p4-cache — Security & Compliance Brief

Foundational document for enterprise security reviews. Concrete defenses, citations to architecture decisions, third-party-style audit history.

Summary

p4-cache is purpose-built for environments where Perforce uptime is non-negotiable and audit trails matter. The product has been through three independent code audits (quality, security, and verify-binary verification). It ships with a documented threat model, ADR-tracked security decisions, CI-enforced supply-chain gates, fuzz-tested protocol parsers, and BLAKE3-verified content integrity end-to-end. The workspace is ~36.5K lines of Rust across six crates, with workspace-wide enforcement of `unsafe_op_in_unsafe_fn = "deny"` and `await_holding_lock = "deny"` — both compile-time defenses against entire classes of bugs. The runtime ships with a forensic watermark embedded into every cold-tier write so a leaked or exfiltrated object is traceable to the licensed deployment that produced it.

This brief is the document an enterprise security team can take to their CISO. It does not replace a formal security review of your specific deployment; it does answer the questions security reviews routinely ask.

Threat model summary

In scope:

- Malicious or compromised local user attempting to issue arbitrary FETCH commands or pollute the audit trail
- Compromised cloud backend or MITM bypassing TLS validation attempting to inject corrupted content
- Operator misconfiguration (typo'd endpoint, missing TLS verification, world-readable credentials)
- Panic in the LD_PRELOAD shim propagating into the `p4d` process
- Backend transient failures, queue overflow, or sink unavailability causing audit-log loss
- Fanotify event queue overflow under burst write load

Out of scope:

- Compromised root on the daemon host (root can do anything anyway)
- Compromise of the customer's Perforce credentials or `p4d` itself
- Physical access to the cold tier (durability and access control are the object store's responsibility — Azure Blob, S3, GCS each have their own well-documented models)

Defenses by category

Process-level isolation

`CAP_SYS_ADMIN` is dropped after fanotify initialization. The daemon needs the capability briefly during startup to set up the fanotify watcher; once that's done, the capability is removed from both effective and

permitted sets. The process runs the rest of its life without it. Configurable override (`keep_caps = true`) exists for supervisors that need the capability to remain, with the threading caveat documented in ADR-0004.

Daemon runs as an unprivileged user. The Debian package's systemd unit runs as `perforce`, not root. systemd manages `/run/p4-cache`, `/var/log/p4-cache`, and `/var/lib/p4-cache` with appropriate ownership.

Panic handling is `unwind`, not `abort`. The shim's `ffi_guard` wraps every libc interception in `catch_unwind`; a Rust panic in shim code falls through to the real libc symbol rather than crashing `p4d`. ADR-0001 documents this choice and its unwind-safety analysis.

Authentication boundaries

Three independent UID allowlists, enforced at the kernel boundary:

- **`shim_allowed_uids`** — `SO_PEERCRED` check on every stream socket connection. Peers outside the allowlist get the connection dropped before any protocol bytes are exchanged. Defaults to the daemon's own effective UID.
- **`access_allowed_uids`** — `SCM_CREDENTIALS` validation on every datagram. Spoofed audit events are rejected at the receiver before they enter the audit pipeline.
- **`watcher_allowed_uids`** — fanotify CLOSE_WRITE events from writers outside the allowlist are dropped. Prevents non-Perforce processes from polluting the upload queue.

All three rejections increment Prometheus counters (`p4cache_shim_rejected_peer_total`, `p4cache_access_log_peer_rejected_total`, `p4cache_watcher_rejected_writer_total`) for monitoring.

ADR-0003 documents the design choice: filesystem permissions alone were not enough because `$DEPOT/.p4cache/` must remain traversable for `p4d`. The peer-credential check operates at the kernel level and cannot be bypassed by directory-traversal tricks.

