

Smart Contract Assessment *Templar Protocol*

HALBORN

Smart Contract Assessment - Templar Protocol

Prepared by: **HALBORN**

Last Updated 08/13/2026

Date of Engagement: July 10th, 2026 - July 23rd, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED



TABLE OF CONTENTS

1. Introduction
1.1 Key themes observed across findings
1.2 Main recommendations
2. Test methodology and approach
3. Technical profile
3.1 Document purpose
3.2 System summary
3.3 Components
3.4 Authority and trust model
3.5 Data model and deployment front door
4. Execution flows
4.1 Document purpose
4.2 Refresh (permissionless price update)
4.3 Read (aggregated_latest and the sep-40 adapter)
4.4 Governance proposal lifecycle
5. Attack surface
5.1 Document purpose

- 5.2 Entry points
- 5.3 Deployment boundary
- 6. Threat model
 - 6.1 Document purpose
 - 6.2 What must be protected
 - 6.3 Actors and trust
 - 6.4 Trust boundaries and assumptions
 - 6.5 Assets at risk
- 7. Dynamic testing and proof-of-concept campaign
 - 7.1 Document purpose
 - 7.2 Approach and coverage
 - 7.3 Results
 - 7.4 Refutations recorded as controls
- 8. Risk methodology
- 9. Scope
- 10. Assessment summary & findings overview
- 11. Findings & Tech Details
 - 11.1 Read-side freshness measures cache-write time, not source time (effective staleness up to about 2x max_age_secs)
 - 11.2 Degraded quorum lets a single malicious surviving source determine the median low price
 - 11.3 Positive aggregated price can be delivered to sep-40 consumers as a valid zero price
 - 11.4 A single budget-exhausting configured source aborts the entire refresh transaction
 - 11.5 Source-outage cache overwrite contradicts the documented last-price grace window
 - 11.6 Freshness bounds are optional and unenforced, allowing stale-forever reads and future-dated prices
 - 11.7 Nonzero circuit-breaker threshold silently persists as zero through the postcard decimal round-trip
 - 11.8 Missing breaker storage is treated as an empty untripped set instead of failing closed
 - 11.9 One oracle address can fill multiple quorum slots via distinct asset aliases
 - 11.10 Source-controlled timestamps can regress, rewrite, or starve the sep-40 history
 - 11.11 Accepted extreme price scale poisons the entire adapter prices() batch
 - 11.12 Config validation accepts inert circuit breakers that can never trip
 - 11.13 Unbounded registered-asset list can make global extend_ttl() exceed transaction resource limits
 - 11.14 Nonzero sample_interval lets monotonicrun miss accepted intra-interval staged ramps
 - 11.15 Circuit-breaker suite has no cumulative-drift defense: a steady sub-threshold price ramp evades all three rules
 - 11.16 Setactionttl (proxyconfigurationmanager) can retime any operation including admin ops, ratcheting admin-recovery timelocks
 - 11.17 Sep-40 adapter can relabel a parent price under an unrelated base without denomination conversion
 - 11.18 Windowedchangedelta ignores extreme interior-window outliers that return to the window baseline
 - 11.19 Decimal saturation aliases distinct extreme window moves and makes windowedchangedelta miss them

11.20 Rearming one breaker clears or replaces the shared accepted baseline used by every other breaker

11.21 `Rearmconfig.armed_after_secs` is interpreted as an absolute timestamp while docs and tests use it as a relative grace duration

11.22 Sep-40 adapter timestamps are not aligned to the advertised resolution

11.23 Renouncing adapter ownership does not decommission it and makes future shutdown impossible

11.24 Governance `extend_ttl` is permissionless but documented as admin-only

11.25 `Refreshfailure` telemetry collapses distinct source failures; documented codes 2 and 4 are unreachable

11.26 `Priceidentifier` json schema contradicts its `serde` hex-string wire format

11.27 Source outage masks an active manual trip in refresh telemetry (status precedence)

11.28 Shared weighted-median kernel counts zero-weight sources toward `min_sources`

11.29 `Decimal::from_str` accepts trailing garbage in fractional strings

11.30 A fresh circuit-breaker trip trips and reports only the first blocking breaker

11.31 No-op `set_manual_trip` emits a `manualtripset` event and clears the cache, contradicting the docs

11.32 Shared governance kernel `create()` accepts a `ttl` below the operation minimum (timelock enforcement not owned by the kernel)

11.33 Shared governance kernel governance has no validating constructor: a corrupt ledger can double-consume a proposal id

11.34 `Decimal::from_str` silently wraps schema-valid whole values at 2^{384} instead of rejecting them

11.35 `Circuitbreakertripped` telemetry drops the normalized exponent and source publish timestamp

11.36 `Weighted medianhigh` can select a minority or zero-weight source

11.37 Conservative decimal scaling can falsely trip a breaker for a representable below-threshold move

11.38 `Monotonicrun` excludes steps exactly equal to its documented minimum threshold

11.39 Successful sep-40 adapter upgrades emit no adapter upgrade event

1. INTRODUCTION

Halborn performed a security assessment of the Templar Protocol proxy oracle, a Soroban (Stellar) smart-contract system that aggregates prices from multiple upstream SEP-40 feeds, applies freshness and circuit-breaker defenses, and re-exposes a normalized price to downstream consumers such as the Blend lending market. The assessment covered the pinned snapshot of Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, path contract/proxy-oracle, branch dev.

The in-scope code comprised two chain-agnostic no_std kernels (the oracle aggregation and circuit-breaker kernel, and the governance proposal-lifecycle kernel), the deployable Soroban runtime proxy oracle contract, the Soroban governance contract, the SEP-40 read adapter, and the shared common and governance-common crates. The excluded near/ runtime was out of scope. The external templar-primitives crate (Decimal, Nanoseconds) was reviewed as a load-bearing boundary dependency rather than as primary scope.

Scope caveat: the review was performed strictly against a local, read-only snapshot pinned to the commit above. No live networks or third-party systems were contacted, and all proof-of-concept tests ran in an isolated scratch clone so the audited tree and its git state stayed pristine.

The assessment surfaced no Critical and no High severity issues. It produced 4 Medium, 19 Low, and 16 Informational findings, supported by 54 Test Plan cases (39 Fail, 15 Pass).

1.1 Key Themes Observed Across Findings

- **Freshness accounting.** The read path measures staleness from the cache write time rather than the source publish time, so a served price can be nearly twice the configured max_age_secs old (MED-001), and the freshness bound itself is optional (LOW-002).
- **Quorum protects availability, not Byzantine integrity.** Under degraded quorum a single surviving source can determine the MedianLow price (MED-002), and one resource-exhausting source can abort an entire refresh transaction (MED-004).
- **Circuit breakers stop sudden jumps, not gradual drift.** A steady sub-threshold ramp evades all three breaker rules (LOW-019), a limitation PROOF.md documents as bounded only by multi-source quorum.
- **Boundary conversions and adapter metadata.** A strictly positive aggregated price can reach a SEP-40 consumer as a valid zero (MED-003), and adapter base and resolution are not bound to the parent feed (LOW-011, LOW-016).
- **Governance is timelocked and sound.** Role-based access control, last-Admin protection, and execute atomicity all held. The residual items are a non-Admin timelock ratchet (LOW-018) and two shared-kernel API robustness gaps that are not reachable on-chain (INFO-016, INFO-017).

1.2 Main Recommendations

- Bound reads by both the cache age and the source publish time against `max_age_secs`, and require a freshness bound for production feeds.
- Treat `min_sources` as an availability threshold. For Byzantine tolerance require at least three valid survivors, fail closed on excessive spread, and isolate per-source work so one source cannot abort a batch refresh.
- Add a cumulative-drift breaker that bounds deviation from a long-window baseline so a slow ramp accumulates into a trip, and disallow a `min_sources` of one for manipulation-resistant feeds.
- Make `normalized_to_sep40` fail closed (or round up) rather than emit a zero price, and bind adapter base and resolution to the parent runtime.
- Require the Admin role for `SetActionTtl` over Admin operations, and add validating constructors to the shared kernels.

2. Test Methodology And Approach

The review combined an exhaustive line-by-line manual audit with a dynamic proof-of-concept campaign. Manual review proceeded through eleven phases: reconnaissance and threat model (Phase 0); the numeric foundation and Soroban decimal wire type (Phase 1); untrusted source ingestion and conversion (Phase 2); aggregation and the freshness filter (Phase 3); circuit-breaker math and rules (Phase 4); the circuit-breaker set state machine, sampling, and history (Phase 5); the runtime contract entrypoints, cache, storage, and TTL (Phase 6); the SEP-40 adapter (Phase 7); the governance proposal-lifecycle kernel (Phase 8); governance common, the action union, roles, and TTL policy (Phase 9); the governance contract RBAC, dispatch engine, and sub-invocation authorization (Phase 10); and configuration, dependencies, build gates, and existing test coverage (Phase 11). Each suspected defect became a falsifiable hypothesis, then a test proved or refuted it in an isolated scratch clone.

3. Technical Profile

3.1 Document Purpose

This section describes the components, authority model, and data flow that the assessment relied on. It is background for the findings and not a findings report.

3.2 System Summary

The proxy oracle normalizes upstream SEP-40 price feeds into a single price per asset and serves it to downstream consumers. It is built from two layers: chain-agnostic kernels that own the security logic, and thin Soroban runtime contracts that own storage, authorization, and the host boundary.

3.3 Components

- **Oracle kernel** (kernel/): the Price type and scaled comparison, weighted-median and priority aggregation, the freshness filter, and the circuit-breaker math, rules (StepwiseChange, MonotonicRun, WindowedChangeDelta), set state machine, and ring buffer.
- **Governance kernel** (governance-kernel/): a storage-agnostic and authorization-agnostic proposal ledger that owns proposal ordering, maturity, and anti-replay.
- **Runtime proxy oracle** (soroban/contract/): the deployable contract exposing the permissionless refresh, the owner-gated configuration mutators, the per-asset cache, history, and TTL management, and the fail-closed read entrypoints.
- **Governance contract** (soroban/governance-contract/): stellar-access role-based access control, the proposal create, cancel, and execute engine, and the sub-invocation path that turns a matured proposal into a privileged runtime call.
- **SEP-40 adapter** (soroban/sep40-adapter-contract/): a per-feed read surface that rescales the runtime price to a configured decimals and re-exposes it in SEP-40 form.
- **Shared crates**: soroban/common (DTOs, the SorobanDecimal wire type, normalized_to_sep40, and ContractError) and soroban/governance-common (the GovernanceAction union, Role, OperationKind, the TtlConfig macro, required_role, and validate_action).

3.4 Authority And Trust Model

The runtime proxy oracle is owned by the governance contract. Every state-mutating runtime entrypoint is owner-gated, while refresh is permissionless. The governance contract enforces a multi-member Admin role with last-Admin protection, plus three operational roles: ProxyConfigurationManager for price-source and breaker configuration, CircuitBreakerOperator for rearm and enforcement, and ManualTripper for manual trip. Every privileged action flows through a timelocked proposal. Upstream source feeds are trusted to report honestly, yet the design (see PROOF.md) explicitly aims to tolerate a compromised source.

3.5 Data Model And Deployment Front Door

Per-asset persistent storage holds the proxy configuration, the postcard-serialized circuit-breaker set, the accepted-price cache, and a bounded history of 32 records. Downstream consumers read through the SEP-40 adapter (lastprice, price, prices), which calls the runtime aggregated_latest. Keepers drive prices by calling the permissionless refresh, which pulls each configured source, aggregates, evaluates breakers, and writes the cache.

4. Execution Flows

4.1 Document Purpose

This section walks the major operations end to end, naming the inputs, internal steps, authorization required, validation and rejection points, and outputs that the assessment exercised.

4.2 Refresh (Permissionless Price Update)

1. A caller invokes refresh with a list of assets. No authorization is required.
2. For each configured source the runtime reads base, decimals, and lastprice through the generated PriceFeedClient. Host or contract errors are swallowed to None, and a base mismatch drops the source.
3. Surviving observations are converted to the kernel Price form (an i128 to i64 downscale with exponent tracking), filtered for freshness and validity, checked against min_sources, aggregated with MedianLow, and evaluated by the circuit-breaker set.
4. On acceptance the price and its updated_at timestamp are cached and pushed to history. On a breaker block or resolve failure the cache is overwritten with a non-accepted status, so subsequent reads fail closed.

4.3 Read (Aggregated_latest And The SEP-40 Adapter)

1. aggregated_latest returns the cached accepted price only when now minus updated_at is within max_age_secs, or unconditionally when max_age_secs is None.
2. The SEP-40 adapter gates on asset match, calls aggregated_latest, and rescales the normalized price to its configured decimals through normalized_to_sep40, returning None on conversion error.

4.4 Governance Proposal Lifecycle

1. create_proposal authenticates the caller, checks Admin or the action required role, computes effective_ttl (the maximum of the requested TTL and the operation minimum, capped at 180 days), reserves the next id, and stores the proposal body.
2. execute_proposal reloads the body by id, re-checks authorization against the proposal's own operation, enforces maturity in the kernel, then dispatches the side effect through a narrowly scoped authorize_as_current_contract sub-invocation, all inside one atomic transaction.
3. cancel_proposal removes a pending proposal after the same authorization check.

5. Attack Surface

5.1 Document Purpose

This section enumerates the entry points where untrusted data or callers enter the system, what the caller controls at each, and the deployment boundary.

5.2 Entry Points

- **Permissionless refresh.** Any caller can invoke refresh for any registered asset and choose the cadence. The caller does not control source values directly but controls timing (sampling, breaker-state churn, and event volume) and which assets are batched together.
- **Upstream source feeds.** A configured source contract controls the values and timestamps it returns for base, decimals, and lastprice, including extreme i128 values, large decimals, a mismatched base, future or zero timestamps, None, and trapping or resource-exhausting behavior. Source compromise is inside the threat model.
- **Governance proposal payloads.** A role holder submits GovernanceAction payloads covering proxy and breaker configuration, roles, TTLs, ownership, and upgrade. Payloads are validated statically by `validate_action` and again by the runtime sub-call, which fails the whole transaction on rejection.

5.3 Deployment Boundary

The SEP-40 adapter is the read-side surface that downstream consumers such as Blend actually call. The boundary rests on the governance contract owning the runtime, the Soroban host authorization model (`require_auth` and `authorize_as_current_contract` with empty sub-invocations), and the periodic TTL extension the design assumes for liveness.

6. Threat Model

6.1 Document Purpose

This section states what the system must protect, the actors and their trust level, the trust boundaries and assumptions, and the assets at risk. Assumptions that testing showed do not fully hold are called out explicitly.

6.2 What Must Be Protected

The integrity and freshness of the price served to lending markets, and the availability of governance control over the oracle.

6.3 Actors And Trust

- Unauthenticated callers, through the permissionless refresh.
- Configured source feeds, trusted to be honest but assumed possibly compromised by design.
- Operational role holders (ProxyConfigurationManager, CircuitBreakerOperator, ManualTripper), semi-trusted and timelocked.
- The multi-member Admin, trusted and timelocked, with last-Admin protection.

6.4 Trust Boundaries And Assumptions

- **Multi-source quorum resists a single compromised source.** This holds at full quorum but not under degraded quorum with MedianLow (MED-002), and min_sources may be configured as one. Testing showed this assumption does not fully hold.
- **Circuit breakers stop the manipulation class from the Blend incident.** They stop sudden jumps, but a steady sub-threshold ramp evades all three rules (LOW-019). Testing showed this assumption holds only for abrupt moves.
- **Reads are fail-closed and fresh.** Reads do fail closed on a breaker block or failed refresh, but freshness is measured from the cache write time and can reach nearly twice max_age_secs (MED-001), and the bound is optional (LOW-002).
- **Governance is timelocked and recoverable.** Confirmed, except that a non-Admin ProxyConfigurationManager can inflate Admin-operation timelocks, and the anti-fast-lowering rule then delays restoration (LOW-018).

6.5 Assets At Risk

Downstream collateral valuations and liquidations that consume the served price, and the timeliness of governance incident response.

7. Dynamic Testing And Proof-Of-Concept Campaign

7.1 Document Purpose

This section describes the dynamic approach, the coverage, what it looked for, and the verdicts and refutations.

7.2 Approach And Coverage

Every suspected defect became a falsifiable hypothesis, proven or refuted by a test that ran in an isolated scratch clone path-dependent on copies of the audited crates, so the audited tree stayed pristine. The campaign reused the kernel rtest and property suites and the Soroban testutils integration harness (Bootstrap, a mock upstream oracle, and ledger advancement helpers). The shipped suites all pass at the pinned commit (kernel 210, runtime 59, adapter 21, the governance kernel, common, and contract suites, and 68 cross-contract integration tests).

7.3 Results

The Test Plan records 54 cases: 39 Fail linked to findings and 15 Pass controls. Confirmed issues were reproduced with proof-of-concept tests, including a 50,000-case property model of the governance maturity predicate, an end-to-end walk of the SEP-40 lastprice past all three enforced breakers from an external attacker perspective, and a budget-exhaustion abort of an entire refresh.

7.4 Refutations Recorded As Controls

Several suspected issues were disproven and recorded as Pass controls: the read path fails closed on a breaker block; proposal execution is atomic and cannot be replayed; last-Admin removal is rejected; the anti-fast-lowering rule blocks shortening a sensitive operation timelock faster than that operation itself; an evicted proposal body does not permanently strand its pending slot because persistent storage is archived and restorable; and the deployed release-soroban-checked build profile compiles the in-scope crates with overflow-checks enabled, which validates the arithmetic panic-safety assumption behind the numeric findings. Independent parallel reviews of the governance and cross-contract surfaces corroborated these refutations and surfaced no additional Critical or High issue.

8. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

8.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker’s control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

8.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75
	Critical (Y:C)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

8.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (<i>C</i>)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (<i>r</i>)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (<i>s</i>)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient *C* is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score *S* is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

9. SCOPE

REMEDATION COMMIT ID: ^

- e8b542d
- fb16191
- 4d9e432
- b289b21
- 5c93031
- bcf357f
- 6d12edf
- dd566da

Out-of-Scope: New features/implementations after the remediation commit IDs.

10. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDiation DATE
READ-SIDE FRESHNESS MEASURES CACHE-WRITE TIME, NOT SOURCE TIME (EFFECTIVE STALENESS UP TO ABOUT 2X MAX_AGE_SECS)	MEDIUM	SOLVED - 08/08/2026
DEGRADED QUORUM LETS A SINGLE MALICIOUS SURVIVING SOURCE DETERMINE THE MEDIANLOW PRICE	MEDIUM	SOLVED - 08/08/2026
POSITIVE AGGREGATED PRICE CAN BE DELIVERED TO SEP-40 CONSUMERS AS A VALID ZERO PRICE	MEDIUM	SOLVED - 08/08/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
A SINGLE BUDGET-EXHAUSTING CONFIGURED SOURCE ABORTS THE ENTIRE REFRESH TRANSACTION	MEDIUM	SOLVED - 08/08/2026
SOURCE-OUTAGE CACHE OVERWRITE CONTRADICTS THE DOCUMENTED LAST-PRICE GRACE WINDOW	LOW	SOLVED - 08/08/2026
FRESHNESS BOUNDS ARE OPTIONAL AND UNENFORCED, ALLOWING STALE-FOREVER READS AND FUTURE-DATED PRICES	LOW	SOLVED - 08/08/2026
NONZERO CIRCUIT-BREAKER THRESHOLD SILENTLY PERSISTS AS ZERO THROUGH THE POSTCARD DECIMAL ROUND-TRIP	LOW	SOLVED - 08/08/2026
MISSING BREAKER STORAGE IS TREATED AS AN EMPTY UNTRIPPED SET INSTEAD OF FAILING CLOSED	LOW	SOLVED - 08/08/2026
ONE ORACLE ADDRESS CAN FILL MULTIPLE QUORUM SLOTS VIA DISTINCT ASSET ALIASES	LOW	SOLVED - 08/08/2026
SOURCE-CONTROLLED TIMESTAMPS CAN REGRESS, REWRITE, OR STARVE THE SEP-40 HISTORY	LOW	SOLVED - 08/08/2026
ACCEPTED EXTREME PRICE SCALE POISONS THE ENTIRE ADAPTER PRICES() BATCH	LOW	SOLVED - 08/08/2026
CONFIG VALIDATION ACCEPTS INERT CIRCUIT BREAKERS THAT CAN NEVER TRIP	LOW	SOLVED - 08/08/2026
UNBOUNDED REGISTERED-ASSET LIST CAN MAKE GLOBAL EXTEND_TTL() EXCEED TRANSACTION RESOURCE LIMITS	LOW	SOLVED - 08/08/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NONZERO SAMPLE_INTERVAL LETS MONOTONICRUN MISS ACCEPTED INTRA-INTERVAL STAGED RAMPS	LOW	SOLVED - 08/08/2026
CIRCUIT-BREAKER SUITE HAS NO CUMULATIVE-DRIFT DEFENSE: A STEADY SUB-THRESHOLD PRICE RAMP EVADES ALL THREE RULES	LOW	SOLVED - 08/08/2026
SETACTIONTTL (PROXYCONFIGURATIONMANAGER) CAN RETIME ANY OPERATION INCLUDING ADMIN OPS, RATCHETING ADMIN-RECOVERY TIMELOCKS	LOW	SOLVED - 08/08/2026
SEP-40 ADAPTER CAN RELABEL A PARENT PRICE UNDER AN UNRELATED BASE WITHOUT DENOMINATION CONVERSION	LOW	SOLVED - 08/08/2026
WINDOWEDCHANGEDELTA IGNORES EXTREME INTERIOR- WINDOW OUTLIERS THAT RETURN TO THE WINDOW BASELINE	LOW	SOLVED - 08/08/2026
DECIMAL SATURATION ALIASES DISTINCT EXTREME WINDOW MOVES AND MAKES WINDOWEDCHANGEDELTA MISS THEM	LOW	SOLVED - 08/08/2026
REARMING ONE BREAKER CLEARS OR REPLACES THE SHARED ACCEPTED BASELINE USED BY EVERY OTHER BREAKER	LOW	SOLVED - 08/08/2026
REARMCONFIG.ARMED_AFTER_SECS IS INTERPRETED AS AN ABSOLUTE TIMESTAMP WHILE DOCS AND TESTS USE IT AS A RELATIVE GRACE DURATION	LOW	SOLVED - 08/08/2026
SEP-40 ADAPTER TIMESTAMPS ARE NOT ALIGNED TO THE ADVERTISED RESOLUTION	LOW	SOLVED - 08/08/2026
RENOUNCING ADAPTER OWNERSHIP DOES NOT DECOMMISSION IT AND MAKES FUTURE SHUTDOWN IMPOSSIBLE	LOW	SOLVED - 08/08/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDiation DATE
GOVERNANCE EXTEND_TTL IS PERMISSIONLESS BUT DOCUMENTED AS ADMIN-ONLY	INFORMATIONAL	SOLVED - 08/08/2026
REFRESHFAILURE TELEMETRY COLLAPSES DISTINCT SOURCE FAILURES; DOCUMENTED CODES 2 AND 4 ARE UNREACHABLE	INFORMATIONAL	SOLVED - 08/08/2026
PRICEIDENTIFIER JSON SCHEMA CONTRADICTS ITS SERDE HEX-STRING WIRE FORMAT	INFORMATIONAL	SOLVED - 08/08/2026
SOURCE OUTAGE MASKS AN ACTIVE MANUAL TRIP IN REFRESH TELEMETRY (STATUS PRECEDENCE)	INFORMATIONAL	SOLVED - 08/08/2026
SHARED WEIGHTED-MEDIAN KERNEL COUNTS ZERO-WEIGHT SOURCES TOWARD MIN_SOURCES	INFORMATIONAL	SOLVED - 08/08/2026
DECIMAL::FROM_STR ACCEPTS TRAILING GARBAGE IN FRACTIONAL STRINGS	INFORMATIONAL	SOLVED - 08/08/2026
A FRESH CIRCUIT-BREAKER TRIP TRIPS AND REPORTS ONLY THE FIRST BLOCKING BREAKER	INFORMATIONAL	SOLVED - 08/08/2026
NO-OP SET_MANUAL_TRIP EMITS A MANUALTRIPSET EVENT AND CLEARS THE CACHE, CONTRADICTING THE DOCS	INFORMATIONAL	SOLVED - 08/08/2026
SHARED GOVERNANCE KERNEL CREATE() ACCEPTS A TTL BELOW THE OPERATION MINIMUM (TIMELOCK ENFORCEMENT NOT OWNED BY THE KERNEL)	INFORMATIONAL	SOLVED - 08/08/2026
SHARED GOVERNANCE KERNEL GOVERNANCE HAS NO VALIDATING CONSTRUCTOR: A CORRUPT LEDGER CAN DOUBLE-CONSUME A PROPOSAL ID	INFORMATIONAL	SOLVED - 08/08/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDiation DATE
DECIMAL::FROM_STR SILENTLY WRAPS SCHEMA-VALID WHOLE VALUES AT 2^384 INSTEAD OF REJECTING THEM	INFORMATIONAL	SOLVED - 08/08/2026
CIRCUITBREAKERTRIPPED TELEMETRY DROPS THE NORMALIZED EXPONENT AND SOURCE PUBLISH TIMESTAMP	INFORMATIONAL	SOLVED - 08/08/2026
WEIGHTED MEDIANHIGH CAN SELECT A MINORITY OR ZERO-WEIGHT SOURCE	INFORMATIONAL	SOLVED - 08/08/2026
CONSERVATIVE DECIMAL SCALING CAN FALSELY TRIP A BREAKER FOR A REPRESENTABLE BELOW-THRESHOLD MOVE	INFORMATIONAL	SOLVED - 08/08/2026
MONOTONICRUN EXCLUDES STEPS EXACTLY EQUAL TO ITS DOCUMENTED MINIMUM THRESHOLD	INFORMATIONAL	SOLVED - 08/08/2026
SUCCESSFUL SEP-40 ADAPTER UPGRADES EMIT NO ADAPTER UPGRADE EVENT	INFORMATIONAL	SOLVED - 08/08/2026

11. FINDINGS & TECH DETAILS

11.1 READ-SIDE FRESHNESS MEASURES CACHE-WRITE TIME, NOT SOURCE TIME (EFFECTIVE STALENESS UP TO ABOUT 2X MAX_AGE_SECS)

// MEDIUM

Description

The proxy oracle applies its freshness filter (`max_age_secs`) to a source's own `publish_time` only at refresh time, when the price is aggregated. When an aggregated price is accepted, the runtime caches it with `updated_at` set to the current ledger time, not to the source's timestamp.

Every read then measures staleness against `cached.updated_at` in `cached_accepted_no_older_than`, and never against the source's own `timestamp`. A price that was already almost `max_age_secs` old at the moment it was accepted therefore stays readable for another full `max_age_secs` window.

The effective staleness a consumer can observe is bounded by about 2x `max_age_secs` (source age at acceptance plus the post-acceptance cache window), and only while no intervening failed or blocked refresh overwrites the cache entry.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs
pub fn cached_accepted_no_older_than(
    cached: &CachedProxyPrice,
    max_age_secs: u64,
    now: u64,
) -> Option<NormalizedPrice> {
    let CachedStatus::Accepted(price) = &cached.status else {
        return None;
    };
    // Bound staleness by the cache update time, not the upstream source's
    // claimed timestamp.
    if now.saturating_sub(cached.updated_at) > max_age_secs {
        return None;
    }
    Some(price.clone())
}
```

Impact

A consumer can receive a price up to about twice as old as the configured freshness bound. A protocol that sized `max_age_secs` as its staleness risk tolerance actually tolerates up to 2x that age, which can matter for liquidation and valuation decisions in fast markets.

Proof of Concept

Confirmed by an executed test against the pinned commit. The test admits a source observation that is already `max_age_secs` old, then reads it again a full `max_age_secs` later and shows the read still succeeds with the original source timestamp.

```
running 1 test
test tests::reaudit_cache_freshness_window_can_double_source_age ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Apply the audit PoC patch and run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_cache_freshness_window_can_double_source_age`.

PoC Script

```
#[test]
fn reaudit_cache_freshness_window_can_double_source_age() {
    let (env, proxy, source, asset) = setup(); // max_age_secs = 30, now = 100
    // Source value is already 30s old (ts=70) but still admitted.
    source.set_price(&asset, &5_000_000_000_i128, &70_u64);
    let accepted = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
    assert!(matches!(accepted.get(0).unwrap().1, RefreshStatus::Accepted(_)));

    // 30s after the refresh the observation is 60s old, but the cache entry
    // is only 30s old, so the read still returns it.
    set_ledger(&env, 130);
    let returned = proxy.aggregated_latest(&asset).unwrap();
    assert_eq!(returned.timestamp, 70);
    assert_eq!(130_u64 - returned.timestamp, 60);
}
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

To mitigate this issue, the following improvements are recommended:

- In `cached_accepted_no_older_than`, reject the cached entry when either `now - cached.updated_at` or `now - price.timestamp` exceeds `max_age_secs`.
- Keep the existing `updated_at` bound so future-dated source timestamps cannot appear indefinitely fresh.

Until fixed, operators should size `max_age_secs` at roughly half their true staleness tolerance to account for the doubled window.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `cached_accepted_no_older_than` also reject the cache when the source's own `price.timestamp` exceeds `max_age_secs`, not just the cache-write time, closing the window that allowed prices up to 2x stale.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/e8b542dc54f5b49871c6ce2e82c21012ef723216>

11.2 DEGRADED QUORUM LETS A SINGLE MALICIOUS SURVIVING SOURCE DETERMINE THE MEDIANLOW PRICE

// MEDIUM

Description

The Soroban runtime always aggregates with `MedianLow` and treats `min_sources` purely as an availability threshold: `refresh` only short-circuits when every source returns nothing, and any surviving prices are handed to the median regardless of how many honest sources dropped out.

With the documented hardening configuration of three sources and `min_sources = 2`, a single honest source becoming unavailable, stale, or malformed leaves exactly two survivors. `MedianLow` of two values returns the lower of the two, so one compromised source that reports a low price fully determines the accepted result, and the runtime still emits `RefreshSuccess`.

The bound: this requires one compromised configured source plus one honest source degraded at the same time, and breakers can still catch the move once a healthy accepted-history baseline exists. It does not let an attacker move the price at full quorum, and the MedianLow bias means the attacker can push the result down more easily than up. This contradicts the PROOF.md claim that one compromised source cannot determine the price in this configuration.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs (only the all-None case is short-circuited)
if prices.iter().all(Option::is_none) {
    return RefreshStatus::SourceUnavailable;
}
// kernel/src/proxy/aggregator/method/median/mod.rs (MedianLow of survivors)
let min_sources = self.min_sources.max(1);
let valid_sources = values.len() / 2;
if valid_sources < min_sources as usize { /* error */ }
```

Impact

One compromised source gains asymmetric downward control of the feed during any single-source outage. The quorum check protects price integrity only at full participation; it provides no Byzantine tolerance in the degraded state that partial outages routinely produce.

Proof of Concept

Confirmed by an executed test. Three sources with `min_sources = 2`; one honest source offline; the malicious survivor dictates the median and the honest survivor cannot pull it up.

 Copy Code

```
running 1 test
test tests::reaudit_degraded_two_of_three_quorum_gives_low_survivor_full_control ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_degraded_two_of_three_quorum_gives_low_survivor_full_control`.

PoC Script

 Copy Code

```
// 3 sources, min_sources = 2, one honest source offline.
malicious.set_price(&asset, &1_i128, &100_u64);
honest.set_price(&asset, &100_i128, &100_u64);
let downward = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
// Accepted, and MedianLow of the two survivors returns the attacker value 1.
assert_eq!(downward_mantissa, 1);
// Reverse case: attacker high, honest 100 -> MedianLow returns 100 (no upward control).
malicious.set_price(&asset, &1_000_i128, &100_u64);
assert_eq!(upward_mantissa, 100);
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Treat `min_sources` explicitly as an availability threshold and, for one-fault Byzantine tolerance, require at least three valid survivors before accepting (or fail closed when only two survive).
- Consider rejecting results with excessive spread between survivors, and document that MedianLow biases downward.
- Correct PROOF.md to describe the degraded-quorum behavior.

Until fixed, operators should not rely on quorum as manipulation protection during partial source outages, and should keep tight StepwiseChange breakers armed.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by requiring `min_sources ≥ 3` in `validate_proxy_config`, so a single source outage on the recommended 3-source setup now leaves too few survivors and the refresh fails closed instead of accepting a manipulable 2-source `MedianLow`. `PROOF.md` was also corrected to reflect this.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/fb16191412cf1168abe333550c84844503dcab5c>

11.3 POSITIVE AGGREGATED PRICE CAN BE DELIVERED TO SEP-40 CONSUMERS AS A VALID ZERO PRICE

// MEDIUM

Description

The kernel guarantees that every accepted aggregated price is strictly positive. The SEP-40 adapter re-scales that price to its own `decimals` through `normalized_to_sep40`. When the target scale is negative (adapter decimals smaller than the price's precision), the conversion performs an i128 truncating division by `10|scale|` with no check that a nonzero input stays nonzero.

A small but strictly positive price therefore truncates to exactly `0` and is returned as `Ok(PriceData { price: 0 })`. The adapter only maps `Err` to `None`, so the zero is surfaced to consumers as a valid, non-`None` price rather than a fail-closed absence.

The bound: this requires a true price smaller than one unit at the adapter's configured decimals (a genuinely tiny price, or a small manipulated median on a single-source or degraded feed) and a downstream consumer that treats zero as an actionable price. It is a per-adapter, configuration-dependent boundary issue, not a global mispricing.

Code Location

 Copy Code

```
// soroban/common/src/lib.rs : normalized_to_sep40 (negative-scale branch)
} else {
    value /= 10_i128
        .checked_pow(scale.unsigned_abs())
        .ok_or(ContractError::ConversionOverflow)?;
}
Ok(PriceData { price: value, timestamp: price.timestamp })
```

Impact

The strictly-positive-price invariant is lost at the final SEP-40 hop. A consumer can receive a zero price and use it for valuation, liquidation, or as a divisor, which can cause wrong liquidations, zero valuations, or divide-by-zero downstream.

Proof of Concept

Confirmed by two executed tests: the pure conversion returns `Some(0)` for a positive input, and the adapter's `lastprice` surfaces that zero end to end.

 Copy Code

```
running 2 tests
test tests::reaudit_sep40_downscale_can_return_some_zero_for_positive_price ... ok
test tests::reaudit_med003_adapter_lastprice_returns_some_zero_for_positive_price ... ok

test result: ok. 2 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_sep40_downscale_can_return_some_zero_for_positive_price` and the adapter test `reaudit_med003_adapter_lastprice_returns_some_zero_for_positive_price`.

PoC Script

 Copy Code

```
// Pure conversion: positive input, adapter decimals below its precision.
let normalized = NormalizedPrice { mantissa: 1, expo: -8, timestamp: 100 };
assert_eq!(normalized_to_sep40(&normalized, 7).unwrap().price, 0);

// End to end through the adapter (decimals = 7):
let p = f.adapter.lastprice(&f.asset).expect("Some, not None");
assert_eq!(p.price, 0);
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

To mitigate this issue, the following improvements are recommended:

- In `normalized_to_sep40`, return an error (or round up to the smallest representable unit) when a nonzero input would truncate to zero at the target scale.
- Have adapters treat a zero result as fail-closed (`None`) rather than a valid price.

Until fixed, integrators must reject a zero price from the adapter instead of acting on it, and adapters should be configured with enough decimals for the asset's price range.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by wrapping `normalized_to_sep40` in a new `normalized_to_adapter` function that rejects the conversion when a nonzero price truncates to zero at the adapter's decimals, which every adapter method (`price`, `prices`, `lastprice`) now propagates as a fail-closed `None`.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

11.4 A SINGLE BUDGET-EXHAUSTING CONFIGURED SOURCE ABORTS THE ENTIRE REFRESH TRANSACTION

// MEDIUM

Description

`compute_refresh` calls each configured source (`base`, `decimals`, `lastprice`) up front, before the quorum check. Ordinary source failures (panic, wrong ABI, wrong base, no price) are caught by the `try_*` cross-contract calls and are treated as an unavailable source, so quorum can still be met by the other sources.

Host resource exhaustion is different. If a source consumes the transaction's CPU or memory budget, the host raises `Error(Budget, ExceededLimit)`, which is an unrecoverable trap that `try_invoke_contract` does not catch. It escalates to a panic that unwinds the whole `refresh` transaction, so a healthy source that alone satisfies `min_sources` does not save the call.

