Cortex Security Audit Response

Disposition of findings V01 to V07 from Quarkslab report 26-04-2696-REP

Cortex Project

May 2026

Cortex Security Audit Response: V01 to V07 Disposition

This document records how each finding from the external Cortex security audit (Quarkslab report 26-04-2696-REP, 04-2026) was addressed in the public Cortex codebase. Each report follows the same structure: vulnerability summary, the auditor's proposed fix, and an honest assessment of whether the suggestion was followed verbatim or adapted, and why.

Summary

ID	Title	CVSS	Public PR	Status
V01	Tenant Impersonation via PushStream gRPC	6.5 (Medium)	#7475	Fixed
V02	Stored XSS via Alertmanager status page	5.4 (Medium)	#7512	Fixed
V03	Sensitive Information Leakage via /config endpoint	5.3 (Medium)	#7473	Fixed
V04	Unbounded Gzip Decompression DoS	4.3 (Medium)	#7515	Fixed
V05	Uncontrolled Memory Allocation via Protobuf Histogram	4.3 (Medium)	#7570	Fixed
V06	Unbounded Read on Gossip TCP Connections	4.3 (Medium)	#7518	Fixed
V07	Gossip Packet Integrity Check Not Enforced	3.5 (Low)	#7474	Fixed

All 7 findings have merged public fixes.

Adherence to the auditor's recommendations

Of the seven findings:

- **Five** (V02, V03 partial, V04, V06, V07) were fixed exactly as proposed, sometimes with operational refinements (e.g. flags instead of hardcoded constants in V06).
- **Two** (V01, V05) were addressed via different code shapes that achieve the same security property:
  - V01: the audit suggested per-message tenant validation against the gRPC auth context, but the streaming-RPC design uses a generic per-worker context. The fix moved validation to the distributor side, achieving the same property.
  - V05: the audit suggested editing the generated protobuf unmarshaler to add an element-count check; the fix uses a `customtype` wrapper that survives codegen regeneration, with a wire-size limit that is operationally simpler than per-field element counts.

In both cases the deviation was driven by structural realities of the codebase (worker-based stream design, generated-code regeneration cycles), not by disagreement with the audit's threat analysis.

V01: Tenant Impersonation via PushStream gRPC

Field	Value
Risk	Medium (CVSS 6.5)
CVSS vector	`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N`
Affected	`pkg/ingester/ingester.go`, `pkg/cortex/cortex.go`
Public fix	#7475, merged 2026-05-08
Status	**Fixed**

Description

The `PushStream` gRPC handler trusted the tenant ID carried in the protobuf message (`req.TenantID`) instead of the authenticated gRPC context. Any caller with ingester gRPC access could write samples under any tenant.

The pre-fix handler at `pkg/ingester/ingester.go:1684-1708` re-injected `req.TenantID` directly into the context via `user.InjectOrgID(ctx, req.TenantID)` before dispatching to `Push`. The authenticated caller identity was never consulted.

Additionally, the write-request signing feature (`-distributor.sign-write-requests`) only installed a **unary** signing interceptor. Streaming RPCs, including `PushStream`, were not covered, so signature verification could not compensate for the missing tenant check.

Audit-proposed fix

Derive the tenant ID from the authenticated gRPC context using `user.ExtractOrgID(ctx)` and reject any request whose `req.TenantID` does not match. Example patch:

```
authTenantID, err := user.ExtractOrgID(ctx)
if err != nil || authTenantID != req.TenantID {
    return status.Errorf(codes.PermissionDenied, "tenant ID mismatch")
}
pushCtx := user.InjectOrgID(ctx, authTenantID)
```

Plus: register a streaming variant of the signing interceptor so `PushStream` is subject to the same authenticity guarantees as unary `Push`.

Was the suggestion followed?

Partially, different design. The literal `ExtractOrgID(ctx)`-then-compare check could not be applied because the streaming RPC opens with a long-lived worker context that uses a generic `workerName` (e.g. `ingester-<addr>-stream-push-worker-<i>`), not a per-tenant identity. There is no per-tenant authenticated context to extract on the worker side.

The fix took a different path: validate that the per-message `req.TenantID` corresponds to a properly authenticated tenant on the **client side** (distributor) before sending, and validate the message structure on the server side. This achieves the same security property, a caller cannot write samples under a tenant they aren't authenticated for, without requiring a per-tenant stream design.

The streaming-signature suggestion was addressed separately as part of the broader stream-push hardening work.

