Security Audit Report

Stellar Core Protocol 23 Changes

v1.0

October 17, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Stellar Development Foundation to perform a security audit of changes for Stellar Core Protocol 23.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/stellar/stellar-core
Commit	`9819d1b3bcfc13c60dcf88a948811fd2f96e7eef`
Scope	Changes to the repository since commit `2d8d764cdb800235d8bc33286197bc64a875931b` were reviewed, including an assessment of how these modifications were integrated into the existing codebase. The review is restricted to the following files: • Overlay module • Crypto module

- Database module
- Main module
- SCP module
- Utils module
- Transactions module except:
    - `src/transactions/ParallelApplyStage.cpp`
    - `src/transactions/ParallelApplyStage.h`
    - `src/transactions/ParallelApplyUtils.cpp`
    - `src/transactions/ParallelApplyUtils.h`
    - `src/transactions/TransactionFrame.cpp`
    - `src/transactions/TransactionFrame.h`
    - `src/transactions/TransactionFrameBase.cpp`
    - `src/transactions/TransactionFrameBase.h`
    - `src/transactions/ExtendFootprintTTLOpFrame.cpp`
    - `src/transactions/ExtendFootprintTTLOpFrame.h`
    - `src/transactions/FeeBumpTransactionFrame.cpp`
    - `src/transactions/FeeBumpTransactionFrame.h`
    - `src/transactions/InvokeHostFunctionOpFrame.cpp`
    - `src/transactions/InvokeHostFunctionOpFrame.h`
    - `src/transactions/RestoreFootprintOpFrame.cpp`
    - `src/transactions/RestoreFootprintOpFrame.h`
    - `src/transactions/EventManager.cpp`
    - `src/transactions/EventManager.h`
    - `src/transactions/LumenEventReconciler.cpp`
    - `src/transactions/LumenEventReconciler.h`
    - `src/transactions/TransactionMeta.cpp`

	○ src/transactions/TransactionMeta.h ● Ledger module except: ○ src/ledger/LedgerManager.h ○ src/ledger/LedgerManagerImpl.cpp ○ src/ledger/LedgerManagerImpl.h ● Herder module except: ○ src/herder/ParallelTxSetBuilder.cpp ○ src/herder/ParallelTxSetBuilder.h ○ src/herder/TxSetFrame.cpp ○ src/herder/TxSetFrame.h ○ src/herder/TxSetUtils.cpp ○ src/herder/TxSetUtils.h ○ src/herder/SurgePricingUtils.cpp ○ src/herder/SurgePricingUtils.h
Fixes verified at commit	a80161d100b17fff94098435ed0a328fb33766c7 Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Repository	https://github.com/stellar/stellar-core
Scope	The scope is restricted to the changes applied in the following pull requests: ● https://github.com/stellar/stellar-core/pull/4711 reviewed at commit 933734a774ff7fcc8356607fd68d5cb9288cbc72, base branch at 29e4c3b6301f67449864b8c5b9551283a072c40a. ● https://github.com/stellar/stellar-core/pull/4704 reviewed at commit d83424abb7e15eee34cab8d11f31aaeb541655c5, base branch at 71a63a6bb46d863c37a34aa150dfe02bcd0b9915. ● https://github.com/stellar/stellar-core/pull/4699 reviewed at commit 17a9df417958e974ac6589c61964c3622681e3ed, base branch at c6209da5ec7eccd2af357be29f4c1b898c944610.

	• https://github.com/stellar/stellar-core/pull/4713 reviewed at commit `c5b46a800a50e539ae70fbbea6e505f688969103`, base branch at `b4b21fc57f353beefae9d00aa213a17ffe971ff7`.
Fixes verified at commit	`a80161d100b17fff94098435ed0a328fb33766c7` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Repository	https://github.com/stellar/stellar-core
Commit	`9819d1b3bcfc13c60dcf88a948811fd2f96e7eef`

Scope	Changes to the repository since commit `2d8d764cdb800235d8bc33286197bc64a875931b` were reviewed,
	
	including an assessment of how these modifications were integrated into the existing codebase. The review is restricted to the following files:
	
	• Transactions module:
	  ◦ `src/transactions/ParallelApplyStage.cpp`
	  ◦ `src/transactions/ParallelApplyStage.h`
	  ◦ `src/transactions/ParallelApplyUtils.cpp`
	  ◦ `src/transactions/ParallelApplyUtils.h`
	  ◦ `src/transactions/TransactionFrame.cpp`
	  ◦ `src/transactions/TransactionFrame.h`
	  ◦ `src/transactions/TransactionFrameBase.cpp`
	  ◦ `src/transactions/TransactionFrameBase.h`
	  ◦ `src/transactions/ExtendFootprintTTLOpFrame.cpp`
	  ◦ `src/transactions/ExtendFootprintTTLOpFrame.h`
	  ◦ `src/transactions/FeeBumpTransactionFrame.cpp`
	  ◦ `src/transactions/FeeBumpTransactionFrame.h`
	  ◦ `src/transactions/InvokeHostFunctionOpFrame.cpp`

- `src/transactions/InvokeHostFunctionOpFrame.h`
- `src/transactions/RestoreFootprintOpFrame.cpp`
- `src/transactions/RestoreFootprintOpFrame.h`

- Ledger module:
  - `src/ledger/LedgerManager.h`
  - `src/ledger/LedgerManagerImpl.cpp` (includes a substantial number of changes, along with miscellaneous updates related to other tasks)
  - `src/ledger/LedgerManagerImpl.h`
- Herder module:
  - `src/herder/ParallelTxSetBuilder.cpp`
  - `src/herder/ParallelTxSetBuilder.h`
  - `src/herder/TxSetFrame.cpp`
  - `src/herder/TxSetFrame.h`
  - `src/herder/TxSetUtils.cpp`
  - `src/herder/TxSetUtils.h`
  - `src/herder/SurgePricingUtils.cpp`
  - `src/herder/SurgePricingUtils.h`
- Events and invariants:
  - `src/transactions/EventManager.cpp`
  - `src/transactions/EventManager.h`
  - `src/transactions/LumenEventReconciler.cpp`
  - `src/transactions/LumenEventReconciler.h`
  - `src/transactions/TransactionMeta.cpp`
  - `src/transactions/TransactionMeta.h`
  - `src/invariant/EventsAreConsistentWithEntryDiffs.cpp`
  - `src/invariant/EventsAreConsistentWithEntryDiffs.h`

Fixes verified at commit	`2978ff0aa366219e87d31235b3026d56e6c71a1c` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Repository	https://github.com/stellar/stellar-core
Commit	`9819d1b3bcfc13c60dcf88a948811fd2f96e7eef`
Scope	Changes to the repository since the previous audit, which was performed at commit `2d8d764cdb800235d8bc33286197bc64a875931b`, including a review of the integration of these changes into the existing codebase. • `src/bucket/*` • `src/catchup/*` • `src/ledger/InMemorySorobanState.cpp` • `src/ledger/InMemorySorobanState.h` • https://github.com/stellar/stellar-core/pull/4752 • https://github.com/stellar/stellar-core/pull/4806 • https://github.com/stellar/stellar-core/pull/4810 • https://github.com/stellar/stellar-core/pull/4809
Fixes verified at commit	`4dd19a2985f1799f9b9dd2c5780f314c7a95d9ed` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Repository	https://github.com/stellar/stellar-core
Label	`stellar-core`
Commit	`f4a4e6a05d4a633e41621681348ef4a5f1873928`
Repository	https://github.com/stellar/rs-soroban-env
Label	`soroban`
Commit	`0ee19322795bd0ff9097a1984b39210d0c58a6ea`

Scope	Changes to the `soroban` repository since the previous audit, which was performed at commit `f0bc81b861efbb07f9406790cad736db90147e12`, including a review of the integration of these changes into the existing codebase. Additionally, changes to `stellar-core` related to CAP-65, 66 and 70 as well as some Rust bridge updates were included in the scope.
Fixes verified at commit	`7f2dc6c08712e0a1aa53aa3e2d50e9db5231be2e` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Stellar is a decentralized, open-source network designed for fast, low-cost payments and asset issuance. The network's core is the Stellar Consensus Protocol (SCP), a Federated Byzantine Agreement (FBA) system that enables consensus without relying on mining. The network is supported by the non-profit Stellar Development Foundation (SDF).

Stellar has been extended with Soroban, a smart contracts platform designed for scalability and a familiar developer experience using Rust and WebAssembly (Wasm). This allows developers and users to perform cross-border payments, trade digital assets, and build sophisticated DeFi applications on a single, integrated network.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	High	-
Code readability and clarity	Low-Medium	Due to the code's complexity and the stratification of the codebase caused by supporting multiple protocol versions over the years, the code is difficult to read.
Level of documentation	Medium	The documentation is extensive; however, it is not always accurate and often refers to older versions.
Test coverage	-	Since the audit was conducted on the diff between two versions, it was not possible to determine the test coverage of the code in scope.

Summary of Findings

No	Description	Severity	Status
1	Deleted ledger entries not propagated to thread state in parallel execution	Critical	Resolved
2	Missing application of transaction sorting results in incorrect execution order and ledger state divergence	Critical	Resolved
3	Unauthenticated Denial of Service via unsafe character handling in error messages	Major	Resolved
4	Bypass of memo and muxed account restrictions in fee-bumped Soroban transactions	Minor	Resolved
5	Null pointer dereference in overlay signature pre-validation	Minor	Resolved
6	Validation of Hot Archive buckets is skipped completely	Minor	Resolved
7	Infinite loop in database initialization due to missing cursor advancement	Minor	Resolved
8	Incorrect global operator new and `new[]` override under `USE_TRACY_MEMORY_TRACKING`	Minor	Resolved
9	Argument corruption in subprocess execution	Minor	Acknowledged
10	Unauthenticated remote access via public HTTP command interface	Minor	Acknowledged
11	Out of memory risk from unvalidated XDR records	Minor	Acknowledged
12	Denial of Service via unbounded stop survey signature verification	Minor	Acknowledged
13	Insufficient error handling causes node termination during bucket state recovery	Minor	Acknowledged
14	Missing thread assertion on main-thread-only operations	Minor	Resolved
15	Double unlock risk from implicitly copyable lock guards	Minor	Resolved
16	Incorrect optimistic patching of copied ledger entries	Minor	Resolved
17	Missing concurrency assertions may allow unsafe access to ledger state during application	Minor	Resolved
18	Archive decompression enabling Denial of Service	Minor	Acknowledged

19	Unconditional drift guard in ledger application causes unnecessary throttling in sequential catch-up mode	Minor	Acknowledged
20	Static transaction fees fail to reflect increased index scan cost caused by larger page sizes	Minor	Acknowledged
21	Incorrect equality implementation for type `MergeCounters`	Minor	Resolved
22	Redundant filesystem operations decrease performance	Minor	Acknowledged
23	Incorrect unit conversion may underestimate bucket size	Minor	Resolved
24	Missing transaction data in history archives causes catchup failure without proper error handling	Minor	Acknowledged
25	Missing exception handling for corrupted index files	Minor	Resolved
26	Current flat fee structure fails to reflect the variable cost of bucket list searches	Minor	Acknowledged
27	Catchup Denial of Service due to incomplete fallback for oversized buckets	Minor	Acknowledged
28	Eviction scan and search logic may delay ledger close due to blocking call	Minor	Acknowledged
29	Potential performance degradation due to unbounded growth of `assetToPoolID` map	Minor	Acknowledged
30	Implicit disabling of explicitly enabled in-memory indexing	Minor	Acknowledged
31	Eviction does not function on buckets larger than threshold	Minor	Acknowledged
32	Compiler exception during startup causes crash loop	Minor	Acknowledged
33	Result of verification is not stored reliably	Minor	Resolved
34	Prolonged non-productive shutdown time	Minor	Acknowledged
35	Premature update of `mContractsCompiled`	Minor	Acknowledged
36	Silent filtering of host function names may break linker	Minor	Acknowledged
37	Preflight snapshot clones entire tracker vector	Minor	Acknowledged
38	Network config validation lacks upper bounds on several Soroban parameters	Minor	Acknowledged
39	Synchronous contract metadata recalculation during upgrade enables Denial of Service	Informational	Acknowledged
40	Regex allows non-hex input inconsistent with hex	Informational	Acknowledged

	decoding		
41	Unchecked error from `std::remove` may mask deletion failures	Informational	Acknowledged
42	Exponential backoff retry strategy could be applied	Informational	Acknowledged
43	Redundant code	Informational	Acknowledged
44	Magic numbers decrease maintainability	Informational	Acknowledged
45	Inconsistent approach to assertions	Informational	Acknowledged
46	Unsafe vector access in `setLastTxProcessingFeeProcessingChanges`	Informational	Resolved
47	Redundant clear on moved-from object	Informational	Resolved
48	Uninitialized variable usage in quorum checker	Informational	Resolved
49	Inconsistent use of Windows compilation macros	Informational	Acknowledged
50	Redundant protocol version check creates unnecessary code complexity in transaction queue management	Informational	Acknowledged
51	Inconsistent integer types in `ParallelTxSetBuilder::Stage`	Informational	Resolved
52	Lack of error handling in file I/O operations	Informational	Resolved
53	Type mismatch in `getSize` return value	Informational	Resolved
54	Misspelled method name in `LedgerManager` interface	Informational	Resolved
55	Missing null pointer dereference in `ByteSlice` constructor	Informational	Acknowledged
56	Missing virtual destructor for `TransactionFrameBase` class	Informational	Resolved
57	Potential negative signature weight and inconsistent integer types	Informational	Acknowledged
58	Missing insertion check in `setDeltaEntry` may silently discard ledger updates	Informational	Acknowledged
59	Missing main thread assertions in `LedgerManager` may obscure threading assumptions	Informational	Resolved
60	Misplaced thread assertion in `commitChangesFromThread` leads to redundant checks	Informational	Acknowledged
61	Invariant check function modifies state	Informational	Acknowledged

62	Confusing return semantics in `maybeAdoptFailedReplayResult`	Informational	Acknowledged
63	Inconsistent empty container checks reduce code maintainability	Informational	Acknowledged
64	Missing exception handling in asynchronous transaction processing	Informational	Acknowledged
65	Quadratic-time transaction clustering	Informational	Acknowledged
66	Scattered logic for concurrency state management in ledger application	Informational	Acknowledged
67	Data race in concurrent bucket index access	Informational	Acknowledged
68	Missing protocol legality validation for Hot Archive bucket entries	Informational	Acknowledged
69	Missing validation of bucket entry key during XDR parsing	Informational	Acknowledged
70	Index creation reliability comment missing failure case	Informational	Acknowledged
71	`EntrySizeForRent` may undercharge rent and enable Denial of Service	Informational	Acknowledged
72	Fragile parsing and pretty-printing of JSON data	Informational	Acknowledged
73	Panic context lost at core boundary obscures error root cause	Informational	Resolved
74	Potential race conditions while working with filesystem	Informational	Acknowledged
75	Potential race condition in closure	Informational	Resolved
76	Imprecise usage of the errno macro	Informational	Resolved
77	Insufficient error handling	Informational	Acknowledged
78	Templates maintainability can be improved	Informational	Acknowledged
79	Misleading documentation	Informational	Resolved
80	Missing defensive check on `linearTerm`	Informational	Acknowledged
81	Miscellaneous comments	Informational	Partially Resolved

Detailed Findings

1. Deleted ledger entries not propagated to thread state in parallel execution

Severity: Critical

In `src/transactions/ParallelApplyUtils.cpp:598-612`, the `collectClusterFootprintEntriesFromGlobal` function is responsible for constructing the per-thread state (`ThreadParallelApplyLedgerState`) for each transaction cluster at the start of a parallel stage. It extracts the relevant read/write ledger keys from each transaction and attempts to populate the thread-local state using `getLiveEntryOpt` from the current global state.