The bound: this requires one of the governance-configured source contracts to be malicious or compromised (source compromise is explicitly in the project's threat model). It is a denial-of-service only; it cannot produce a wrong price or move funds. The blast radius is the asset plus every other asset batched in the same `refresh([])` call, and it persists until governance removes the source.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs : every source is called before quorum
let mut prices = AllocVec::with_capacity(config.sources.len() as usize);
for source in config.sources.iter() {
    prices.push(source_kernel_price(env, source, &expected_base));
}
```

Impact

One hostile configured source can deny refresh to the asset and to every asset in the same batch.

Because `refresh` is permissionless, no caller can route around the bad source; reads fail closed until a governance proposal removes it. Quorum protects price integrity but provides no availability protection against a single hostile source.

Proof of Concept

Confirmed by an executed test. A source that hashes in a loop until the budget is exhausted, configured alongside a healthy source that alone meets quorum, aborts the whole `refresh` with a budget trap rather than returning the healthy price.

 Copy Code

```
total footprint / CPU budget exceeded
thread '...reaudit_w4_budget_exhausting_source_aborts_whole_refresh' panicked:
HostError: Error(Budget, ExceededLimit)
  [Diagnostic Event] ["contract try_call failed", base, []]
  [Diagnostic Event] ["contract call failed", refresh, [[[Other, BTC]]]]
test ...reaudit_w4_budget_exhausting_source_aborts_whole_refresh - should panic ... ok
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace; soroban-sdk 25 testutils enforce mainnet-style resource limits.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_w4_budget_exhausting_source_aborts_whole_refresh`.

PoC Script

 Copy Code

```
// GriefSource::base() runs ~2,000,000 sha256 host calls, exceeding the budget.
let grief_id = env.register(      GriefSource, ());
// sources = [grief, healthy], min_sources = 1
healthy.set_price(&asset, &5_000_000_000_i128, &100_u64);
// Panics with Error(Budget, ExceededLimit) on the refresh frame:
let _ = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Refresh a single asset per transaction (or cap the batch) so a hostile source cannot take down unrelated assets in one call.
- If and when the host supports sub-invocation resource caps, bound per-source work.
- Document that source availability, not just integrity, depends on every configured source being non-hostile, and keep a fast governance path to remove a source.

Until fixed, operators should monitor per-source refresh cost and be ready to remove a misbehaving source promptly.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by changing `refresh` from a batched `Vec<Asset>` entrypoint to a single-asset call (`refresh(env: Env, asset: Asset) -> RefreshStatus`), so a budget-exhausting hostile source can only abort the transaction for its own asset and can no longer take down unrelated assets refreshed in the same call. This is confirmed by the new test `hostile_source_cannot_abort_a_separate_asset_refresh`.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.5 SOURCE-OUTAGE CACHE OVERWRITE CONTRADICTS THE DOCUMENTED LAST-PRICE GRACE WINDOW

// LOW

Description

`apply_refresh` maps every refresh status except `UnknownAsset` to a cache write. A `Blocked`, `ResolveFailed`, or `SourceUnavailable` refresh therefore overwrites a previously `Accepted` cache entry with a non-accepted status and `updated_at = now`.

Because `aggregated_latest` returns a price only for `CachedStatus::Accepted`, the feed returns `None` on the very first failed or blocked refresh after the last good one. RUNBOOK section 10 states that the cache holds the last accepted price until `max_age_secs` elapses; the code provides no such grace window. The bound: this is the fail-closed (safe) direction and never serves a wrong or stale price. The issue is a documentation and availability-expectation mismatch, not a mispricing. An attacker cannot fabricate a source outage or breaker trip; the flip to `None` only happens when a refresh genuinely fails or blocks.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs
fn status_to_cached(status: &RefreshStatus) -> Option<CachedStatus> {
  match status {
    RefreshStatus::UnknownAsset => None,
    RefreshStatus::Accepted(price) => Some(CachedStatus::Accepted(price.clone())),
    RefreshStatus::Blocked(code) => Some(CachedStatus::Blocked(*code)),
    RefreshStatus::ResolveFailed(code) => Some(CachedStatus::ResolveFailed(*code)),
    RefreshStatus::SourceUnavailable => Some(CachedStatus::ResolveFailed(SOURCE_UNAVAILABLE_CODE)),
  }
}
```

Impact

A dependent market can lose its price immediately on any failed or blocked refresh, with no grace window. Integrators who design around the documented behavior may pause unexpectedly; because `refresh` is permissionless, anyone can trigger the overwrite during a genuine source outage. No wrong-price or fund-loss vector.

Proof of Concept

Confirmed by the project's own shipped test, which asserts that a source outage overwrites the accepted cache and the read then returns `None`.

 Copy Code

```
running 1 test
test tests::failed_refresh_overwrites_accepted_cache_fail_closed ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils failed_refresh_overwrites_accepted_cache_fail_closed`.

PoC Script

 Copy Code

```
// Accept a price, then clear the source and refresh again.
source.set_price(&asset, &5_000_000_000_i128, &100_u64);
proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
source.clear_price(&asset);
let result = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(matches!(result.get(0).unwrap().1, RefreshStatus::SourceUnavailable));
assert_eq!(legacy_lastprice(&proxy, &asset), None); // no grace window
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, choose one of the following:

- Update RUNBOOK section 10 to state that any failed or blocked refresh overwrites the accepted cache and reads fail closed immediately.
- Or, if the grace window is intended, stop overwriting an existing `Accepted` entry on `SourceUnavailable`/`ResolveFailed` and let `max_age_secs` age it out (which then requires `max_age_secs` to be mandatory, see the freshness-enforcement finding).

Until fixed, integrators should not rely on a post-outage grace window.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by correcting `RUNBOOK.md` section 10 (and `AUDIT.md`) to state that any failed or blocked refresh immediately replaces the cached result and reads fail closed, removing the incorrect claim of a post-outage grace window. The fail-closed code behavior itself was already safe and required no change.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.6 FRESHNESS BOUNDS ARE OPTIONAL AND UNENFORCED, ALLOWING STALE-FOREVER READS AND FUTURE-DATED PRICES

// LOW

Description

`ProxyConfig.max_age_secs` and `max_clock_drift_secs` are both `Option` values. If a proxy is configured with `max_age_secs = None`, `aggregated_latest` uses `u64::MAX` as the staleness bound, so a cached accepted price is served indefinitely with no read-time freshness check. If `max_clock_drift_secs = None`, the freshness filter accepts arbitrarily future-dated source prices at refresh time.

Neither the runtime's `validate_proxy_config` nor the governance `validate_action` requires these fields to be present, even though the README, AUDIT, and RUNBOOK all describe freshness as mandatory for production.

The bound: this needs a governance misconfiguration (or an incomplete migration) that omits `max_age_secs`; it is not directly attacker-triggerable. It is a guard-rail gap rather than an exploit primitive.

Code Location

 Copy Code

```
// soroban/contract/src/lib.rs : aggregated_latest
let max_age = proxy_config.max_age_secs.unwrap_or(u64::MAX);
cached_accepted_no_older_than(&cached, max_age, env.ledger().timestamp())
// soroban/contract/src/conversion.rs : validate_proxy_config never requires max_age_secs
```

Impact

A misconfigured feed can serve an arbitrarily old price as if fresh. Downstream markets could consume a very stale last-good price (up to eviction) with no read-time staleness protection, defeating a core oracle safety property.

Proof of Concept

Confirmed by the project's own shipped test, which documents that with `max_age_secs = None` the cached price is returned regardless of age.

 Copy Code

```
running 1 test
test tests::missing_config_lastprice_no_freshness_limit_is_documented_exception ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.

- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils missing_config_lastprice_no_freshness_limit_is_documented_exception`.

PoC Script

 Copy Code

```
// Proxy configured with max_age_secs = None.  
proxy.set_proxy(&asset, &ProxyConfig { sources, min_sources: 1,  
    max_age_secs: None, max_clock_drift_secs: None });  
source.set_price(&asset, &5_000_000_000_i128, &100_u64);  
proxy.refresh(&Vec::from_array(&env, [asset.clone()]));  
// Advance far past any normal max_age; the price is still returned.  
set_ledger(&env, 999_999);  
assert!(legacy_lastprice(&proxy, &asset).is_some());
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Require `max_age_secs.is_some()` (and consider requiring `max_clock_drift_secs`) in `validate_proxy_config`, and mirror the check in governance `validate_action`.
- Alternatively, make `aggregated_latest` fail closed when `max_age_secs` is `None` instead of defaulting to `u64::MAX`.

Until fixed, operators must always set explicit freshness bounds for production feeds.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `validate_proxy_config` require both `max_age_secs` and `max_clock_drift_secs` to be `Some` (rejecting `None` with `InvalidInput`, and bounding both to sane maximums), a check now shared by governance's `validate_action`. As defense in depth, `aggregated_latest` also fails closed via `proxy_config.max_age_secs?` instead of defaulting to `u64::MAX`.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.7 NONZERO CIRCUIT-BREAKER THRESHOLD SILENTLY PERSISTS AS ZERO THROUGH THE POSTCARD DECIMAL ROUND-TRIP

// LOW

Description

`circuit_breaker_from_config` rejects a threshold only when `Decimal::is_zero()` is true. A wire value such as `Decimal::from_repr([1, 0, ...])` is strictly nonzero and passes validation. However, the `Decimal` serde format retains only 38 fractional decimal digits, so a value below `10-38` serializes to the string `"0"`.

The breaker set is persisted with postcard (serde) on every store and reload. A validated-nonzero but sub-precision threshold therefore round-trips to exactly `Decimal::ZERO` in storage. On reload the breaker's threshold is zero.

The bound: the submitter already holds breaker-configuration authority and the value was effectively near zero to begin with, so this needs a privileged (mis)configuration. The concern is that validate-nonzero followed by persist-zero can silently turn an intended tiny threshold into a hair-trigger breaker.

Code Location

 Copy Code

```
// soroban/contract/src/conversion.rs : only zero is rejected
if max_relative_change.is_zero() {
    return Err(ContractError::InvalidInput);
}
// soroban/contract/src/storage.rs : breaker set persisted via postcard (serde)
let bytes = to_allocvec(breakers)...;
            from_bytes(&bytes.to_alloc_vec())...;
```

Impact

A breaker validated as nonzero can become zero in storage. A zero StepwiseChange or WindowedChangeDelta threshold trips on any nonzero move, and a zero MonotonicRun step counts every move, so an enforced breaker can block the feed on a one-unit price change (deterministic feed unavailability).

Proof of Concept

Confirmed by an executed test: a nonzero `Decimal::from_repr([1,0,...])` threshold passes validation, persists as zero, and then a one-unit price move trips the breaker.

 Copy Code

```
running 1 test
test tests::reaudit_nonzero_decimal_threshold_rounds_to_zero_when_persisted ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_nonzero_decimal_threshold_rounds_to_zero_when_persisted`.

PoC Script

 Copy Code

```
let submitted = Decimal::from_repr([1, 0, 0, 0, 0, 0, 0, 0]);
assert!(!submitted.is_zero()); // passes validation
let breaker_id = proxy.add_breaker(&asset,
    &CircuitBreakerConfig::StepwiseChange(SorobanStepwiseChangeConfig {
        max_relative_change: SorobanDecimal::from_decimal(&env, submitted) }));
let persisted = stored_breakers(&env, &proxy.address, &asset);
let CircuitBreaker::StepwiseChange(rule) = ... ;
assert!(rule.max_relative_change.is_zero()); // persisted as zero
// a 5_000_000_000 -> 5_000_000_001 move now trips (Blocked)
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Re-check `is_zero()` on the reloaded `Decimal` after the storage round-trip, or reject values that quantize to zero at the serde precision at ingest.
- Alternatively, canonicalize `SorobanDecimal` to the serde precision on ingest so validation sees the value that will actually be stored.

Until fixed, operators must avoid sub- 10^{-38} breaker thresholds.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by rejecting zero (and sub-precision) thresholds at breaker-add time via a new `is_valid_for` check, and by re-validating the breaker set after every postcard storage round-trip in `write_breakers`, failing the write with `StorageError` if any threshold degenerates to zero.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.8 MISSING BREAKER STORAGE IS TREATED AS AN EMPTY UNTRIPPED SET INSTEAD OF FAILING CLOSED

// LOW

Description

`load_breakers` returns `CircuitBreakerSet::empty()` when the `Breakers(asset)` persistent key is absent, rather than treating a missing key as a storage error. If a proxy config exists but the breaker key is genuinely absent, a subsequent successful refresh persists that empty set, permanently discarding all configured breakers and any manual-trip state. `get_breaker_set_view` then reports a legitimate-looking empty set. This contradicts AUDIT.md, which states that `refresh_one` fails closed on missing storage keys. The bound: this is not reachable through public entrypoints in normal operation. On Soroban Protocol 23+, an expired persistent entry is archived and auto-restored to its original value (a transaction referencing it fails until restored); it does not read back as absent. The precondition is a faulty migration or upgrade, manual state deletion, or a malformed deployment that leaves `Proxy` present while `Breakers` is truly absent, so ordinary TTL eviction does not trigger it.

Code Location

 Copy Code

```
// soroban/contract/src/storage.rs
pub fn load_breakers(env: &Env, asset: &Asset) -> Result<CircuitBreakerSet, ContractError> {
    let Some(bytes) = env.storage().persistent()
        .get::(<_, Bytes>(&DataKey::Breakers(asset.clone()))) else {
        return Ok(CircuitBreakerSet::empty()); // fail-open on missing key
    };
    from_bytes(&bytes.to_alloc_vec()).map_err(|_| ContractError::StorageError)
}
```

Impact

All circuit-breaker and manual-trip protection for an asset can be silently and permanently lost. After the empty set is rewritten, subsequent large jumps are accepted. High impact if the precondition occurs, but it requires an operational error rather than an attacker action or ordinary archival.

Proof of Concept

Confirmed by an executed test that removes only the `Breakers(asset)` key and shows the next refresh is accepted with a zero breaker count.

 Copy Code

```
running 1 test
test tests::reaudit_missing_breaker_storage_fails_open ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.

- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_missing_breaker_storage_fails_open`.

PoC Script

 Copy Code

```
// Install a breaker, then delete only the Breakers key (simulating bad migration).
env.as_contract(&proxy.address, || {
    env.storage().persistent().remove(&DataKey::Breakers(asset.clone()));
});
set_ledger(&env, 101);
source.set_price(&asset, &10_000_000_000_i128, &101_u64); // large jump
let after = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(matches!(after.get(0).unwrap().1, RefreshStatus::Accepted(_));
assert_eq!(proxy.get_breaker_set_view(&asset).unwrap().breaker_count, 0);
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Once `Proxy(asset)` exists, treat a missing `Breakers(asset)` as `ContractError::StorageError` (fail closed) instead of returning an empty set.
- Keep empty-set initialization exclusively in `set_proxy`.

Until fixed, migrations and upgrades must guarantee the breaker key is always written whenever a proxy config is present.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `load_breakers` return `ContractError::StorageError` when the `Breakers(asset)` key is missing, instead of silently falling back to an empty untripped set.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.9 ONE ORACLE ADDRESS CAN FILL MULTIPLE QUORUM SLOTS VIA DISTINCT ASSET ALIASES

// LOW

Description

`validate_proxy_config` rejects only exact duplicate `(oracle, asset)` pairs. The same oracle address can therefore occupy several source slots under different asset symbols, and a shipped test explicitly allows the same oracle with different assets.

A malicious oracle that ignores the `asset` argument (returning the same manipulated price for every asset it is asked about) then casts multiple median votes from a single trust domain, defeating the independent-sources assumption behind quorum.

The bound: this requires governance to configure one oracle address under multiple asset aliases (a plausible convenience configuration), and then that oracle to be compromised. It does not affect configurations that use distinct oracle addresses per slot.

Code Location

 Copy Code

```
// soroban/contract/src/conversion.rs : dedup is on the full (oracle, asset) pair only
for i in 0..sources_len {
  let src_i = config.sources.get(i).ok_or(ContractError::InvalidInput)?;
  for j in (i + 1)..sources_len {
    let src_j = config.sources.get(j).ok_or(ContractError::InvalidInput)?;
    if src_i.oracle == src_j.oracle && src_i.asset == src_j.asset {
      return Err(ContractError::InvalidInput);
    }
  }
}
```

Impact

One compromised contract can control the median without compromising a majority of independent oracles. For example, with `[A/BTC, A/WBTC, B]` and `min_sources = 3`, oracle A supplies two of the three votes and dominates the median.

Proof of Concept

Confirmed by an executed test: the same oracle configured under two asset aliases satisfies a two-of-... quorum and its repeated vote controls the result.

 Copy Code

```
running 1 test
test tests::reaudit_one_oracle_contract_can_satisfy_multi_source_quorum_via_asset_aliases ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo `Templar-Protocol/contracts` at commit `fde891da949ccdc615198175847de72cfdaef828`, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_one_oracle_contract_can_satisfy_multi_source_quorum_via_asset_aliases`.

PoC Script

 Copy Code

```
// Same oracle address under two asset aliases, min_sources = 2.
let sources = Vec::from_array(&env, [
    SourceConfig { oracle: oracle.clone(), asset: asset.clone() },
    SourceConfig { oracle, asset: alias.clone() },
]);
proxy.set_proxy(&asset, &ProxyConfig { sources, min_sources: 2,
    max_age_secs: Some(30), max_clock_drift_secs: Some(5) });
source.set_price(&asset, &9_000_000_000_i128, &100_u64);
source.set_price(&alias, &9_000_000_000_i128, &100_u64);
let result = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(matches!(result.get(0).unwrap().1, RefreshStatus::Accepted(_)));
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Enforce unique oracle addresses across a proxy's source list, or count quorum by unique trust domain rather than by `(oracle, asset)` slot.
- Alternatively, make each source's returned asset binding explicit and verified so one oracle cannot answer for multiple aliases.

Until fixed, governance must configure each quorum slot with a distinct, independent oracle address.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `validate_proxy_config` reject any duplicate oracle address across a proxy's source list, regardless of the asset it is paired with, so a single oracle can no longer occupy multiple quorum slots.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.10 SOURCE-CONTROLLED TIMESTAMPS CAN REGRESS, REWRITE, OR STARVE THE SEP-40 HISTORY

// LOW

Description

The runtime stores accepted-price history keyed by the source-provided timestamp. `push_history` deduplicates by exact timestamp: if the incoming timestamp already exists it removes the prior entry and appends the new price at the back. Timestamps are bounded relative to the current ledger by the freshness filter but are not required to be monotonically increasing, and `aggregated_history` applies no freshness filter.

A source selected by the median can therefore emit a regressing timestamp (producing out-of-order history), or reuse a recent timestamp (an in-place rewrite of that datapoint rather than an append), and can churn the 32-entry buffer.

The bound: this affects the history read surface (which is documented as freshness-unfiltered), not `aggregated_latest`. It requires a source that becomes the MedianLow selection with timestamps inside the freshness window.

Code Location

 Copy Code

```
// soroban/contract/src/storage.rs : dedup-by-timestamp then append
if let Some(index) = history.iter()
    .position(|entry| entry.timestamp == price.timestamp)
    .and_then(|i| u32::try_from(i).ok()) {
    history.remove(index);
}
history.push_back(price.clone());
```

Impact

Consumers of the history surface (for example a TWAP) can be fed non-chronological or rewritten data. A source can overwrite its own recent datapoints instead of appending, distorting any history-derived value.

Proof of Concept

Confirmed by an executed test: a regressing timestamp yields out-of-order history, and a reused timestamp rewrites an existing entry rather than appending.