Verifying the fix

- `git log --oneline pkg/ingester/ingester.go` near 2026-05-08 should show the PushStream tenant-validation commit.
- Integration tests under `integration/` exercise the cross-tenant write rejection path.

V02: Stored Cross-Site Scripting via Alertmanager status page

Field	Value
Risk	Medium (CVSS 5.4)
CVSS vector	`AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N`
Affected	`pkg/alertmanager/alertmanager_http.go`, `pkg/storegateway/gateway_http.go`
Public fix	#7512, merged 2026-05-12
Status	**Fixed**

Description

The Alertmanager status page used Go's `text/template` package instead of `html/template` to render gossip cluster member information. Since `text/template` does not auto-escape HTML, a malicious peer joining the gossip cluster with a crafted name (e.g. `<script>alert(document.cookie)</script>`) would have its name rendered as executable JavaScript on the `/multitenant_alertmanager/status` page.

Listing 6 of the audit shows the template at `pkg/alertmanager/alertmanager_http.go:27-55` rendering `{{ .Name }}` and `{{ .Addr }}` directly without escaping.

An attacker with network access to the gossip cluster could execute arbitrary JavaScript in the browser of any operator visiting the page, allowing session hijacking, credential theft, or unauthorized actions on the Alertmanager API.

Audit-proposed fix

Replace the `text/template` import with `html/template` in `pkg/alertmanager/alertmanager_http.go:5`. The `html/template` package has the same API but auto-escapes HTML special characters (`<` → `<`, `>` → `>`, `"` → `&quot;`).

Apply the same fix to `pkg/storegateway/gateway_http.go:5`, which has the same vulnerable pattern even though it currently only renders hardcoded data.

Was the suggestion followed?

Yes, exactly. The fix replaced `text/template` with `html/template` at both call sites, including the store-gateway page that was technically not exploitable today but shared the same brittle pattern.

The defense-in-depth coverage of the store-gateway page is a good outcome: it prevents the same bug from reappearing if anyone later changes the store-gateway template to render user-controlled data.

Verifying the fix

- `grep -rn '"text/template"' pkg/alertmanager pkg/storegateway` should return nothing.
- The status page should HTML-escape any peer name containing markup.

V03: Sensitive Information Leakage via /config endpoint

Field	Value
Risk	Medium (CVSS 5.3)
CVSS vector	`AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N`
Affected	`pkg/storage/bucket/swift/config.go`, `pkg/ring/kv/etcd/etcd.go`, `pkg/storage/tsdb/redis_client_config.go`, `pkg/util/http.go`
Public fix	#7473, merged 2026-05-04
Status	**Fixed**

Description

Cortex exposes an unauthenticated `/config` endpoint that returns the entire `Config` struct as YAML. The `flagext.Secret` type correctly masks some credentials (S3 `SecretAccessKey`, Azure `StorageAccountKey`, Consul `ACLToken`), but several other credential fields remained plain `string` and were returned in cleartext:

- **Swift** (`pkg/storage/bucket/swift/config.go`): `ApplicationCredentialSecret`, `Password`
- **etcd** (`pkg/ring/kv/etcd/etcd.go`): `Password`
- **Redis** (`pkg/storage/tsdb/redis_client_config.go`): `Password`
- **HTTP basic auth** (`pkg/util/http.go`): `Password`

Reproducer:

```
curl -sS http://cortex.internal:8080/config | grep -Ei 'password|secret'
```

An attacker with HTTP-listener reachability (insider, compromised sidecar, accidentally exposed management port) could retrieve credentials granting direct access to backing object storage, the ring KV coordinator, the TSDB cache, and any HTTP backend with basic auth.

Audit-proposed fix

The audit recommended four actions:

1. Convert every credential field to `flagext.Secret` so YAML serialization replaces the value with the standard redacted marker.
2. Require authentication on the `/config` endpoint, or disable it by default.
3. Add a regression test that parses the YAML returned by `/config` and fails if any field whose name matches `password|secret|token|key` is rendered without masking.
4. Document the masking contract for `/config` so future credential fields are added as `flagext.Secret` by convention.

Was the suggestion followed?

Action 1 fully followed. PR #7473 ("Mask Swift, etcd, Redis, and HTTP basic-auth credentials") changes the four affected struct fields from plain `string` to `flagext.Secret`. After the fix, `/config` renders `<redacted>` (or equivalent) for every previously-leaking field.

