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 streaming-MD5-verified restore integrity against p4d's own `db.storage.digest`. The workspace is ~40K 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 trigger requests 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)
- Backend transient failures, queue overflow, or sink unavailability causing audit-log loss
- Post-build binary tampering (patched-out license check, NOP'd audit writes)
- Wall-clock manipulation across restarts to load a license that's past its grace window (containers, unprivileged user namespaces, virtualised time)
- `P4CACHE_LICENSE` env-var redirect pointing at an attacker-controlled file
- License-audit-log truncation to hide over-capacity transitions
- Signature-algorithm downgrade against the license envelope

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

Per-call trigger binary; no in-process libc hooks in p4d. The v2 integration is a per-call `p4cache-trigger` static binary that p4d spawns via its `+X` archive trigger. The trigger speaks postcard frames over a local Unix socket (or named pipe on Windows) to the daemon and exits when done. A trigger crash exits non-zero and surfaces as a stock p4d archive-op failure; it cannot affect p4d's process state.

Daemon runs as an unprivileged user. Production systemd units run the daemon as `perforce`, not root.

Panic handling is `unwind`, not `abort`. The daemon's per-connection trigger handler is wrapped in `catch_unwind`; a panic during a single request becomes a structured ERROR log rather than killing the listener.

Authentication boundaries

Peer-credential gating on the trigger socket. On Unix, `SO_PEERCRED` is checked against the configured `trigger_allowed_uids` allowlist on every accepted connection. Peers outside the allowlist get the connection dropped before any protocol bytes are exchanged. On Windows the named-pipe peer-credentialing story is documented in `docs/windows-port-status.md`; today the listener fails-closed on any peer-check failure (including the Windows "Unsupported" verdict) rather than admit a connection silently.

Input validation at the trust boundary. Depot paths and revision strings are validated against traversal, NUL, LF/CR, oversize, and `-`-prefix argv tricks before any handler reaches the filesystem, the bucket key, or `p4` argv. The validators are shared between the trigger and the daemon so a compromised trigger or operator-misconfigured invocation cannot push junk through the socket.

Reserved depot subtree. No depot file may live under any `.p4cache` component at any path depth — enforced by the shared `validate_rel_path_bytes` validator and property-tested in `shared/src/lib.rs`.

Data integrity

Streaming MD5 against `db.storage.digest`, end-to-end. Every cold restore is hashed as bytes stream from the cloud backend into the cache temp file. Before the temp is atomically renamed onto the live archive path, the streaming digest is compared against `db.storage.digest` — the same MD5 p4d itself stores in `db.storage`. Mismatches fail the restore; rejected bytes never reach p4d.

Why MD5, not BLAKE3, on the restore path. p4d already maintains an MD5 for every archive. Re-using that digest as the verification anchor means a corrupted cold-tier object fails closed against p4d's own ground truth, without the daemon maintaining a parallel hash database that could drift. BLAKE3 still drives the license-audit MAC chain, the integrity stamp, and the object-store download content-hash (`p4cache_shared::compute_blake3_file`, which cross-checks parallel-range downloads) — none of which is the restore install gate; that stays MD5 against `db.storage.digest`.

Atomic cache writes. Every write goes through temp + fsync + rename + fsync(parent). The on-disk archive layout matches p4d's exactly, verified by `rust/tests/p4_gzip_parity.sh`.

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 RFC 8785 JCS-canonicalized JSON) carries a capacity claim and an expiry. The runtime drives a four-state capacity machine off the ratio of managed bytes to the soft ceiling (`capacity_tib × 1 TiB`):

State	Range	Behavior
Green	< 80%	normal
Warn	80–100%	warn log, serve everything
Grace	100–120%	SEV-1 log, serve everything
Refused	≥ 120%	refuse new admissions and new cold-tier uploads; reads, restores of already-tracked files, and uploads of already-Dirty files continue

All four states surface via Prometheus (`p4cache_license_managed_bytes`, `p4cache_license_soft_ceiling_bytes`, `p4cache_license_hard_ceiling_bytes`) so operator alerting can fire long before Refused. State transitions are atomic — capacity and expiry pack into a single `AtomicU16` swap so the hot-path admission guard never blocks.

Algorithm pinning. `VendorPubKey` carries an explicit `signature_alg` field; the verifier requires both the envelope's declared algorithm and the matched key's algorithm to be `Ed25519`. Key-id comparison is constant-time via `subtle::ConstantTimeEq`. Signature parsing is explicit-length-checked rather than infallible — wrong-size signatures get `LicenseError::MalformedSignature` instead of panicking through `try_into`.