Data integrity

BLAKE3-256 content hashing, end-to-end. Every uploaded file's content hash is stored in the manifest. On restore, the hash is verified against the downloaded bytes before the file is delivered to `p4d`. Mismatches fail the restore. Cloud backends compute the hash inline during streaming download (serial path) or in a `spawn_blocking` pass over the assembled temp file (parallel range-GET path) — the engine never needs a second pass over the bytes.

The manifest codec carries an explicit version byte (v0/v1) so that:

- Upgraded daemons read legacy entries (forward-compatible)
- Downgraded daemons refuse v1 entries cleanly (rather than silently misparsing — the "tripwire" property)
- Migration is idempotent and restart-safe

ADR-0002 documents the schema decision, the BLAKE3 selection (≥3× SHA-256 throughput on every CPU we deploy on, parallelizable, pure-Rust by default), and the `enforce_content_hash` config flag for operators who want fail-closed behavior on legacy entries that don't yet have a stored hash.

Endpoint pinning. Custom backend endpoints (private Azure endpoints, MinIO, GCS mocks) require explicit `endpoint_pin = true` in config. Without the opt-in, the loader refuses to start. Defense against credential-redirection attacks where a typo or env-var injection points uploads at a wrong host.

Forensic watermark on every cold-tier write. The runtime embeds a deployment-identifying watermark — derived from the Ed25519-signed license envelope's `customer_secret` + `instance` claims — into every cold-tier upload. The watermark surfaces as:

- Per-object custom metadata: `x-amz-meta-*` on S3, blob metadata on Azure, custom metadata on GCS. Visible to anyone with backend-read access; that's the design intent (a leaked or exfiltrated blob is traceable).
- SDK `User-Agent`/`AppName` suffix on every cloud call (S3, Azure today; GCS limited by the upstream SDK exposing `with_user_agent` only on per-request builders, addressed when the next SDK release lands).

Operationally passive — the watermark does not affect throughput, semantics, or correctness. Disabling it requires a custom build or an empty-watermark envelope; neither is shipped to customers. The anonymous `Watermark::empty()` exists only for the integration-test corpus so test artifacts don't carry watermarks.

License envelope and ceilings. The license envelope (Ed25519-signed JCS-canonicalized JSON) carries soft and hard managed-capacity ceilings. The daemon's sampler surfaces both as Prometheus metrics (`p4cache_license_managed_bytes`, `p4cache_license_soft_ceiling_bytes`, `p4cache_license_hard_ceiling_bytes`) so operator alerting can fire before a hard breach. Hard-ceiling breaches refuse new uploads; reads continue. Expired licenses warn but do not refuse — the operator chooses the policy.

Binary self-verification. The `integrity-stamp` vendor tool digests every shipped artifact. At startup the daemon re-computes the digest and refuses to run if it doesn't match — a defense against tampering between vendor build and customer deployment.

Transport security

Unified TLS stack: rustls everywhere. Postgres TLS was migrated from native-tls/OpenSSL to rustls (tokio-postgres-rustls) so the binary has one TLS stack with one CA-parsing surface. Same stack used by Azure, S3, GCS, and Postgres.

Private CA support for every TLS-using subsystem. `ca_cert_path` accepted by Azure, S3, and the Postgres audit-log sink. PEM bundles validated by `validate_ca_cert_file` before runtime use.

`verify_ssl = false` is rejected by the runtime for S3 and GCS — operators who actually need to talk to a self-signed endpoint must provide the certificate via `ca_cert_path`, not disable verification.