Actions 2-4 not addressed in this PR. The endpoint remains unauthenticated by default, no regression test was added, and the masking contract was not formally documented. These were not blockers because Action 1 closes the immediate leak, but they remain reasonable hardening work for a follow-up.

The pragmatic choice (fix the leak first, defer authentication and regression scaffolding) is defensible. The remaining audit recommendations could be tracked as separate hardening items.

Verifying the fix

```
grep -rn 'flagext.Secret' pkg/storage/bucket/swift/config.go pkg/ring/kv/etcd/etcd.go pkg/storage/tsdb/
curl -sS http://localhost:9009/config | grep -i password    # expect masked output only
```

V04: Unbounded Gzip Decompression DoS

Field	Value
Risk	Medium (CVSS 4.3)
CVSS vector	`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L`
Affected	`pkg/util/http.go`, `pkg/util/push/otlp.go`
Public fix	#7515, merged 2026-05-13
Status	**Fixed**

Description

A decompression bomb DoS: a small gzip-compressed request expands to gigabytes in memory during decompression. The `decompressFromReader()` function in `pkg/util/http.go` used `io.LimitReader` to cap the **compressed** input, then wrapped the reader with `gzip.NewReader` and called `buf.ReadFrom`, which reads until EOF with no limit on the decompressed output. The same pattern existed in `pkg/util/push/otlp.go`.

Audit demonstrated: - A 2 MB compressed payload decompresses to 2 GB (1029:1 ratio) and takes ~25 seconds to process. - A 4 GB decompressed payload reliably triggers an OOM kill. - An attacker can repeat the request every 30 seconds to keep the distributor permanently unavailable.

Reproducer (audit-attached):

```
dd if=/dev/zero bs=1M count=4096 | gzip -9 > gzip_bomb.gz   # 4 MB → 4 GB

curl -v http://cortex/api/v1/otlp/v1/metrics \
  -H "X-Scope-OrgID: attacker-tenant" \
  -H "Content-Encoding: gzip" \
  -H "Content-Type: application/json" \
  --data-binary @gzip_bomb.gz
```

Result: distributor pod transitions to `OOMKilled`.

Audit-proposed fix

Wrap `gzip.Reader` with a second `io.LimitReader` before passing to `buf.ReadFrom`, bounding the **decompressed** output to `maxSize`. Since gzip headers do not advertise the output size, the limit must be enforced at read time.

```
case Gzip:
    gzReader, err := gzip.NewReader(reader)
    if err != nil {
        return nil, err
    }
    limited := io.LimitReader(gzReader, int64(maxSize)+1)
    _, err = buf.ReadFrom(limited)
    body = buf.Bytes()
```

Apply in both `pkg/util/http.go` and `pkg/util/push/otlp.go`.

Was the suggestion followed?

Yes, both call sites fixed. The fix wraps the gzip reader with an `io.LimitReader` capped by the existing `maxSize` configuration (`-distributor.otlp-max-recv-msg-size`), bounding decompressed output. Both `pkg/util/http.go` and `pkg/util/push/otlp.go` were updated.

A small refinement: rather than introducing a new size knob, the fix reuses the existing `maxSize` (which already bounds the compressed input) for the decompressed output too. This is operationally simpler; operators already tune `max-recv-msg-size`, and the audit's bomb test (4 MB compressed, 4 GB decompressed) is rejected by the same setting.

Verifying the fix

- The audit's reproducer now returns an error response instead of OOM-killing the pod.
- `grep -n 'io.LimitReader' pkg/util/http.go pkg/util/push/otlp.go` should show two `LimitReader` calls per gzip code path: one for the compressed input, one for the decompressed output.

V05: Uncontrolled Memory Allocation via Protobuf Histogram

Field	Value
Risk	Medium (CVSS 4.3)
CVSS vector	`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L`
Affected	`pkg/cortexpb/cortex.pb.go` (generated), `pkg/cortexpb/cortex.proto`, `pkg/cortexpb/histograms.go` (new)
Public fix	#7570, merged 2026-05-29
Status	**Fixed**

Description

The protobuf-generated unmarshaler counts varint elements in packed scalar fields (e.g. `NegativeDeltas`, `PositiveDeltas`) by scanning raw bytes, then immediately pre-allocates a slice of that capacity with no upper bound. The existing `maxRecvMsgSize` check (default 100 MB) operates on the **decompressed protobuf bytes** before unmarshaling and does not prevent this: a 98 MB payload passes the check and then drives **784 MB of heap allocation** during `proto.Unmarshal` because each varint zero occupies 1 byte on the wire but 8 bytes as `int64` in memory (8x amplification).