 Copy Code

```
running 1 test
test tests::reaudit_source_timestamps_can_regress_and_rewrite_history ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.

- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_source_timestamps_can_regress_and_rewrite_history`.

PoC Script

 Copy Code

```
// Accept ts=100, then ts=99 (regression) -> history is [100, 99].
source.set_price(&asset, &5_100_000_000_i128, &99_u64);
proxy.refresh(...);
let regressed = proxy.aggregated_history(&asset, &2).unwrap();
assert_eq!(regressed.get(1).unwrap().timestamp, 99);
// Reuse ts=100 -> the earlier ts=100 entry is removed and rewritten.
source.set_price(&asset, &5_200_000_000_i128, &100_u64);
proxy.refresh(...);
let rewritten = proxy.aggregated_history(&asset, &2).unwrap();
assert_eq!(rewritten.get(1).unwrap().mantissa, 5_200_000_000);
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Enforce strictly increasing timestamps in `push_history` (reject or ignore non-monotonic samples).
- Alternatively, key history on ledger time rather than the source-provided timestamp.

Until fixed, history-derived consumers should not assume chronological ordering from the adapter.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by requiring strictly increasing timestamps in the history update path: a new sample is only appended when its timestamp is greater than the last stored one, otherwise the update is a no-op.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.11 ACCEPTED EXTREME PRICE SCALE POISONS THE ENTIRE ADAPTER PRICES() BATCH

// LOW

Description

`source_price_to_kernel` accepts extreme source scales (for example `decimals = i32::MAX`, giving exponent about -2,147,483,647, or `decimals = 0` with `price = i128::MAX`) into the kernel, cache, and history. The runtime returns `Accepted` for such a price.

The SEP-40 adapter's `prices()` maps any single `normalized_to_sep40` error to `None` for the whole batch via `.ok()?`. A single unconvertible history entry therefore makes the entire `prices(N)` window return `None`, and history is not cleared on proxy reconfigure, so honest samples added afterward remain unreadable until the poison entry rolls off the 32-entry buffer.

The bound: this needs a source (during bootstrap or via median control) returning an extreme decimals/scale, and an adapter with normal decimals. It is an availability/consistency issue at the adapter read surface that self-heals as history rolls over.

Code Location

 Copy Code

```
// soroban/sep40-adapter-contract/src/lib.rs : one bad entry aborts the whole batch
for entry in history.iter() {
    out.push_back(normalized_to_sep40(&entry, config.decimals).ok()?);
}
```

Impact

One accepted extreme sample makes the adapter's batched history read unavailable and puts the runtime (`Accepted`) and adapter (`None`) into disagreement until the entry ages out of the buffer.

Proof of Concept

Confirmed by an executed test: an accepted extreme-scale sample persists in history, and a later honest sample cannot be read back via the batch because the poison entry fails conversion.

 Copy Code

```
running 1 test
test tests::reaudit_accepted_unconvertible_scale_poison_persists_in_history ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo `Templar-Protocol/contracts` at commit `fde891da949ccdc615198175847de72cfdaef828`, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_accepted_unconvertible_scale_poison_persists_in_history`.

PoC Script

 Copy Code

```
source.init(&base, &(i32::MAX as u32), &1_u32); // extreme decimals
source.set_price(&asset, &1_i128, &100_u64);
let extreme = proxy.refresh(...); // Accepted
set_ledger(&env, 101);
source.init(&base, &8_u32, &1_u32);
source.set_price(&asset, &5_000_000_000_i128, &101_u64);
proxy.refresh(...); // Accepted honest sample
let history = proxy.aggregated_history(&asset, &2).unwrap();
assert!(normalized_to_sep40(&history.get(0).unwrap(), 8).is_err()); // poison entry
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Bound the acceptable source **decimals** / resulting exponent at ingest and fail closed before caching an unrepresentable scale.
- Make the adapter skip individual unconvertible history entries rather than aborting the whole batch.
- Clear history on proxy reconfigure.

Until fixed, operators should vet source decimals and treat a sudden adapter **prices()** outage as a possible poisoned-history condition.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by bounding source decimals to 18 at ingest, making the SEP-40 adapter's **prices()** skip individual unconvertible history entries instead of aborting the whole batch, and clearing history on proxy reconfiguration.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

11.12 CONFIG VALIDATION ACCEPTS INERT CIRCUIT BREAKERS THAT CAN NEVER TRIP

// LOW

Description

`circuit_breaker_from_config` rejects only zero thresholds, `max_streak == 0`, `window_len < 2`, and `lookback_windows == 0`. Several configurations pass validation yet make a breaker unable to ever trip: (a) a relative-change threshold near `Decimal::MAX` is nonzero but larger than any real move, so the breaker never trips; (b) a `WindowedChangeDelta` needs `window_len * (lookback_windows + 1)` samples, but nothing relates the installed rule to the set's `history_len`, so `change_delta` returns `None` forever if history is too small; (c) a `MonotonicRun.max_streak` larger than the retained history can supply steps for can never reach the streak.

The bound: this is a governance/operations misconfiguration guardrail gap, not an attacker-triggered condition. RUNBOOK warns operators to size `history_len` for every rule, but the contract does not enforce it.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/rule/windowed_change_delta.rs
let current_start = history.len().checked_sub(window_len)?;
let lookback_offset = lookback_windows.checked_mul(window_len)?;
let previous_start = current_start.checked_sub(lookback_offset)?; // None if history too small
```

Impact

An installed, armed, enforced breaker can silently provide zero protection. Operators and monitoring believe the feed is defended when it is not; it is the mirror image of the persist-to-zero over-sensitive breaker issue.

Proof of Concept

Confirmed by an executed test that installs each of the three inert configurations and shows none trips even on a violent move.

 Copy Code

```
running 1 test
test tests::reaudit_p4_low008_inert_breaker_configs_never_trip ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaf828, full workspace.

- Run `cargo test -p templar-proxy-oracle-kernel`
`reaudit_p4_low008_inert_breaker_configs_never_trip`.

PoC Script

 Copy Code

```
// (a) huge threshold: a 100x jump does not trip.
let inert = StepwiseChange { max_relative_change: Decimal::MAX };
assert!(!inert.should_trip(&history([100, 1_000_000])));
// (b) window needs 6 samples but only 4 in history: never trips.
let windowed = WindowedChangeDelta { window_len: 2, lookback_windows: 2,
    max_relative_change_delta: dec("0.01") };
assert!(!windowed.should_trip(&history([1, 1_000_000, 1, 1_000_000])));
// (c) max_streak larger than history can supply: never trips.
let monotonic = MonotonicRun { max_streak: 100, min_relative_step_change: dec("0.01") };
assert!(!monotonic.should_trip(&history([100, 110, 121, 133])));
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- At config/add time, require `history_len >= window_len * (lookback_windows + 1)` for `WindowedChangeDelta` and `history_len > max_streak` for `MonotonicRun`.
- Consider an upper bound (or a below-100-percent sanity cap) on relative-change thresholds, and/or emit a warning event on a suspicious configuration.

Until fixed, operators must manually verify each installed breaker can actually trip under the configured history length.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by adding an `is_valid_for` check that requires `history_len >= window_len * (lookback_windows + 1)` for `WindowedChangeDelta`, `max_streak < history_len` for `MonotonicRun`, and bounds every relative-change threshold to `(0, 1]`, rejecting configurations that could never trip.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.13 UNBOUNDED REGISTERED-ASSET LIST CAN MAKE GLOBAL `EXTEND_TTL()` EXCEED TRANSACTION RESOURCE LIMITS

// LOW

Description

Each new proxy asset is appended to a single global `Assets` vector with no maximum count. Two public maintenance paths iterate that whole vector in one transaction: `refresh([])` refreshes every registered asset, and `extend_ttl()` renews the `Proxy`, `Breakers`, `Cache`, and `History` keys for every asset. There is no paginated or per-asset TTL-extension endpoint.

Once enough assets are configured, `extend_ttl()` exceeds the transaction footprint limit and aborts. In the PoC, after configuring 81 assets the call emits `TtlExtended { asset_count: 81 }` and then the host aborts with `Error(Budget, ExceededLimit)` (`total footprint ledger entries: 327 > 100`). The bound: this requires privileged configuration (`ProxyConfigurationManager` or Admin) to add many assets, and it is availability/operations only, not a wrong-price issue. Per-asset explicit `refresh([asset])` still works, but there is no per-asset `extend_ttl`.

Code Location

 Copy Code

```
// soroban/contract/src/lib.rs : extend_ttl iterates all registered assets
let assets = get_assets(&env);
for asset in assets.iter() {
    extend_persistent_ttl(&env, &DataKey::Proxy(asset.clone()));
    extend_persistent_ttl(&env, &DataKey::Breakers(asset.clone()));
    extend_persistent_ttl(&env, &DataKey::Cache(asset.clone()));
    extend_persistent_ttl(&env, &DataKey::History(asset));
}
```

Impact

Global TTL maintenance can become unusable. If keepers rely on global `extend_ttl()`, persistent entries can eventually archive and require manual restoration; global `refresh([])` can likewise become unusable, degrading availability of every affected feed.

Proof of Concept

Confirmed by an executed test that configures 81 assets and shows `extend_ttl()` aborts at the mainnet-style footprint limit.

 Copy Code

```
[Contract Event] topics:[ttl_extended], data:{asset_count: 81}
invocation resource limits are exceeded: total footprint ledger entries: 327 > 100
HostError: Error(Budget, ExceededLimit)
test tests::reaudit_unbounded_registered_assets_drive_extend_ttl_resource_growth - should panic ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace; soroban-sdk 25 testutils enforce mainnet-style resource limits.

- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_unbounded_registered_assets_drive_extend_ttl_resource_growth`.

PoC Script

 Copy Code

```
// Register 81 assets, then call extend_ttl().
for i in 0..80_u32 {
    let asset = Asset::Other(Symbol::new(&env, &format!("A{i}")));
    proxy.set_proxy(&asset, &ProxyConfig { sources, min_sources: 1,
        max_age_secs: Some(30), max_clock_drift_secs: Some(5) });
}
assert_eq!(proxy.registered_assets().len(), 81);
proxy.extend_ttl(); // panics: Error(Budget, ExceededLimit), footprint 327 > 100
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Add a global cap on the number of registered assets.
- Expose a paginated or per-asset TTL-extension endpoint (for example `extend_ttl(asset)` or `extend_ttl_range(start, limit)`), and discourage global `refresh([])` for large deployments.

Until fixed, operators should keep the registered-asset count well below the resource-limit threshold and monitor `asset_count`.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making both `refresh` and `extend_ttl` per-asset endpoints instead of iterating the full registered-asset list, and by adding a global cap (`MAX_REGISTERED_ASSETS`) on the number of assets a proxy can register.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/e8b542dc54f5b49871c6ce2e82c21012ef723216>

11.14 NONZERO SAMPLE_INTERVAL LETS MONOTONICRUN MISS ACCEPTED INTRA-INTERVAL STAGED RAMPS

// LOW

Description

When `sample_interval_ns > 0`, `try_accept_price` appends a candidate to a cloned `proposed_accepted_history` but only commits it to the persistent `accepted_history` when `should_persist_sample` is true (that is, when the sample interval has elapsed since the last observed sample). `MonotonicRun` evaluates its streak only over the supplied history, which is the persisted accepted history plus the single current candidate.

As a result, multiple same-direction accepted refreshes inside one sample interval never build a multi-step streak: each candidate is seen as a single step from the last persisted sample. The runtime still updates the cache and returns `Accepted` to callers for each of them.

The bound: this is configuration-dependent (nonzero `sample_interval` and reliance on `MonotonicRun`) and can be partially mitigated by a tight `StepwiseChange` threshold. It weakens the documented staged-ramp defense but does not by itself grant fund theft.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/set.rs
let should_persist_sample = self.should_persist_sample(now);
// ... breakers evaluate proposed_accepted_history ...
if acceptance.value.is_ok() && should_persist_sample {
    self.0.accepted_history = proposed_accepted_history; // only committed when sampled
}
```

Impact

A staged ramp delivered as several accepted refreshes within one sample interval avoids `MonotonicRun`.

Downstream consumers see the ramp via `aggregated_latest`, but the streak-based defense does not trip until enough sampled commits accumulate.

Proof of Concept

Confirmed by an executed test: baseline 100 committed at t=1; 102, 104, 106 accepted within a 10s interval without being committed, so no streak forms and even a later 108 is one step from 100.

 Copy Code

```
running 1 test
test tests::reaudit_w2_sample_interval_hides_intra_interval_monotonic_streak ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.

- Run `cargo test -p templar-proxy-oracle-kernel`

`reaudit_w2_sample_interval_hides_intra_interval_monotonic_streak`.

PoC Script

 Copy Code

```
let mut set = breaker_set(Nanoseconds::from_secs(10), 8); // 10s sample interval
set.add(0, CircuitBreaker::MonotonicRun(MonotonicRun {
    max_streak: 2, min_relative_step_change: dec("0.01") })).unwrap();
set.try_accept_price(price(100), Nanoseconds::from_secs(1)).unwrap(); // committed
for (t, p) in [(2, 102), (3, 104), (4, 106)] {
    let o = set.try_accept_price(price(p), Nanoseconds::from_secs(t)).unwrap();
    assert!(o.value.is_ok()); // accepted
    assert_eq!(set.accepted_history().len(), 1); // not committed, no streak
}
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Evaluate rules against the accepted candidate stream even inside a sample interval (sample only an auxiliary history for storage/gas), or have MonotonicRun consider intra-interval accepted candidates.
- Alternatively, document that `sample_interval_secs = 0` is required for MonotonicRun to detect per-refresh ramps, and enforce a maximum refresh frequency.

Until fixed, deployments relying on MonotonicRun should set `sample_interval_secs = 0` or pair it with a tight StepwiseChange threshold.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by requiring `sample_interval_ns == 0` as a precondition for adding or reconfiguring a `MonotonicRun` breaker, eliminating the configuration that let intra-interval ramps go undetected.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.15 CIRCUIT-BREAKER SUITE HAS NO CUMULATIVE-DRIFT DEFENSE: A STEADY SUB-THRESHOLD PRICE RAMP EVADES ALL THREE RULES

// LOW

Description

The proxy oracle's defense against price manipulation is a set of three circuit-breaker rules (StepwiseChange, MonotonicRun, WindowedChangeDelta) that gate whether a newly aggregated price is accepted into the cache that downstream SEP-40 consumers read. A steady price ramp whose per-step relative change stays just below the configured thresholds evades all three rules at once, so a party that controls the accepted (aggregated) price can walk the reported price arbitrarily far while every enabled and enforced breaker stays silent.

The root cause is that no rule compares the price level against a longer-term baseline. StepwiseChange only inspects the two most recent observations, so any per-step move at or below `max_relative_change` passes. MonotonicRun classifies any step whose relative change is at or below `min_relative_step_change` as Minor, and a Minor step breaks its streak scan, so a ramp of sub-threshold steps never accumulates a streak. WindowedChangeDelta compares the signed relative change of two equal-length windows; during a constant-velocity ramp both windows change by the same amount, so the delta is approximately zero - it detects acceleration, not sustained drift.

This is a documented limitation rather than a novel defect: PROOF.md's Limitations section states that gradual sub-1%-per-window moves "can stay under MonotonicRun, but still require compromising a majority of independent sources". The two audit-relevant refinements beyond that note are: (1) the same ramp also evades StepwiseChange and WindowedChangeDelta, so PROOF.md's defense-in-depth table over-credits WindowedChangeDelta as a sustained-move / statistical-anomaly catch it does not provide; and (2) `validate_proxy_config` permits `min_sources = 1`, which collapses the sole documented mitigation (multi-source quorum) to a single source. The total move is bounded only by the attacker's patience: a smaller `min_relative_step_change` merely forces a slower cadence.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/rule/monotonic_run.rs - a Minor step breaks the streak scan
let step_change = classify_step_change(&previous.price, &current.price, self.min_relative_step_change);
if step_change == StepChange::Minor {
    break;
}

// kernel/src/proxy/circuit_breaker/math.rs - a sub-threshold move IS classified Minor
if relative_abs_change_exceeds(previous, current, min_relative_step_change) {
    step_change
} else {
    StepChange::Minor
}

// kernel/src/proxy/circuit_breaker/rule/windowed_change_delta.rs - delta of two windows'
// internal change is ~0 for a constant-velocity ramp (acceleration, not drift)
Some(current.abs_diff(previous))
```

Impact

Given control of the accepted price (`min_sources = 1` with a compromised source, the degraded-quorum survivor of the MedianLow finding, or a compromised source majority - all within PROOF.md's compromised-source threat model), an attacker walks the SEP-40 lastprice that a lending market reads arbitrarily far while every enforced breaker stays silent. Consumed by a lending market this enables over-borrowing against an inflated collateral price, or wrongful liquidation against a deflated one. For the recommended $n \geq 3$ / `min_sources=majority` configuration this is exactly the documented, accepted limitation; the residual risk is a deployer who trusts the defense table's sustained-move rows, or a single-source (`min_sources=1`) feed.

Proof of Concept

Two kernel PoCs show no rule trips at any prefix of the ramp; a third shows the same evasion under a conservative 0.5%-min-step / 5%-threshold config; and an end-to-end integration PoC (external attacker perspective) shows the SEP-40 lastprice a lending market reads is walked about 2.8x while every enforced breaker stays silent. A passing test here demonstrates the evasion (green is bad).

[Copy Code](#)

```
running 3 tests
test proxy::circuit_breaker::tests::reaudit_p4_high_steady_ramp_evades_all_three_rules ... ok
test proxy::circuit_breaker::tests::reaudit_p4_high_steady_ramp_accepted_through_enforced_breaker_set ..
test proxy::circuit_breaker::tests::reaudit_p4_high_ramp_evades_even_conservative_thresholds ... ok
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 211 filtered out

running 1 test
test reaudit_e2e_attacker_ramps_price_past_all_enforced_breakers ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7 filtered out
```

Environment / Reproduce

[Copy Code](#)

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_p4_high
cargo test -p templar-proxy-oracle-soroban-integration-tests --test breakers reaudit_e2e
```

PoC Script

End-to-end, external attacker perspective: the admin installs a `min_sources=1` feed plus all three breakers, enabled and enforced, at 10% thresholds. The attacker controls only one compromised source and calls the permissionless refresh.

