



Security Review For **Re.xyz**



Collaborative Audit Prepared For:
Lead Security Expert(s):

Re.xyz
KupiaSec
vinica_boy
July 15 - July 18, 2026

Date Audited:

Introduction

Re Protocol is a decentralized platform that channels stablecoin capital into fully collateralized reinsurance contracts through licensed reinsurers. Users deposit stable tokens, which are used as collateral to back reinsurance agreements. In return, depositors earn yield derived from a blend of on-chain and off-chain sources, plus a token-specific spread.

Scope

Repository: `resilience-foundation/smart-contracts`

Audited Commit: `1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d`

Final Commit: `007c484a37d4672a0bd06ddb1792ce529f45c74d`

Files:

- `src/nav/core/NAVOracle.sol`
- `src/nav/feed/NAVFeedProxy.sol`
- `src/nav/feed/NAVFeed.sol`
- `src/nav/providers/CREReportProvider.sol`
- `src/nav/sinks/LegacyConsumerSinkAdapter.sol`
- `src/nav/sinks/NAVPushSinks.sol`

Final Commit Hash

`007c484a37d4672a0bd06ddb1792ce529f45c74d`

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	1	5

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Velocity budget understates net NAV growth when reports are split into small steps [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/10>

Summary

`NAVOracle._spendVelocity` charges each report as bps vs the **then-current** NAV ($\Delta / \text{current}$). That means the velocity budget can be “bypassed” in net-NAV terms by splitting one large uptick into several smaller `submitReport` calls: the same (or greater) final NAV costs **less** bucket than a single jump, and leftover budget can push even higher.

Vulnerability Detail

The automated path’s primary frequency-independent rail is the velocity token bucket. On each accepted report, spend is:

```
uint256 moveWad = Math.mulDiv(navValue - current, MAX_BPS * BPS_WAD, current);
```

So charge depends on the NAV **at that step**, not on net growth from a fixed anchor. Docs/NatSpec describe capacity as $\sim \text{maxCumulativeBps}$ per window (up to $\sim 2\times$ with refill). That bounds $\sum (\Delta_i / \text{current})$, not the multiplicative outcome $(\prod (1 + \Delta_i / N_{i-1}))$.

Nothing else limits how often `submitReport` may be called (only band, staleness, and monotonic `observationTimestamp`). An authorized submitter can therefore drain the bucket more efficiently with many small steps than with one large step.

Example (`maxCumulativeBps` = 5000 $\rightarrow$ 50% budget, start NAV = 100; each step must also fit `normalBandBps`):

```
One report: 100  $\rightarrow$  150  $\rightarrow$  charges 50%, budget empty
Five steps: 100 $\rightarrow$ 110 $\rightarrow$ 120 $\rightarrow$ 130 $\rightarrow$ 140 $\rightarrow$ 150
              charges 1000+909+833+769+714 4225 bps (~42%)
              ~8% budget left  $\rightarrow$  can climb past 150
```

With a full bucket (B) (as a fraction), many tiny steps approach net multiple $(\approx e^B)$ vs one-shot $(1+B)$ (e.g. $(B = 0.5) \rightarrow \sim 1.65\times$ vs $1.5\times$).

Impact

An authorized (or compromised) `NAV_SUBMITTER` can inflate on-chain NAV, feed reads, and sink prices **further** than a naive reading of “`maxCumulativeBps` = max net move” implies, within the same window. Downstream mint/redeem/pricing that trusts the feed/sinks sees the higher value. Fast correction still requires Protection Mode + timelocked markdown ($\sim 24\text{h}$).

Economic size scales with configured `maxCumulativeBps` / `normalBandBps` (larger when the band allows meaningful step sizes; production docs use smaller bands than the 50% example).

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L400-L415>

Tool Used

Manual Review

Recommendation

Either:

1. Charge movement against a **fixed window anchor** (NAV at bucket refill / window start) so spend tracks additive net growth; or
2. Enforce an explicit **net multiple cap** per window; or
3. If path-dependent spend is intentional, document clearly that the rail bounds $\frac{\text{NAV}}{\text{NAV}_{\text{current}}}$, not net NAV %, and that many small reports can leave budget while exceeding a one-shot move of size `maxCumulativeBps`.

Issue L-1: A sink guarding its own price movement can desync from the oracle with no on-chain way back [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/7>

Summary

The oracle fans NAV out to sinks on a best-effort basis and offers `syncSink` as the recovery tool when a sink misses an update. But if a sink rejects a push because the *value moved too far* (like the share-price calculator's 15% divergence rail), `syncSink` cannot fix it - it re-presents the exact same value and gets rejected the exact same way.