However, both deleted entries (removed earlier in the same ledger) and non-existent entries return `std::nullopt` from `getLiveEntryOpt`.

This has the effect that the function logic fails to distinguish between these cases, as `collectClusterFootprintEntriesFromGlobal` simply skips any entry that returns `nullopt`. As a result, ledger keys deleted earlier in the same ledger are not recorded in the thread-local view.

This omission causes downstream issues during transaction execution. When a thread attempts to access such a key via `getLiveEntryOpt`, it finds no local state and falls back to the original snapshot (`mLiveSnapshot->load(key)`), which predates transaction execution.

Consequently, the thread retrieves stale data, effectively "resurrecting" deleted entries. Transactions may interact with deleted accounts, access invalid contract storage, or make incorrect state assumptions.

Recommendation

We recommend enhancing `collectClusterFootprintEntriesFromGlobal` by handling and propagating deleted entries.

This can be achieved by populating the thread state with `cleanErased` state for both deleted and non-existing entries. In addition, you might extend the lookup in the global state to differentiate between populated erased entries and entries that are not present.

Status: Resolved

2. Missing application of transaction sorting results in incorrect execution order and ledger state divergence

Severity: Critical

The `sortedForApplyParallel` function in `src/herder/TxSetFrame.cpp:387–409` is intended to enforce a transaction execution order during parallel application. It constructs a sorted copy of the transaction stages (`sortedStages`), applying intra-cluster transaction sorting and reordering the clusters based on the hash of their first transaction.

However, the function incorrectly returns the original unsorted input (`stages`) instead of the sorted copy (`sortedStages`). This results in the complete loss of all ordering logic computed within the function.

Consequently, transactions within each cluster remain unordered, and the execution order of clusters becomes dependent on the input state. Since transaction ordering directly impacts state transitions, this behavior leads to incorrect ledger states.

Additionally, different transaction ordering and mismatched state hashes across nodes break consensus guarantees, threatening network finality.

Recommendation

We recommend correcting the return statement to return `sortedStages` to ensure the computed ordering is applied.

Status: Resolved

3. Unauthenticated Denial of Service via unsafe character handling in error messages

Severity: Major

In `src/overlay/Peer.cpp:1597`, the `Peer::recvError` function sanitizes incoming error strings by iterating through their bytes and invoking `std::isalnum` on a plain `char`. On platforms where char is signed, any byte with a value greater than or equal to `0x80` is interpreted as a negative integer.

However, `std::isalnum` only permits arguments that are representable as unsigned `char` or `EOF`; passing a negative value invokes undefined behavior.

Since the `ERROR_MSG` message type is explicitly permitted before authentication and excluded from `HMAC` verification even after the `HELLO` handshake, any remote host with access to `PEER_PORT` can send an `ERROR_MSG` containing a single non-ASCII byte.

This is sufficient to trigger undefined behavior and eventually crash the process before the peer is disconnected. Message size restrictions offer no mitigation, as a one-byte payload is enough to exploit the flaw. This results in an unauthenticated denial of service condition.

Recommendation

We recommend casting each byte to unsigned char before passing it to `std::isalnum`, ensuring well-defined behavior across platforms and preventing maliciously crafted error messages from triggering process crashes.

Status: Resolved

4. Bypass of memo and muxed account restrictions in fee-bumped Soroban transactions

Severity: Minor

The restriction on memos and muxed source accounts in Soroban transactions that use non-source authorization can be circumvented when the transaction is wrapped in a fee bump.

The only enforcement point, `validateSorobanMemo` in `src/herder/TransactionQueue.cpp`, performs checks only when the outer envelope type is `ENVELOPE_TYPE_TX`. In `TransactionQueue::canAdd`, this helper is invoked after general validity checks, but for any `ENVELOPE_TYPE_TX_FEE_BUMP`, it returns immediately, allowing the restrictions to be bypassed.

As a consequence, a Soroban transaction with non-source authorization can include a memo or muxed source account in its inner V1 envelope, and once wrapped in a fee bump, it will be admitted into the queue.

There is no compensating enforcement in `FeeBumpTransactionFrame::checkValid`, which delegates entirely to `mInnerTx->checkValid`, nor in `TransactionFrame::checkValid`, which only verifies fees, resources, and operational consistency.

The absence of enforcement undermines the invariant that Soroban transactions using non-source authorization must not contain memos or muxed accounts. Violations of this policy can lead to metadata confusion, misrouting in downstream systems that depend on the invariant, and potential privacy exposure.

Recommendation

We recommend updating `validateSorobanMemo` to unwrap fee-bumped transactions to their inner V1 form and apply the same validation.

Status: Resolved

5. Null pointer dereference in overlay signature pre-validation

Severity: Minor

In `src/overlay/Peer.cpp:59-96`, when both `BACKGROUND_OVERLAY_PROCESSING` and `EXPERIMENTAL_BACKGROUND_TX_SIG_VERIFICATION` are enabled, the overlay thread pre-validates transaction signatures during the construction of `CapacityTrackedMessage`.

In this path, the function `populateSignatureCache` retrieves an account object via `LedgerSnapshot::getAccount(tx->getFeeSourceID())` and immediately dereferences it using `sourceAccount.current().data.account().thresholds[THRESHOLD_HIGH]` without confirming that the account exists.

If a malicious peer transmits a transaction, or fee-bump transaction, with a fee-source account absent from the current ledger state, `getAccount` returns an empty wrapper. The subsequent dereference causes a null pointer access, leading to a crash on the overlay thread.

This occurs before the main thread's standard validation logic would reject the transaction with `txNO_ACCOUNT`. As a result, a remote peer can trigger a denial of service (DoS) by completing the overlay handshake and sending a single malformed `TRANSACTION` message.

Recommendation

We recommend adding explicit existence checks before dereferencing the result of `LedgerSnapshot::getAccount`.

Status: Resolved

6. Validation of Hot Archive buckets is skipped completely

Severity: Minor

In `src/bucket/BucketInputIterator.cpp:54-64`, the validation of `mMetadata` is implemented that verifies type and version of the metadata extension. If the version is not `1` or the type is not "Hot Archive", a runtime error is thrown.

However, this validation is conditioned by `constexpr (std::is_same_v<BucketT, HotArchiveBucketEntry>)`. The type predicate contains a typo, so the actual condition can never be satisfied. Type `BucketT` can only be either `HotArchiveBucket` or `LiveBucket`. It cannot be `HotArchiveBucketEntry`.

As a consequence, the validation does not ever execute.

Recommendation

We recommend correcting the condition and comparing the type `BucketT` to the `HotArchiveBucket` type.

Status: Resolved

7. Infinite loop in database initialization due to missing cursor advancement

Severity: Minor

In `src/database/Database.cpp:461-469`, the function `Database::initialize` executes `PRAGMA database_list` to locate the main database. The query is run with `st.execute(true)`, which already retrieves the first row.

However, the execution then enters a loop while `st.got_data` is `true`, but the loop body never advances the cursor. If the first row returned is not the main database, the condition remains `true` indefinitely because no call to `st.fetch` is made.

This results in an infinite loop that consumes a CPU core and prevents initialization from completing.

Recommendation

We recommend advancing the result set on each iteration by calling `st.fetch` within the loop.

Status: Resolved

8. Incorrect global operator `new` and `new[]` override under `USE_TRACY_MEMORY_TRACKING`

Severity: Minor

In `src/main/main.cpp:289-290,305-306`, the global operators `new` and `new[]` are overridden when `USE_TRACY_MEMORY_TRACKING` is enabled without ASAN. These implementations delegate to `malloc` and return its result directly without validating the pointer.

According to the C++ standard, the non-nothrow versions of operator `new` and `new[]` must invoke the new-handler and ultimately throw `std::bad_alloc` if memory allocation fails. Returning a `null` pointer is explicitly undefined behavior. The current implementation violates this requirement: if `malloc` returns `null`, the function simply propagates the `null` pointer, leading to undefined program behavior under memory exhaustion.

Although the codebase installs a new-handler using `std::set_new_handler(outOfMemory)` in line `325`, the custom operator `new` overrides bypass this mechanism, as they do not invoke the handler when allocation fails. The consequence is that, in out-of-memory conditions, the application may crash unpredictably, corrupt memory, or exhibit erratic behavior instead of failing safely.

Recommendation

We recommend updating the custom operator `new` and `new[]` implementations to validate the return value from `malloc`. If a `null` pointer is returned, they should either call the configured new-handler or throw `std::bad_alloc` directly. This ensures standard-compliant behavior, predictable error handling, and improved robustness under memory exhaustion.

Status: Resolved

9. Argument corruption in subprocess execution

Severity: Minor

In `src/herder/RustQuorumCheckerAdaptor.cpp:495-502`, `RustQuorumCheckerAdaptor` constructs subprocess commands as a single flat string and relies on naive whitespace-based tokenization.

This approach does not respect quoting or escaping rules, leading to corrupted arguments and platform-specific security risks. Executable paths, such as the current binary or temporary file paths, are concatenated into the command string and passed to `ProcessManager::runProcess`.

On POSIX systems, the command string is split using a regular expression, and quotes are discarded before invoking `posix_spawnp`, which causes arguments containing spaces to be broken into multiple entries.

On Windows, the code calls `CreateProcess` with `lpApplicationName` set to `nullptr`, passing only the raw string. This behavior relies on ambiguous system parsing and exposes a known unquoted path hijack vector.

As a result, paths such as `C:\Program Files\stellar-core\file.exe` may be incorrectly split into multiple arguments, leading to the wrong command or malformed parameters being executed. Even when configuration values are quoted, the splitting logic discards the quotes before process creation, rendering them ineffective.

Recommendation

We recommend avoiding flat string command construction and instead passing arguments as a structured vector to process creation functions.

On POSIX systems, each argument should be explicitly defined in the `argv` array to preserve spaces and special characters correctly.

On Windows, `CreateProcess` should be invoked with a properly quoted executable path in `lpApplicationName`, while arguments are separately constructed to avoid unquoted path hijack risks.

Status: Acknowledged

The client acknowledges the issue stating that this behavior is mostly intentional.

Specifically, command strings provided by operators are subject to simple, naive space-splitting before being passed to `posix_spawnp`, making such splitting necessary and acceptable. The client further clarifies that this approach does not introduce new risks when delegating to Windows command-line space-splitting in the Windows environment.

Additionally, user-controlled data never reaches this stage; if it did, the resulting concerns would be significantly more severe than space-splitting itself.

10. Unauthenticated remote access via public HTTP command interface

Severity: Minor

In `src/main/CommandHandler.cpp:53`, when `PUBLIC_HTTP_PORT` is enabled, the HTTP command handler exposes sensitive administrative endpoints over the network without authentication or authorization, effectively creating an unauthenticated remote API. By default, the server binds to `127.0.0.1`, restricting access to local clients.

However, when `PUBLIC_HTTP_PORT` is set to `true`, the logic in switches the binding to `0.0.0.0`. Route registration through `addRoute` and `safeRouter` occurs without additional access control checks.

This configuration exposes a range of critical endpoints, including operational commands, system and log management, and ledger or transaction operations.

A remote attacker with network access could leverage this exposure to modify log configurations, rotate logs, or tamper with logging output, drop or ban peers to disrupt connectivity, manually close ledgers in standalone mode, inject arbitrary transactions into the local queue or trigger operational and maintenance tasks.

Recommendation

We recommend protecting remote administration commands by authentication and authorization mechanisms, such as TLS with client certificates or token-based access control.

Status: Acknowledged

The client acknowledges the issue stating that the observation is correct; however, in production systems this configuration is never enabled.

Furthermore, the documentation and example configurations explicitly warn operators against using it. The client notes that, given these safeguards, there is insufficient motivation to address the issue at this time.

11. Out of memory risk from unvalidated XDR records

Severity: Minor

In `src/util/XDRStream.h`, both `XDRInputFileStream::readOne` and `XDRInputFileStream::readPage` rely on the 4-byte length prefix of an XDR record and immediately resize their internal buffer to the claimed size without validation.

This permits a malicious or corrupted history archive to present a deceptively small `.xdr.gz` file that decompresses into an XDR stream advertising an extremely large record size, such as `0x7fffffff` (approximately 2 GB).