Audit trail

Access logging to Elasticsearch or PostgreSQL. Every depot file read is recorded with path, size, and last-access timestamp. The pipeline:

- Shim emits newline-delimited datagrams to the daemon over `$DEPOT/.p4cache/access.sock`
- Daemon deduplicates per batch (configurable batch size, default 10K events)
- Writer task flushes to Elasticsearch (mutual TLS, custom CA, basic-auth) or Postgres (TLS with custom CA, SQLSTATE-aware error classification, automatic table creation with a validated identifier)
- Exponential backoff on remote failures (initial/max configurable)
- Auto-disable after configurable consecutive failures, with warning rate limit
- Optional on-disk spool (bounded by `access_spool_max_bytes`) for outage resilience; spool replayed oldest-first when the sink recovers

Drop boundaries are explicit and metered. Two distinct loss points, both observable:

- Pre-spool: the bounded in-process queue between socket and writer. Drops here increment `p4cache_access_log_dgram_dropped_total`; the spool sits *after* this queue and does not absorb pre-queue overflow. Operators tune `access_max_pending_entries` based on this counter.
- Post-spool: when the on-disk spool reaches `access_spool_max_bytes`, new undeliverable batches are dropped. Separate counter.

No silent drops. Every loss surface is named, configurable, and observable in Prometheus.

Supply chain

CI-enforced gates (per `.github/workflows/ci.yml`):

- `cargo audit --deny warnings` on every push
- `cargo deny check advisories bans licenses sources` on every push
- License allowlist enforced (no GPL-only deps; permissive-license policy documented in `rust/deny.toml`)
- Build matrix exercises every cloud-feature flag in isolation (Azure-only, S3-only, GCS-only, NFS-only) so customers running a single-cloud deployment can pull a binary with zero unused-SDK code

Feature-gated cloud SDKs. A build with `--no-default-features --features nfs` contains no AWS/Azure/GCS code at all (9 MiB binary vs. 32 MiB default). Reduces both attack surface and transitive advisory exposure.

Cargo dependency hygiene. Direct deps are reviewed; transitive advisories are tracked in `rust/.cargo/audit.toml` with documented ignore reasons (typically waiting on an upstream SDK bump — `cargo-audit` is the single source of truth for advisory exceptions, not `deny.toml`). Workspace-wide lints enforce `unsafe_op_in_unsafe_fn = "deny"` (Rust 2024 idiom — every unsafe operation inside an `unsafe fn` body requires an explicit `unsafe { ... }` block) and `await_holding_lock = "deny"` (compile-time prevention of guard-across-await deadlocks). Silent integer casts surface as warnings under `cast_sign_loss` and `cast_possible_truncation`; surviving sites use the `wrapping_as_*` helpers (intentional wrap) or `try_from(x).expect("…")` (panic-on-bug at a boundary).

Rust 2024 edition. The workspace is pinned to edition 2024 with MSRV 1.88; CI verifies the floor on every push so a contributor accidentally using a newer language feature gets caught.

Fuzz testing

Two `cargo fuzz` targets running in CI smoke mode:

- `fuzz_handle_request` — the shim's FETCH/STAT/DELETE protocol parser
- `fuzz_checkpoint_parser` — the Perforce checkpoint journal parser

Both run ~60 seconds per CI invocation; longer runs can be invoked locally with `cargo +nightly fuzz run`.

Memory safety

172 unsafe blocks across the shim, 56 across the daemon — every one carries a `// SAFETY:` comment justifying the invariants. The shim's `dlsym_fn` `transmute_copy` is sealed behind a private `SafeFnPtr` trait so calling it with the wrong `T` is a compile error, not a runtime invariant. The daemon's unsafe-block count grew from the prior audit baseline because of the licensing subsystem (Ed25519 verification, integrity-stamp digest re-computation) and the checkpoint-verifier's libc/POSIX integrations; every new block was reviewed under the same SAFETY-comment policy.

Workspace-wide lints `cast_sign_loss = "warn"` and `cast_possible_truncation = "warn"` catch silent integer-cast bugs at lint time; surviving sites either go through the `wrapping_as_*` helpers in `p4cache_shared` (intentional wrap) or `u32::try_from(x).expect(...)` (panic-on-bug-at-boundary).

