BMB NEXUS · GENESIS PLATFORM

VEIL: Code Integrity on Untrusted Hosts

How a proprietary cognitive system is delivered onto hardware the operator fully controls without handing over its internals. Sealed-build transformation, binary-level capability removal that anyone can verify independently, disclosure boundaries, and fail-closed behaviour — with an explicit account of what client-side integrity can and cannot claim.

Technical Specification · BMB Nexus
Updated 2026-08-04 · 9 pages
https://genesis.bmbnexus.ai/whitepapers/veil-integrity

Code Integrity · Anti-Tamper · Untrusted Host · Local-First · Software Protection · Threat Modelling

VEIL: Code Integrity on Untrusted Hosts

Technical Whitepaper v1.0

Protecting a proprietary cognitive system on hardware the operator fully controls.

BMB Nexus · Genesis Platform
Specification v1.0 · 2026-08-04 · genesis.bmbnexus.ai/whitepapers/veil-integrity
Security correspondence: care@bmbnexus.ai

Abstract

We present VEIL, the code-integrity layer of the Genesis platform. VEIL addresses a problem that server-hosted software never has to solve: **how do you deliver a proprietary system onto a machine you do not control, and not simply hand over its internals?** Local-first software has extraordinary economics — no marginal infrastructure cost per customer, no dependency on vendor availability — but it inverts the usual security posture. The runtime is owned by someone else. VEIL's response has three parts: **sealed-build transformation**, which converts distributable source into an encrypted bytecode artefact; **binary-level capability removal**, which disables the standard instrumentation entry points beneath the layer any application code could re-enable; and **runtime disclosure boundaries**, which prevent the system from revealing its own implementation even when legitimately asked. We state precisely what each layer achieves, and we are equally precise about what client-side integrity cannot claim: this is **resistance, not impossibility**. We also identify the one property of VEIL that a skeptic can verify independently on a shipped binary without trusting a word of this document.

Keywords: Code Integrity, Untrusted Host, Anti-Tamper, Software Protection, Bytecode, Capability Removal, Fail-Closed Design, Local-First Software, Disclosure Boundary, Supply-Chain Integrity

---

1. Introduction: The Local-First Paradox

Shipping software to run on the customer's own machine is the cheapest distribution model that exists. There is no per-user compute, no scaling curve, no infrastructure bill that grows with success, and no outage that takes every customer offline simultaneously. It is also the model that best serves the customer: their data stays with them, and the software keeps working whether or not the vendor does.

Almost nobody ships this way. The reason is not technical difficulty — it is that local delivery inverts the security posture that commercial software depends on. On a server, the vendor controls the runtime absolutely. On a local install, the operator controls the disk, the operating system, the memory, and the debugger. **Everything the vendor ships is in the hands of a party with total control over the environment in which it runs.**

Four consequences follow, and each is independently sufficient to deter a vendor from shipping local: entitlement logic can be modified, because there is no authority to check against; proprietary implementation is exposed, because the artefact is on the customer's disk; the vendor becomes responsible for data they cannot reach; and any collaborative feature appears to drag them back into hosting.

Genesis resolves all four, and VEIL resolves the second. Its role within the platform is narrow and specific: SERAPHUM binds entitlement to hardware, FORTRESS governs runtime secrets and protected data, AETHER protects what leaves the machine, and **VEIL protects the executable identity and integrity of the system itself.**

---

2. Threat Model: The Untrusted Host

VEIL is designed for the **untrusted-host** model, which we state formally because it is easily misread as an accusation.

2.1 Definition. The operator controls the disk, the operating system, the process environment, and the physical machine. They may therefore be *modelled* as an adversary with respect to protected implementation boundaries. This is a statement about capability, not about intent. It says nothing about any individual customer, in the same way that a bank's threat model says nothing about any individual depositor.

2.2 The distinction that makes this coherent. Authorisation to *use* a system is not authorisation to *dismantle* it. These are routinely conflated, and separating them is the core of VEIL's design.

The legitimate operator is trusted to:

- use the software for its intended purpose;
- access their own information through supported interfaces;
- authorise operations on their own behalf;
- export their own data in supported formats.

The operator is **not** automatically trusted by the architecture to:

- obtain raw cryptographic key material;
- attach instrumentation to a protected process;
- read protected implementation;
- modify entitlement logic;
- extract the system's internals or produce an unauthorised derivative.

2.3 A consequence worth stating plainly. Holding the master password authorises the software to operate on the user's behalf and to open the user's own data. It does not, by itself, expose raw key material, disable integrity controls, reveal protected implementation, or authorise modification of the security boundary. **Protecting the user's data and protecting the system's internals are different promises**, and the platform keeps both without one implying the other.

2.4 Adversary capability assumed. We assume an adversary who is patient, technically capable, has unrestricted physical and administrative access, may run any tooling, and may attempt the process any number of times. We do not assume they are prevented from trying.

