonce and a timelock slot; keep the execute-time checks as the authoritative guard.

Update

Client's response

Both signer-rotation proposal entrypoints validate the source chain where applicable, non-empty signer sets, threshold validity, signer addresses, duplicates, and maximum size before creating a proposal.

Oxorio's response

Both signer-rotation proposal entrypoints now validate the source chain (where applicable), a non-empty signer set, threshold validity, signer addresses, duplicates, and maximum size before a proposal is created. A malformed rotation now fails at propose time rather than after burning a nonce and a timelock slot, closing the finding, while the execute-time checks remain as the authoritative guard.

Fixed at: `1864605b`

I-21	`setLZAdapter` / `UpdateLZAdapter` lack a zero-check and a rotate-vs-disable event distinction in `Bridge.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
Bridge.sol	`setLZAdapter`	104-108
MultisigProxy.sol	`UpdateLZAdapter`	877-881

Description

In the contract `Bridge` the function `setLZAdapter` and the corresponding `UpdateLZAdapter` path in `MultisigProxy` treat setting the adapter to `address(0)` as an intentional "close the path" operation, but both rotate and disable emit the same `LZAdapterUpdated` event, which complicates off-chain monitoring (an operator cannot distinguish a routine rotation from a path shutdown by topic alone).

```
// Bridge.sol
function setLZAdapter(address newAdapter) external override onlyOwner {
 lzAdapter = newAdapter; // no address(0) guard
 emit LZAdapterUpdated(old, newAdapter); // same event for rotate AND disable
}
```

Recommendation

We recommend splitting rotate from disable: add an explicit `disableLZAdapter` that emits a dedicated `LZAdapterDisabled` event, make `setLZAdapter` (`Bridge.sol`) revert on `address(0)`, and mirror the same branch in `MultisigProxy._executeByType`'s `UpdateLZAdapter`.

Update

Client's response

Adapter rotation rejects address(0) in both Bridge and MultisigProxy. Deliberate shutdown uses separate disable operations that clear the adapter and emit LZAdapterDisabled, allowing monitoring to distinguish non-zero rotation from closure of the LZ path.

Oxorio's response

Adapter rotation now rejects `address(0)` in both `Bridge` and `MultisigProxy`, and a deliberate shutdown uses a separate disable operation that clears the adapter and emits `LZAdapterDisabled`. Monitoring can now distinguish a non-zero rotation from a path closure by event, and an accidental zero rotation is rejected, so the finding is closed.

Fixed at: `cd6d17af`

I-22	Inconsistent compiler pragma across sources in `MultisigProxy.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
MultisigProxy.sol	`pragma solidity ^0.8.20;` (floating) vs exactly `0.8.20` elsewhere	—
IMultisigProxy.sol	`pragma solidity ^0.8.20;` (floating) vs exactly `0.8.20` elsewhere	—

Description

In these files (`MultisigProxy.sol` and `IMultisigProxy.sol`) the pragma is `pragma solidity ^0.8.20;` (floating), while every other source file pins exactly `0.8.20`. A floating pragma admits non-deterministic builds across compiler minor versions.

```
// MultisigProxy.sol and interfaces/IMultisigProxy.sol
pragma solidity ^0.8.20; // floating — every other file pins `pragma solidity 0.8.20;`
```

Recommendation

We recommend changing line 2 of `MultisigProxy.sol` and `IMultisigProxy.sol` to the pinned `pragma solidity 0.8.20;`, and pinning `solc_version = "0.8.20"` in `foundry.toml`.

Update

Client's response

MultisigProxy.sol, IMultisigProxy.sol, and the remaining Solidity sources now use the exact 0.8.35 pragma, and Foundry pins solc_version = "0.8.35".

Oxorio's response

`MultisigProxy.sol`, `IMultisigProxy.sol`, and the remaining sources now pin the exact `0.8.35` pragma, and Foundry pins `solc_version = "0.8.35"`. The floating-pragma non-determinism is removed, closing the finding.