During catchup or verification, nodes download and decompress these files and invoke `readOne` to parse ledger headers, transactions, or results. Without size checks, `std::vector::resize(sz)` attempts to allocate the advertised memory. Because most call sites do not handle `std::bad_alloc`, this results in process crashes due to out-of-memory conditions or termination by the operating system. The failure recurs on every restart until the affected checkpoint is skipped or the archive is corrected. An attacker with control of, or the ability to spoof, a configured history archive can use this flaw to cause every syncing node to crash at the targeted checkpoint.

A similar issue exists in `readPage`. Although it enforces a configured page size, the buffer still expands beyond this limit if a record claims to cross the page boundary, enabling corrupted bucket files to trigger excessive memory use during disk-index scans.

Recommendation

We recommend validating the advertised XDR record size before resizing buffers. Enforce a strict upper bound based on the maximum expected size for the data type and fail fast when the prefix exceeds this threshold.

Status: Acknowledged

The client acknowledges the issue stating that History Archive Sources are trusted and explicitly selected by validators. In the event that an archive behaves maliciously, the system logs the source of the download, allowing operators to identify and remove the offending archive from the trusted set if it causes crashes.

The client emphasizes that this does not represent a significant risk, since a crashed node in this scenario is not part of the active network (as it is still in the process of catching up) and will simply crash and retry with another archive chosen at random. The client further notes

that, even if this issue were addressed, other potential out-of-memory attacks (such as zip bombs) remain possible.

12. Denial of Service via unbounded stop survey signature verification

Severity: Minor

In `src/overlay/SurveyManager.cpp`, handling of `TIME_SLICED_SURVEY_STOP_COLLECTING` performs only a minimal ledger-window check before immediately invoking an expensive Ed25519 signature verification.

However, there is no per-surveyor or per-ledger rate limiting applied on this path. In `validateStopSurveyCollecting`, the sole precondition is that the ledger number falls within a small window; `onSuccessValidation` then verifies the signature for every packet, even if no active survey exists or the nonce is invalid.

Because the permission check (`surveyorPermitted`) is evaluated before signature verification, any connected peer can spoof a permitted `surveyorID` and still trigger a signature verification that predictably fails.

This allows an attacker to generate a stream of unique `Stop` messages that bypass deduplication and caching, forcing repeated operations. The attacker's effort remains negligible while the receiver's CPU can be steadily consumed, enabling a low-cost denial-of-service attack.

Recommendation

We recommend introducing pre-signature rate limiting for `Stop` survey messages to ensure that spoofed or invalid packets cannot trigger unbounded signature verification.

Status: Acknowledged

The client acknowledges the issue stating that peers transmitting data with invalid signatures are dropped immediately. As a result, an attacker would need to continuously reconnect in order to sustain the attack. The client highlights that re-connection itself serves as a throttling mechanism, since it is inherently slow, requires a full handshake, and may not succeed if the target node has already exhausted its available connection slots.

13. Insufficient error handling causes node termination during bucket state recovery

Severity: Minor

In `src/ledger/LedgerManagerImpl.cpp:566-574`, the `loadLastKnownLedger` function handles failures of the `AssumeStateWork` operation during node startup by calling

`releaseAssertOrThrow(mApp.isStopping())`, which terminates the node process if the application is not in a stopping state.

However, the current implementation assumes that `AssumeStateWork` can only fail during graceful shutdown scenarios, as indicated by the inline audit comment acknowledging this limitation. The code does not account for legitimate failure conditions that may occur during crash recovery, such as corrupted bucket files, insufficient disk space, or memory pressure. When `AssumeStateWork` fails for any reason other than graceful shutdown, the node terminates immediately rather than attempting recovery procedures.

Consequently, nodes experiencing bucket state corruption or resource constraints during startup become unable to rejoin the network without manual intervention. Multiple validators experiencing similar failures simultaneously could reduce network consensus participation and impact overall network availability. The lack of graceful degradation mechanisms means that transient issues such as temporary disk I/O errors or memory pressure can cause permanent node exclusion from consensus operations.

Recommendation

We recommend implementing proper error handling for `AssumeStateWork` failures by adding conditional logic to distinguish between graceful shutdown and unexpected failure scenarios, implementing recovery strategies such as bucket file validation, archive re-download, and fallback to the last known good checkpoint before terminating the node process.

Status: Acknowledged

The client acknowledges the issue stating that this is intended behavior. A node that cannot assume state or has corrupted state poses a risk of causing an unrecoverable fork if it were to join the network. For this reason, the system is designed to crash and require manual intervention, which serves as a deliberate safety measure.

14. Missing thread assertion on main-thread-only operations

Severity: Minor

Several methods in `src/main/AppConnector.cpp` and `src/main/PersistentState.cpp` are designed or documented as main-thread-only operations but lack runtime thread safety assertions, creating inconsistent enforcement of threading contracts and increasing the risk of race conditions.

In `src/main/AppConnector.cpp:67-70`, the method `getSorobanMetrics` is documented in the header (line `40`) as main-thread-only but does not enforce this with a `releaseAssert(threadIsMain())` check. The method accesses `LedgerManager::getSorobanMetrics`, which likely maintains non-thread-safe state and may be corrupted if accessed concurrently.

In `src/main/PersistentState.cpp`, multiple database operations are missing thread assertions:

- Lines `193–196`: `getState` performs database reads without validation.
- Lines `200–204`: `setState` performs database writes without validation.
- Lines `289–290`: `updateDb` executes SQL updates without validation.
- Lines `402–403`: `getFromDb` executes SQL queries without validation.

These functions directly access `mApp.getDatabase` and issue SQL statements, which typically require serialized access to preserve transaction consistency and prevent connection pool conflicts. The database layer appears to assume main-thread execution, but the absence of explicit guards leaves the contract unenforced at runtime.

This inconsistency becomes clear when compared to methods such as `shouldYield` in the same file (line `132`), which correctly enforces main-thread execution with `releaseAssert(threadIsMain())`.

Recommendation

We recommend adding explicit thread assertions.

Status: Resolved

15. Double unlock risk from implicitly copyable lock guards

Severity: Minor

In `src/util/ThreadAnnotations.h:160–163,218–221`, the classes `MutexLockerT` and `SharedLockShared` each define a reference data member (`MutexType& mut` and `SharedMutex& mut`, respectively).

However, they do not declare copy or move constructors or assignment operators. As a result, the compiler implicitly generates these functions, allowing the lock guard objects to be copied.

When such RAII lock guards are copied, whether by pass-by-value, return-by-value, or insertion into containers, multiple guard objects reference the same mutex. Both the original and the copy invoke `Unlock` in their destructors, causing a double-unlock condition that results in undefined behavior. This can manifest as process crashes, memory corruption, or unpredictable data races. The impact is significant, as these lock guards are widely used across cryptographic operations, transaction processing, and concurrent data access. Although the codebase generally uses them correctly in local scope, accidental copying in function calls, lambda captures, or container operations remains a realistic risk.

Recommendation

We recommend explicitly deleting the copy and move constructors and assignment operators for both `MutexLockerT` and `SharedLockShared`. Their destructors should also be declared `noexcept` to enforce correct RAII semantics. These changes prevent accidental copying, eliminate the possibility of double-unlock scenarios, and make the intended threading contracts explicit.

Status: Resolved

16. Incorrect optimistic patching of copied ledger entries

Severity: Minor

In `src/transactions/ParallelApplyUtils.cpp`, the functions `ThreadParallelApplyLedgerState::setEffectsDeltaFromSuccessfulOp` and `setLedgerChangesFromSuccessfulOp` are responsible for recording the effects delta and ledger changes after successful transaction execution. These functions create copies of modified ledger entries, which are later used for invariant validation.

However, both functions modify the copied entries by updating their `lastModifiedLedgerSeq` field after the copy is made. This follows the assumption that `LedgerTxn` will update this field for persistent changes.

Consequently, any code operating on the copied state before this patching step may observe inconsistent metadata. For example, old copies of updated entries created for historical comparison will reflect stale `lastModifiedLedgerSeq` values. This inconsistency also applies to event metadata, which may capture a partially updated state.

As a result, future invariant checks may fail erroneously due to operating on deltas and events that are only partially patched, undermining the reliability of execution validation.

Recommendation

We recommend updating the `lastModifiedLedgerSeq` field on ledger entries immediately as they are modified in the ledger state. This ensures that all components (including invariant checks, effects delta tracking, and event recording) operate on a consistent and accurate view of state changes.

Applying metadata updates early in the modification lifecycle eliminates timing discrepancies and prevents downstream logic from observing stale or incomplete entry state.

Status: Resolved

17. Missing concurrency assertions may allow unsafe access to ledger state during application

Severity: Minor

Several functions in `src/ledger/LedgerManagerImpl.cpp` are accessible from multiple threads and require concurrency safeguards. The `mCurrentlyApplying` flag, accessed via `isApplying`, is used to track whether the ledger is actively being modified.

However, some functions lack explicit assertions to enforce that they are not invoked during concurrent ledger application.

In particular, functions such as `setLastClosedLedger`, `loadLastKnownLedger`, and `compileAllContractsInLedger` can safely include a `releaseAssert(!mCurrentlyApplying)` to prevent unsafe access during critical sections.

Without such assertions, these functions may be called in invalid contexts, leading to race conditions, state corruption, or subtle concurrency bugs.

Recommendation

We recommend adding `releaseAssert(!mCurrentlyApplying)` to all functions that must not be invoked during ledger application.

Status: Resolved

18. Archive decompression enabling Denial of Service

Severity: Minor

In `src/historywork/GetAndUnzipRemoteFileWork.cpp`, the `GetAndUnzipRemoteFileWork` function fetches and decompresses remote history files (e.g., buckets, ledgers) without enforcing strict size constraints.

During the retrieval phase, `GetRemoteFileWork::getCommand` issues shell commands such as `curl -sf -o` to download files without validating the size or `Content-Length` header. Subsequently, `GunzipFileWork` decompresses the archive using `gzip -d` without restrictions on the inflated file size or the time required for decompression.

However, validation of the archive contents via `VerifyBucketWork::onRun` occurs only after decompression, at which point a significant amount of disk space and CPU resources may already have been consumed.

An attacker can exploit this by serving a malicious `.gz` archive using DEFLATE compression, which decompresses into a disproportionately large file, potentially tens or hundreds of

gigabytes. Repeating this operation can deplete node resources persistently, keeping validator nodes in a resource-starved state.

We are reporting this with minor severity since archive providers are expected to be trusted.

Recommendation

We recommend implementing strict validation checks before and during the download and decompression stages. This includes enforcing maximum file size limits, validating the `Content-Length` header, and applying decompression timeouts.

Status: Acknowledged

The client acknowledges the issue stating that in the Stellar security model, History Archive providers are trusted entities chosen by node operators. If a trusted quorum member behaves maliciously, the expected behavior is for the node to crash and attribute fault to the byzantine actor. This is achieved through an info-level log in `GetRemoteFileWork.cpp:41`, ensuring that if a node were to run out of memory and crash, the logs would clearly indicate which quorum member was responsible. This allows operators to identify the malicious participant and take appropriate corrective action.

19. Unconditional drift guard in ledger application causes unnecessary throttling in sequential catch-up mode

Severity: Minor

In the `LedgerApplyManagerImpl::tryApplySyncingLedgers` function, defined in `src/catchup/LedgerApplyManagerImpl.cpp:516-523`, a design inconsistency exists between the intended behavior and the implemented logic of the `MAX_EXTERNALIZE_LEDGER_APPLY_DRIFT` constraint.

According to documentation in `src/catchup/LedgerApplyManagerImpl.h:18-24`, the 12-ledger drift limit should apply only when `Config.parallelLedgerClose` returns `true`. However, in practice, the drift guard is evaluated unconditionally, prior to checking the parallel mode flag.

This results in unintended throttling even when `parallelLedgerClose` is false (i.e., sequential mode). In such cases, the method restricts application to just `12` buffered ledgers per invocation, despite using a static `lcl` (last closed ledger) reference that does not update during the loop.

Although repeated calls to `tryApplySyncingLedgers` eventually process the backlog, the unnecessary drift guard imposes redundant round-trips and degrades performance, particularly during high-volume catch-up operations.

Recommendation

We recommend refactoring the logic to conditionally apply the drift constraint only when `parallelLedgerClose` is enabled, as originally intended. Additionally, consider updating the `lcl` reference dynamically within the apply loop to ensure correct drift accounting, should the guard be retained for sequential mode in the future.

Status: Acknowledged

The client acknowledges the issue stating that while this is a valid observation, the impact is considered minor since, in practice, nodes rarely buffer more than 12 ledgers. The client confirms that a fix will be implemented in the next stability release.

20. Static transaction fees fail to reflect increased index scan cost caused by larger page sizes

Severity: Minor

In `src/bucket/DiskIndex.cpp:59-86`, the `DiskIndex::scan` function performs a lookup using a combination of in-memory index data and a bloom filter. The performance of this scan is heavily influenced by the `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT` setting, which controls the structure and size of the range index.

When `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT = 0`, the system uses a full in-memory index, enabling fast O(1) lookups at the cost of high memory usage. When page size is increased, memory usage drops, but lookup performance degrades due to additional iteration and potentially less cache locality.

Despite this tradeoff, the transaction fee model applies a static cost (`FEE_READ_LEDGER_ENTRY`, `FEE_READ_1KB`), making no distinction between fast in-memory lookups and slower paged scans. This disconnect means that transactions performing high-cost reads under large page settings are underpriced relative to their actual runtime footprint.

While a performance monitoring metric (Counters) is already in place to help track these effects, the fee model remains agnostic, leading to fee/performance misalignment.

Recommendation

We recommend incorporating index mode awareness into the fee model. Potential approaches:

1. Differentiate fees for `HotArchiveBucketIndex`, `LiveBucketIndex` with paging, and `LiveBucketIndex` with full in-memory index.