---

3. Why Ordinary Distribution Is Insufficient

Most desktop applications built on web technology ship as readable source, or as source with cosmetic transformation. For a system whose implementation *is* the product, that is equivalent to publishing it.

3.1 Minification and obfuscation are not protection. Source-level obfuscation is reversible by widely available tooling in a time measured in seconds, not hours. We consider it a speed bump with a marketing department. VEIL does not rely on it as a boundary, and we say so because a document that presented obfuscation as security would tell a reader everything they need to know about the rest of the design.

3.2 Server-side checks do not transfer. A licence check that runs on the customer's machine is code the customer controls. Any scheme whose enforcement is a conditional branch in shippable source is defeated by editing the branch. Enforcement must be cryptographic (see SERAPHUM) and the artefact containing it must not be trivially editable — which is VEIL's job.

3.3 The realistic goal. VEIL raises the cost of extraction from *trivial* to *expert and sustained*, removes the standard instrumentation entry points beneath the layer application code can influence, and ensures failure is loud and terminal rather than silent and permissive. Section 10 is explicit that this is resistance rather than impossibility.

---

4. Sealed-Build Transformation

VEIL's primary mechanism is applied at build time: the distributable artefact is not the source, and is not derived from the source by a reversible transformation.

4.1 The pipeline, in outline.

- **Build identity.** Each build is stamped with a unique identity and with markers that survive later transformation stages, establishing artefact provenance.
- **Consolidation.** The runtime is resolved and consolidated into self-contained bundles, eliminating loose modules that could be individually replaced.
- **Bytecode compilation.** The bundles are compiled to V8 bytecode. This is a substantive step rather than a cosmetic one: recovering intent from bytecode requires engine-internals expertise, not a decompiler run.
- **Encryption.** The compiled artefacts are sealed under **AES-256-GCM**. The shipped artefact is ciphertext, and the authenticated mode means modification is detected rather than silently tolerated.
- **Key transition.** The key used for distribution is not the key used thereafter. At first unlock, the artefact is re-keyed to material derived within FORTRESS, binding the runtime artefact to the protected key hierarchy rather

than to anything shipped alongside it.

- **Native integrity.** Native components are hashed at build time so that substitution is detectable.
- **Loader hardening.** The component that performs decryption and loading is itself compiled and hardened rather than shipped as readable logic.

4.2 The measure that matters. After transformation, **exactly one readable source file ships** — a minimal bootstrap whose entire function is to hand control to the compiled loader. Everything else is bytecode under authenticated encryption. We state that as a single number because it is checkable by inspection of the package, and because "we protect our code" means nothing without one.

4.3 Bundle-integrity enforcement. The build refuses to produce an artefact whose runtime resolution could escape the sealed boundary. This check is performed by parsing the module graph rather than by pattern-matching text, because a pattern-matching version of the same check was found to both miss a real violation and reject valid constructs. **A guard that produces false results in both directions is worse than no guard**, since it is trusted while being wrong.

---

5. Binary-Level Capability Removal

Sealing an artefact is insufficient if the runtime that loads it can be trivially redirected. The standard techniques for subverting a desktop runtime do not involve attacking the application at all — they involve instructing the runtime to behave differently before the application starts.

VEIL therefore removes those capabilities **at the binary level**, beneath the layer any application code could reach:

- **Runtime option injection is disabled.** The standard environment mechanism for injecting a module into a process at startup — the most common single technique for subverting an application of this class — is removed from the binary.
- **Alternative execution modes are disabled.** The runtime cannot be invoked in a mode that bypasses the application shell.
- **Debug and inspection arguments are disabled.** The built-in remote-instrumentation entry points are removed.
- **Package integrity is enforced.** A modified application package is detected by the runtime itself.
- **Out-of-package loading is blocked.** Code cannot be sideloaded from outside the verified package.

5.1 Why this layer is different from everything else in this document. These controls are fused into the shipped binary before distribution. **No JavaScript — ours, or an attacker's — can re-enable them at runtime.** They are not checks that can be patched out; they are absent capabilities.

5.2 The property a skeptic can verify without trusting us. This is the unusual part, and it is the claim we most want scrutinised. Anti-tamper documentation is almost always unfalsifiable: a vendor asserts protections, and a reader has no way to check without reverse-engineering the product. **The fuse state of a shipped binary is**

externally readable using the standard published tooling for the runtime. Anyone holding the installer can confirm these capabilities are disabled, independently, in about a minute, without our cooperation and without taking a single sentence of this paper on faith.

We regard that as the most important sentence in this document. A security claim that cannot be checked is a marketing claim.

---

6. Runtime Boundaries

Two disclosure boundaries operate while the system runs. Both exist because a sufficiently capable system can be induced to describe itself, and self-description is extraction by other means.