Fixed at: `65973812`

I-23	Code-quality set: missing `indexed` event fields, `PUSH0` target note, unused custom error in `ICommissionManager.sol`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
ethereum/src	events missing `indexed`; `PUSH0` opcode / `evm_version` per target	—
ICommissionManager.sol	`InvalidEthUsdFeed` custom error declared but unused	59

Description

In these files a small set of code-quality issues appears:

- Several events omit `indexed` on fields that off-chain indexers filter by, raising integration cost.
- The `PUSH0` opcode is supported on Arbitrum (Shanghai+) so it is a no-op there, but a redeploy to a non-Shanghai chain would break — `evm_version` should be set explicitly per target.
- In the contract `ICommissionManager` the `InvalidEthUsdFeed` custom error is declared but unused.

```
// interfaces/ICommissionManager.sol
error InvalidEthUsdFeed; // declared, never referenced in ethereum/src
```

Recommendation

We recommend adding `indexed` to the fields the backend filters on (e.g. `operationId` / `recipient` / `burnId`), setting `evm_version` explicitly in `foundry.toml` (`shanghai` for Arbitrum, `paris` for any pre-Shanghai target), and removing the unused `error InvalidEthUsdFeed;` at `ICommissionManager.sol`.

Update

Client's response

The Bridge build explicitly targets shanghai, while the multi-network usdt0 build targets paris to avoid accidental PUSH0 incompatibility. Canonical correlation fields such as Bridge operationId and burnId and LZ compose operationId are indexed, and the unused InvalidEthUsdFeed declaration was removed.

Oxorio's response

Verified fixed. EVM targets are explicit, correlation fields are indexed, and the unused error was removed. Regression coverage exercises this invariant in the remediation suite.

Fixed at: `745eea9f`, `1c976562`, `834d4015`

4. APPENDIX

4.1 SECURITY ASSESSMENT METHODOLOGY

Oxorio's smart contract security audit methodology is designed to ensure the security, reliability, and compliance of smart contracts throughout their development lifecycle. Our process integrates the Smart Contract Security Verification Standard (SCSVS) with our advanced techniques to address complex security challenges. For a detailed look at our approach, please refer to the full version of our methodology. Here is a concise overview of our auditing process:

1. Project Architecture Review

All necessary information about the smart contract is gathered, including its intended functionality and dependencies. This stage sets the foundation by reviewing documentation, business logic, and initial code analysis.

2. Vulnerability Assessment

This phase involves a deep dive into the smart contract's code to identify security vulnerabilities. Rigorous testing and review processes are applied to ensure robustness against potential attacks.

This stage is focused on identifying specific vulnerabilities within the smart contract code. It involves scanning and testing the code for known security weaknesses and patterns that could potentially be exploited by malicious actors.

3. Security Model Evaluation

The smart contract's architecture is assessed to ensure it aligns with security best practices and does not introduce potential vulnerabilities. This includes reviewing how the contract integrates with external systems, its compliance with security best practices, and whether the overall design supports a secure operational environment.

This phase involves a analysis of the project's documentation, the consistency of business logic as documented versus implemented in the code, and any assumptions made during the design and development phases. It assesses if the contract's architectural design adequately addresses potential threats and integrates necessary security controls.

4. Cross-Verification by Multiple Auditors

Typically, the project is assessed by multiple auditors to ensure a diverse range of insights and thorough coverage. Findings from individual auditors are cross-checked to verify accuracy and completeness.

5. Report Consolidation

Findings from all auditors are consolidated into a single, comprehensive audit report. This report outlines potential vulnerabilities, areas for improvement, and an overall assessment of the smart contract's security posture.

6. Reaudit of Revised Submissions

Post-review modifications made by the client are reassessed to ensure that all previously identified issues have been adequately addressed. This stage helps validate the effectiveness of the fixes applied.

7. Final Audit Report Publication

The final version of the audit report is delivered to the client and published on Oxorio's official website. This report includes detailed findings, recommendations for improvement, and an executive summary of the smart contract's security status.

4.2 CODEBASE QUALITY ASSESSMENT REFERENCE

The tables below describe the codebase quality assessment categories and rating criteria used in this report.