2. Alternatively, dynamically account for scan latency or bloom filter misses and translate that into fee multipliers.

Status: Acknowledged

The client acknowledges the issue stating that this seems unnecessarily complex compared to the potential benefit; it's also hard to abuse the slight read time differences due to caching.

21. Incorrect equality implementation for type `MergeCounters`

Severity: Minor

In `src/bucket/BucketUtils.cpp:94-134`, the `operator==` is implemented for the `MergeCounters` type. In the same file, `operator+=` is implemented as well. Although implementations of the two functions have to be consistent with each other, the `operator==` does not compare two fields:

- `mPreShadowRemovalProtocolMerges`
- `mPostShadowRemovalProtocolMerges`

As a consequence, equalities involving `MergeCounters` are not precise. There are multiple usages of the `operator==` throughout the codebase. Incorrect implementation of the equality relation can cause issues when using set-like data structures.

Recommendation

We recommend completing the `operator==` implementation by comparing values of the `mPreShadowRemovalProtocolMerges` and `mPostShadowRemovalProtocolMerges` fields.

Status: Resolved

22. Redundant filesystem operations decrease performance

Severity: Minor

In `src/bucket/BucketBase.cpp:120-134`, the `randomFileName` function is defined. This function, parameterized by string values `tmpDir` and `ext`:

1. Generates 8 random bytes.
2. Composes filepath that includes `tmpDir`, `ext` and the generated bytes.
3. Checks existence of the filepath by accessing the filesystem.
4. Starts over with step 1, if the filepath is already occupied.

While the random filepath generation is secure and fast enough, filesystem operations can cause bottleneck in the performance of the overall system since the `randomFileName` function is used in multiple index operations:

- `LiveBucket::fresh`
- `LiveBucket::mergeInMemory`
- `BucketBase<BucketT, IndexT>::merge`
- `BucketManager::mergeBuckets`
- `DiskIndex<BucketT>::DiskIndex` constructor
- `BucketLevel<LiveBucket>::prepareFirstLevel`

All of these and other operations, calling the `randomFileName` function, block I/O thread and wait until the filesystem operation completes although they do not write any actual data on disk. This can significantly reduce performance of the database.

Recommendation

We recommend removing the filesystem operation from the `randomFileName` function. The chance of filepath collision is low enough even when using only 8 random bytes, but it is feasible to reduce collision rate even further by using more random bytes, a cryptographic hash function or UUID. Alternatively, lazy name generation can be implemented that is called immediately before the actual data write.

Status: Acknowledged

The client acknowledges the issue stating that this micro optimization does not seem worth the added complexity.

23. Incorrect unit conversion may underestimate bucket size

Severity: Minor

In `src/bucket/LiveBucketIndex.cpp:32-36`, the expression interprets `BUCKETLIST_DB_INDEX_CUTOFF` using decimal megabytes (`1'000'000`) rather than binary (`1024 * 1024`), which may cause subtle misclassifications near threshold values. As a result, `bucketSize` might fall below the cutoff unexpectedly, causing the function to return a page size of zero even when the bucket is non-trivially large.

This misalignment can lead to skipped index generation or inconsistencies in performance characteristics for near-cutoff bucket sizes, undermining operator expectations and tuning strategies.

Recommendation

We recommend correcting the unit convention. If binary MB is intended, replace the multiplier with `1024 * 1024`.

Status: Resolved

24. Missing transaction data in history archives causes catchup failure without proper error handling

Severity: Minor

In `src/catchup/ApplyCheckpointWork.cpp:133-162`, the `getCurrentTxSet` function attempts to read transaction history entries from an XDR file. When `getHistoryEntryForLedger` cannot find a transaction set for a given ledger sequence in the history file, the function silently falls back to creating an empty transaction set.

This empty transaction set is then used in `getNextLedgerCloseData`, where its hash is verified against the ledger header's expected `txSetHash`. This verification will fail for any ledger that actually contained transactions, causing the entire catchup process to abort with an exception.

An attacker who controls or corrupts a history archive can exploit this behavior to create a denial-of-service condition. By providing transaction history files with systematically missing entries, they can prevent nodes from successfully completing catchup. The attack is particularly effective because the failure occurs late in the apply phase after significant resources have been consumed downloading and processing files.

Additionally, the error handling does not attempt to retry with alternative history archives at this stage.

Recommendation

We recommend modifying `getCurrentTxSet` to distinguish between legitimately empty transaction sets and missing data. The function should explicitly track which ledger sequences are expected to be present in the history file based on the checkpoint range. Additionally, throw a specific exception when required transaction data is missing, allowing the catchup process to fail fast and potentially retry with an alternative archive.

Status: Acknowledged

The client acknowledges the issue stating that History Archive Providers are trusted entities chosen by node operators. If they are malicious and catchup cannot be completed, the node will crash. On startup, another archive will be chosen at random. We log the archive we are fetching from during catchup such that malicious actors can be blamed via logs. Given that

archives are assumed to be honest and this only affects new nodes joining the network, not nodes already in sync, no action is needed.

25. Missing exception handling for corrupted index files

Severity: Minor

In `src/bucket/BucketIndexUtils.cpp:58`, the `loadIndex` function opens and deserializes index files without any exception handling.

A corrupted or malicious index file could trigger unhandled exceptions during deserialization, causing immediate process termination. This creates a denial-of-service vulnerability where a single bad index file prevents the node from starting.

Recommendation

We recommend wrapping the entire deserialization process in a try-catch block and return an appropriate error indicator when deserialization fails.

Status: Resolved

26. Current flat fee structure fails to reflect the variable cost of bucket list searches

Severity: Minor

In `src/bucket/BucketListSnapshotBase.cpp:85-129`, the `SearchableBucketListSnapshotBase<BucketT>::load` function performs a linear search over the bucket list to locate a given `LedgerKey`. This function is invoked frequently via `LedgerTxnRoot::Impl::getNewestVersion` during transaction execution.

The computational cost of this search is directly influenced by the number and arrangement of buckets, yet the current fee model applies a static charge using `FEE_READ_LEDGER_ENTRY = 5,000` and `FEE_READ_1KB = 1,000`, regardless of how many buckets are scanned. This flat fee structure fails to capture the actual work done, leading to an inaccurate and potentially exploitable pricing model for read-heavy transactions.

Recommendation

We recommend adopting a dynamic fee model that incorporates the search effort involved in reading from the bucket list. This can be informed by the number of buckets scanned per load call.

Status: Acknowledged

The client acknowledges the issue stating that this seems unnecessarily complex compared to the potential benefit; it's also hard to abuse the slight read time differences due to caching.

27. Catchup Denial of Service due to incomplete fallback for oversized buckets

Severity: Minor

In `src/historywork/VerifyBucketWork.cpp:58-73`, catchup verification fails fast when a bucket exceeds the allowed `MAX_HISTORY_ARCHIVE_BUCKET_SIZE`. This ensures catchup nodes avoid processing anomalously large buckets.

However, publishing logic is more lenient - oversized buckets are logged with a `FATAL` error but still accepted to avoid stalling the network (`src/history/HistoryArchive.cpp:549-564`). This leads to asymmetric behavior - a bucket deemed valid for publishing might later be rejected by catchup nodes.

The intended mitigation is a manual fallback, where nodes restart catchup from a ledger before the oversized bucket was introduced (`src/history/HistoryArchive.h:70-81`). But this fallback is not automatic, leaving room for operator error and degraded UX during recovery from oversized state.

Recommendation

We recommend aligning behavior or adding explicit automated fallback in the catchup pipeline to handle such edge cases gracefully.

Status: Acknowledged

The client acknowledges the issue stating that honest nodes should not generate Buckets larger than the limit (with a warning to indicate if this occurs, as this would be a bug). Given that the bug is not fatal to the liveness of the network (just new nodes joining the network), we've elected to warn on this error instead of halting the network. The fail fast behavior on catchup is to protect against malicious archives.

28. Eviction scan and search logic may delay ledger close due to blocking call

Severity: Minor

In `src/bucket/BucketManager.cpp:1108-1131`, the `resolveBackgroundEvictionScan` function synchronously waits for the result of a background eviction scan using `mEvictionFuture.get()`. This is called during every

ledger application via `LedgerManagerImpl::sealLedgerTxnAndTransferEntriesToBucketList`.

Although the eviction scan is initiated proactively (scan for block N+1 starts during block N), the synchronous `get()` block if the scan is not complete by the time it is needed, risking delays in ledger close time.

Additionally, in `src/bucket/BucketSnapshot.cpp:236`, the scan processing logic invokes `bl.loadKeys(keysToSearch, "eviction")` to retrieve potentially large sets of entries from the bucket list. This search is unmetered and lacks explicit bounds on processing effort or data volume.

While the system includes a bound on the eviction scan size (to limit scan scope), this does not fully mitigate the issue. As noted in a related issue, the eviction size bound limits the data volume but does not guarantee timely scan completion or responsiveness, especially under constrained I/O or CPU conditions.

Additionally, if configuration changes during the scan, a new scan is initiated and the function blocks again, further increasing the chance of delay.

NOTE: The client notes that the scan size is bounded by network configuration parameters, which are conservative: scans typically complete within a few hundred milliseconds and are allowed up to 5 seconds. This mitigates the risk under normal conditions.

However, residual risks remain:

- The `SCAN_SIZE` parameter alone does not guarantee timely completion under constrained I/O or CPU conditions.
- In `src/bucket/BucketSnapshot.cpp:236`, the call to `bl.loadKeys(keysToSearch, "eviction")` remains unmetered, potentially involving large data volumes without explicit effort bounding.
- If configuration changes occur during a scan, a new scan is initiated, which can further increase latency.

Recommendation

Although the risk is reduced by conservative network configuration, the blocking behavior `(mEvictionFuture.get())` and unmetered search `(bl.loadKeys)` introduce residual latency risks. We recommend explicitly bounding search operations and considering mechanisms to skip or defer eviction if results are unavailable within a defined latency window.

Status: Acknowledged

The client acknowledges the issue stating that the scan is limited by network config. The network config is very conservative, as it usually completes in a few hundred MS, but has 5 seconds to complete.

29. Potential performance degradation due to unbounded growth of `assetToPoolID` map

Severity: Minor

At `src/bucket/DiskIndex.cpp:217-221`, the logic appends liquidity pool IDs to `assetToPoolID`, a `std::map<Asset, std::vector<PoolID>>`, under the assumption that the number of pools per asset remains reasonably small. However, over time, the vector associated with a frequently used or adversarially targeted asset can grow indefinitely.

NOTE: *The client notes that this map is part of BucketIndex, which is relatively short-lived and immutable. Bucket indexes are routinely garbage collected and recreated during bucket merges. Since the database layer is already de-duplicated, there is no need for filtering or deduplication when populating the map. The only way to increase the size of this vector is by creating more pools, which is considered standard network behavior.*

While growth of this vector is indeed expected as pools increase, the concern remains that under extreme or adversarial conditions, a single hot asset could accumulate an unusually large number of pools. In such cases, the associated vector may grow disproportionately, introducing localized memory pressure and increasing the cost of bucket merges. This does not impact correctness but does introduce a performance and availability risk that operators should remain aware of.

Recommendation

We recommend considering introducing size limits, eviction policies, or deduplication.

Status: Acknowledged

The client acknowledges the issue stating that this is actually bounded by the fact that a single operation can only cross 1000 offers, and at most 1000 operations can make it into a single ledger. The cache should also be cleared after each ledger.

30. Implicit disabling of explicitly enabled in-memory indexing

Severity: Minor

In `src/bucket/HotArchiveBucketIndex.cpp:31-42`, the `getPageSize` function is defined that receives the `cfg` parameter of `Config` type and the `bucketSize` parameter. While the `bucketSize` is not used, the `cfg` config is used to retrieve the actual page size.

This is achieved using the `getPageSizeFromConfig` function, defined in `src/bucket/BucketIndexUtils.cpp:19-28`. This function handles the config in generic manner, and returns `0` if the configuration value `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT` is set to `0`. In case when the `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT` is not set, it defaults to `14` as it is seen

from `src/main/Config.cpp:172`. This means that the `getPageSizeFromConfig` function returns `0` only when the exponent is explicitly set to `0` by the operator.

However, the `HotArchiveBucketIndex::getPageSize` function overrides the `0` page size with the hard-coded page size of 16 kilobytes. The commentary explains this as "HotArchive does not support individual indexes, use default 16 KB value even if config is set to 0".

This implicit overriding of even explicitly provided value `0` can easily be missed by the operator, complicate deployment or pass unnoticed and result in suboptimal node performance. If the operator sets `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT` to `0`, the "in-memory indexing" is expected to be turned on.

Recommendation

We recommend implementing several enhancements to improve the deployment experience:

1. Terminate the node early if the explicitly selected "in-memory indexing" mode cannot be turned due to the specific index implementation selected.
2. Implement dedicated configuration value to enable or disable "in-memory indexing".
3. Remove implicit enabling of the "in-memory indexing" and not derive it from the page size or exponent.

Status: Acknowledged

The client acknowledges the issue stating that it's an explicit design decision to not support in-memory index of Hot Archive Buckets. Page size of 0 is not recommended in production and is really just a testing mode anyway.

However, we believe that this is not an explicit design, as the expected approach should be not allowing the operator to set `BUCKETLIST_DB_INDEX_PAGE_SIZE_EXPONENT` for the hot archive.

31. Eviction does not function on buckets larger than threshold

Severity: Minor

At `src/bucket/LiveBucketList.cpp:157-174`, the `period * scanSize < b->getSize()` check is intended to detect whether a bucket can be fully scanned before it is updated (spilled). When a spill occurs, `bucketFileOffset` resets to zero, restarting the eviction scan on the same bucket file (the eviction iterator remains stuck on the same bucket level and file). For large bucket files at higher levels, this threshold can silently fail, causing the eviction scan to never complete because each spill restarts scanning from the beginning of the file, leading to perpetual rescanning of the large bucket files.

As a result, the eviction iterator never resets to the start (level 0) to evict later-created TTL entries. Consequently, newly created TTL-based (evictable) entries accumulate in the live

bucket because the eviction iterator is stuck on the last bucket file. This causes continuous resource waste, persistent buildup of un-evicted state, and prevents eviction from progressing to more recent data.