Vulnerability Detail

`_pushSinks` is intentionally best-effort: a sink that reverts just emits `SinkPushFailed` and the fan-out keeps going. The recovery for a missed sink is `syncSink`, which pushes the current NAV again and reverts if it fails. That can help if the failure reason is temporary but it breaks for a sink whose failure reason is the value itself.

The share-price calculator sink has a 15% per-update divergence guard. The NAVOracle, on the other hand, permits much larger per-update moves: the deviations supported through `submitReport()` and `forceNAVUpdate()` are bounded up to 50% and 100% respectively (`MAX_NORMAL_BAND_BPS` and `MAX_BPS`), while in production these are expected to be set lower, the ceiling is well above 15%. So if we have a deviation between NAV updates of more than 15%, the NAV value saved in the NAVOracle will diverge from the one in the SharePriceCalculator, and `syncSink` would not help us here.

Two normal situations push the calculator past its 15% rail in a single delivered step:

- A markdown, which can drop NAV in one write down toward `MIN_MARKDOWN_NAV`.
- Accumulated small moves during a sink outage while the calculator is paused / its role is being rotated, the oracle keeps accepting reports. When the calculator comes back, the gap between its last-stored price and the oracle's current NAV can already exceed 15%.

Once the gap is over 15%, the calculator rejects the push. The oracle continues, so the calculator is now frozen. Because oracle NAV is monotonic non-decreasing on the report path, the gap only widens – so every future push is also >15%, including `syncSink`, which presents the same current value and hits the same guard:

```
function syncSink(address target) external restricted nonReentrant {
    if (!_hasSink(target)) revert SinkNotFound(target);
    uint256 nav = _currentNav(); // same value the guard already
    ↪ rejected
```

```
    if (!_pushOne(target, nav)) revert SinkSyncFailed(target, nav);  
}
```

There is no in-contract path that delivers an intermediate value or resets the calculator, so the contract's own recovery tool cannot resolve a divergence-guard rejection.

Impact

- A live sink (the share-price calculator) can serve a stale price indefinitely while the oracle has moved on, and `syncSink` – the documented recovery – cannot restore it.
- Recovery depends entirely on off-chain monitoring of `SinkPushFailed` plus manual governance steps on the external calculator. A missed event = a silently stale downstream price.

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/d64aee11bb2b0e6405eb1b914594e1ff7faca0bc/smart-contracts/src/nav/core/NAVOracle.sol#L510-L514>

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/d64aee11bb2b0e6405eb1b914594e1ff7faca0bc/smart-contracts/src/nav/sinks/NAVPushSinks.sol#L72-L83>

Tool Used

Manual Review

Recommendation

This can be acknowledged if the risk is accepted and markdown value is explicitly chosen to pass sinks guards

Issue L-2: Re-pushing a stale oracle value to the legacy consumer marks it fresh [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/8>

Summary

The oracle deliberately keeps its `updatedAt` frozen when no new reports arrive, so integrators can spot a stale NAV. But when governance re-pushes the current NAV to the legacy consumer via `syncSink`, the legacy consumer stamps `latestTimestamp = block.timestamp`. A value that is genuinely old in the oracle shows up in the legacy consumer as "just updated," defeating any staleness check on that side.

Vulnerability Detail

The oracle's freshness model relies on `updatedAt` staying put between updates so downstream staleness checks can react. `submitReport` enforces staleness on `accept`, but between reports the stored NAV simply persists and `_currentNav()` can be arbitrarily old.

`syncSink` pushes that possibly-old `_currentNav()` to a single sink. For the legacy path, the sink is `LegacyConsumerSinkAdapter`, which forwards it as a force update:

```
// LegacyConsumerSinkAdapter.setSharePrice
legacyConsumer.forceNAVUpdate(nav, UPDATE_REASON);
```

`syncSink` exists precisely for the "sink was disconnected and needs to catch up" case which is the exact situation where the oracle value being replayed may be stale. The `INAVReceiver.setSharePrice(uint256)` interface carries only the value, no observation/data timestamp, so the adapter has nothing to forward even if it wanted to.

Inside the legacy `NAVConsumer`, `forceNAVUpdate` unconditionally stamps the current time:

```
latestNAV = _navValue;
latestTimestamp = block.timestamp;
```

A legacy reader calling `getCurrentNAV()` then receives `(oldNAV, now)` and treats it as fresh which is the opposite of what the oracle's frozen `updatedAt` was trying to communicate.