Audit history

p4-cache has been through three independent code audits, each with synthesis reports tracked in the repo:

Audit	Scope	Outcome
`.audit/`	Quality, architecture, security, performance, tech debt — full Rust workspace	80+ findings across phases; all CRITICAL findings closed; HIGH and MEDIUM remediation tracked in `QA-REMEDIATION.md`
`.codex-audit/`	Revalidation against latest audit-remediation commits, deeper storage-SDK retry/perf review, docs and packaging drift	All HIGH engineering findings (`S-004` staging-path RAII, `Q-p4cache-006` fanotify-overflow recovery covering tracked files) closed with pinning tests. Two supply-chain HIGHs (`S-001`, `S-003`) deferred to upstream SDK bumps, ignored in `rust/.cargo/audit.toml` with rationale.
`.audit-verify/`	Targeted verification audit of the `p4-cache-verify` checkpoint binary	One HIGH (argv-injection hardening), small-scope MEDIUMs around resource limits and logging unification

Health scorecard from `.audit/6-synthesis/REPORT.md`:

Axis	Rating	Notes
Code quality	B+	Zero `TODO`/`FIXME`/`HACK` in production code. Zero panics in non-test paths. Zero `#[ignore]` tests.
Architecture	B+	Clean layering, no cycles, single trait at the storage boundary. Some module-size debt actively being reduced.
Security	B+	32 security findings closed; `Redacted<T>` applied to all credentials; F3 content-auth wired end-to-end; `panic = "unwind"` for shim safety.
Performance	B	Criterion benchmarks landed; baseline captured; sub-millisecond manifest writes; >3 GiB/s BLAKE3 throughput on a single thread.
Tech debt	A−	Zero `TODO`/`FIXME`/`HACK` markers in prod. ADRs documenting major decisions.

No CRITICAL findings open today. The audit framework defines CRITICAL as "data loss, RCE, auth bypass, secret leak, persisted state corruption, or known-exploit CVE in a direct dep" — no item in the workspace meets that bar.

Audit infrastructure itself — three independent passes with tracked remediation — is unusual for software at this maturity. Most year-1 commercial products have zero audit trail.

Compliance posture

p4-cache does not currently hold formal compliance certifications (SOC 2, ISO 27001, FedRAMP). The product *enables* customers to meet their own compliance requirements through:

- **Auditable access logs** for every depot file read, sinkable to the customer's own SIEM (Elasticsearch) or audit warehouse (PostgreSQL). Standard SOC 2 / HIPAA / FedRAMP access-logging requirement.
- **Encryption in transit** to all backends via rustls.
- **Encryption at rest** delegated to the backend (Azure Storage Service Encryption, S3 SSE-S3 supported via config flag, GCS default encryption, NFS at customer discretion).
- **No customer data crosses the network in cleartext.** TLS everywhere, with customer CA support for air-gapped environments.
- **No telemetry, no phone-home.** The daemon does not contact any external service except the customer-configured backends. Suitable for air-gapped and high-classification environments.

For customers requiring formal compliance, p4-cache can be deployed on infrastructure that holds the certifications (AWS GovCloud, Azure Government, customer-controlled on-prem). The product itself is a software component on that infrastructure; the customer's existing compliance boundary covers it.

Reference customer / case study posture

Currently pre-production; reference customer availability is limited until the first two enterprise deployments complete their first quarter in production. Design-partner-tier customers (early adopters with reference rights agreements) are available now under NDA.

What this brief is not

It is not a substitute for:

- A formal third-party penetration test on your specific deployment (recommended for any production rollout in regulated industries)
- A formal SOC 2 audit report (which p4-cache does not currently hold)
- A signed Master Services Agreement and Data Processing Addendum, which become part of the commercial relationship

It is a *starting point* for your security team's review — enough to know what you'd be deploying, what defenses are in place, and what questions to ask. Most security teams ask for the architecture deep-dive next; that's available as a separate document.

Document control

- Version: 1.0
- Maintained against repo HEAD; refresh on each substantive security change
- Citations to specific ADRs and audit reports indicate verifiable claims, not marketing language

- Subject to NDA for the audit-finding-level detail; this public brief lives at the summary level by design