Let's break down the math:

```
At level 10, the update period is:

period = 2^(2 * 10 - 1) = 2^19 = 524,288 ledgers

Given scanSize = 100 KB
total scan throughput per update period is:
total_scanned = period × scanSize = 524,288 × 100KB ≈ 53 GB
```

If the bucket size exceeds this threshold (53GB), the eviction scan cannot complete, causing repeated rescanning and state buildup.

Another reason could be misconfiguration of `scanSize` bytes.

Recommendation

We recommend :

1. Dynamically adjusting `scanSize` based on bucket size or level.
2. Exploring resumable eviction checkpoints to avoid full restarts.

Status: Acknowledged

The client acknowledges the issue stating that this is a known limitation of the eviction scan. We have a warning mechanism for this case via `LiveBucketList::checkIfEvictionScanIsStuck`, where the expected remedy is to increase the eviction scan size.

32. Compiler exception during startup causes crash loop

Severity: Minor

In `stellar-core:src/ledger/SharedModuleCacheCompiler.cpp:102-113`, the `popAndCompileWasm` function attempts to compile next WASM blob retrieved from the `mWasms` deque, by calling `mModuleCache->compile(ledgerVersion, slice)` on line `104`. In case of failure, the error is being logged and re-thrown on line `112`.

The re-thrown exception is not caught by the thread entry `SharedModuleCacheCompiler::start` function, therefore the C++ runtime calls `std::terminate`. In `stellar-core:main.cpp:336`, Stellar sets a global terminate-handler that simply aborts the whole process using the `printBacktraceAndAbort` function.

The `SharedModuleCacheCompiler::start` function is called during node startup. Upon restart, it spawns a loader thread that scans the bucket-list and enqueues every unique `ContractCodeEntry` for compilation in `stellar-core:src/ledger/SharedModuleCacheCompiler.cpp:136-191`. Stellar node will attempt to compile the same WASM blob again without potential to succeed. The process will be aborted again.

A single smart contract failing to compile can cause network-wide DoS once it is uploaded to the network. Transaction validation for contract uploads calls the `InvokeHostFunctionOpFrame` function defined in `stellar-core:src/transactions/InvokeHostFunctionOpFrame.cpp:1165`. The validation step only checks the size limit and does not attempt to parse or pre-compile WASM blobs. This means that an attempt to compile the contract occurs only after the transaction is considered valid and added to the ledger.

Since the transaction with the malicious contract is in the ledger, the only remediation possible for this issue is via a network-wide upgrade that disables the malicious contract. This would require development and coordination efforts. It is not enough to simply erase the node cache and restart it.

While the impact is critical, and multiple potential panics have been observed in the WASM compiler, no specific vector has been identified yet as exploitable. In general, any contract causing a compiler exception will trigger the issue.

Recommendation

We recommend ensuring that an uploaded contract compiles, before including the corresponding transaction into the ledger. Additionally, make sure that the WASM compiler cannot panic or mark non-compilable contracts as invalid without stalling the node.

Status: Acknowledged

The client acknowledges the issue stating that this is not an issue, only valid (as in 'compile-able') contracts can be actually uploaded to the network. The check happens during the transaction application, not validation (because it is expensive and should be metered).

However we believe that while it is mostly an error handling issue, there is still the nuance that if the compilation fails after initial compilation (for example, after compiler upgrade although they separate versions) then the failing contract needs to be removed by the operator.

33. Result of verification is not stored reliably

Severity: Minor

In `stellar-core:src/historywork/WriteVerifiedCheckpointHashesWork.cpp`

:309, the dir parameter of the fs::durableRename function is specified incorrectly. The value used is mOutputPath.relative_path().string, while mOutputPath.parent_path().string must be used.

The dir parameter of the fs::durableRename function is used to call fsync on the directory containing the file that is being renamed. This is required to sync the directory metadata, so the renamed file can actually be discovered.

In rare events like power outages or the process being killed, the file will effectively disappear and the node will have to regenerate it. In the regular scenario, the file will be inaccessible for longer than necessary.

As a consequence, the result of the WriteVerifiedCheckpointHashesWork::endOutputFile function is not stored reliably or cannot be observed externally for some time.

Recommendation

We recommend using the correct value for the dir parameter of the fs::durableRename function.

Status: Resolved

34. Prolonged non-productive shutdown time

Severity: Minor

In stellar-core:src/ledger/SharedModuleCacheCompiler.cpp:33-39, the ~SharedModuleCacheCompiler destructor is defined, which calls the join method on every thread from the mThreads pool.

The mThreads pool contains threads that pull every live ContractCodeEntry from the bucket-list and compile the contained WASM modules in order to store them in the reusable module cache.

If the node begins shutdown, for instance when handling SIGINT or SIGTERM, while a large compilation batch is in progress, the main thread will call the ~SharedModuleCacheCompiler destructor that blocks until all modules finish compiling.

As a consequence, node shutdown takes longer than necessary. More importantly, while waiting for the destructor to complete, the node has already stopped listening to HTTP traffic and is de-facto offline.

Recommendation

We recommend aborting running threads instead of waiting for their completion.

Status: Acknowledged

The client acknowledges the issue stating that it's a few hundred milliseconds and only if we shut the process down mid-compile which is very rare.

35. Premature update of `mContractsCompiled`

Severity: Minor

At `stellar-core:rc/ledger/SharedModuleCacheCompiler.cpp:70`, the `mContractsCompiled` is updated immediately after loading contracts, but compilation may not have completed yet.

This means the counter reflects all seen contracts rather than the number of successfully compiled contracts.

Recommendation

We recommend updating `mContractsCompiled` only after compilation finishes, and ensure it reflects successfully compiled contracts rather than total seen entries.

Status: Acknowledged

The client acknowledges the issue stating that it's never observed either by printing or metric publication until compilation is finished.

36. Silent filtering of host function names may break linker

Severity: Minor

At `soroban:soroban-env-host/src/vm/parsed_module.rs:240-254`, host function imports with module or function names longer than `SYM_LEN_LIMIT` are silently filtered, which risks breaking the linker unexpectedly.

While current configs avoid exceeding the limit, silently dropping entries hides potential misconfigurations instead of enforcing constraints explicitly.

Recommendation

We recommend returning an error when symbol names exceed `SYM_LEN_LIMIT`, rather than filtering, to make violations explicit and prevent subtle runtime issues.

Status: Acknowledged

The client acknowledges the issue stating that they can't change this right now since it's a protocol-observable change, but it could theoretically go in a protocol update. in any case it's mis-titled: it won't and can't "break the linker", it's just suppressing notice of an invalid

condition rather than reporting it as a noisy error. The condition is still invalid, and nobody should create it anyways.

37. Preflight snapshot clones entire tracker vector

Severity: Minor

In `soroban:soroban-env-host/src/auth.rs:1149`, the `AuthorizationManager::snapshot` in recording branch clones the whole `Vec<RefCell<AccountAuthorizationTracker>>`.

For complex transactions with many trackers, this deep clone takes into account allocations even when only a small portion of the structure changes per frame. The effect is limited to recording's mode preflight, but it increases preflight latency and memory overhead.

Recommendation

We recommend snapshoting only minimal rollback state or use a copy-on-write mutation log so restore can rebuild without full vector clones.

Status: Acknowledged

The client acknowledges the issue stating that while this might truly present a bit of unmetered work, updating this is not worth the complexity. The snapshots only happen on the contract call boundaries, and the overall number of contract calls possible is reasonably low thanks to budget metering, so spending a bit more time on snapshots is not a big deal compared to the overall amount of work done for calling a contract.

38. Network config validation lacks upper bounds on several Soroban parameters

Severity: Minor

In `stellar-core:src/ledger/NetworkConfig.cpp:1023`, the `isValidConfigSettingEntry` function for `SorobanNetworkConfig` enforces minimums and relational constraints, like non negative fees, but does not define global maxima for several Soroban specific limits.

These parameters are for example code and data size caps, event size and count, as well as CPU and memory ceilings. Extremely large values, whether accidental or malicious, could lead to instability and unwanted behavior across nodes.

Recommendation

We recommend introducing explicit upper bounds for key Soroban limits, validate them in `isValidConfigSettingEntry`, and reject configs outside those ranges.

Status: Acknowledged

The client acknowledges the issue stating that it is hard to come up with an exact upper bound for a lot of these parameters and given that the values can only be accepted by a validator consensus the probability of an actually harmful mistake is low. Also, lower bounds are there first and foremost to not bring the network to the state where it's no longer upgradeable; too large limits still allow validators to roll back the bad configuration reasonably quickly.

39. Synchronous contract metadata recalculation during upgrade enables Denial of Service

Severity: Informational

In `src/ledger/LedgerManagerImpl.cpp:860-874`, the protocol upgrade process triggers an inline call to `recomputeContractCodeSize`, which recalculates memory usage metadata for all contracts deployed since protocol version `20`. This recalculation is performed synchronously within the `ApplyLedgerState` execution path, making it a blocking operation for ledger progression.

However, this design introduces a denial of service vector: a malicious actor can preemptively deploy a large number of contracts prior to the protocol upgrade, inflating the workload of the recalculation phase.

As a result, during the upgrade, the node is forced to process a substantial number of entries, delaying ledger advancement. If processing exceeds the consensus-imposed execution window, this may cause timeouts, node restarts, or outright failure to complete the upgrade, thereby jeopardizing network liveness and upgrade reliability.

Recommendation

We recommend isolating and deferring the recalculation process to a non-blocking background task or distributing it incrementally across multiple ledgers post-upgrade.

Introduce execution time constraints and checkpointing mechanisms to prevent prolonged blocking during upgrades.

Status: Acknowledged

The client acknowledges the issue stating that while it is correct that the contract code size is synchronously recomputed on protocol upgrades, this operation is expected to be sufficiently fast to not pose a concern. They note that the current number of contracts on-chain is in the

order of hundreds, and even scaling to a few thousand would require unreasonable financial expenditure. At such a scale, memory limitations would be encountered before the recomputation becomes problematic.

40. Regex allows non-hex input inconsistent with hex decoding

Severity: Informational

In `src/bucket/BucketManager.cpp:192-203`, the `isBucketFile` function uses a regular expression to identify valid bucket filenames. However, the regex pattern `^bucket-[a-z0-9]{64}\\.xdr(\\.gz)?$` permits all lowercase alphanumeric characters, including non-hex characters like `'g'` or `'z'`.

This is inconsistent with `extractFromFilename`, which slices out the 64-character hash portion and passes it to `hexToBin256`, a function that internally relies on `sodium_hex2bin`, which strictly accepts only hexadecimal characters `([a-f0-9])`.

This mismatch can result in runtime errors when a filename passes the regex check but fails hex decoding. Such cases could arise from operator errors, manual file system changes, or malformed test data.

Recommendation

We recommend updating the regex pattern in `isBucketFile` from `[a-z0-9]{64}` to `[a-f0-9]{64}` to ensure strict hex compliance and alignment with the downstream decoding logic.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

41. Unchecked error from `std::remove` may mask deletion failures

Severity: Informational

In `src/bucket/BucketManager.cpp:464` the system attempts to delete redundant bucket files using `std::remove(filename.c_str())`. However, the return value of this call is not checked.

`std::remove()` returns a non-zero value on failure, for example due to permission issues, file locks, missing files, or underlying file system errors. Ignoring this result may lead the system to believe the deletion succeeded, which can cause redundant or orphaned files to persist undetected. This undermines cleanup guarantees and may lead to resource bloat over time.

Recommendation

We recommend checking the return value of `std::remove`. On failure, emit a warning log (like `CLOG_WARNING`) including the filename and error context (use `errno` if necessary). This will improve observability and help operators diagnose filesystem-level issues.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

42. Exponential backoff retry strategy could be applied

Severity: Informational

In `src/catchup/CatchupWork.cpp:301-304`, the `getAndMaybeSetHistoryArchiveState` function retries the process 10 times. During periods of increased load, a short time interval might be not enough to push the task towards success.

Instead of retrying the task at fixed intervals, exponential backoff retry strategy could be applied, increasing the wait time between retries as failures continue. This would help avoid overwhelming a system that is already heavily loaded or experiencing errors.

Recommendation

We recommend increasing the wait time between retries.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

43. Redundant code

Severity: Informational

Throughout the codebase multiple instances of redundant code have been discovered:

- In `src/ledger/InMemorySorobanState.cpp:563-566`, the `getSize` function is implemented that validates the `mStateSize` variable for being non-negative and returns it. The return type is declared to be `uint64_t`. However, prior to returning the value of `mStateSize`, it is being cast to `int64_t`. This is redundant since the `mStateSize` is already of type `int64_t`. The conversion to the `uint64_t` type is performed implicitly.
- In `src/ledger/LedgerTypeUtils.cpp:69-71`, the `entrySizeForRent` of type `uint32_t` is computed based on the `totalSize` of type `uint64_t`. If the value of the `totalSize` variable exceeds the maximum value for the `uint32_t`

type, the result is saturated. In theory, such saturation would cause an invalid state. However, in practical terms the maximum value denotes 4 GB. Since such size of a single contract code is unrealistic, it is both more practical and more safe to throw an error instead of saturation.