Impact

The oracle's staleness signal is lost across the legacy sink boundary. Legacy readers that gate on `latestTimestamp` get a false "fresh" reading for a stale NAV.

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/d64aee11bb2b0e6405eb1b914594e1ff7faca0bc/smart-contracts/src/nav/core/NAVOracle.sol#L510-L514>

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/d64aee11bb2b0e6405eb1b914594e1ff7faca0bc/smart-contracts/src/nav/sinks/LegacyConsumerSinkAdapter.sol#L44-L47>

Tool Used

Manual Review

Recommendation

Can be acknowledged if the risk is accepted. You can also document that the legacy consumer's `latestTimestamp` reflects delivery time to the sink, not the actual delivery time.

Issue L-3: Chainlink-canonical NewRound / AnswerUpdated events are emitted from the core with a raw 18-decimal NAV, not a feed answer [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/9>

Summary

NAVOracle emits the Chainlink-canonical NewRound and AnswerUpdated events. Two things about them can be misleading: they're emitted from the oracle core rather than the integrator-facing feed, and the answer they carry is the raw 18-decimal NAV value, not the "answer" a consumer reads back through the feed.

```
NavFeed.sol
    /// @notice Feed decimals. `AnswerUpdated` and `NewRound` are emitted by the
    → oracle core, not
    ///         this view-only adapter.
```

```
NavFeedProxy.sol
    //The proxy does not synthesize AnswerUpdated/NewRound events
```

Vulnerability Detail

AnswerUpdated and NewRound are part of the vendored Chainlink V2 AggregatorInterface, and standard tooling expects them from the aggregator address it reads prices from. In this stack the integrator-facing address is NAVFeed / NAVFeedProxy, but those are view-only and cannot emit - so the events are emitted from NAVOracle._record instead. This part is intentional and already noted in the code

The less obvious part is the value. AnswerUpdated's first field is named current (the "answer"), but the oracle passes the raw 18-decimal NAV:

```
emit NewRound(roundId, msg.sender, block.timestamp);
emit AnswerUpdated(int256(navValue), roundId, block.timestamp);
```

Meanwhile a consumer reading through the feed gets that same NAV down-scaled to the feed's decimals (conventionally 8) via integer division. So the answer carried by the event does not equal latestAnswer() / getRoundData().answer in scale - the event value is 18-dec NAV, the read value is 8-dec. The field is semantically the NAV, not a feed-decimals aggregator answer.

Impact

Off-chain only. An indexer or monitor that treats `AnswerUpdated.current` as the feed's answer will be off by the decimals difference

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/d64aee11bb2b0e6405eb1b914594e1ff7faca0bc/smart-contracts/src/nav/core/NAVOracle.sol#L453-L456>

Tool Used

Manual Review

Recommendation

Document at the emit site or at the event declaration that `AnswerUpdated.current` is the raw 18-decimal core NAV emitted from the oracle address, and does not match the feed's scaled `latestAnswer()`.

Issue L-4: currentDataTimestamp can move backward after forceNAVUpdate then submitReport [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/11>

Summary

forceNAVUpdate sets `_currentDataTimestamp = block.timestamp` but does **not** advance `lastObservationTs`. A later `submitReport` only needs `observationTimestamp > lastObservationTs`, so an observation older than the force's block time can still be accepted. `_record` then overwrites `_currentDataTimestamp` with that smaller value – the integrator-facing “data as of” timestamp regresses while a **new** round is still recorded.

Vulnerability Detail

`_record` always assigns the caller-supplied data time:

```
_rounds[roundId] = Round(uint192(navValue), uint64(block.timestamp));
_currentDataTimestamp = uint64(dataTimestamp);
```

Force commits with `dataTimestamp = block.timestamp` and intentionally leaves `lastObservationTs` unchanged so a CRE report observed just before the force can still land. Report acceptance only checks:

- `observationTimestamp > lastObservationTs`
- not future, not stale, band, velocity, etc.

It does **not** require `observationTimestamp >= currentDataTimestamp()`.