[Copy Code](#)

```
// integration-tests/tests/breakers.rs (abridged)
b.configure_default_feed(); // BTC/USD, one source, min_sources = 1
// admin adds StepwiseChange(0.10), MonotonicRun(max_streak 3, min_step 0.10),
// WindowedChangeDelta(window 3, lookback 1, 0.10) - all enforced
let mut price: i128 = 1_000_000_000;
b.push_upstream_price(&b.asset_btc, price, ts);
let start = b.adapter.lastprice(&b.asset_btc).unwrap().price;
for _ in 0..12 {
    advance_secs(&b.env, 1); ts += 1;
    price = price * 109 / 100; // +9% per refresh, under every 10% threshold
    b.push_upstream_price(&b.asset_btc, price, ts);
    assert!(matches!(b.refresh_one(&b.asset_btc), RefreshStatus::Accepted(_)));
    assert!(!b.runtime.get_breaker_set_view(&b.asset_btc).unwrap().is_blocking());
}
```

```
let end = b.adapter.lastprice(&b.asset_btc).unwrap().price;  
assert!(end >= start * 2); // walked ~2.8x, no breaker ever tripped
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Add a defense that bounds cumulative drift, and remove the configuration that collapses quorum:

- Add a circuit-breaker rule that bounds the accepted price's deviation from a longer-term baseline (for example, a maximum relative change versus an N-sample trailing reference or a long-window EMA), so a slow ramp accumulates into a trip regardless of per-step size.
- Alternatively, make MonotonicRun accumulate sub-threshold same-direction drift (sum the moves) instead of discarding any step at or below min_relative_step_change and letting a Minor step break a genuine run.
- Disallow min_sources = 1 in validate_proxy_config, or require an external anchor / peer-deviation check for single-feed assets, since min_sources = 1 reduces the only documented mitigation to a single source.
- Correct PROOF.md: either implement the average/level comparison its WindowedChangeDelta description claims, or amend the defense table and description to state that WindowedChangeDelta detects acceleration, not sustained drift, and that gradual manipulation is bounded only by quorum.

Until fixed, operators must not rely on the breaker suite to stop a slow ramp; treat multi-source quorum (n>=3 with an honest majority) as the actual defense and monitor for sustained directional drift out of band.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding a new **CumulativeChange** circuit-breaker rule that compares the accepted price against a fixed baseline, so a steady sub-threshold ramp accumulates into a trip instead of evading all rules.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.16 SETACTIONTTL (PROXYCONFIGURATIONMANAGER) CAN RETIME ANY OPERATION INCLUDING ADMIN OPS, RATCHETING ADMIN-RECOVERY TIMELOCKS

// LOW

Description

Governance timelocks are the per-operation TTLs changed by the SetActionTtl action. required_role maps SetActionTtl to the non-Admin ProxyConfigurationManager role, yet SetActionTtl can target ANY operation kind, including Admin-only operations (SetRole, Upgrade, TransferOwnership). The anti-fast-lowering rule prevents shortening an operation's timelock faster than that operation's own current timelock, but the same rule is asymmetric and forms a ratchet a rogue config manager can use against Admin recovery.

minimum_ttl for SetActionTtl(kind, _) returns max(ttl[SetActionTtl], ttl[kind]). Lowering kind's TTL is therefore gated by kind's own current TTL, so the "shrink a sensitive op ahead of an attack" concern is refuted. But raising kind's TTL is gated only by kind's current (low) value, while restoring a short TTL afterwards is gated by the new (high) value. A ProxyConfigurationManager can thus inflate the timelock of Admin recovery operations to the 180-day MAX_PROPOSAL_TTL cheaply, after which neither the manager nor Admin can restore a short timelock for up to 180 days.

Bound: the config manager cannot create or execute Admin operations (those still require the Admin role); it can only change their timing. The inflation proposal is itself timelocked, visible, and cancellable by Admin, and Admin retains the immediate cancel_proposal lever. The impact is therefore delayed governance recovery (availability), not fund loss or privilege escalation.

Code Location

 Copy Code

```
// soroban/governance-common/src/lib.rs - SetActionTtl is authorized by a non-Admin role
Self::SetProxy(_, _) | Self::RemoveProxy(_) | Self::ConfigureBreakers(_, _, _)
  | Self::AddBreaker(_, _) | Self::RemoveBreaker(_, _)
  | Self::SetActionTtl(_, _) => Role::ProxyConfigurationManager,

// anti-fast-lowering minimum_ttl - max of own and target TTL (asymmetric ratchet)
Self::SetActionTtl(kind, _) => {
  let set_action_ttl = ttls.get(OperationKind::SetActionTtl);
  let target_ttl = ttls.get(*kind);
  set_action_ttl.max(target_ttl)
}
```

Impact

A rogue or key-compromised ProxyConfigurationManager can inflate the SetRole and Upgrade timelocks to 180 days, delaying Admin's ability to revoke the manager or upgrade the contract for that period. During the window Admin can still cancel individual malicious proposals but must babysit governance rather than removing the manager outright. No fund loss; governance availability and incident response are degraded.

Proof of Concept

Unit PoCs show the ratchet at the policy layer and end-to-end through the deployed governance contract: a non-Admin manager raises Upgrade's TTL while the gate is 0 (cheap), after which the restore gate is the 180-day maximum.

 Copy Code

```
running 3 tests
test tests::reaudit_p9_setactionttl_value_boundaries ... ok
test tests::reaudit_p9_setactionttl_ratchet_inflates_admin_recovery ... ok
test tests::reaudit_p9_setactionttl_lowering_gated_by_target_ttl ... ok
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 10 filtered out

running 1 test
test tests::reaudit_p10_setactionttl_ratchet_inflates_admin_recovery_end_to_end ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 29 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-governance-common reaudit_p9_setactionttl
cargo test -p templar-proxy-oracle-soroban-governance-contract reaudit_p10_setactionttl
```

PoC Script

 Copy Code

```
// governance-contract/src/tests.rs (abridged) - non-Admin manager, end to end
let grant = submit_now(&env, &governance_id, &admin,
    GovernanceAction::SetRole(manager.clone(), Role::ProxyConfigurationManager, true));
accept_now(&env, &governance_id, &admin, grant);
// raise gate is the current (0) TTL - cheap
assert_eq!(get_effective_proposal_ttl(SetActionTtl(Upgrade, MAX_PROPOSAL_TTL_NS), 0), 0);
let raise = submit_now(&env, &governance_id, &manager,
    GovernanceAction::SetActionTtl(OperationKind::Upgrade, MAX_PROPOSAL_TTL_NS));
accept_now(&env, &governance_id, &manager, raise);
// restoring a short timelock is now gated by the inflated 180-day value
assert_eq!(get_effective_proposal_ttl(SetActionTtl(Upgrade, 0), 0), MAX_PROPOSAL_TTL_NS);
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Bind the authority to change an operation's timelock to at least the operation's own privilege level, and give Admin an escape hatch:

- Require Role::Admin for SetActionTtl(kind, _) when kind.required_role() is Admin, or gate all SetActionTtl behind Admin.
- Alternatively, give Admin a fast path exempt from anti-fast-lowering when restoring a timelock, and/or cap Admin-operation timelocks below the 180-day maximum.
- Keep the anti-fast-lowering rule for the lowering direction; it correctly prevents shrinking a sensitive operation's timelock ahead of an attack.

Until fixed, restrict the ProxyConfigurationManager role to trusted holders and monitor for SetActionTtl proposals that target Admin operations.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by requiring `Role::Admin` for `SetActionTtl(kind, _)` whenever the targeted operation kind itself requires Admin, closing the ratchet a non-Admin config manager could otherwise use against Admin-recovery timelocks.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/5c930310554c4272f0dbbf3cef5fe8c45373d74f>

11.17 SEP-40 ADAPTER CAN RELABEL A PARENT PRICE UNDER AN UNRELATED BASE WITHOUT DENOMINATION CONVERSION

// LOW

Description

The runtime guarantees every source feed reports in its configured immutable source_base (for example USD). Each SEP-40 adapter, however, independently declares its own SEP-40 base and lets its owner change it through set_metadata. The adapter never checks that its base equals the runtime's source base and performs no FX/denomination conversion; its read paths only re-scale the same normalized mantissa by decimals.

An adapter owner can therefore change the adapter label from USD to EUR while continuing to return the unchanged BTC/USD numeric value as though it were BTC/EUR. The numeric price is correct for the wrong denomination.

Bound: this requires a trusted-owner action (or owner-key compromise) and affects only that adapter's consumers; the runtime cache, source validation, and other adapters are unaffected. The adapter owner can already upgrade the adapter, which bounds severity.

Code Location

 Copy Code

```
// soroban/sep40-adapter-contract/src/lib.rs - adapter declares/mutates its own base,
// and read paths only rescale decimals; they never validate or convert the denomination.
fn lastprice(env: Env, asset: Asset) -> Option<PriceData> {
    let config = load_config(&env);
    if config.asset != asset { return None; }
    let normalized = ProxyOracleClient::new(&env, &config.parent_oracle).aggregated_latest(&asset)?;
    normalized_to_sep40(&normalized, config.decimals).ok() // rescales decimals only
}
```

Impact

Downstream consumers that trust the adapter's base() can interpret a price in the wrong denomination, causing misvaluation of collateral or debt for that adapter's consumers. Isolated to the misconfigured adapter.

Proof of Concept

An end-to-end integration PoC configures an adapter whose base is set to EUR while the parent runtime reports USD, then shows lastprice returns the unchanged USD numeric value labelled EUR.

 Copy Code

```
test reaudit_low011_adapter_can_relabel_usd_price_as_eur_without_conversion ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 6 filtered out
```

Environment / Reproduce

```
cargo test -p templar-proxy-oracle-soroban-integration-tests reaudit_low011
```

PoC Script

Abridged: deploy the runtime (source_base USD) and an adapter constructed with base EUR for the same asset; drive a USD price through refresh; assert the adapter's base() is EUR while lastprice().price equals the unconverted USD mantissa.

```
// adapter base = EUR, parent source_base = USD, same asset, no FX conversion
assert_eq!(adapter.base(), Asset::Other(Symbol::new(&env, "EUR")));
let p = adapter.lastprice(&asset).unwrap();
assert_eq!(p.price, usd_mantissa); // USD value served as EUR
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Bind the adapter denomination to the runtime's:

- Expose the runtime denomination through ProxyOracleTrait (or call the runtime's source_base entrypoint) and require adapter base equality at construction.
- Make the adapter base immutable after deployment.
- Any cross-base adapter must perform an explicit, auditable FX conversion rather than relabel.

Until fixed, treat adapter base() as owner-asserted metadata and verify it out of band against the parent runtime's source base.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by validating at construction that the adapter's declared base equals the parent runtime's source base, and by removing the ability to change the base after deployment.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

11.18 WINDOWEDCHANGEDELTA IGNORES EXTREME INTERIOR-WINDOW OUTLIERS THAT RETURN TO THE WINDOW BASELINE

// LOW

Description

PROOF.md documents WindowedChangeDelta as a statistical outlier rule that compares recent and historical window averages. The implementation does not inspect or average interior samples; it computes only the signed relative change from the first to the last observation in each window.

An extreme interior spike that returns to the window's starting price has a zero net first-to-last change and is therefore invisible to the rule. In the PoC, the historical window [100,100,100] has average 100 and the current window [100,10_000,100] has average 3,400 (over 33x), yet both first-to-last changes are 0 and the breaker does not trip at a 10% threshold.

Bound: a correctly configured StepwiseChange breaker catches the individual spike, so impact requires relying on WindowedChangeDelta without a sufficiently tight StepwiseChange rule (or another breaker being inert/bypassed).

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/rule/windowed_change_delta.rs - only first/last per window
let current = relative_signed_change(&current_first.price, &current_last.price?);
let previous = relative_signed_change(&previous_first.price, &previous_last.price?);
Some(current.abs_diff(previous)) // interior samples never inspected
```

Impact

A manipulated price can be accepted and exposed to consumers during the interior of a window without this rule detecting it, provided the attacker returns the price to the window baseline by the window end. No direct fund-loss primitive on its own; it is a gap versus the documented statistical defense.

Proof of Concept

A kernel PoC builds an interior-outlier window and shows WindowedChangeDelta does not trip (green demonstrates the miss).

 Copy Code

```
test proxy::circuit_breaker::tests::reaudit_low012_windowed_rule_ignores_interior_window_outlier ... ok
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_low012
```

PoC Script

Abridged: history [100,100,100,100,10000,100]; WindowedChangeDelta(window_len 3, lookback 1, threshold 0.10). Both windows have first==last, so delta 0 and should_trip is false despite the 100x interior spike.

 Copy Code

```
let rule = WindowedChangeDelta { window_len: 3, lookback_windows: 1, max_relative_change_delta: dec("0.1")
let h = history([100, 100, 100, 100, 10_000, 100]);
assert(!rule.should_trip(&h)); // interior 100x spike invisible
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Align the implementation with the documented defense, or the documentation with the implementation:

- Implement the documented average-vs-average comparison over every sample in each window (with checked/saturating arithmetic).
- Or rename/document the rule as a first-to-last window-change comparator and state explicitly that it does not detect interior round-trip spikes.
- Require a complementary StepwiseChange breaker so single-sample spikes are still caught.

Until fixed, always pair WindowedChangeDelta with a tight StepwiseChange rule.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by rewriting **WindowedChangeDelta** to compare the average of every sample in each window (**max_relative_mean_change**), aligning the implementation with the documented average-vs-average defense instead of a first-to-last comparison.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.19 DECIMAL SATURATION ALIASES DISTINCT EXTREME WINDOW MOVES AND MAKES WINDOWEDCHANGEDelta MISS THEM

// LOW

Description

Decimal::mul_pow10 conservatively returns None for some products that actually fit U512. The breaker math maps that None to Decimal::MAX. Distinct, representable, enormous relative changes can therefore both collapse to the same MAX value, so their SignedDecimal::abs_diff becomes zero.

In the PoC the previous window change is about $2e115$ and the current window change is about $3e115$ (they differ by about $1e115$, far above threshold 1), but both saturate to the same sentinel and

WindowedChangeDelta does not trip.

Bound: ordinary 0-18 decimal feeds are unaffected. A malicious source must control aggregation and use unusual exponent gaps (for example source decimals 115 and 0) while no complementary breaker catches the movement.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/math.rs - any mul_pow10 failure becomes MAX/MIN
fn scale_decimal(value: Decimal, exponent: i32) -> Decimal {
    value.mul_pow10(exponent).unwrap_or_else(|| {
        if exponent.is_negative() { Decimal::MIN } else { Decimal::MAX }
    })
}
// two distinct over-range changes both become MAX, so abs_diff == 0
```

Impact

An extreme, code-permitted exponent/price sequence can cause a false negative in WindowedChangeDelta. Reachable only with abnormal decimals/exponent gaps and reliance on WindowedChangeDelta without effective Stepwise/Monotonic protection.

Proof of Concept

A kernel PoC constructs two distinct extreme window moves that both saturate to MAX; WindowedChangeDelta does not trip (green demonstrates the miss).

 Copy Code

```
test proxy::circuit_breaker::tests::reaudit_low013_windowed_saturation_aliases_distinct_extreme_moves ..
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_low013
```

PoC Script

Abridged: build a history whose previous window moves $\sim 1e-115$ to 2 (change $\sim 2e115$) and current window moves $\sim 1e-115$ to 3 (change $\sim 3e115$). Both relative changes saturate to `Decimal::MAX`, `abs_diff` is 0, and `should_trip` is false at threshold 1.

 Copy Code

```
// prev window change ~2e115, current ~3e115 (differ by ~1e115 > threshold 1)
// both saturate to Decimal::MAX => abs_diff == 0
assert(!windowed.should_trip(&history_with_extreme_exponent_gaps));
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Stop aliasing distinct over-range values to one sentinel before differencing:

- Fix `Decimal::mul_pow10` to use checked/overflowing multiplication against the actual product (and test the actual division result for zero) rather than conservative bit-length bounds.
- In breaker math, avoid collapsing distinct over-range values to a single sentinel before computing the delta.
- Reject unsupported exponent gaps at source ingest.

Until fixed, constrain configured source decimals to a normal range and pair `WindowedChangeDelta` with `Stepwise/Monotonic` breakers.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by rewriting the breaker comparison math to use exact U512 arithmetic (checked/overflowing multiplication and division) instead of a conservative bit-length heuristic, so distinct extreme values no longer collapse to the same **`Decimal::MAX`** sentinel.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/bcf357fbdb513fdd06569b4d301e9610ffc52655>

11.20 REARMING ONE BREAKER CLEARS OR REPLACES THE SHARED ACCEPTED BASELINE USED BY EVERY OTHER BREAKER

// LOW

Description

A `CircuitBreakerSet` stores one `accepted_history` shared by every installed rule. The public `rearm` operation is addressed to a single `breaker_id`, emits a `Rearmed { breaker_id, ... }` event for that breaker, and changes only that breaker's status, but its `AcceptedHistorySource` option clears or replaces the set-wide `accepted` history.

Rearming one breaker with `Empty` therefore erases the baseline used by all other armed/enforced breakers. In the PoC, breaker 1 is a strict 10% `StepwiseChange` with `accepted` baseline 100 and is never rearmed; governance rearms only permissive breaker 0 with `Empty`; breaker 1 then accepts a 100% jump to 200 because the shared proposed history contains only one sample.

Bound: governance-gated (a `CircuitBreakerOperator` or `Admin`), time-bounded while the baseline rebuilds, and the same privileged role can rearm breakers intentionally. This is a surprising cross-breaker side effect / misconfiguration guardrail, not privilege escalation.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/set.rs - rearm(breaker_id, ...) changes one breaker's status
// but its AcceptedHistorySource clears or replaces the SET-WIDE accepted_history,
// while the Rearmed event names only breaker_id.
```

Impact

An operator intending to rearm one breaker can temporarily disable history-dependent protection for every other breaker until enough new `accepted` samples accumulate. Observed seeding similarly replaces every rule's baseline with trip-time observed prices. The event mentions only the rearmed breaker, obscuring the global side effect.

Proof of Concept

A kernel PoC installs a strict breaker and a permissive breaker sharing history, rearms only the permissive one with `Empty`, and shows the strict breaker then accepts a 100% jump (green demonstrates the disabled protection).

 Copy Code

```
test proxy::circuit_breaker::tests::reaudit_low014_rearming_one_breaker_clears_shared_baseline_for_all_b
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_low014
```

PoC Script

Abridged: breaker 0 permissive, breaker 1 strict StepwiseChange(0.10) with accepted baseline 100.
rearm(breaker_id=0, Empty) clears the shared history; a subsequent 100 to 200 update is accepted even though breaker 1 (never rearmed) should have blocked it.