- In `src/ledger/InMemorySorobanState.cpp`, multiple code snippets are duplicated between handlers of contract code and contract data. For instance, both `createContractDataEntry` and `createContractCodeEntry` include the same code block that adopts orphaned TTLs from the `mPendingTTLs` map.
- In `soroban:soroban-env-host/src/vm/module_cache.rs:140`, the `parse_and_cache_module_simple` function is unused.
- In `src/historywork/WriteVerifiedCheckpointHashesWork.h:95-98`, the `mAppendToFile` field is declared but not used, suggesting partial or abandoned behavior.
- In `soroban:soroban-env-host/src/e2e_invoke.rs:372`, the total memory cost of type `u64` is truncated to `u32::MAX` in order to fit into the `u32` type. This is safe since 4GB of memory is never reached in practice. However, this means that the truncation is redundant and should be replaced with assertion.
- In `stellar-core:src/history/CheckpointBuilder.cpp:132-136`, the code is identical to lines `158-162`.
- In `src/bucket/LiveBucketIndex.cpp:22-26`, the `typeNotSupported` function is defined to only list the `OFFER` type as unsupported. However, `OFFER` entries are actually supported. Almost all usages assume that `typeNotSupported` is `true` only for the `OFFER` type:
  - `src/ledger/LedgerTxn.cpp:2764-2771`
  - `src/invariant/BucketListIsConsistentWithDatabase.cpp:313-315`
  - `src/invariant/BucketListIsConsistentWithDatabase.cpp:123-130`

Recommendation

We recommend removing the redundant code.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

44. Magic numbers decrease maintainability

Severity: Informational

Throughout the codebase, hard-coded number literals without context or a description are used. Using such "magic numbers" goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers have been discovered in the following locations:

- `src/bucket/BucketBase.cpp:298`, `src/bucket/InMemoryIndex.cpp:127`, `src/bucket/DiskIndex.cpp:173` use same number `1000` to check for shutdown
- `src/bucket/BucketApplicator.cpp:247` specifies `1000000` for microsecond-to-second conversion
- `src/catchup/CatchupWork.cpp:304` defines 10 retries
- `src/bucket/LiveBucketIndex.cpp:32` incorrectly defines number `1'000'000` as number of bytes in a megabyte

Using magic numbers leads to maintainability issues and confusion, especially when they need to be updated or dynamically configured.

Recommendation

We recommend using named constants or configuration values instead of directly embedding literals to improve clarity and reduce potential errors in future development.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

45. Inconsistent approach to assertions

Severity: Informational

In `src/ledger/InMemorySorobanState.h:189-195`, the `QueryKey::get` function is disabled using the "QueryKey::get() called - this is a logic error" error message. However, throughout the codebase another, more robust, approach is utilized that relies on the `releaseAssertOrThrow` macro.

The `releaseAssertOrThrow` macro throws a runtime error after printing the failing expression and also its location, but the direct throw only emits a hard-coded string.

Recommendation

We recommend consistently using the `releaseAssertOrThrow` macro for increased maintainability.

Status: Acknowledged

The client acknowledges the issue stating that they will fix this in a future release.

46. Unsafe vector access in `setLastTxProcessingFeeProcessingChanges`

Severity: Informational

In `src/ledger/LedgerCloseMetaFrame.cpp:101-102`, the function `setLastTxProcessingFeeProcessingChanges` unconditionally calls `back` on the `txProcessing` vector across all version cases. The implementation assumes that at least one transaction entry has already been pushed into the vector.

Under normal conditions, this assumption holds because `pushTxProcessingEntry` is invoked before `setLastTxProcessingFeeProcessingChanges`.

However, this creates a fragile implicit dependency. If the function is ever invoked out of order, such as during transaction failures, corrupted state recovery, or after future refactoring, the `back` call would attempt to access an empty vector, likely crashing the process.

Recommendation

We recommend adding a check to ensure `txProcessing` is not empty before calling `back`.

Status: Resolved

47. Redundant clear on moved-from object

Severity: Informational

In `src/main/QueryServer.cpp:292-295`, the function `getLedgerEntry` performs unnecessary operations on a moved-from object. After executing `std::move(*liveEntriesOp)` to transfer ownership of the vector contained in the optional, the code immediately calls `liveEntriesOp->clear()`.

Once the move is performed, `liveEntriesOp` remains in a valid but unspecified state, which for a vector is typically empty. Calling `clear` on an already-empty vector has no effect, consumes unnecessary CPU cycles, and adds visual clutter. Furthermore, `liveEntriesOp` is destroyed automatically when it goes out of scope, making the explicit `clear` redundant.

Recommendation

We recommend removing the redundant `clear` operation.

Status: Resolved

48. Uninitialized variable usage in quorum checker

Severity: Informational

In `src/herder/RustQuorumCheckerAdaptor.cpp:354-424`, the function `checkQuorumIntersectionInner` declares the local variable `QuorumCheckerResource` usage without initialization before entering a `try` block. If `rust_bridge::network_enjoys_quorum_intersection` throws an exception, the `catch` block still attempts to read `usage.time_ms` and `usage.mem_bytes` to update metrics and log totals.

However, since the variable may never have been assigned a value, this behavior constitutes undefined memory access.

Uninitialized memory may corrupt telemetry by inflating counters, producing misleading log entries, and generating inconsistent measurements. Such corruption risks could obscure real failures and could influence operators to make incorrect decisions based on invalid resource accounting.

Recommendation

We recommend checking `usage.time_ms` and `usage.mem_bytes` before logging them.

Status: Resolved

49. Inconsistent use of Windows compilation macros

Severity: Informational

The codebase generally applies the `_WIN32` macro for Windows-specific conditional compilation, but in `src/util/XDRStream.h`, the macro `WIN32` is used instead. This inconsistency introduces portability and maintainability risks.

As a result, code guarded by `#ifdef WIN32` may be excluded under some compiler or build settings, leading to missing functionality or compilation errors on Windows builds. This discrepancy undermines the reliability of platform-specific logic and complicates long-term maintainability.

Recommendation

We recommend standardizing on `_WIN32` as the sole macro for Windows-specific compilation guards throughout the codebase.

Status: Acknowledged

50. Redundant protocol version check creates unnecessary code complexity in transaction queue management

Severity: Informational

In `src/herder/HerderImpl.cpp:1492`, the `triggerNextLedger` function performs a protocol version check using `protocolVersionStartsFrom` to determine whether to call `mSorobanTransactionQueue->ban` for invalid Soroban transactions. This check verifies that the ledger version supports the Soroban protocol before accessing the Soroban transaction queue. The function already conditionally initializes `mSorobanTransactionQueue` earlier in line `2269` using the same protocol version check, ensuring the queue exists only when Soroban is supported.

However, the second protocol version checks in lines `1492` and `1418` duplicates validation logic that has already been established through the conditional initialization pattern. The `mSorobanTransactionQueue` member is only non-null when the protocol version supports Soroban features, making the subsequent version check redundant. This creates unnecessary conditional branching and adds complexity to the transaction processing flow without providing additional safety or functionality.

Consequently, this redundant check introduces maintenance overhead by requiring developers to understand and maintain duplicate validation logic across multiple code paths. The additional conditional statement increases cognitive load during code review and debugging, as readers must verify that both checks are semantically equivalent and properly synchronized. This pattern also creates potential for inconsistency if future modifications update one check but not the other, leading to subtle bugs where the initialization and usage logic diverge.

Recommendation

We recommend simplifying the code by replacing the protocol version check with a null pointer check on `mSorobanTransactionQueue`, allowing the conditional initialization pattern to serve as the single source of truth for Soroban protocol support determination.

Status: Acknowledged

The client acknowledges the issue stating that this is primarily a matter of style preference.

The protocol version check is intentionally more defensive than a simple queue existence check and provides flexibility for future changes, such as creating the queue eagerly. While the team understands the perceived readability concern, they do not consider it an actual issue.

51. Inconsistent integer types in `ParallelTxSetBuilder::Stage`

Severity: Informational

In `src/herder/ParallelTxSetBuilder.cpp:371-374`, the `Stage` class defines `mBinInstructions` as `std::vector<uint64_t>` to track per-bin instruction counts, but declares mInstructions as `int64_t` for the total stage instruction count. This inconsistency introduces arithmetic risks and deviates from the non-negative semantics of instruction counts.

The mismatch manifests in multiple parts of the class. In `inPlaceBinPacking` (lines `227` and `237`), operations mix signed and unsigned values, such as subtracting `mInstructions` from `mBinInstructions[binId]`. If the cluster's instruction count exceeds the bin's current value, the subtraction causes unsigned underflow, producing a large positive result instead of failing safely.

Similarly, in `createNewClusters` (lines `257-261`), the signed `newInstructions` variable accumulates unsigned counts, risking signed integer overflow in scenarios with very large values. Comparisons between signed and unsigned types further complicate correctness by invoking implicit conversions that may produce unexpected results.

Recommendation

We recommend unifying all instruction-related counters in the `Stage` class to use `uint64_t`.

Status: Resolved

52. Lack of error handling in file I/O operations

Severity: Informational

In `src/herder/RustQuorumCheckerAdaptor.cpp:486-490`, the function `runQuorumIntersectionCheckAsync` writes a quorum map JSON file to disk for use by an external Rust subprocess that performs quorum intersection analysis.

The current implementation uses `std::ofstream` to write, flush, and close the file, but does not check whether these operations succeed. Instead, it relies solely on implicit cleanup by the file stream's destructor. File I/O can fail for numerous reasons, including insufficient disk space, permission issues, filesystem corruption, or transient I/O errors. If the file is not written correctly, the Rust subprocess may fail to start or consume corrupted data, producing incorrect quorum intersection results.

Recommendation

We recommend adding explicit error checking similar to the pattern used in the `writeResults` function.

Status: Resolved

53. Type mismatch in `getSize` return value

Severity: Informational

In `src/ledger/InMemorySorabanState.cpp:563-566`, the function `getSize` is declared to return a `uint64_t`, representing the total in-memory Soroban state size in bytes used for rent fee computation. However, the implementation incorrectly casts the internal member `mStateSize` to `int64_t` before returning, rather than to the declared `uint64_t`.

The `mStateSize` variable is deliberately stored as an `int64_t` to simplify arithmetic and avoid underflow when applying size deltas, as shown in `updateStateSizeOnEntryUpdate`. While this internal design choice is valid, the public API is intended to expose the state size as a non-negative quantity in bytes. Casting to a signed type during return introduces an unnecessary inconsistency between the declared return type and the actual value being produced.

Recommendation

We recommend casting `mStateSize` directly to `uint64_t` when returning from `getSize`.

Status: Resolved

54. Misspelled method name in `LedgerManager` interface

Severity: Informational

In `src/ledger/LedgerManager.h:233`, the virtual method `getLastClosedSnaphot` contains a typographical error: it is missing the letter "s" and should be spelled `getLastClosedSnapshot`.

The incorrect spelling introduces unnecessary friction for developers, increasing the likelihood of mistakes or confusion when navigating the ledger management APIs.

Recommendation

We recommend renaming the method to `getLastClosedSnapshot` across both interface and implementation, updating all dependent call sites accordingly.

Status: Resolved

55. Missing `null` pointer dereference in `ByteSlice` constructor

Severity: Informational

In `src/crypto/ByteSlice.h:79-83`, the constructor `ByteSlice(char const* str)` invokes `strlen(str)` without verifying that str is non-null. If a `null` pointer is passed, `strlen` attempts to dereference it, resulting in an immediate segmentation fault.

The constructor forwards to `ByteSlice((void const*)str, strlen(str))`, where the unchecked `strlen` call is always executed.

This flaw affects cryptographic operations that rely on `ByteSlice` when instantiated with C-string input, including hashing functions such as `sha256` and `blake2`, as well as HMAC operations and signature verification routines.

Although most of the codebase uses safer abstractions like `std::string` or `std::vector<uint8_t>`, the existence of this constructor leaves open the possibility that a `null` pointer may be inadvertently passed, exposing critical cryptographic flows to crashes. The impact extends to transaction signing, key derivation, and other security-sensitive processes.

Recommendation

We recommend introducing an explicit null pointer check in the `ByteSlice(char const* str)` constructor.

Status: Acknowledged

The client acknowledges the issue stating that there is a minor risk of passing a `nullptr` in this case, and indicates that adding an assertion could be a reasonable safeguard.

56. Missing virtual destructor for `TransactionFrameBase` class

Severity: Informational

In `src/transactions/TransactionFrameBase.h:239-241`, the `TransactionFrameBase` class is declared as a polymorphic base class but does not define a virtual destructor. The class exposes multiple pure virtual methods (lines `136-241`) and functions as the interface for concrete implementations such as `TransactionFrame` and `FeeBumpTransactionFrame` (as shown in `TransactionFrameBase.cpp:20-22`).

This omission introduces a potential undefined behavior scenario in C++. Although the current codebase primarily manages these objects using `std::shared_ptr<TransactionFrameBase>` (via the `TransactionFrameBasePtr` alias), which ensures proper polymorphic deletion through type erasure, the lack of a virtual destructor violates a fundamental rule for polymorphic base classes.

The risk arises if, through refactoring or accidental misuse, a derived object is deleted through a raw `TransactionFrameBase*` pointer. In such a case, only the base destructor would be invoked, preventing derived-class cleanup and potentially leaking resources or leaving objects in an inconsistent state. These failures are subtle and notoriously difficult to debug.

Recommendation

We recommend adding a virtual default destructor to the `TransactionFrameBase` class. This ensures that objects deleted through base pointers are correctly destroyed, aligning with modern C++ best practices and the C++ Core Guidelines (C.35)

Status: Resolved

57. Potential negative signature weight and inconsistent integer types

Severity: Informational

In `src/transactions/TransactionFrameBase.h:171-173`, the interface uses inconsistent integer types that create both semantic and practical issues across the transaction framework.

The `checkSignature` method declares its `neededWeight` parameter as `int32_t`, permitting negative values. In Stellar's authorization model, signature weights and thresholds are strictly non-negative: zero represents no weight, while positive values represent cryptographic voting power. Passing a negative neededWeight produces meaningless comparisons such as `totalWeight >= neededWeight`, where any accumulated positive weight trivially satisfies a negative threshold. While protocol-level safeguards likely prevent negative thresholds from reaching this point, the API design itself introduces unnecessary risk of programming errors and defensive checks.

Beyond this semantic flaw, the header demonstrates inconsistent use of integer types, which increases complexity and the likelihood of subtle bugs:

- The code mixes XDR aliases (`int64` in line `235`) with standard types (`int64_t` in lines `182-187`), creating ambiguity about which convention applies in different contexts.
- Weights, which are inherently non-negative, are represented as `int32_t`, while other non-negative quantities may use unsigned types.
- Most fee-related methods use `int64_t` (lines `182-187`), but `declaredSorobanResourceFee` uses `int64` (line `235`), introducing potential overload ambiguities and inconsistent formatting.

