Zero-NPM-Dependency Security Boundary for Cloud-Native AI

SaaS, platform engineering, and DevSecOps teams

White paper — A FIPS-mode framework boundary with zero transitive deps, plus a host admission gate that audits every plugin's supply chain before it loads

Enclawed LLC **May 5, 2026**

Executive summary

Modern cloud-native AI stacks live or die by their dependency graph. A single `npm install` on a typical AI repo pulls **300–800 transitive packages**, each one a separate maintainer, key-rotation cadence, and supply-chain-compromise opportunity. The 2024 `xz-utils` backdoor and the ongoing typosquatting / dependency-confusion incidents have made the cost obvious; what hasn't been obvious is what to do about it.

enclawed-enclaved addresses the supply-chain attack surface in two layers, with different scopes:

- **The framework boundary itself has zero runtime NPM dependencies.** The closed-tree primitives, the inherited `enclawed-oss` framework, and the admission gate that authorizes every extension load are all implemented using only Node.js's `node:` built-in modules. The supply-chain attack surface of the framework, by construction, is the surface of Node itself plus libcrypto.
- **The bundled extension catalog (channels, cloud providers, tool plugins) does carry its own NPM dependencies.** The full upstream OpenClaw extension catalog is mirrored so the open flavor retains broad ecosystem reach. In the enclaved flavor, every such extension is gated by the admission layer: it loads only if its manifest is signed by a trust-root key, declares its capabilities explicitly, and (if it requires network access) carries an explicit per-extension destination allowlist. Extensions whose supply chain you do not trust are simply not signed; admission denies them at load.

Key point. Zero transitive NPM deps in the framework boundary. The framework's `package.json` declares no `dependencies`. Cloning the framework and running `node -test` works without `npm install`. A CI guard fails the build if a `dependencies` block reappears. Bundled extensions live outside this boundary and are gated by the admission layer at load time rather than vendored into the framework's dependency closure.

1 Why zero NPM dependencies matters

1.1 The dependency-graph attack surface

A cloud-native AI repo's `package-lock.json` typically includes:

- 1 to 5 declared production dependencies;
- 50 to 300 transitive runtime dependencies;
- 200 to 1500 transitive dev dependencies;
- authored by 100–500 individual maintainers, none of whom your security team has vetted.

The risk. Every one of those maintainers can ship code into your production AI stack. A single compromised maintainer account (via SIM-swap, password reuse, or a 2FA bypass) is sufficient to push malicious code into your build via a normal-looking patch release.

1.2 The actual incidents

- **xz-utils** (2024): a multi-year social-engineering operation against an open-source maintainer placed a backdoor in a transitive dependency of OpenSSH.
- **event-stream / flatmap-stream** (2018): a compromised npm maintainer added a wallet-stealer to a package with 2M weekly downloads.
- **ua-parser-js / coa / rc** (2021): npm compromises injecting cryptominers + stealers, each affecting hundreds of thousands of installs in a 24-hour window.
- **Continuous typosquatting / dependency- confusion** (2020–present): malicious packages published under near-miss names of internal corporate packages, automatically pulled by misconfigured `npm install`.

2 What enclawed-enclaved gives you

2.1 Zero runtime dependencies in the framework boundary

Surface	Dependency declaration
Framework primitives (closed-tree + inherited OSS)	no `dependencies`
Admission gate	no `dependencies`
Closed-tree extensions	no `dependencies`

Table 1: The framework boundary — the code path that actually runs every gating decision — carries no transitive NPM dependencies.

2.2 Bundled extensions: not vendored, but gated

Surface	Dependency posture
Mirrored OpenClaw extensions	each carries its own `dependencies`; loaded only via the admission gate
Open-flavor admission	unsigned extensions load with a warning
Enclaved-flavor admission	unsigned, mis-signed, or under-verified network-capable extensions are denied at load
Per-extension egress narrowing	each admitted extension is restricted at runtime to its declared destinations

Table 2: Extensions are admitted by signed manifest, not absorbed into the framework's dependency graph. Their NPM deps live in their own package, behind the seal.
The CI workflow includes a guard that fails the build if the *framework boundary* accumulates runtime dependencies:

```
- name: Verify framework boundary has zero runtime deps
  run: |
    node -e "const p=require('./package.json');
             if (p.dependencies && Object.keys(p.dependencies).length)
               process.exit(1);"
```

2.3 FIPS Mode of Operation built on the host's validated provider

The closed tree's *FIPS Mode of Operation* engages the validated FIPS provider already shipping with your government- build host (RHEL FIPS, Ubuntu Pro FIPS, SLES FIPS, AL2023 FIPS-mode, Windows CNG). It does not bundle its own cryptographic implementation.
The runtime layer is a JS-side allowlist enforcement on top of the OpenSSL FIPS Provider's enforcement: defense in depth without adding a dependency.