 Copy Code

```
set.rearm(0, /* Empty */ ...);           // rearm ONLY breaker 0
// shared accepted_history now empty; strict breaker 1 has no baseline
let outcome = set.try_accept_price(price(200), t); // 100 -> 200 (+100%)
assert!(outcome.value.is_ok());           // accepted despite breaker 1 being strict
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Decouple per-breaker rearm from the shared baseline:

- Maintain accepted history per breaker, or make history reseeding an explicit set-level governance operation that clearly applies to all breakers and emits a global event.
- If history remains shared, rearm(breaker_id, ...) should not mutate it implicitly; require a separate ReseedAcceptedHistory action and re-evaluate/rearm all affected breakers.

Until fixed, treat any rearm with Empty/Observed as resetting protection for all breakers on that asset and re-establish a baseline before relying on them.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by removing the ability for a per-breaker **rearm** call to mutate the shared accepted-history baseline; rearming now only changes the addressed breaker's own status.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.21 REARMCONFIG.ARMED_AFTER_SECS IS INTERPRETED AS AN ABSOLUTE TIMESTAMP WHILE DOCS AND TESTS USE IT AS A RELATIVE GRACE DURATION

// LOW

Description

The kernel expects an absolute arm timestamp (`ArmedAfter { timestamp_ns }`). The Soroban surface names the field `armed_after_secs`, and both the `RUNBOOK` and the integration tests use small values (3600 / 60) as though they were relative grace durations. The runtime does not add the current ledger time; it converts and stores those small values as absolute Unix timestamps.

At any modern ledger time those values are already in the past, so the breaker is armed immediately. In the PoC a breaker is rearmed at ledger time 101 with `armed_after_secs = 60`; at $t = 102$ (only one second into the intended 60-second grace) a large move trips again because $102 \geq \text{absolute timestamp } 60$.

Bound: this does not admit a malicious price; it is availability/operations in the fail-closed direction, and it is governance-gated.

Code Location

 Copy Code

```
// soroban/contract/src/lib.rs - the value is forwarded as an absolute ArmedAfter timestamp
// (no env.ledger().timestamp() addition); kernel compares now >= timestamp.
ArmedAfter { timestamp_ns: Nanoseconds::from_secs(config.armed_after_secs) }
// RUNBOOK.md and integration tests pass 3600 / 60 as a relative grace
```

Impact

Operators following the documented/example semantics do not get the intended post-rearm stabilization window: a feed can re-trip immediately, preventing recovery and prolonging unavailability. No wrong price is admitted.

Proof of Concept

An integration PoC rearms with `armed_after_secs = 60` at ledger time 101 and shows the breaker re-trips one second later (green demonstrates the missing grace window).

 Copy Code

```
test reaudit_low015_rearm_grace_seconds_are_treated_as_absolute_timestamp ... ok
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 6 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-integration-tests reaudit_low015
```

PoC Script

Abridged: trip a StepwiseChange breaker, rearm with RearmConfig { armed_after_secs: 60 } at ledger time 101, advance one second to 102, and observe a large move trips again because 102 >= the absolute timestamp 60.

 Copy Code

```
// rearm at t=101 with a value meant as a 60s grace
Rearm(asset, 0, RearmConfig { armed_after_secs: 60, accepted_history_source_code: 1 });
    advance_secs(&env, 1); // t = 102, only 1s into the intended 60s grace
// large move re-trips: 102 >= absolute timestamp 60
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Choose and enforce one unambiguous API:

- Rename to armed_after_unix_secs and require an absolute future timestamp, updating RUNBOOK and tests; or
- Rename to arming_delay_secs and compute env.ledger().timestamp().checked_add(delay) in the runtime.
- Reject values in the past when an absolute timestamp is intended.

Until fixed, operators must pass an absolute future Unix timestamp (current time plus the intended grace), not a relative duration.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by renaming the field to **arming_delay_secs** and computing the absolute arm timestamp at execution time via **env.ledger().timestamp().checked_add(arming_delay_secs)**, so operators now pass a relative grace duration as documented.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.22 SEP-40 ADAPTER TIMESTAMPS ARE NOT ALIGNED TO THE ADVERTISED RESOLUTION

// LOW

Description

The adapter advertises a configurable SEP-40 resolution but does not apply it to returned timestamps or validate that parent history is aligned to it. `lastprice` and `prices copy raw` parent timestamps, while `price(asset, timestamp)` requires an exact raw timestamp match.

In the PoC an adapter advertises resolution 60 but returns timestamp 101; `price(asset, 60)` (the advertised 60-second bucket) returns `None`, while only the non-aligned raw query `price(asset, 101)` succeeds.

Bound: numeric price integrity is unchanged; this is an integration/availability mismatch on time-bucketed reads, and it is owner-configured.

Code Location

 Copy Code

```
// soroban/sep40-adapter-contract/src/lib.rs - resolution is metadata only;
// price(asset, timestamp) does an exact lookup and lastprice/prices copy raw parent timestamps.
// soroban/common/src/lib.rs - normalized_to_sep40 copies the parent timestamp unchanged
// and receives no resolution.
```

Impact

Consumers that derive uniform time buckets from `resolution()` can fail to retrieve recorded prices or build nonuniform TWAP/history windows. No price-integrity impact.

Proof of Concept

An adapter PoC advertises resolution 60 but stores/returns timestamp 101; the aligned query `price(asset, 60)` returns `None` while `price(asset, 101)` succeeds (green demonstrates the mismatch).

 Copy Code

```
test tests::reaudit_adapter_returns_timestamp_not_aligned_to_advertised_resolution ... ok
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 17 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-sep40-adapter-contract reaudit_adapter_returns_timestamp_not_
```

PoC Script

Abridged: configure adapter resolution 60; refresh a parent price at timestamp 101; assert `price(asset, 60)` is `None` (advertised bucket) while `price(asset, 101)` is `Some` (raw).

```
assert_eq!(adapter.resolution(), 60);  
// parent price stored at raw timestamp 101  
assert!(adapter.price(&asset, &60).is_none()); // advertised 60s bucket misses  
assert!(adapter.price(&asset, &101).is_some()); // only raw timestamp works
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Make resolution meaningful and consistent:

- Derive/enforce adapter resolution from the parent feed and reject mismatches, or normalize returned/history timestamps to $\text{floor}(\text{timestamp}/\text{resolution}) * \text{resolution}$ with a clear collision policy.
- Make resolution immutable after deployment.

Until fixed, consumers must query by raw parent timestamps rather than resolution-derived buckets.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding a **bucket_timestamp** helper that aligns returned/history timestamps to **$\text{floor}(\text{timestamp}/\text{resolution}) * \text{resolution}$** , and by making the adapter's resolution immutable after deployment.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

11.23 RENOUNCING ADAPTER OWNERSHIP DOES NOT DECOMMISSION IT AND MAKES FUTURE SHUTDOWN IMPOSSIBLE

// LOW

Description

The README suggests adapter owners can decommission by upgrading to a no-op, renouncing, transferring to a burn address, or stopping publication. Renouncing ownership only clears the owner; it does not change base, assets, price, prices, or lastprice. Reads continue to call the parent and extend instance TTL.

After renounce, no owner remains to call `set_metadata` or `upgrade`, so the adapter continues publishing and cannot be disabled through its own admin surface. Transfer to an inaccessible burn address has the same property.

Bound: owner-initiated and adapter-local. The parent runtime can still remove the proxy (affecting all adapters for that asset) or be upgraded, so the adapter is not absolutely unstoppable.

Code Location

 Copy Code

```
// soroban/sep40-adapter-contract/src/lib.rs - renounce clears owner only;  
// all read methods ignore ownership/decommission state and keep serving + extending TTL.  
// soroban/README.md lists renounce as an adapter decommission option.
```

Impact

An operator following the documented renounce decommission path can create a permanently unmanaged adapter that keeps serving prices, which downstream users may keep treating as active/official, with no owner able to stop it.

Proof of Concept

An adapter PoC renounces ownership, then shows lastprice still serves and `set_metadata/upgrade` are no longer callable (green demonstrates the continued, unmanageable publication).

 Copy Code

```
test tests::reaudit_low017_renounce_does_not_decommission_adapter ... ok  
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 17 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-sep40-adapter-contract reaudit_low017
```

PoC Script

Abridged: configure a working adapter, `renounce_ownership`, then assert lastprice still returns Some and there is no owner left to call `set_metadata/upgrade` to disable it.

```
adapter.renounce_ownership();
assert_eq!(adapter.get_owner(), None);
assert!(adapter.lastprice(&asset).is_some()); // still publishing, unmanageable
```

BVSS

AO:A/AC:L/AX:L/R:L/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Provide a real decommission path:

- Add an explicit owner-gated decommission state checked by all price read methods (return None once set), with a decommission event.
- Require decommission before renounce, or remove renounce/burn-transfer from the documented decommission options and require an upgrade to a no-op before ownership is discarded.

Until fixed, do not treat renounce or burn-transfer as decommissioning; upgrade the adapter to a no-op (or remove the parent proxy) instead.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding an owner-gated **decommission()** state that all price read methods check and return **None** once set, providing a real shutdown path before ownership is renounced.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

11.24 GOVERNANCE EXTEND_TTL IS PERMISSIONLESS BUT DOCUMENTED AS ADMIN-ONLY

// INFORMATIONAL

Description

The governance contract's `extend_ttl` takes no `caller` and performs no `require_auth` or role check, so any account can call it. The soroban README correctly describes it as permissionless, but RUNBOOK section 7 shows an `extend_ttl --caller <ADMIN>` invocation and labels it admin-only, and the ABI has no such parameter.

The bound: extending TTL only renews storage the protocol wants alive, and the caller pays the fee, so there is no identified profit or griefing vector. The issue is a documentation and auth-surface-modeling inaccuracy, not an exploitable condition.

Code Location

```
// soroban/governance-contract/src/lib.rs
pub fn extend_ttl(env: Env) -> Result<(), GovernanceError> {
    extend_instance_ttl(&env);
    extend_active_proposal_ttls(&env, &load_header(&env)?);
    TtlExtended {}.publish(&env);
    Ok(())
}
```

 Copy Code

Impact

No direct security impact. Tooling or runbooks built on the admin-only assumption are wrong, and reviewers may mis-model the authorization surface.

Proof of Concept

Confirmed by an executed test: a non-admin caller successfully calls governance `extend_ttl`. The signature has no `caller` parameter and there is no role gate.

```
running 1 test
test tests::reaudit_info001_extend_ttl_is_callable_by_non_admin ... ok

test result: ok. 1 passed; 0 failed
```

 Copy Code

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-governance-contract --features testutils reaudit_info001_extend_ttl_is_callable_by_non_admin`.

PoC Script

```
let result = env.as_contract(&governance_id, || {  
  ProxyOracleGovernance::extend_ttl(env.clone())  
});  
assert!(result.is_ok()); // permissionless, no caller parameter
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, choose one of the following:

- Fix RUNBOOK section 7 to show `extend_ttl` with no caller (matching the runtime and adapter entries).
- Or, if admin-gating is actually intended, add the auth check to the code.

Until fixed, treat governance `extend_ttl` as permissionless in tooling and monitoring.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by correcting RUNBOOK.md section 7 to describe governance `extend_ttl` as permissionless with no caller parameter, matching the runtime and the README.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.25 REFRESHFAILURE TELEMETRY COLLAPSES DISTINCT SOURCE FAILURES; DOCUMENTED CODES 2 AND 4 ARE UNREACHABLE

// INFORMATIONAL

Description

`source_kernel_price` discards the specific failure reason with `.ok()`: ABI error, callee panic, wrong base, missing price, and conversion overflow all collapse to `None`. When every source yields `None`, refresh reports `SourceUnavailable` (event code 5).

Consequently the documented code 4 (conversion overflow) is never emitted, because conversion errors are swallowed before they can surface. Code 2 (a circuit-breaker resolve error) is also unreachable through the public refresh path: inputs carry `conf = 0` and invalid prices are filtered before aggregation, so the only refresh-time breaker error (`InvalidPrice`) cannot fire. A responding wrong-base source likewise produces code 5, and a partial failure that still meets quorum emits only `RefreshSuccess` with no survivor count. The bound: this is an observability and documentation issue; it does not affect price correctness or fail-closed behavior.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs : failure reasons discarded
let base = client.try_base().ok()?.ok()?;
let decimals = client.try_decimals().ok()?.ok()?;
let price = client.try_lastprice(&source.asset).ok()?.ok()??;
source_price_to_kernel(price, decimals).ok() // conversion error -> None
```

Impact

Monitoring cannot distinguish an outage from malicious metadata, conversion overflow, ABI mismatch, or degraded-but-quorum redundancy. Incident response keyed to codes 2 or 4 will never trigger.

Proof of Concept

Confirmed by an executed test: a source with an unrepresentable decimals value fails conversion, but refresh reports `SourceUnavailable` and caches `ResolveFailed(5)`, not the documented code 4.

 Copy Code

```
running 1 test
test tests::reaudit_source_conversion_overflow_is_reported_as_unavailable_code_five ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaf828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_source_conversion_overflow_is_reported_as_unavailable_code_five`.

PoC Script

 Copy Code

```
source.init(&base, &(i32::MAX as u32 + 1), &1_u32); // decimals unrepresentable
source.set_price(&asset, &1_i128, &100_u64);
let result = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(matches!(result.get(0).unwrap().1, RefreshStatus::SourceUnavailable));
assert!(matches!(proxy.get_cached(&asset).unwrap().status,
    CachedStatus::ResolveFailed(SOURCE_UNAVAILABLE_CODE))); // code 5, not 4
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Preserve typed per-source failure reasons and expose valid/failed source counts in the refresh events.
- Or, fix the RUNBOOK code table to list only the reachable codes and their true meanings.

Until fixed, do not build alerting on codes 2 or 4, and treat code 5 as covering multiple distinct failure classes.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by reassigning refresh-failure codes so unexpected breaker failures map to the storage code, and correcting RUNBOOK.md to document only the codes actually reachable through the public refresh path.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/e8b542dc54f5b49871c6ce2e82c21012ef723216>

11.26 PRICEIDENTIFIER JSON SCHEMA CONTRADICTS ITS SERDE HEX-STRING WIRE FORMAT

// INFORMATIONAL

Description

`PriceIdentifier([u8; 32])` uses custom serde `serialize_with`/`deserialize_with` that encode it as a single 64-character hex string. It does not provide a custom `JsonSchema`, so the derived Schemars schema describes the underlying 32-element integer array. Schema and wire format therefore disagree;

`AccountId` avoids this by supplying a manual string schema.

The bound: there is no on-chain effect. `PriceIdentifier` is unused in the Soroban scope (assets are the `Asset` type), so this is dead code for this deployment and purely an off-chain tooling hazard.

Code Location

 Copy Code

```
// kernel/src/primitive/price_id.rs : hex-string serde, but no custom JsonSchema
pub struct PriceIdentifier(
    #[cfg_attr(feature = "serde", serde(
        serialize_with = "hex::serde::serialize",
        deserialize_with = "hex::serde::deserialize"))]
    pub [u8; 32],
);
```

Impact

Off-chain validators or codegen built from the schema reject the real hex-string wire form, and vice versa.

This can cause integration failures or canonicalization disagreement in off-chain or NEAR tooling; no Soroban on-chain impact.

Proof of Concept

Confirmed by an executed test: the serde value is a string, while the derived schema models an array.

 Copy Code

```
running 1 test
test primitive::price_id::tests::reaudit_schema_disagrees_with_hex_string_serde_shape ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo `Templar-Protocol/contracts` at commit `fde891da949ccdc615198175847de72cfdaef828`, full workspace.
- Run `cargo test -p templar-proxy-oracle-kernel --features 'serde,schemars' reaudit_schema_disagrees_with_hex_string_serde_shape`.

PoC Script

```
let id = PriceIdentifier([0_u8; 32]);
assert!(to_value(id).unwrap().is_string());
let schema = schema_for!(PriceIdentifier).schema;
assert_ne!(schema.instance_type, Some(InstanceType::String.into()));
assert!(schema.array.is_some()); // schema models the byte array, not the hex string
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, the following improvement is recommended:

- Give `PriceIdentifier` a manual string `JsonSchema` mirroring `AccountId` (fixed-length hex pattern), so the schema matches the serde wire format.

Until fixed, off-chain consumers must not rely on the generated schema for `PriceIdentifier`.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by giving `PriceIdentifier` a manual `JsonSchema` implementation that describes the exact 64-character hex string wire format, mirroring `AccountId`'s approach.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/6d12edfb72a2f1136e56ee48ee6dfb51b306bb85>

11.27 SOURCE OUTAGE MASKS AN ACTIVE MANUAL TRIP IN REFRESH TELEMETRY (STATUS PRECEDENCE)

// INFORMATIONAL

Description

In `compute_refresh`, the all-sources-`None` check returns `SourceUnavailable` before the breaker state is loaded. If a feed is manually tripped and all sources also return `None`, refresh reports `SourceUnavailable` (code 5, `RefreshFailure`) instead of `Blocked`/`CacheBlocked`.

The bound: this is purely a telemetry precedence issue. Reads still fail closed and the breaker view still shows the feed as manually tripped, so there is no safety impact; only the emitted failure code and event are misleading. PROOF.md implies a tripped feed always signals a block event.

Code Location

 Copy Code

```
// soroban/contract/src/refresh.rs : all-None returns before breaker load
if prices.iter().all(Option::is_none) {
    return RefreshStatus::SourceUnavailable; // precedes manual-trip/breaker check
}
```

Impact

A manually tripped, source-down feed is reported as a source outage rather than a manual block.

Monitoring keyed to block events could misattribute the state, though the feed remains correctly unavailable to readers.

Proof of Concept

Confirmed by an executed test: with the feed manually tripped and all sources cleared, refresh returns `SourceUnavailable`/code 5 while `is_manually_tripped` stays true and reads stay `None`.

 Copy Code

```
running 1 test
test tests::reaudit_info005_source_outage_masks_active_manual_trip ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaf828, full workspace.
- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_info005_source_outage_masks_active_manual_trip`.

PoC Script

```

proxy.set_manual_trip(&asset, &true, &None);
source.clear_price(&asset);
let result = proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(matches!(result.get(0).unwrap().1, RefreshStatus::SourceUnavailable));
assert!(proxy.get_breaker_set_view(&asset).unwrap().is_manually_tripped);
assert_eq!(legacy_lastprice(&proxy, &asset), None); // still fail-closed

```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, choose one of the following:

- Check manual-trip/blocking state before the all-sources-unavailable early return so the emitted status reflects the block.
- Or document the status precedence (source availability is reported before block state).

Until fixed, monitoring should treat a tripped feed as blocked regardless of a concurrent source-unavailable code.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding a **blocking_reason()** check that runs before the all-sources-unavailable early return, so a manually tripped feed now reports **Blocked** instead of the misleading **SourceUnavailable** code.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.28 SHARED WEIGHTED-MEDIAN KERNEL COUNTS ZERO-WEIGHT SOURCES TOWARD MIN_SOURCES

// INFORMATIONAL

Description

The shared median kernel permits zero-weight sources (`WeightedSource::new` accepts `weight: u32` including 0), but its quorum check counts all valid source slots (`valid_sources = values.len() / 2`) rather than positive-weight slots. A `min_sources` quorum can therefore be satisfied with zero total voting weight (all-zero weights fall back to an ordinary positional median) or with only one positive-weight source while zero-weight sources fill the quorum.