Evidence from `TransactionFrame::checkExtraSigners` in lines `378-386` further highlights this inconsistency: the implementation includes a static assertion and explicit cast to `int32_t`.

Recommendation

We recommend two corrective changes:

- Replace `int32_t` with an unsigned type for all signature weight parameters to align with their strictly non-negative semantics.
- Standardize the use of integer types throughout the header, eliminating the mix of XDR aliases and standard types to ensure clarity, consistency, and maintainability.

Status: Acknowledged

The client acknowledges the issue stating that signature weights are clamped at `u8::MAX`, eliminating any overflow risk in this context. The client clarifies that the use of unsigned types for "semantically positive" values is not applied consistently across the codebase, and in fact there is an argument against adopting that practice universally. As such, the current code is considered acceptable.

However, the client agrees that the `int64/int64_t` inconsistency represents a broader concern and acknowledges that it warrants a large-scale cleanup effort across the entire codebase.

58. Missing insertion check in `setDeltaEntry` may silently discard ledger updates

Severity: Informational

In `src/transactions/ParallelApplyStage.h`, the `setDeltaEntry` function inserts updated ledger entries into the `mDelta.entry` map using `emplace`, but does not check whether the insertion was successful. If an entry with the same key already exists in the map, the insertion will fail silently, and the new delta will be discarded without warning.

This behavior can lead to silent loss of state updates, resulting in inconsistencies during delta processing or downstream invariant validations.

Recommendation

We recommend adding a `releaseAssert` to verify the success of the emplace operation, as a failed insertion should not occur under expected conditions.

Status: Acknowledged

59. Missing main thread assertions in `LedgerManager` may obscure threading assumptions

Severity: Informational

Several functions in `src/ledger/LedgerManagerImpl.cpp` are intended to be invoked exclusively from the main thread, and this constraint is typically enforced via explicit

assertions. These checks help ensure thread safety and make threading assumptions clear during development and debugging.

However, some functions that appear to have similar constraints currently lack such assertions. In particular, `loadLastKnownLedger` and `rebuildInMemorySorobanStateForTesting` are likely main-thread-only and would benefit from an assertion to enforce that requirement.

Additionally, the function `applyLedger` may be invoked from either the main thread or the application thread. In this case, using `threadInvariant`, which verifies the call originates from an allowed thread, would improve clarity and robustness.

Recommendation

We recommend adding `mainThreadOnly` assertions to `loadLastKnownLedger` and `rebuildInMemorySorobanStateForTesting`, and applying `threadInvariant` to `applyLedger` to explicitly define and enforce valid threading contexts. These safeguards will reduce the risk of unintended concurrent access and clarify execution context expectations for future maintainers.

Status: Resolved

60. Misplaced thread assertion in `commitChangesFromThread` leads to redundant checks

Severity: Informational

In `src/transactions/ParallelApplyUtils.cpp`, the function `commitChangesFromThread` asserts that it is being executed from either the main thread or the application thread. While this check is necessary to enforce correct thread context, placing the assertion inside `commitChangesFromThread` results in the assertion being evaluated repeatedly, once per thread.

However, this function is only invoked via `commitChangesFromThreads`, which iterates over all thread-local states and delegates to `commitChangesFromThread` for each.

Since all invocations occur within a single context, the thread assertion could be more efficiently and clearly enforced at the start of `commitChangesFromThreads`.

Recommendation

We recommend relocating the thread context assertion from the function that applies changes per thread to the higher-level function that orchestrates the application across all threads. This adjustment preserves the correctness of the thread safety check while ensuring it is evaluated only once per operation

Status: Acknowledged

61. Invariant check function modifies state

Severity: Informational

The function `LedgerManagerImpl::checkAllTxBundleInvariants` in `src/ledger/LedgerManagerImpl.cpp` is intended to perform post-transaction invariant validation. However, it also mutates transaction effects by invoking `setDeltaHeader` during execution.

This blending of validation and state mutation introduces a risk of misaligned state during invariant checks.

Recommendation

We recommend refactoring the function to decouple invariant validation from any mutation of transaction effects.

Status: Acknowledged

62. Confusing return semantics in `maybeAdoptFailedReplayResult`

Severity: Informational

The function `maybeAdoptFailedReplayResult` in `src/transactions/TransactionFrame.cpp` exhibits counterintuitive behavior by returning `true` when no replay result is adopted.

This return value conflicts with expectations implied by the function name, which suggests that a return value of `true` would indicate a successful adoption. Additionally, its purpose as a no-op under normal execution is not immediately evident from its naming or placement.

Recommendation

We recommend inverting the return logic of the function so that it returns `true` when the replay result is adopted and `false` otherwise.

Status: Acknowledged

The client acknowledges the issue stating that it will likely be addressed as part of a future cleanup effort.

63. Inconsistent empty container checks reduce code maintainability

Severity: Informational

In `src/herder/TxSetFrame.cpp`, the codebase exhibits inconsistent patterns when evaluating empty container states.

Specifically, it alternates between using the `empty` method and the `size() == 0` comparison within identical contexts and even within the same class, such as `TxSetPhaseFrame`.

For example, `TxSetPhaseFrame::empty` evaluates emptiness via `sizeTx() == 0`, while nearby logic directly invokes `mStages.empty`. Similar inconsistencies are present in validation routines and classes like `ApplicableTxSetFrame`.

Although both methods are semantically equivalent, this stylistic divergence increases the cognitive overhead for developers, hinders code readability and maintainability, and poses a latent risk for semantic divergence if container implementations change or are refactored in the future.

Recommendation

We recommend standardizing all empty container checks throughout the codebase by adopting the idiomatic `empty` method.

Status: Acknowledged

The client acknowledges the issue stating that it will likely be addressed as part of a future cleanup effort.

64. Missing exception handling in asynchronous transaction processing

Severity: Informational

The `applySorobanStageClustersInParallel` function, defined in `src/ledger/LedgerManagerImpl.cpp:2119-2154`, initiates parallel transaction execution using `std::async` but omits explicit exception handling during the result collection phase.

When exceptions are thrown from any parallel task and caught via `threadFuture.get` in line `2149`, they propagate immediately without structured error handling, relying instead on the destructor behavior of `std::future` for cleanup.

While the destructor of a `std::future` obtained via `std::async` blocks until the associated thread completes, this implicit synchronization introduces a secondary risk. Blocking during stack unwinding may lead to unpredictable timing, delayed error propagation, or even deadlocks.

Moreover, this diverges from the project's established pattern, where exceptions in parallel contexts are explicitly caught and logged using `printErrorAndAbort` (e.g., `TransactionFrame::parallelApply`, lines `1953-1955`).

Recommendation

We recommend invoking `printErrorAndAbort` or a similar explicit logging and fail-fast mechanism upon exception detection.

Status: Acknowledged

65. Quadratic-time transaction clustering

Severity: Informational

The transaction clustering logic in `src/herder/ParallelTxSetBuilder.cpp:308-321` applies a first-fit-decreasing bin packing strategy with a worst-case time complexity of `O(n^2)`, where n represents the number of transactions.

The performance bottleneck lies in the `Stage::addTx` method, which triggers a full recomputation of transaction clusters after each transaction insertion. Specifically, when transactions exhibit overlapping access footprints, they may repeatedly merge existing clusters, causing the bin packing algorithm to re-execute over the full dataset.

Although the bin packing loop runs in `O(C * B)` time per invocation (where C is the number of clusters and B the number of bins), repeated cluster invalidation and reconstruction across

transactions results in an overall computational cost of `O(n^2 * B)`. This quadratic behavior creates a denial-of-service (DoS) vector through excessive CPU usage.

A malicious actor can exploit this by submitting Soroban transactions with deliberately overlapping state access patterns, provoking maximum conflict resolution effort. This would lead to repeated cluster rebuilds, stalling validator nodes during ledger finalization and risking missed consensus deadlines or transaction drops.

While current limits, such as maximum transactions per ledger and single-cluster-per-stage enforcement, partially mitigate the issue, it remains a latent scalability risk.

Recommendation

We recommend refactoring the clustering mechanism to use an incremental or amortized update strategy that avoids full recomputation on each transaction.

Status: Acknowledged

The client acknowledges the issue stating that while this scenario is theoretically possible, it does not appear to present a concern in the near term given that the system currently operates on the order of hundreds, rather than thousands, of transactions. The client further confirms that this case will be included in upcoming benchmarks to ensure proper coverage and that a reasonable optimization path is available.

66. Scattered logic for concurrency state management in ledger application

Severity: Informational

To manage concurrency during ledger application, the catch-up procedure invokes `beginApply` on the `LedgerManager` to signal the start of application. Later, once ledger processing concludes, the application state flag `mCurrentlyApplying` may be reset to `false` within `ledgerCloseComplete`, but only if no further ledgers remain in the application queue.

However, the logic for setting and clearing `mCurrentlyApplying` is currently scattered across the codebase, with over `1,000` lines separating these two control points. This separation obscures the lifecycle of the flag and complicates reasoning about thread safety and state ownership.

Recommendation

We recommend introducing a dedicated `maybeEndApply` function in `src/ledger/LedgerManagerImpl.cpp` to encapsulate the logic responsible for resetting `mCurrentlyApplying`.

Status: Acknowledged

The client acknowledges the issue stating that this is a valid point. The underlying cause is that the externalize, apply, trigger next ledger flow is asynchronous and distributed across multiple modules, which necessitates the use of `mCurrentlyApplying`. The client notes that efforts are underway to simplify this flow, referencing the proposed changes in pull request #4725.

67. Data race in concurrent bucket index access

Severity: Informational

In `src/bucket/BucketManager.cpp:1073`, the `maybeSetIndex` function sets a bucket's index pointer without holding any mutex on the individual `LiveBucket` object.

While most bucket operations are protected by `mBucketMutex`, the mIndex member of individual buckets can be accessed concurrently. The `getIndex` method dereferences `mIndex` without synchronization, creating a data race condition. This can lead to crashes or corrupted index data when one thread is setting the index while another is reading it.

Recommendation

We recommend adding a mutex protection to the individual bucket's index operations. Either use a per-bucket mutex or ensure all index access happens under the `mBucketMutex` lock.

Status: Acknowledged

The client acknowledges the issue stating that this function is only called on initialization when only a single thread has access to the object, so there is no race. Only after calling `maybeSetIndex` is the `Bucket` accessible outside of `BucketManager`, i.e. accessible to other threads

68. Missing protocol legality validation for Hot Archive bucket entries

Severity: Informational

In `src/bucket/BucketInputIterator.cpp:75` the code validates protocol legality only for `LiveBucket` entries.

There is no corresponding validation for `HotArchiveBucket` entries. This means that Hot Archive buckets could contain entries that violate protocol rules without detection, leading to consensus failures or other undefined behavior.

Recommendation

We recommend implementing protocol legality checking for `HotArchiveBucket` entries similar to the existing check for `LiveBucket`, or document why such validation is not necessary for Hot Archive buckets.

Status: Acknowledged

The client acknowledges the issue stating that the protocol legality check is for a protocol change where the Live `BucketList` added new entry types. There is no such change for Hot Archive buckets to check.

69. Missing validation of bucket entry key during XDR parsing

Severity: Informational

In `src/util/XDRStream.h`, lines `117-163` define the `readOne` function while lines `165-232` define the `readPage` function. Both functions process the input stream reading a bucket entry from disk - one reads the whole entry as-is, and another reads a certain number of bytes. Both functions end with calls to `xdr::xdr_argpack_archive`.

However, the `readPage` function additionally validates the parsed entry using its `key` parameter. The `readOne` function does not receive such a parameter and does not validate the bucket entry. As a consequence, in `src/bucket/BucketSnapshot.cpp:59-73`, the `getEntryAtOffset` function utilizes the `k` parameter of `LedgerKey` type only when `pageSize > 0`, while the case `pageSize == 0` ignores the `k` parameter.

Recommendation

We recommend utilizing the `k` parameter in the `getEntryAtOffset` function by passing it into the `readOne` function and implementing the validation similarly to how it is done in the `readPage` function.

Status: Acknowledged

The client acknowledges the issue stating that this is actually dead code, as `pageSize != 0` in this function, since when `pageSize == 0`, a memory index is used.

70. Index creation reliability comment missing failure case

Severity: Informational

In `src/bucket/BucketIndexUtils.cpp:33-50`, the function responsible for returning a `BucketT::IndexT` object contains a comment indicating that a `std::runtime_error` may occur when `BucketManager` shuts down. However, the comment lacks completeness regarding other probabilistic failure conditions.

Specifically, index creation can also fail due to binary fuse filter memory access errors, even after being retried multiple times (typically 10). This condition is not documented, reducing the clarity and debuggability for downstream developers trying to understand non-deterministic index creation failures.

Recommendation

We recommend extending the comment in the catch block to include the binary fuse filter memory access failure scenario.

Status: Acknowledged

The client acknowledges the issue stating that the binary fuse error case is included for completeness, but is statistically impossible.

71. `EntrySizeForRent` may undercharge rent and enable Denial of Service

Severity: Informational

In `src/ledger/LedgerTypeUtils.cpp:69-72`, the `totalSize` variable is correctly computed as a `uint64_t` sum of `entrySizeForRent` and the `contract_code_memory_size_for_rent`. However, the result is then downcast to a `uint32_t` using `std::min(totalSize, UINT32_MAX)`. This truncation introduces a subtle issue - once `totalSize` exceeds 4GB (the maximum value of `uint32_t`), the computed `entrySizeForRent` caps at 4GB. As a result, for contract sizes above this threshold.

This discrepancy can be exploited by adversaries to deploy oversized contracts while only paying rent for a much smaller footprint. If the network permits a large number of such contracts, this could lead to memory exhaustion and denial of service. The exploitability of this vector depends on broader network parameters such as the maximum number of contracts allowed or enforced quotas

While current protocol parameters (`MAX_CONTRACT_SIZE = 2KB`) prevent this scenario, the logic itself does not enforce it or fail loudly.