2.4 Reproducible builds, deterministic prompts

The closed tree's prompt-cache stability rules ensure turn-to-turn determinism: same inputs produce same model prompts, byte-for-byte. The CI runs a regression test that fails the build on any cache-prefix divergence.

2.5 NIST OSCAL emit: SOC 2 / ISO 27001 evidence as JSON

The boundary emits the full **NIST OSCAL 1.2.2** FedRAMP submission model set on every assessment cycle: *Component Definition* (which 800-53 controls the boundary implements, with status and evidence pointers), *Assessment Results* (per-sample observations + per-control findings), *System Security Plan starter* (deployment-specific fields hard-gated for the platform team to fill in — FIPS-199 categorisation, authorization boundary, information types, users), and *Plan of Action and Milestones* (deployer-owned residuals and any not-satisfied finding). Bundled SVG architecture, audit-chain, and admission-gate diagrams ride along as OSCAL back-matter resources. All four documents are schema-validated against NIST's published JSON Schemas, Ed25519-signed, and recorded in the same hash-chained audit log as every gate decision. For a SaaS or platform-engineering team, this is the JSON the GRC platform (Drata, Vanta, Hyperproof, ServiceNow GRC) imports directly: SOC 2 Trust Services Criteria evidence, ISO/IEC

27001 Annex A control attestation, and machine-readable FedRAMP-package input — without an additional manual evidence-collection sprint per audit cycle.

3 Compliance posture for SaaS / platform engineering

Framework	Coverage
SOC 2 **CC8.x** (Change Management)	Module-signed releases, integrity manifest
ISO 27001 **A.5.20** (Supplier relationships)	Vendored OSS, gitignored signing key
ISO 27001 **A.8.32** (Change Management)	Re-sealing requires trust-root key
SLSA Level 3	Provenance + non-falsifiable build
NIST SSDF (SP 800-218)	PO + PS + PW practices
EU CRA (Cyber Resilience Act)	SBOM + vulnerability handling
NIST AI RMF 1.0 *Map / Measure / Manage*	Classification, audit, cloud-security monitor
OWASP LLM Top 10	Prompt-shield, DLP, two-layer egress allowlist, biconditional extension admission, mandatory boot-time accreditor, audit

4 Integration patterns

4.1 Pattern A: Sidecar gateway

Run enclawed-enclaved as a sidecar process in front of your LLM endpoint. Every request from your application is routed through prompt-shield, DLP, classification, egress allowlist, and the audit-log writer before reaching the LLM. The sidecar is FIPS-mode-engaged on RHEL/Ubuntu Pro/AL2023 hosts.

4.2 Pattern B: In-process library

For Node-native AI applications, import enclawed-enclaved directly. The closed-tree primitives compose with your existing code; you wrap your model calls with the gating primitives. No network hop, no extra container.

4.3 Pattern C: Tarball distribution

For air-gapped or isolated deployments, the customer tarball is self-contained: native binaries, public key, integrity manifest. No internet access required at deploy time, no `npm install`, no package registry resolution.

5 Performance + footprint

Metric	Value
Cold start	< 50 ms (Node 22, RHEL 9)
Per-request gating overhead (full path)	< 5 ms
Audit-log write throughput	> 50 k records/sec single-node
Accreditor witness round (M-of-N=3)	< 20 ms
Tarball size (without binaries)	8 MB

Customer-side native binary	4.5 MB (statically linked libcrypto)

6 Direct empirical evidence

This is not a marketing claim. We back it with a published statistical in-vivo harness (`enclawed/test/security/in-vivo/llm-narrative.mjs` in the public repository) that mediates 1600 chat-message samples through three subjects against real Discord and Telegram bot endpoints. Across the full run, upstream OpenClaw achieves **recall = 0.000** on every failure mode (F1 gate-bypass, F2 audit-forgery, F3 silent-host-failure, F4 wrong-target); both enclawed-oss and enclawed-enclaved achieve **precision = recall = F1 = 1.000** on all four. Total wall-clock for the full pass: 42 seconds. Per-sample ground-truth labels and per-subject decisions are written to a CSV; every gate decision lands in a tamper-evident audit log; every witness record is independently re-verifiable from a journal file. The companion paper, *Architectural Obsolescence of Unhardened Agentic-AI Runtimes*, formalizes the methodology.

Talk to us

Enclawed LLC
Alfredo Metere <alfredo.metere@enclawed.com>
A trial deployment and the technical-design package are available on request.