Past-expiry claims rejected at parse time. `Claims::validate` refuses any envelope whose `expires_at` is older than `build_date - 7 days`, so a stale envelope can't quietly load. Capacity values ≥ 1 EiB are also rejected at parse time (overflow defense).

Binary self-verification — hard-fail at startup. The `integrity-stamp` vendor tool digests every shipped artifact and bakes the value into a dedicated block in `.rodata`. At startup the daemon walks its own program headers via `dl_iterate_phdr`, hashes every `PT_LOAD` segment with the `PF_X` (executable) flag set in `p_vaddr` order, and refuses to start on mismatch. Hashing the loaded segments (not `/proc/self/exe`) catches in-memory patches; hashing every executable LOAD (not just `.text`) catches LTO/PGO/`-ffunction-sections` split-text patches that would slip past a single-section check. An opt-out (`P4CACHE_INTEGRITY_SOFT_FAIL=1`) exists for build-pipeline use and is loudly logged + audit-logged when honored.

License-load env hardening. The `P4CACHE_LICENSE` env var still lets operators point the daemon at a non-default license path, but the loader now refuses the override unless the target file is root-owned, mode 0600, and not a symlink. Every honored override emits an audit-grade tracing event at startup with the resolved path.

Cross-restart clock-skew anchor. The first successful license load persists the wall-clock view of "now" to `<state_dir>/license/monotonic-anchor`. Every subsequent startup compares the current clock to the

anchor and refuses to load if the clock has moved backward by more than `CLOCK_SKEW_TOLERANCE_SECS` (1 hour — absorbs NTP corrections, leap-second-style backward steps, and DST mis-handling without false-positive). Closes the "rewind the clock to load an expired license" path that virtualised-time environments otherwise admit.

Tamper-evident license audit log. Every license-relevant event (load, capacity-state transition, expiry transition, integrity-check result) appends one NDJSON line to `<state_dir>/license-audit.log` with a BLAKE3 keyed-MAC chain. The MAC key is the per-deployment `customer_secret`. The format includes monotonic `seq` numbers, an RFC 8785 canonical HMAC input (via `serde_jcs`, so nested-object payloads can't produce ambiguous bytes), and a cross-restart anchor file (`license-audit.anchor.json`) recording `(latest_seq, latest_hmac)`. On startup the loader compares the anchor to the file's max seq and warns on mismatch — closing the "truncate the tail to hide bad events while keeping the chain internally valid" path. Developer-build licenses (all-zero `customer_secret`) switch to plaintext "unchained" lines with a `# DEVELOPER BUILD — UNCHAINED` header so operators don't mistake them for production audit history.

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:

- The trigger listener emits access events directly into an in-process channel as it serves each archive — the sole producer in the pipeline (no datagram socket is bound)
- 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 bounded. Two distinct loss points:

- Pre-spool: the bounded in-process channel between the trigger listener and the writer task. When the writer is behind, the producer `try_send`s and sheds the event rather than awaiting, so the audit pipeline can never apply backpressure to the hot serve path. Operators size this channel with `access_max_pending_entries`.
- Post-spool: when the on-disk spool reaches `access_spool_max_bytes`, new undeliverable batches are dropped, and invalid/unreadable spool entries are surfaced via the `p4cache_access_spool_invalid_dropped_total` Prometheus counter.

The serve path never blocks on audit delivery: pre-spool overflow is a deliberate, bounded shed governed by `access_max_pending_entries`; the spool absorbs sink outages, and its drop surface is named, configurable, and observable in Prometheus.

Supply chain

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

- `cargo audit` 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`)
- Cross-build job (`windows-shared-build`) compiles lib + trigger + daemon on `windows-latest` and publishes the trigger artifact

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). 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 daemon's trigger request handler
- `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

Every unsafe block in the daemon carries a `// SAFETY:` comment justifying the invariants. The unsafe surface covers `dl_iterate_phdr` self-hash over executable LOAD segments, cross-restart clock anchor, license audit-log v2, SCM_RIGHTS fd-passing on the trigger socket, and the checkpoint-verifier's libc/POSIX integrations; every block is 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 (e.g. `S-004` staging-path RAII) 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+	Two tracked `TODO` markers; zero panics in non-test paths; zero `#[ignore]` tests.
Architecture	B+	Clean layering, no cycles, single trait at the storage boundary, on-disk archive tree as source of truth.
Security	B+	All HIGH security findings closed; `Redacted<T>` applied to all credentials; restore-time MD5 against `db.storage.digest`; `panic = "unwind"` for the trigger handler's `catch_unwind`.
Performance	B	Criterion benchmarks landed; baselines captured in `docs/perf/`.
Tech debt	A−	Two tracked `TODO` 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