Recommendation

We recommend adding an explicit assertion or logging if `totalSize > UINT32_MAX` to fail safe..

Status: Acknowledged

The client acknowledges the issue stating that the u32 limit on the entry size overshoots any reasonable assumptions on what the entry size can be by several orders of magnitude; it's not

really worth making the code more complex because of this. Truncation is good enough as a fail safe to prevent accidental crashes on mis-configuration.

72. Fragile parsing and pretty-printing of JSON data

Severity: Informational

The codebase employs manual strings manipulation as the main approach towards reading and writing data in JSON format.

- In `stellar-core:src/historywork/WriteVerifiedCheckpointHashesWork.cpp`, the output JSON is built by manual concatenation of `"["`, lines with trailing commas and final `"[0, \"\"]\n]"` constant.
- In `stellar-core:src/historywork/WriteVerifiedCheckpointHashesWork.cpp:272-293`:
  - The `trustedHashFile` is copied as-is without any validation. The output file must result in correct JSON, but the copied file is not ensured to have correct structure. The content of `trustedHashFile` is also not limited by size.
  - The fallback terminator `"[0, ""]\n]"` is not used, assuming that the `trustedHashFile` has the correct structure. If the assumption does not hold, the output could remain `"["` and then be closed and renamed, silently producing invalid JSON without any error.

Recommendation

We recommend either validating the JSON inputs and/or the whole output file, or simply adopting a reputable library for handling JSON.

Status: Acknowledged

73. Panic context lost at core boundary obscures error root cause

Severity: Informational

In `soroban-core:src/soroban_proto_any.rs:327`, the `invoke_host_function_or_maybe_panic` wraps execution in `panic::catch_unwind` and, on panic, returns a generic "`contract host panicked`" error.

The original panic payload is discarded, taking away from the operator the actionable context of what was the panic reason and why it happens, and complicating analysis when rare panics occur. The catch boundary is correct for safety, but dropping the payload harms observability.

Recommendation

We recommend downcasing the panic payload and include it in the returned error.

Status: Resolved

74. Potential race conditions while working with filesystem

Severity: Informational

Two similar issues have been discovered stemming from gaps between “Time-of-Check” and “Time-of-Use” while working with the filesystem. Such issues introduce subtle race conditions and, although unlikely in the current version of the codebase, could become a problem in the future:

- In `stellar-core:src/historywork/WriteVerifiedCheckpointHashesWork.cpp:273`, the `endOutputFile` function checks for existence of the `mTrustedHashPath` in the filesystem. If the check is successful, the variable is used on line `280` where the input stream is opened and assigned to the `trustedHashFile` variable.

  Further, if the input stream has been opened successfully, it is used to retrieve lines to copy into the output file. However, the `if` expression started on line `281` has no `else` clause. If the input stream has not been opened, which could be caused by a missing file, the problem is silently ignored. In fact, the check on line `273` is not protective enough and should be transformed into the handler of missing file error.

- Similarly, in `stellar-core:src/historywork/VerifyBucketWork.cpp:61-62`, the `spawnVerifier` function checks the size of the bucket file by querying the filesystem by filename. Then, on line `79`, it starts a background thread and passes the `filename` value to it. On line `97`, the `filename` is passed into the `createIndex` function and is actually used only in `stellar-core:src/bucket/BucketIndexUtils.cpp:42`.

  Between `stellar-core:src/historywork/VerifyBucketWork.cpp:61` and `stellar-core:src/bucket/BucketIndexUtils.cpp:42` the conditions of the bucket file could change - bucket size validation can get stale or the bucket file can move if it is being manipulated by another thread, by the system or by the operator. If the file became larger, it would violate the `MAX_HISTORY_ARCHIVE_BUCKET_SIZE` limit. If the file has moved, the whole operation could fail.

These issues do not pose immediate risk, since no simultaneous writes to the aforementioned files have been identified. Theoretically, these issues could be triggered by the operating system managing the files or by unskilled operator intervention. We are reporting these issues

merely as “bad practice” that could become a more serious issue in future versions of the codebase.

Recommendation

We recommend performing all necessary file validations atomically with the acquiring of the file descriptor. Existence check can be replaced with the check for errors during the file opening. Size check can be performed directly on the file descriptor using the `fstat` system call.

Status: Acknowledged

The client acknowledges the issue stating that there are no real cases: either "immutable files" or "race would only be with ourselves, outside any real threat model".

75. Potential race condition in closure

Severity: Informational

In `stellar-core:src/historywork/VerifyBucketWork.cpp:77-80`, the `spawnVerifier` function saves the reference to the `VerifyBucketWork` object into a variable of `std::weak_ptr` type. Further, it starts a new background thread using the closure that captures the `mIndex` member of the same `VerifyBucketWork` object as the `index` reference using the `&index = mIndex` expression.

On lines `84-89`, the `weak` pointer is converted into a stronger reference of `shared_ptr` type and the lambda function is aborted in case of failure to do so. This serves as protection from writing the result of work by the pointer to a non-existing object.

Past the line `89`, the `self` variable points to the `VerifyBucketWork` object, which has a `mIndex` member and which is guaranteed to live as long as the `self` pointer is valid. This means that assigning to the `index` reference on lines `97-99` should be safe.

On other hand, the `index` reference has been taken from `mIndex` before the `weak.lock`, so there could be another reader or writer of it which would introduce the race condition. It is safer to use `self->mIndex` because `self` is guaranteed to be a strong pointer of the `shared_ptr` type.

Recommendation

We recommend removing the binding expression `&index = mIndex` on line `80` and assigning to the `self->mIndex` on line `97`.

Status: Resolved

76. Imprecise usage of the `errno` macro

Severity: Informational

In `stellar-core:src/history/HistoryManagerImpl.cpp:767-769`, the error in removing a stale checkpoint is being logged. The error code is referred to by `strerror(errno)`, however the call that is supposed to set the errno has been performed long before by `fs::removeWithLog(f)` on line `753`.

Since the `strerror(errno)` is called long after the failed syscall, the error message can report the wrong cause. Between the actual failure and the message, various events could clobber `errno`: from thread sleep to logging, depending on platform/libraries.

Recommendation

We recommend saving `errno` immediately after the call to a function that sets it, in general. However, the `fs::removeWithLog(f)` function defined in `stellar-core:src/util/Fs.cpp:506-521` already logs the `errno`, so simply removing `strerror(errno)` on line `769` would suffice.

Status: Resolved

77. Insufficient error handling

Severity: Informational

Throughout the codebase, multiple error handling issues have been discovered:

- In `soroban:soroban-env-host/src/vm/module_cache.rs:58`, the `map.lock` call occupies the mutex, and on line `59` all potential errors are mapped into `InternalError`. Mutex poisoning is not handled and causes any thread-local panic to be propagated as global `InternalError` and crash the node. Ideally, mutex recovery must be implemented.
- In `soroban:soroban-env-host/src/vm.rs:68-69`, VM instantiation time is recorded in the `VmInstantiationTimer` implementation of the `Drop` trait. The "`let _ =`" pattern silently ignores potential errors from timing collection. While such failures are rare (only `RefCell` borrow conflicts), the potential error should be logged for better observability rather than silently ignored.
- In `stellar-core:src/history/FileTransferInfo.cpp:27`, the `std::runtime_error` is thrown but it does not include the actually occurred filesystem error.
- In `stellar-core:src/rust/src/soroban_proto_all.rs:579`, the `unwrap` call should be replaced with a proper error message.
- Usages of `unwrap_or` and `unwrap_or_default` mask errors with fallback values.

Recommendation

We recommend handling errors and attach descriptive messages to them in order to increase maintainability and developer experience.

Status: Acknowledged

78. Templates maintainability can be improved

Severity: Informational

Several maintainability issues related to templates have been discovered:

- In `stellar-core:src/history/HistoryUtils.cpp`, the `getHistoryEntryForLedger` template function does not validate that the template `T` type has a `ledgerSeq` member before accessing it.

  If it has no `ledgerSeq` member, the expressions `currentEntry.ledgerSeq` on lines `26` and `33` will cause compilation errors.

- Similarly, in `stellar-core:src/historywork/VerifyBucketWork.cpp`, the `VerifyBucketWork` class is templated but does not validate the bucket type at compile time.

  Missing `IndexT` or `createIndex` members will cause compilation errors in `VerifyBucketWork.h:31` and `VerifyBucketWork.cpp:97` correspondingly.

While there is no security concern, the developer experience of tracing the cause of failed compilation can be highly challenging due to incomprehensible template traces.

Recommendation

We recommend adding a `static_assert` to produce a clear, localized error message, for example "`T must have ledgerSeq`" to provide better developer experience.

Status: Acknowledged

79. Misleading documentation

Severity: Informational

In `src/historywork/WriteVerifiedCheckpointHashesWork.h:45-46`, the `loadLatestHashPairFromJsonOutput` function is declared. The inline documentation on lines `43-44` states that it returns `std::nullopt` if the file does not exist.

However, the actual definition in `stellar-core:src/historywork/WriteVerifiedCheckpointHashesWork.cpp:25-31` is implemented to throw `std::runtime_error` in case of non-existing path.

Additionally, in `soroban:soroban-env-host/src/cost_runner/util.rs:22,40` two error messages are defined, which are wrong, having in mind a function within which they are returned. For `ecdsa_secp256r1_recover_key`, the error mentions `ECDSA-secp256k1`, and for `ecdsa_secp256k1_verify_signature` the error mentions `secp256r1`.

Recommendation

We recommend correcting the docs and errors returned.

Status: Resolved

80. Missing defensive check on `linearTerm`

Severity: Informational

At `stellar-core:src/ledger/LedgerManagerImpl.cpp:983-987`, the code assigns `linearTerm` directly from `memParams[(size_t)stellar::VmInstantiation]` without ensuring it is non-zero.

If it is zero, it will trigger `startCompilingAllContracts` every block, placing unnecessary load on the critical ledger apply path, and since the parameter may fallback to `5000` when absent form configurations, this weakens correctness by allowing an incomplete network config.

Recommendation

We recommend adding a check to enforce `linearTerm != 0` and throw an error if the parameter is missing, making it compulsory in the network configuration.

Status: Acknowledged

The client acknowledges the issue stating that in general, cost model mis-calibration may lead to the potential issues and DOS concerns, which is why it's not updated often and is thoroughly audited. That said, in this particular case the mis-calibration impact is not too high and there is no good reason to halt the network because of it. We could use the default 'safety' value if the config value is 0 though.

81. Miscellaneous comments

Severity: Informational

The following issues were identified as minor code quality or consistency concerns. While they do not pose security risks, addressing them would improve maintainability, readability, and adherence to project standards:

- `src/transactions/ParallelApplyUtils.cpp`
  - `maybeMergeRoTTLBumps`: Contains unnecessary nesting that can be flattened for better readability.
  - Line `176`: Manual construction of `ParallelApplyEntry` should be replaced with a call to `cleanPopulated` for consistency.
  - `buildRoTTLSet` and `getReadWriteKeysForStage`: These functions have overlapping functionality but significantly different names, consider renaming for clarity.
- `src/transactions/TransactionFrame.cpp`
  - Documentation is outdated regarding assumptions about host-side fee calculations.
  - Line `789`: Error message incorrectly refers to protocol `v0`; it should reference protocol `v1`.
  - Lines `654–671`: Splitting logic into two separate if statements for computing `numDiskReads` and checking total reads (for protocol version $\geq$ `23`) would improve code clarity and logical chunking.
- `src/ledger/LedgerManagerImpl.cpp`
  - `evictFromModuleCache`: The ledgerVersion argument is currently unused, consider removing or documenting its intended future use.
- `src/transactions/InvokeHostFunctionOpFrame.cpp`
  - `checkIfReadWriteEntryIsMarkedForAutorestore`: The `lk` argument is unused and should either be removed or its purpose clarified.
- `src/util/TxResource.h`
  - The enumerator `DISK_READ_BYTES` uses an inconsistent naming convention relative to the other enum members, standardization is recommended.
- `src/transactions/LumenEventReconciler.cpp`
  - Include a reference to `CAP-067` to explain the rationale behind the implemented workaround.

- `src/invariant/EventsAreConsistentWithEntryDiffs.cpp`
    - Line `583`: The empty string return value is inconsistent with how similar cases are handled elsewhere in the file, should be aligned for uniformity.
- `src/herder/TxSetFrame.cpp`
    - Line `1987`: A validation check is performed only in the main thread; it should also be enforced in the apply thread to ensure consistency in execution semantics.
- `soroban:soroban-env-host/src/host/invocation_metering.rs:354`
    - Multiplication of three variables of type `i64` and one variable of type `u32` is assigned to the `rent_ledger_bytes` variable of type `i64`. In general case, this multiplication can overflow. Although further investigation revealed that this code is enabled by the `testutils` feature, this code seems to be possible to re-purpose in future. It is better to explicitly handle the potential overflow.
- `stellar-core:src/ledger/LedgerManagerImpl.cpp:980`
    - The `scale` is hardcoded as `128`. This introduces a hidden dependency since the value must always remain consistent with `COST_MODEL_LIN_TERM_SCALE_BITS` defined on the Rust side, and divergence would cause mismatches in contract cost calculations. It is better to replace the magic number with a named constant.
- `stellar-core:src/ledger/SharedModuleCacheCompiler.cpp:122`
    - The duration in microseconds `dur_us` is added to the total time `mTotalCompileTime`, stored in nanoseconds. While it is safe due to the overloaded `+=` operator, which performs the conversion implicitly, it would be more clear and maintainable to perform the conversion manually using `duration_cast`.

Recommendation

We recommend performing a systematic cleanup of minor code inconsistencies, unused parameters, outdated documentation, and naming misalignments across the codebase. While these issues do not impact runtime correctness or security, resolving them improves developer experience, reduces cognitive overhead during maintenance, and prevents potential misunderstandings in future code changes.

Status: Partially Resolved