Category	Description
Access Control	Evaluates the effectiveness of mechanisms controlling access to ensure only authorized entities can execute specific actions, critical for maintaining system integrity and preventing unauthorized use.
Arithmetic	Focuses on the correct implementation of arithmetic operations to prevent vulnerabilities like overflows and underflows, ensuring that mathematical operations are both logically and semantically accurate.
Complexity	Assesses code organization and function clarity to confirm that functions and modules are organized for ease of understanding and maintenance, thereby reducing unnecessary complexity and enhancing readability.
Data Validation	Assesses the robustness of input validation to prevent common vulnerabilities like overflow, invalid addresses, and other malicious input exploits.
Decentralization	Reviews the implementation of decentralized governance structures to mitigate insider threats and ensure effective risk management during contract upgrades.
Documentation	Reviews the comprehensiveness and clarity of code documentation to ensure that it provides adequate guidance for understanding, maintaining, and securely operating the codebase.
External Dependencies	Evaluates the extent to which the codebase depends on external protocols, oracles, or services. It identifies risks posed by these dependencies, such as compromised data integrity, cascading failures, or reliance on centralized entities. The assessment checks if these external integrations have appropriate fallback mechanisms or redundancy to mitigate risks and protect the protocol's functionality.
Error Handling	Reviews the methods used to handle exceptions and errors, ensuring that failures are managed gracefully and securely.
Logging and Monitoring	Evaluates the use of event auditing and logging to ensure effective tracking of critical system interactions and detect potential anomalies.
Low-Level Calls	Reviews the use of low-level constructs like inline assembly, raw `call` or `delegatecall`, ensuring they are justified, carefully implemented, and do not compromise contract security.

Category	Description
Testing and Verification	Reviews the implementation of unit tests and integration tests to verify that codebase has comprehensive test coverage and reliable mechanisms to catch potential issues.

4.2.1 Rating Criteria

Rating	Description
Excellent	The system is flawless and surpasses standard industry best practices.
Good	Only minor issues were detected; overall, the system adheres to established best practices.
Fair	Issues were identified that could potentially compromise system integrity.
Poor	Numerous issues were identified that compromise system integrity.
Absent	A critical component is absent, severely compromising system safety.
Not Applicable	This category does not apply to the current evaluation.

4.3 FINDINGS CLASSIFICATION REFERENCE

4.3.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

Title	Description
CRITICAL	Issues that pose immediate and significant risks, potentially leading to asset theft, inaccessible funds, unauthorized transactions, or other substantial financial losses. These vulnerabilities represent serious flaws that could be exploited to compromise or control the entire contract. They require immediate attention and remediation to secure the system and prevent further exploitation.
MAJOR	Issues that could cause a significant failure in the contract's functionality, potentially necessitating manual intervention to modify or replace the contract. These vulnerabilities may result in data corruption, malfunctioning logic, or prolonged downtime, requiring substantial operational changes to restore normal performance. While these issues do not immediately lead to financial losses, they compromise the reliability and security of the contract, demanding prioritized attention and remediation.
WARNING	Issues that might disrupt the contract's intended logic, affecting its correct functioning or making it vulnerable to Denial of Service (DDoS) attacks. These problems may result in the unintended triggering of conditions, edge cases, or interactions that could degrade the user experience or impede specific operations. While they do not pose immediate critical risks, they could impact contract reliability and require attention to prevent future vulnerabilities or disruptions.
INFO	Issues that do not impact the security of the project but are reported to the client's team for improvement. They include recommendations related to code quality, gas optimization, and other minor adjustments that could enhance the project's overall performance and maintainability.

4.3.2 Status Level Reference

Based on the feedback received from the client's team regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

Title	Description
NEW	Waiting for the project team's feedback.

Title	Description
FIXED	Recommended fixes have been applied to the project code and the identified issue no longer affects the project's security.
ACKNOWLEDGED	The project team is aware of this finding and acknowledges the associated risks. This finding may affect the overall security of the project; however, based on the risk assessment, the team will decide whether to address it or leave it unchanged.
NO ISSUE	Finding does not affect the overall security of the project and does not violate the logic of its work.

4.4 ABOUT OXORIO

OXORIO is a blockchain security firm that specializes in smart contracts, zk-SNARK solutions, and security consulting. With a decade of blockchain development and five years in smart contract auditing, our expert team delivers premier security services for projects at any stage of maturity and development.

Since 2021, we've conducted key security audits for notable DeFi projects like Lido, 1Inch, Rarible, and deBridge, prioritizing excellence and long-term client relationships. Our co-founders, recognized by the Ethereum and Web3 Foundations, lead our continuous research to address new threats in the blockchain industry. Committed to the industry's trust and advancement, we contribute significantly to security standards and practices through our research and education work.

Our contacts:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

THANK YOU FOR CHOOSING