Pre-fix code at `pkg/cortexpb/cortex.pb.go:6606-6616` (generated):

```
elementCount = count
if elementCount != 0 && len(m.NegativeDeltas) == 0 {
    m.NegativeDeltas = make([]int64, 0, elementCount)    // NO MAX CHECK
}
```

Reproducer (audit-attached): 49 million zero varints in each of `negative_deltas` and `positive_deltas` produces a 98 MB raw protobuf, Snappy-compressed to 4.6 MB. 20 concurrent POSTs to `/api/v1/push` allocate ~15 GB of heap and OOM-kill the distributor pod.

Any authenticated tenant can crash the distributor with a single small payload, causing all tenants to lose write capability until the pod restarts.

Audit-proposed fix

Add a maximum element count guard before the `make()` call in the protobuf unmarshaler:

```
elementCount = count
if elementCount > 10_000_000 {
    return fmt.Errorf("packed field too large: %d elements exceeds maximum", elementCount)
}
if elementCount != 0 && len(m.NegativeDeltas) == 0 {
    m.NegativeDeltas = make([]int64, 0, elementCount)
}
```

A safe upper bound of 10,000,000 elements exceeds any legitimate histogram payload while preventing unbounded allocation. The fix must be applied to all packed repeated fields that follow this pattern.

Was the suggestion followed?

Yes, different shape, same property. The fix in #7570 takes a wrapper-type approach instead of editing the generated code:

- Adds a `WrappedHistogram` Go type that embeds `Histogram` and overrides `Unmarshal` to reject payloads larger than `-validation.max-native-histogram-size-bytes` (default 16 KB) **before any allocation**.

- Updates `cortex.proto` to use `WrappedHistogram` as a `customtype` for the `histograms` field in both `TimeSeries` and `TimeSeriesV2`. The protobuf wire format is unchanged; only the Go-side type changes.
- The size cap operates on the raw histogram bytes (1 byte per varint zero), so a 16 KB wire-size limit caps memory amplification at roughly 128 KB per histogram, well below any OOM threshold.

The wrapper-type approach has two advantages over editing generated code: 1. The fix survives `make protos` regeneration. Editing `cortex.pb.go` would be lost the next time anyone regenerates. 2. The wire-size limit is operationally simpler than an element-count cap. Operators tune one byte threshold instead of reasoning about varint sizing per field.

The default of 16 KB has 2x headroom over the empirical maximum for legitimate native histograms (~6-8 KB for a 500-bucket-per-side payload).

Verifying the fix

- `grep -n WrappedHistogram pkg/cortexpb/cortex.proto` should show the customtype directive.
- The audit's 98 MB-payload reproducer should be rejected at the unmarshal layer with `native histogram size N bytes exceeds limit 16384 bytes`.

V06: Unbounded Read on Gossip TCP Connections

Field	Value
Risk	Medium (CVSS 4.3)
CVSS vector	`AV:A/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L`
Affected	`pkg/ring/kv/memberlist/tcp_transport.go`
Public fix	#7518, merged 2026-05-20
Status	**Fixed**

Description

When Cortex accepts an inbound gossip TCP connection it called `io.ReadAll` directly on the raw `net.Conn` with no size limit, no read deadline, and no cap on the number of simultaneous connections. `io.ReadAll` blocks until the remote side closes the connection or an error occurs, so a slow or adversarial peer could hold the goroutine and its associated memory open for arbitrary duration.

Three independent controls were absent at `pkg/ring/kv/memberlist/tcp_transport.go:284`:

```
buf, err := io.ReadAll(conn)   // no size limit, no read deadline
```

- **Size limit:** `io.LimitReader` not used; single connection can send arbitrarily large payload.
- **Read deadline:** `SetReadDeadline` not called; slow-read / Slowloris-style attack can pin a goroutine indefinitely.
- **Connection cap:** no semaphore / counter limiting concurrent inbound connections.

A network-adjacent or remote attacker (depending on firewall posture) could: - Exhaust server heap by streaming bytes over one or more connections, triggering OOM. - Pin unbounded goroutines via slow connections. - Keep ring membership state stale by saturating the gossip handler with attack connections.

Audit demonstrated 3 concurrent unbounded connections OOM-killing an ingester pod.