Scenario:

```
1. forceNAVUpdate(...)
   → _currentDataTimestamp = block.timestamp (T_force)
   → lastObservationTs unchanged (still T_old < T_force)

2. submitReport(nav, obsTs) with T_old < obsTs < T_force
   → passes monotonicity vs lastObservationTs
   → _currentDataTimestamp = obsTs (< previous value)
   → new round recorded (Round.updatedAt = now)
```

`getCurrentNAV() / currentDataTimestamp() / NAVFeed.latestObservationTimestamp()`
then report an older “data as of” than after the force, even though NAV and round id moved forward. Docs acknowledge this force→report case; round acceptance time (`Round.updatedAt / feed.updatedAt`) stays monotonic.

Impact

Fail-closed for freshness that uses a “not too old” check against wall clock (data looks older). Harmful if any integrator or monitor assumes `currentDataTimestamp` is **non-decreasing**, or alerts on timestamp flaps after recovery forces. Does not by itself open stale-price acceptance in the usual Chainlink `updatedAt` heartbeat path (that uses acceptance time on the feed).

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L421-L431>

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L435-L451>

Tool Used

Manual Review

Recommendation

Clamp on report (and any path that should not regress data time):

```
if (dataTimestamp > _currentDataTimestamp) {  
    _currentDataTimestamp = uint64(dataTimestamp);  
}
```

Alternatively reject reports with `observationTimestamp < currentDataTimestamp()` after a force – at the cost of dropping in-flight pre-force observations. Document for integrators that `currentDataTimestamp` is not a monotonic watermark; use feed `updatedAt` for liveness.

Issue L-5: Force path can raise NAV far faster than the report velocity rail; governance cannot bind force to velocity [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-07-re-july-13th-2026/issues/12>

Summary

With production params, `forceNAVUpdate` can raise NAV ~150 bps every 4h with **no velocity spend** (~9%/day, on the order of a week to `maxNAV`), while reports are capped near ~200 bps / 14d. The contract also **requires** `forceBandBps >= normalBandBps`, so force's per-write band cannot be tighter than `submitReport`'s. Markdown still needs protection + 24h. Governance can lower `forceBandBps` only down to `normalBandBps` – that shrinks each step but cannot put force under the velocity rail or change the fixed 4h cadence, so there is no on-chain way to contain frequent `forceNAVUpdate` if that key is misused.

Vulnerability Detail

`forceNAVUpdate` is velocity-exempt by design: it does not call `_spendVelocity`, and does not refill or reset the report bucket. Limits that do apply:

- up-or-flat within `forceBandBps`
- `MIN_FORCE_UPDATE_INTERVAL = 4 hours` (immutable constant)
- `maxNAV`
- blocked in Protection Mode

Ctor / `setForceBandBps` enforce:

```
if (bps < normalBandBps || bps > MAX_BPS) revert InvalidBand();
```

So force's per-step band is always **at least** as wide as the report band (up to 100%). Admin can soften force only by lowering `forceBandBps` to equal `normalBandBps` – not below, and not by attaching velocity or lengthening the 4h interval.

Production docs (reUSD-style): `forceBandBps = 150`, `normalBandBps = 50`, `maxCumulativeBps = 200 / 14d`:

Path	Approx. allowed climb
Report velocity	~200 bps / 14d (burst ~2× bucket)
Force (max steps)	$150 \text{ bps} \times 6/\text{day} = \textbf{-900 bps/day}$ → ~week to hit <code>maxNAV</code> from bootstrap

Production wiring keeps force on **ADMIN** (governance). The residual is compromised or misused governance break-glass: upward inflation outruns the “primary” frequency-independent rail, while downward correction remains slow.

Impact

A misused force/governance key can inflate NAV (and thus feed/sink prices) much faster than the automated velocity policy implies. Users/integrators that price off the oracle stack can over-mint or misprice until a Protection Mode + timelocked markdown (⌘24h) lands. Not an unprivileged bug; severity is trust-model / centralization residual unless sponsors intended force to obey velocity.

Code Snippet

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L281-L295>

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L177>

<https://github.com/sherlock-audit/2026-07-re-july-13th-2026/blob/1a60a7bda1f78256b3952ec6ba2d6d56ec0abf8d/smart-contracts/src/nav/core/NAVOracle.sol#L478-L481>

Tool Used

Manual Review

Recommendation

Confirm whether this asymmetry is accepted residual risk for a trusted force key. If not:

1. Make force **spend** the velocity bucket (or a separate force budget); and/or
2. Allow `forceBandBps < normalBandBps` (or cap force band at `maxCumulativeBps`); and/or
3. Make `MIN_FORCE_UPDATE_INTERVAL` governance-configurable (longer than 4h).

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.