The bound: this is not reachable through the current Soroban runtime, which constructs every `WeightedSource` with weight 1. It is a shared-kernel/API guardrail issue that only affects a runtime or off-chain consumer that exposes non-unit weights.

Code Location

 Copy Code

```
// kernel/src/proxy/aggregator/method/median/mod.rs
let min_sources = self.min_sources.max(1);
let valid_sources = values.len() / 2; // counts slots regardless of weight
if valid_sources < min_sources as usize { /* error */ }
```

Impact

A weighted configuration could appear to require N sources while being determined by fewer positive-weight sources. The current Soroban deployment is not exposed (all weights are 1); the risk is latent for other kernel consumers.

Proof of Concept

Confirmed by an executed kernel test: an all-zero-weight quorum still returns a price, and one positive-weight source with two zero-weight sources satisfies `min_sources = 3` while the positive-weight source alone determines the result.

 Copy Code

```
running 1 test
test proxy::aggregator::method::median::tests::reaudit_p1234_zero_weight_sources_count_toward_min_source
test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo `Templar-Protocol/contracts` at commit `fde891da949ccdc615198175847de72cfdaef828`, full workspace.
- Run `cargo test -p templar-proxy-oracle-kernel reaudit_p1234_zero_weight_sources_count_toward_min_sources`.

PoC Script

 Copy Code

```
// One positive-weight source, two zero-weight sources, min_sources = 3.  
let prices = vec![Some(price(100, 0, 0)), Some(price(1, 0, 0)), Some(price(1, 0, 0))];  
let result = median_low(&[1, 0, 0], 3).aggregate(prices).unwrap();  
assert_eq!(result.price, 100); // quorum satisfied, one source decided it
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, the following improvements are recommended:

- Reject `weight == 0` in the `WeightedSource` constructor or config validation.
- Or compute quorum over positive-weight survivors and require total survivor weight greater than zero.

Until fixed, any non-Soroban consumer of the kernel must avoid zero-weight sources.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by counting only positive-weight sources toward `valid_sources` in the shared median kernel's quorum check, so zero-weight sources can no longer fill quorum slots.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/fb16191412cf1168abe333550c84844503dcab5c>

11.29 DECIMAL::FROM_STR ACCEPTS TRAILING GARBAGE IN FRACTIONAL STRINGS

// INFORMATIONAL

Description

The external `templar-primitives` `Decimal` `FromStr` parser iterates the fractional component with `frac.chars().take(38)` and breaks on the first non-digit, then returns `Ok`. Malformed strings such as `"1.2abc"`, `"1.2.3"`, and `"1.2_3"` therefore parse as exactly `1.2`. The `serde` deserializer delegates to this parser, while the `Schemars` pattern requires strict decimal syntax, so schema validation and `serde` parsing disagree.

The bound: this affects off-chain or JSON paths that accept operator or user-provided `Decimal` strings. It is not reachable through the Soroban contract ABI, where breaker configuration uses `SorobanDecimal(BytesN<64>)` rather than decimal strings.

Code Location

 Copy Code

```
// primitives/src/number.rs : FromStr fractional loop breaks on non-digit, returns Ok
for c in frac.chars().take(FRACTIONAL_DECIMAL_DIGITS) {
    if let Some(d) = c.to_digit(10) { /* accumulate */ } else { break; }
}
Ok(Self { repr: whole.saturating_add(...) })
```

Impact

Off-chain tools, JSON APIs, or non-Soroban runtimes can silently accept malformed operator or user inputs that a schema validator rejects. This can cause canonicalization disagreement or hidden configuration typos; no direct on-chain Soroban impact.

Proof of Concept

Confirmed by an executed test: several malformed fractional strings all parse to the same value as the clean prefix.

 Copy Code

```
running 1 test
test number::tests::reaudit_decimal_parser_accepts_trailing_fraction_garbage ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo `Templar-Protocol/contracts` at commit `fde891da949ccdc615198175847de72cfdaef828`, full workspace (external primitives crate).
- Run `cargo test -p templar-primitives reaudit_decimal_parser_accepts_trailing_fraction_garbage`.

PoC Script

 Copy Code

```
let clean = Decimal::from_str("1.2").unwrap();
assert_eq!(Decimal::from_str("1.2abc").unwrap(), clean);
assert_eq!(Decimal::from_str("1.2.3").unwrap(), clean);
assert_eq!(Decimal::from_str("1.2_3").unwrap(), clean);
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, the following improvement is recommended:

- Make `FromStr` reject any non-digit character in the fractional component (and, if desired, reject an empty fractional suffix) instead of truncating at the first invalid character.

Until fixed, off-chain consumers must validate `Decimal` strings against the strict pattern before parsing.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by rewriting `Decimal::from_str` to reject any non-digit character in the fractional component instead of truncating at the first invalid character.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/dd566daa069d05e7138d398a9aa8a6e445ad919a>

11.30 A FRESH CIRCUIT-BREAKER TRIP TRIPS AND REPORTS ONLY THE FIRST BLOCKING BREAKER

// INFORMATIONAL

Description

When a single accepted price update simultaneously meets the trip conditions of several enforced circuit breakers, `apply_armed_breaker_transitions` iterates breakers in id order and returns at the first one that blocks. Only the lowest-id breaker is tripped and emits a `Tripped` event, and

`PriceBlockedReason::BreakerTripped` lists only that single id.

The remaining breakers whose conditions were also met are not evaluated or tripped that round, and while the first breaker stays blocking they are never tripped on later rounds either, because the already-blocking short-circuit fires first. That already-blocking path, by contrast, reports the full set of currently-blocking ids, so the two code paths return different id lists for what is conceptually the same blocked state.

The bound: this is an internal state-machine and telemetry behavior with no attacker interaction. The feed is correctly blocked in every case; only the per-breaker trip events and the reported id list are incomplete.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/set.rs : apply_armed_breaker_transitions
for (breaker_id, breaker) in &mut self.0.breakers {
    if breaker.is_armed_at(now) && breaker.breaker.should_trip(proposed_accepted_history) {
        events.push(breaker.trip(*breaker_id, price_update, now));
    }
    if breaker.is_blocking() {
        return CircuitBreakerOutcome::new(Err(PriceBlockedReason::BreakerTripped {
            blocking_breaker_ids: vec![*breaker_id], // only the first blocking id
        }))
        .with_events(events);
    }
}
```

Impact

Observability only. Per-breaker trip telemetry and the reported `blocking_breaker_ids` under-report which breakers' conditions were actually met on a fresh trip, which can complicate incident analysis and attribution. There is no wrong-price, fund-loss, or availability impact; on Soroban all blocked outcomes map to the same cache reason code (2).

Proof of Concept

Confirmed by an executed test. Two enforced StepwiseChange breakers (both 10%) see a 10x jump; both conditions are met, but only breaker 0 trips and is reported.

 Copy Code

```
running 1 test
test proxy::circuit_breaker::tests::reaudit_p5_only_first_blocking_breaker_trips_and_is_reported ... ok
test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.
- Run `cargo test -p templar-proxy-oracle-kernel reaudit_p5_only_first_blocking_breaker_trips_and_is_reported`.

PoC Script

 Copy Code

```
// Two enforced StepwiseChange breakers, both max_relative_change = 0.10.
set.try_accept_price(price(100), Nanoseconds::from_secs(1)).unwrap();
let outcome = set.try_accept_price(price(1000), Nanoseconds::from_secs(2)).unwrap();
assert_eq!(outcome.value, Err(PriceBlockedReason::BreakerTripped {
    blocking_breaker_ids: vec![0], // only breaker 0, though both met the condition
}));
let tripped: Vec<u32> = outcome.events.iter().filter_map(|e| match e {
    CircuitBreakerEvent::Tripped { breaker_id, .. } => Some(*breaker_id), _ => None }).collect();
assert_eq!(tripped, vec![0]); // only one Tripped event
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, the following improvement is recommended:

- Evaluate and trip every armed breaker whose condition is met before returning, collecting all tripped ids into `blocking_breaker_ids` and emitting one `Tripped` event per breaker, so fresh-trip telemetry matches the already-blocking path.

Until fixed, incident responders should not treat the reported blocking id as the exhaustive set of breakers whose conditions were met.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `apply_armed_breaker_transitions` evaluate and trip every armed breaker whose condition is met, collecting all blocking breaker ids into a single event instead of stopping at the first match.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.31 NO-OP SET_MANUAL_TRIP EMITS A MANUALTRIPSET EVENT AND CLEARS THE CACHE, CONTRADICTING THE DOCS

// INFORMATIONAL

Description

The kernel's `set_manual_trip` returns an empty outcome (no event) when the requested manual-trip state equals the current state, so a no-op does not produce a kernel event. The runtime entrypoint, however, wraps the call in `with_breakers` (which always invalidates the cache and re-stores the set) and then publishes its own `ManualTripSet` event unconditionally, regardless of whether anything changed. As a result, a no-op manual trip (for example setting the flag to `false` while it is already `false`) still emits a `ManualTripSet` event and clears a previously-accepted cache entry. This contradicts the documentation, which states that no-op manual-trip executions do not emit a manual-trip event.

The bound: the action is governance-gated (the `ManualTripper` role or Admin, via a matured proposal) and has no wrong-price or fund-loss impact. It is an observability and minor availability issue only.

Code Location

 Copy Code

```
// soroban/contract/src/lib.rs : runtime set_manual_trip
with_breakers(&env, &asset, |breakers| {
    let outcome = breakers.set_manual_trip(is_manually_tripped, /* actor */, kernel_metadata);
    Ok(((), outcome.events)) // kernel emits no event on a no-op
})?;
ManualTripSet { asset, is_manually_tripped, metadata }.publish(&env); // published unconditionally
// with_breakers always calls invalidate_cache(&env, &asset)
```

Impact

Observability and minor availability. Event consumers and monitoring can see a `ManualTripSet` event with no actual state change, contradicting the documented no-op-no-event semantics, and a governed no-op needlessly drops the cached price so `aggregated_latest` returns `None` until the next `refresh`. No wrong-price or fund-loss vector.

Proof of Concept

Confirmed by an executed test. After an accepted refresh, a no-op manual trip (false while already false) emits a `ManualTripSet` event and clears the cache.

 Copy Code

```
running 1 test
test tests::reaudit_info008_noop_manual_trip_emits_event_and_clears_cache ... ok

test result: ok. 1 passed; 0 failed
```

Environment / Reproduce

- Repo Templar-Protocol/contracts at commit fde891da949ccdc615198175847de72cfdaef828, full workspace.

- Run `cargo test -p templar-proxy-oracle-soroban-contract --features testutils reaudit_info008_noop_manual_trip_emits_event_and_clears_cache`.

PoC Script

 Copy Code

```
source.set_price(&asset, &5_000_000_000_i128, &100_u64);
proxy.refresh(&Vec::from_array(&env, [asset.clone()]));
assert!(proxy.get_cached(&asset).is_some());
// No-op: set manual trip to its current value (false -> false).
proxy.set_manual_trip(&asset, &false, &None);
// A ManualTripSet event is emitted and the accepted cache is cleared.
assert_eq!(contract_events(&env, &proxy.address),
    vec![ManualTripSet { asset, is_manually_tripped: false, metadata: None }.to_xdr(...)]);
assert!(proxy.get_cached(&asset).is_none());
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

To mitigate this issue, choose one of the following:

- Gate the runtime `ManualTripSet.publish` (and ideally the cache invalidation) on whether the kernel outcome actually changed state, for example only publish and invalidate when `outcome.events` is non-empty.
- Or update the documentation to state that on Soroban a manual-trip call always emits the event and clears the cache, even on a no-op.

Until fixed, treat a ManualTripSet event as not implying a state change, and expect a cache reset after any manual-trip call.

Remediation Comment

SOLVED: The `Templar Protocol` team has solved this issue by making `with_breakers` compare state before and after the operation, only persisting, publishing the `ManualTripSet` event, and invalidating the cache when the state actually changed.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.32 SHARED GOVERNANCE KERNEL CREATE() ACCEPTS A TTL BELOW THE OPERATION MINIMUM (TIMELOCK ENFORCEMENT NOT OWNED BY THE KERNEL)

// INFORMATIONAL

Description

The shared governance kernel (governance-kernel, a chain-agnostic crate meant to own every invariant of the proposal lifecycle) exposes the per-operation minimum timelock policy through `OperationPolicy::minimum_ttl` and the `Governance::effective_ttl` helper. However, `Governance::create` takes a final `ttl` argument and stores it verbatim; it never compares that value against the operation minimum. A direct caller can pass a `ttl` below the configured minimum (even zero) and obtain an immediately executable proposal.

The minimum-enforcement logic exists only in the separate `effective_ttl` function, which `create()` does not call. Correctness therefore depends on every runtime computing `effective_ttl` and passing its result into `create`. The audited Soroban governance contract does this correctly - `create_proposal` computes `effective_ttl` and additionally caps it at `MAX_PROPOSAL_TTL_NS` before calling the kernel - so there is no bypass in the deployed system.

This is an API footgun in a shared kernel; it is informational because the audited runtime is not exposed. A future or external runtime that integrates the kernel and calls `create` with an untrusted or requested `ttl` instead of the `effective_ttl` result would silently lose all operation timelocks.

Code Location

 Copy Code

```
// governance-kernel/src/lib.rs - create stores the caller-supplied ttl verbatim
pub fn create<Operation, AccountId>(&mut self, id: u64, operation: Operation,
    now: Nanoseconds, created_by: AccountId, ttl: Nanoseconds) -> Result<...> {
    // id / pending-cap / validation checks ...
    Ok(Proposal { operation, created_at: now, ttl, created_by }) // ttl NOT clamped to minimum
}

// minimum enforcement lives only in the separate helper create() does not call:
pub fn effective_ttl<Operation>(&self, operation: &Operation, requested_ttl: Nanoseconds)
    -> Nanoseconds { operation.minimum_ttl(&self.ttls).max(requested_ttl) }
```

Impact

No impact on the deployed Soroban contract, which computes and bounds `effective_ttl` before `create`. The risk is to any future or external kernel consumer that calls `create` directly with a requested `ttl`: all configured operation timelocks (including sensitive ones) could be silently bypassed and executed immediately.

Proof of Concept

A direct kernel PoC constructs a governance whose `Slow` operation has a 10-second minimum, then calls `create` with `ttl=0` and executes immediately - the green test demonstrates the missing enforcement.

```
running 16 tests
test tests::reaudit_info016_create_does_not_enforce_operation_minimum_ttl ... ok
test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 5 filtered out
```

Environment / Reproduce

```
cargo test -p templar-proxy-oracle-governance-kernel reaudit_info016
```

PoC Script

```
// governance-kernel/src/lib.rs tests (abridged)
let mut governance = governance(); // Slow operation minimum TTL = 10 seconds
assert_eq!(governance.effective_ttl(&op, Nanoseconds::zero()), Nanoseconds::from_secs(10));
// but create() ignores the minimum and stores ttl = 0
let proposal = governance.create(0, op, now, "alice", Nanoseconds::zero()).unwrap();
assert_eq!(proposal.ttl, Nanoseconds::zero());
assert!(proposal.can_execute(now)); // immediately executable
governance.execute(0, &proposal, now).unwrap();
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Make the kernel enforce its own invariant so no consumer can skip it:

- Have create compute `operation.minimum_ttl(&self.ttls).max(requested_ttl)` internally (accept a `requested_ttl` rather than a pre-computed final ttl), or make raw-ttl creation private and expose only a `create_with_effective_ttl` entry point.
- At minimum, reject `ttl < operation.minimum_ttl(&self.ttls)` inside create.

The audited Soroban runtime already computes `effective_ttl` before create; this hardening protects future integrations of the shared kernel.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by making **Governance::create** compute **`operation.minimum_ttl(&self.ttls).max(requested_ttl)`** internally, so the kernel itself enforces the per-operation minimum timelock rather than relying on callers.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/5c930310554c4272f0dbbf3cef5fe8c45373d74f>

11.33 SHARED GOVERNANCE KERNEL GOVERNANCE HAS NO VALIDATING CONSTRUCTOR: A CORRUPT LEDGER CAN DOUBLE-CONSUME A PROPOSAL ID

// INFORMATIONAL

Description

The governance kernel's replay and ordering guarantees rest on two `active_ids` invariants: each pending proposal id appears at most once, and every id is less than `next_id`. These are upheld only by the `create/cancel/execute` methods. The `Governance` struct exposes all fields as `pub` and derives `borsh/serde` with no validating `TryFrom` or constructor, so a ledger rebuilt from hand-built or corrupted/attacker-influenced bytes can violate them.

With a duplicate id in `active_ids` the raw kernel consumes the same id twice: each `execute/cancel` removes only the first match. This defeats the no-double-spend-of-a-proposal property the kernel is documented to own. Contrast the circuit-breaker set, which validates parse-time invariants in its own `try_from`.

It is not reachable in the audited Soroban runtime, which is doubly protected: proposal bodies live in per-id persistent storage, and `cancel_proposal/execute_proposal` both require `load_proposal(id)` to succeed and delete the body on first consume, so a corrupt header alone cannot double-fire a side effect. This is an informational shared-kernel robustness gap.

Code Location

 Copy Code

```
// governance-kernel/src/lib.rs - all fields public, no validating constructor / TryFrom
pub struct Governance<TtlConfig> {
    pub next_id: u64,
    pub active_ids: Vec<u64>,
    pub ttls: TtlConfig,
    pub max_pending_proposals: u32,
}

// pending_index (the anti-replay guard) trusts active_ids uniqueness
fn pending_index(&self, id: u64) -> Result<usize, PendingLookupError> {
    if id >= self.next_id { /* IdOutOfBounds */ }
    self.active_ids.iter().position(|candidate| *candidate == id).ok_or(...)
}
```

Impact

No impact on the audited Soroban contract, where per-id body existence is an independent gate. A future or external kernel consumer that deserializes untrusted or migration-corrupted state directly into `Governance` and keys proposal existence solely on kernel `active_ids` membership could execute one proposal id's side effects twice, or leave a consumed id resurrectable.

Proof of Concept

A raw-kernel PoC hand-builds an invalid ledger the public API can never produce (`active_ids = [0, 0]`, `next_id = 1`) and shows `execute` consuming id 0 twice. The green test demonstrates the missing validation.