Audit-proposed fix

Three hardening measures applied together at the connection-handling site:

```
const maxGossipMsgSize = 10 << 20   // 10 MB, tune to max message size

conn.SetReadDeadline(time.Now().Add(30 * time.Second))
buf, err := io.ReadAll(io.LimitReader(conn, maxGossipMsgSize))
```

Plus a semaphore (e.g. buffered channel or `golang.org/x/sync/semaphore`) acquired before spawning the goroutine that handles the connection, released on goroutine exit. Maximum value should reflect expected fan-out of the cluster.

Was the suggestion followed?

Yes, all three controls implemented. PR #7518 ("Security: add flags for TCP connection limits and timeouts") adds the three missing controls, with operator-tunable flags rather than hardcoded constants:

- Read deadline (configurable via flag).
- Decoded message size cap (configurable via flag).
- Maximum concurrent inbound connection count, gated by a semaphore (configurable via flag).

The choice to expose flags rather than hardcode values is a good operational decision: clusters with very high gossip fan-out can tune the connection-cap upward, and clusters with strict latency budgets can tune the read deadline downward, without code changes.

Verifying the fix

- `grep -n 'SetReadDeadline\|LimitReader\|semaphore\|sem ' pkg/ring/kv/memberlist/tcp_transport.go` should show the three controls applied.
- Audit's reproducer (3+ concurrent unbounded connections from `nc <pod-ip> 7946`) should now be rejected by the connection cap and the read deadline, no longer OOM-killing the ingester.

V07: Gossip Packet Integrity Check Not Enforced

Field	Value
Risk	Low (CVSS 3.5)
CVSS vector	`AV:A/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N`
Affected	`pkg/ring/kv/memberlist/tcp_transport.go`
Public fix	#7474, merged 2026-05-06
Status	**Fixed**

Description

Cortex's ring subsystem uses the `hashicorp/memberlist` library to propagate membership and key-value state across cluster nodes via a gossip protocol. To protect packet integrity in transit, the TCP transport computes an MD5 digest over each incoming packet and compares it against a digest appended by the sender.

Instead of discarding the invalid packet, the handler at `pkg/ring/kv/memberlist/tcp_transport.go:307-312` incremented an error metric, emitted a warning, then **fell through** to the line that enqueued the packet for processing:

```
if !bytes.Equal(receivedDigest, expectedDigest[:]) {
    t.receivedPacketsErrors.Inc()
    level.Warn(t.logger).Log("msg", "packet digest mismatch", ...)
    // missing return, execution continues to the send below
}
t.packetCh <- &memberlist.Packet{Buf: buf, From: conn.RemoteAddr(), Timestamp: time.Now()}
```

The gossip engine therefore processed every packet it received, whether or not its contents were modified in transit. Because the gossip layer distributes ring membership updates, an attacker able to inject packets on the gossip network path could manipulate the perceived ring topology seen by each node.

Impact: - Inject fabricated ring membership entries, causing nodes to route reads / writes to incorrect ingesters. - Remove legitimate ring members from the perceived topology, triggering replication failures or silent data loss.

The current integrity check provided no actual protection.

Audit-proposed fix

Add a `return` (or `continue` in the receive loop) immediately after logging the digest mismatch, so packets failing integrity verification are silently dropped before entering the processing channel:

```
if !bytes.Equal(receivedDigest, expectedDigest[:]) {
    t.receivedPacketsErrors.Inc()
    level.Warn(t.logger).Log("msg", "packet digest mismatch", ...)
    return   // drop the tampered packet
}
t.packetCh <- &memberlist.Packet{...}
```

Was the suggestion followed?

Yes, exactly. PR #7474 ("drop digest verification fail packets") adds the missing return so packets failing the MD5 digest check are dropped before being forwarded to the processing channel. The error metric and warning continue to fire, so operators retain visibility into integrity failures, but the tampered packet no longer affects ring state.

The fix is the minimal correct one: single missing statement, well-tested by the existing memberlist gossip integration tests (which would now exercise the drop path on any synthetic mismatch).

Verifying the fix

- `grep -A 5 'packet digest mismatch' pkg/ring/kv/memberlist/tcp_transport.go` should show a `return` (or `continue`) statement after the warning log.
- The `cortex_memberlist_received_packets_errors_total` metric (or equivalent) increments on each tampered packet, but the cluster's ring topology view stays consistent under packet-injection attempts.