```
running 16 tests
test tests::reaudit_p8_corrupt_duplicate_active_ids_double_consume_raw_kernel ... ok
test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 5 filtered out
```

Environment / Reproduce

```
cargo test -p templar-proxy-oracle-governance-kernel reaudit_p8_corrupt
```

PoC Script

```
let mut governance = governance();
governance.next_id = 1;
governance.active_ids = vec![0u64, 0u64]; // invalid: duplicate id
let body = Proposal { operation: Op("fast", Kind::Fast), created_at: Nanoseconds::zero(),
    ttl: Nanoseconds::zero(), created_by: "alice" };
governance.execute(0, &body, Nanoseconds::from_secs(1)).unwrap();
assert_eq!(governance.active_ids(), &[0]); // one copy consumed
governance.execute(0, &body, Nanoseconds::from_secs(1)).unwrap(); // consumed AGAIN
assert!(governance.active_ids().is_empty());
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Enforce the lifecycle invariants at the kernel boundary, mirroring the circuit-breaker set:

- Add a validating constructor / TryFrom that rejects duplicate ids, any id $\geq$ next_id, and active_ids.len() > max_pending_proposals, and route all borsh/serde deserialization through it.
- Prefer keeping the fields private behind accessors.
- At minimum, document that consumers must validate deserialized Governance state and gate proposal existence on an independent per-id record, as the Soroban runtime does.

No change is required for the deployed Soroban contract; this hardens future integrations of the shared kernel.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding a validating **TryFrom** constructor for **Governance** that rejects duplicate/out-of-order active ids, ids beyond **next_id**, and excess pending proposals, routed through both serde and borsh deserialization with private fields.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/5c930310554c4272f0dbbf3cef5fe8c45373d74f>

11.34 DECIMAL::FROM_STR SILENTLY WRAPS SCHEMA-VALID WHOLE VALUES AT 2^{384} INSTEAD OF REJECTING THEM

// INFORMATIONAL

Description

Decimal stores a U512 fixed-point representation with 128 fractional bits, leaving only 384 bits for the whole number. FromStr first parses the whole component as a full U512 and then shifts it left by 128 bits. It never verifies that the whole value fits in 384 bits, so the shift silently truncates the high bits (wraps modulo 2^{384}).

The JSON schema's whole-number pattern permits 116-digit values and does not enforce the numeric maximum. The schema-valid decimal string 2^{384} therefore parses (and serde JSON deserializes) as exactly Decimal::ZERO; $2^{384} + 1$ parses as Decimal::ONE.

Bound: there is no current Soroban on-chain path. SorobanDecimal accepts the raw 64-byte representation and cannot represent a value above the Decimal range. The excluded NEAR runtime and off-chain library/JSON consumers may be affected.

Code Location

 Copy Code

```
// primitives/src/number.rs - whole part parsed as full U512 then shifted, no range check
let whole = U512::from_dec_str(whole_str)?;
let value = whole << FRACTIONAL_BITS; // 128; silently drops bits above 2^384
// schema whole-number pattern [1-9][0-9]{0,115} allows 116 digits, not the numeric max
```

Impact

Off-chain JSON/config consumers or another runtime using Decimal strings can submit a schema-valid large threshold/amount and silently receive a completely different small value, including zero. For a breaker threshold this can invert behavior from effectively inert/huge to hair-trigger/zero. No current Soroban on-chain path.

Proof of Concept

A primitives PoC parses 2^{384} (and via serde JSON) and shows it equals Decimal::ZERO, and $2^{384} + 1$ equals Decimal::ONE (green demonstrates the silent wrap).

 Copy Code

```
test number::tests::reaudit_decimal_parser_wraps_whole_part_at_two_pow_384 ... ok
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 65 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-primitives reaudit_decimal_parser_wraps_whole_part
```

PoC Script

Abridged: parse the schema-valid decimal string for 2^{384} and assert it equals `Decimal::ZERO`; parse $2^{384} + 1$ and assert it equals `Decimal::ONE`; repeat through the serde JSON path.

 Copy Code

```
let two_pow_384: &str = /* decimal string of 2^384 */;
assert_eq!(Decimal::from_str(two_pow_384).unwrap(), Decimal::ZERO);
assert_eq!(Decimal::from_str(two_pow_384_plus_1).unwrap(), Decimal::ONE);
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Reject over-range whole components before shifting:

- Require `whole.bits() <= 512 - FRACTIONAL_BITS (384)`, or use a checked range test against $2^{384} - 1$, and return a parse error otherwise.
- Update the JSON schema to enforce the actual numeric maximum rather than digit count alone.

Not reachable through the Soroban ABI; this hardens off-chain and non-Soroban consumers of the primitive.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by rejecting whole components above the exact 384-bit representable maximum before shifting, instead of silently wrapping, and by regenerating the JSON schema pattern to match the true representable range.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/dd566daa069d05e7138d398a9aa8a6e445ad919a>

11.35 CIRCUITBREAKERTRIPPED TELEMETRY DROPS THE NORMALIZED EXPONENT AND SOURCE PUBLISH TIMESTAMP

// INFORMATIONAL

Description

The kernel trip event contains a full Observation, whose Price includes { price, conf, expo, publish_time_ns }, plus observed_at_ns. The compact Soroban CircuitBreakerTripped event flattens this to { tripped_at_secs, price, timestamp, is_enforced } but omits expo and sets timestamp = observed_at_ns.as_secs(). Therefore timestamp duplicates tripped_at_secs and the upstream source publish time is lost. Because price is only the mantissa, consumers also cannot reconstruct the numeric price without the exponent (especially when source decimals differ or source conversion downscales a large i128). RefreshSuccess, by contrast, emits both mantissa and expo and preserves the source timestamp.

Bound: the feed is still correctly blocked; this is observability only, with no price-integrity, availability, or fund-loss impact.

Code Location

 Copy Code

```
// soroban/contract/src/events.rs - event has price (mantissa) but no expo, and
// timestamp is populated from Observation.observed_at_ns, not price.publish_time_ns;
// tripped_at_secs uses the same ledger time => timestamp duplicates tripped_at_secs.
```

Impact

Monitoring and incident-response systems cannot reconstruct the actual normalized trip price or determine the age of the source observation that caused the trip. A 1-second-old source price that trips at ledger 101 reports both tripped_at_secs=101 and timestamp=101, hiding the publish time 100; the mantissa has no exponent.

Proof of Concept

A contract PoC trips a breaker and inspects the emitted event, confirming expo is absent and timestamp equals tripped_at_secs rather than the source publish time (green demonstrates the telemetry loss).

 Copy Code

```
test tests::reaudit_info011_trip_event_loses_publish_timestamp_and_exponent ... ok
test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 44 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-contract reaudit_info011
```

PoC Script

Abridged: refresh a source price with publish time 100 that trips at ledger time 101; read the CircuitBreakerTripped event and assert timestamp == 101 (== tripped_at_secs, not 100) and that no exponent field is present.

 Copy Code

```
// source publish_time = 100, trips at ledger 101
let ev = last_circuit_breaker_tripped_event();
assert_eq!(ev.tripped_at_secs, 101);
assert_eq!(ev.timestamp, 101); // duplicates tripped_at_secs; source ts 100 lost
// event carries only the mantissa: no expo field to reconstruct the price
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Preserve the price scale and the source time:

- Add expo and a separate publish_timestamp_secs field (derived from price.publish_time_ns) to CircuitBreakerTripped.
- Retain tripped_at_secs (or observed_at_secs) for the ledger observation time, mirroring the kernel/NEAR event's distinction between publish_time_ns and observed_at_ns.

Observability only; no on-chain behavior change is required.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by adding **expo** and **publish_timestamp_secs** fields to the **CircuitBreakerTripped** event, preserving the price scale and the upstream source publish time.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/b289b21ab48b6add6e2e4d15c298ca7826bfdb2a>

11.36 WEIGHTED MEDIANHIGH CAN SELECT A MINORITY OR ZERO-WEIGHT SOURCE

// INFORMATIONAL

Description

For unequal weights, the two-pointer weighted-median helper does not produce a correct high-median index. With two sorted items [(A,2),(B,1)] the standard repeated-weight multiset is [A,A,B], so both the low and high positional medians are A. The helper returns (lo,hi) = (1,0); Low::median selects A (index 0) correctly, but High::median selects B (index 1), even though B has minority weight. With [(A,1),(B,0)] MedianHigh likewise selects the zero-weight B.

An exhaustive model over every length 1..6 weight vector with weights 0..4 found MedianLow matches the reference, isolating the defect to MedianHigh.

Bound: the current Soroban runtime is not affected - kernel_proxy_from_config always constructs MedianLow and hardcodes every source weight to 1. The defect is reachable only via a non-Soroban consumer using MedianHigh with unequal/custom weights.

Code Location

 Copy Code

```
// kernel/src/proxy/aggregator/method/median/mod.rs - weighted median returns a (lo, hi)
// pair that can be reversed for unequal weights; High::median = lo.max(hi) then mis-selects.
// High::median(...) -> chooses lo.max(hi)
```

Impact

A runtime/library consumer using weighted MedianHigh can return a price supplied by a minority or even zero-weight source, defeating the configured weight semantics. No deployed Soroban path (MedianLow, equal weights).

Proof of Concept

A kernel PoC shows MedianHigh selecting a zero/lower-weight item, plus an exhaustive control confirming MedianLow matches a repeated-weight reference (green on the MedianHigh test demonstrates the mis-selection).

 Copy Code

```
test proxy::aggregator::method::median::tests::reaudit_info012_median_high_selects_zero_or_lower_weight_
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_info012
```

PoC Script

Abridged: with items [(A,2),(B,1)] the repeated-weight multiset is [A,A,B]; assert High::median returns B (the minority-weight item) while the reference high median is A; with [(A,1),(B,0)] assert MedianHigh returns the zero-weight B.

 Copy Code

```
// [(A, weight 2), (B, weight 1)] -> multiset [A, A, B]; reference high median = A
assert_eq!(median_high(&[(A, 2), (B, 1)]), B); // BUG: minority-weight B selected
assert_eq!(median_high(&[(A, 1), (B, 0)]), B); // BUG: zero-weight B selected
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Compute weighted medians from cumulative positive weight:

- Implement low/high weighted medians from cumulative positive weight (or an equivalent proven algorithm); do not derive the high median as max(lo,hi) from the current two-pointer pair.
- Test both variants against a repeated-weight reference for mixed, zero, and dominant weights.

No change is required for the deployed Soroban runtime (MedianLow, equal weights); this hardens future integrations using MedianHigh or custom weights.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by computing low/high weighted medians from cumulative positive weight instead of deriving the high median from a two-pointer pair, validated against a repeated-weight reference across exhaustive weight combinations.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/fb16191412cf1168abe333550c84844503dcab5c>

11.37 CONSERVATIVE DECIMAL SCALING CAN FALSELY TRIP A BREAKER FOR A REPRESENTABLE BELOW-THRESHOLD MOVE

// INFORMATIONAL

Description

The positive-exponent `mul_pow10` guard compares the value's leading-zero count with the operand bit length, which is sufficient but not necessary for a product to fit U512. Some representable scaled ratios are rejected (None) and breaker math replaces them with `Decimal::MAX`.

In the PoC the true relative move is about $2.5e115$, below a representable configured threshold of about $3e115$, but scaling returns None, maps to MAX, and `StepwiseChange` falsely trips.

Bound: false blocking only, no wrong-price acceptance. Requires extreme supported exponents (example previous {1,-97}, current { $2.5e18$,0}) and a near-maximum threshold; ordinary feeds are unaffected. Fail-closed (availability only).

Code Location

 Copy Code

```
// primitives/src/number.rs - mul_pow10 rejects some representable products (bit-length guard)
// kernel/src/proxy/circuit_breaker/math.rs - maps that None to Decimal::MAX, so a
// below-threshold move can compare as MAX and trip.
```

Impact

A representable, below-threshold extreme move is falsely blocked. There is no wrong-price acceptance; the failure direction is availability (fail-closed). Requires abnormal source decimals/exponent gaps and a near-maximum threshold.

Proof of Concept

A kernel PoC feeds a representable below-threshold extreme move and shows `StepwiseChange` trips due to the conservative scaling saturating to MAX (green demonstrates the false trip).

 Copy Code

```
test proxy::circuit_breaker::tests::reaudit_info013_representable_scaled_ratio_falsely_saturates_and_trips
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_info013
```

PoC Script

Abridged: previous {1,-97}, current { $2.5e18$,0} gives a true relative move $\sim 2.5e115$; `StepwiseChange` threshold $\sim 3e115$ (representable). Scaling returns None, maps to `Decimal::MAX`, and `should_trip` is true even though the move is below threshold.

```
// true move ~2.5e115 < threshold ~3e115, yet scaling saturates to MAX  
assert!(stepwise.should_trip(&history)); // false positive trip (fail-closed)
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Make scaling exact instead of conservative:

- Replace the conservative bit-length test in `mul_pow10` with actual checked/overflowing U512 multiplication.
- For negative exponents, divide first and then test whether the actual quotient is zero.

Fail-closed and extreme-only; no wrong-price impact. This improves availability at abnormal scales.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by replacing the conservative bit-length overflow guard in **`mul_pow10`** with exact checked/overflowing multiplication and division, so representable below-threshold moves are no longer falsely rejected.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/dd566daa069d05e7138d398a9aa8a6e445ad919a>

11.38 MONOTONICRUN EXCLUDES STEPS EXACTLY EQUAL TO ITS DOCUMENTED MINIMUM THRESHOLD

// INFORMATIONAL

Description

The generic helper tests `change > threshold`, which is correct for maximum thresholds (`StepwiseChange / WindowedChangeDelta`) but is reused for `min_relative_step_change` in `MonotonicRun`. Consequently a step whose relative change is exactly the configured minimum classifies as `Minor` and does not count toward the streak, whereas `PROOF.md` describes steps greater than or equal to the minimum as counting.

This is an off-by-a-boundary case: a step exactly equal to the minimum is excluded rather than included. Bound: kept Informational because a step epsilon-below the minimum already evades `MonotonicRun` by design (the broader gradual-drift limitation is documented and captured separately). This finding is the exact-boundary manifestation of that documented behavior.

Code Location

 Copy Code

```
// kernel/src/proxy/circuit_breaker/math.rs - classify_step_change reuses the strict
// relative_abs_change_exceeds (change > threshold), so change == min_relative_step_change
// is classified Minor.
if relative_abs_change_exceeds(previous, current, min_relative_step_change) {
    step_change
} else {
    StepChange::Minor // includes the exact-minimum case
}
// soroban/PROOF.md describes steps >= the minimum as counting
```

Impact

A staged ramp whose per-step change is exactly the configured minimum is not counted toward the `MonotonicRun` streak, contrary to the documented `>=` semantics. Observability/spec-conformance nuance; the broader sub-threshold ramp behavior is addressed separately.

Proof of Concept

A kernel PoC feeds exact-minimum steps and shows `MonotonicRun` does not build a streak (green demonstrates the exclusion at the boundary).

 Copy Code

```
test proxy::circuit_breaker::tests::reaudit_info014_monotonic_exact_minimum_steps_do_not_count ... ok
test result: ok. 27 passed; 0 failed; 0 ignored; 0 measured; 187 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-kernel --lib reaudit_info014
```

PoC Script

Abridged: build a history of consecutive same-direction steps each exactly equal to `min_relative_step_change`; assert `MonotonicRun` does not trip because each exact-minimum step classifies as `Minor` and never extends the streak.

 Copy Code

```
// steps of exactly min_relative_step_change, same direction, longer than max_streak
assert!(!monotonic.should_trip(&history_of_exact_minimum_steps)); // excluded at boundary
```

BVSS

AO:S/AC:H/AX:H/R:L/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Use inclusive comparison for the minimum-step threshold:

- Introduce a `relative_abs_change_at_least` (change $\geq$ threshold) for `min_relative_step_change`, keeping the strict $>$ comparison for maximum thresholds, so exact-minimum steps count as the documentation states.
- Or amend `PROOF.md` to describe strictly-greater-than semantics.

Spec-conformance only; note the broader gradual-ramp evasion is tracked as a separate finding.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by introducing an inclusive **`relative_abs_change_at_least`** comparison for **`min_relative_step_change`**, so a step exactly equal to the configured minimum now counts toward the streak as documented.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/bcf357fbdb513fdd06569b4d301e9610ffc52655>

11.39 SUCCESSFUL SEP-40 ADAPTER UPGRADES EMIT NO ADAPTER UPGRADE EVENT

// INFORMATIONAL

Description

A successful adapter upgrade calls `env.deployer().update_current_contract_wasm(new_wasm_hash)` and returns without emitting an adapter-specific event. The runtime's equivalent upgrade publishes the new hash via a `ContractUpgraded` event, but adapter event consumers receive no `ContractUpgraded` / `AdapterUpgraded` signal.

The PoC uploads a local test WASM, performs an owner-authorized adapter upgrade successfully, and confirms the adapter's event stream remains empty.

Bound: ledger-level code changes are still observable externally; there is no authorization or price-integrity bypass. Observability of a privileged action only.

Code Location

 Copy Code

```
// soroban/sep40-adapter-contract/src/lib.rs - owner-authenticated code swap, no event
fn upgrade(env: Env, new_wasm_hash: BytesN<32>) {
    // owner auth ...
    env.deployer().update_current_contract_wasm(new_wasm_hash); // returns; no event published
}
// contrast soroban/contract/src/lib.rs upgrade, which emits ContractUpgraded
```

Impact

Monitoring and incident-response systems that rely on contract events cannot detect or attribute an adapter code change from the adapter event stream. No authorization or price-integrity impact.

Proof of Concept

An adapter PoC performs an owner-authorized upgrade to a local test WASM and confirms the adapter event stream stays empty (green demonstrates the missing event).

 Copy Code

```
test tests::reaudit_info015_successful_adapter_upgrade_emits_no_upgrade_event ... ok
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 17 filtered out
```

Environment / Reproduce

 Copy Code

```
cargo test -p templar-proxy-oracle-soroban-sep40-adapter-contract reaudit_info015
```

PoC Script

Abridged: upload a local test WASM, call the owner-authorized adapter upgrade, assert it succeeds, and assert the adapter's emitted-events list is empty afterwards.

```
let hash = env.deployer().upload_contract_wasm(TEST_WASM);
adapter.upgrade(&hash); // owner-authorized, succeeds
assert!(adapter_events(&env, &adapter_id).is_empty()); // no AdapterUpgraded event
```

BVSS

AO:S/AC:H/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Emit an event on adapter code change:

- Publish an AdapterUpgraded { new_wasm_hash } event after the successful update, matching the runtime's ContractUpgraded event, and monitor it.

Observability only; no on-chain behavior change is required.

Remediation Comment

SOLVED: The **Templar Protocol** team has solved this issue by publishing an **AdapterUpgraded { new_wasm_hash }** event after a successful adapter code upgrade, matching the runtime's **ContractUpgraded** event.

Remediation Hash

<https://github.com/Templar-Protocol/contracts/commit/4d9e432225085c8d365d083e23f3ea75667f6904>

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.