6.1 Implementation is unreadable to the system itself. The software cannot read its own protected implementation, configuration, or key material through its ordinary file-access capabilities. The customer's own files remain fully accessible — that is the product's purpose — but the boundary between "the user's data" and "the system's internals" is enforced at the access layer rather than left to judgement.

6.2 The system does not narrate its own architecture. Requests to describe protected internals are answered in terms of *capability* rather than *implementation* — what the system can do, never how it is built or what its components are named. This closes a channel that is unique to systems with natural-language interfaces and that conventional software protection does not have to consider: **an interface that answers questions is an interface that can be interrogated.**

6.3 Transport hardening. Local interfaces between components are authenticated per request and structured so that observation of one component does not yield the ability to impersonate another. Details of this layer are deliberately omitted; see Section 9.

---

7. Fail-Closed Behaviour

Integrity systems are usually defeated at their failure paths rather than at their controls. A check that cannot complete, and therefore permits the operation, is not a check.

VEIL's rule is that **an integrity failure terminates the affected operation rather than degrading to a permissive path.** There is no fallback that proceeds on weaker terms, and no branch by which an unverifiable artefact is loaded anyway.

7.1 A distinction that must be preserved. Fail-closed applies to *integrity*, not to *availability*. An unreachable network endpoint is not an integrity failure and must never be treated as one — a vendor's infrastructure being down is not evidence that a customer's install is compromised. Conflating the two produces software that bricks itself when its vendor has an outage, which is precisely the dependency local-first delivery exists to eliminate. SERAPHUM states the corresponding rule for entitlement: an ambiguous result is never a denial.

8. Cooperation With the Other Layers

VEIL is not self-sufficient and is not designed to be. Its guarantees are meaningful only in composition.

- **With FORTRESS.** VEIL protects the artefact; FORTRESS protects the secrets the artefact uses. The re-key at first unlock (Section 4.1) is the join: the sealed artefact becomes dependent on the protected key hierarchy, so possessing the distributed artefact alone is not sufficient.
- **With SERAPHUM.** SERAPHUM binds entitlement cryptographically; VEIL ensures the binding is not simply edited out. Neither is sufficient alone — cryptographic binding in an editable artefact is theatre, and a sealed artefact with no binding grants everyone everything.
- **With AETHER.** AETHER protects what leaves the machine. VEIL ensures the endpoint doing the protecting is the endpoint we shipped.

Runtime anti-debugging and environment inspection are performed by FORTRESS, not by VEIL, and are documented in the FORTRESS specification. We draw that boundary explicitly rather than claiming a capability that lives in an adjacent layer, because a reader evaluating VEIL is entitled to know exactly which system does what.

---

9. What This Document Deliberately Omits

This paper specifies the architecture and the evidence. It does not specify the implementation map.

Omitted: detection heuristics and their thresholds; timing tolerances; the order in which verifications execute; scheduling of integrity operations; protected-region layouts; container internals; native component locations; decoy structure; termination triggers; and key-reconstruction sequencing.

This is not security through obscurity, and the distinction matters. The design is published; the security rests on the design, not on the design being unknown. What is withheld is operational detail whose only use to a reader is to shorten the path to a bypass. A cryptographic protocol is published in full because publication permits analysis. An anti-tamper implementation map is different in kind: it does not permit analysis of the design, it just supplies coordinates.

We also withhold specifics of any weakness identified during testing that is not yet remediated. Those are disclosed when they are fixed, not while they are live.

---

10. What VEIL Does Not Claim

We state this section in the strongest terms available, because overclaiming here is both technically indefensible and commercially fatal.

10.1 VEIL does not claim that extraction is impossible. Client-side integrity protection is properly described as **resistance, detection, and rapid termination** — never as mathematical impossibility. An adversary with full control of the machine, sufficient expertise, and unlimited time is not provably excluded. Industry guidance on client-side anti-tampering is explicit that such measures raise cost and are not absolute, and we adopt that position rather than contradicting it.

The defensible claim is:

> VEIL is designed to resist, detect, and rapidly terminate unauthorised inspection, modification, instrumentation, extraction, and forking — **including when the adversary controls the local machine.**

10.2 VEIL does not protect a machine already compromised at runtime. An adversary with privileged execution inside the process while it is legitimately unlocked is inside the boundary.

10.3 VEIL does not defend against the operator's own choices. Disclosed credentials and authorised-but-misunderstood actions defeat the layer by legitimate means.

10.4 VEIL has not undergone independent third-party audit. The verifiable property in Section 5.2 is real and checkable, but it covers one layer. The remainder rests on our own testing, and we state that rather than allowing the verifiable part to imply the whole.

10.5 Build-time protection is not continuous runtime attestation. VEIL's strength is concentrated in artefact transformation and capability removal — both applied before the software ever runs. It does not perform continuous runtime re-verification of its own code, and it would be inaccurate to imply otherwise. We consider the fused controls a stronger guarantee than a runtime check in any case, precisely because they cannot be patched out by the code they protect.

---

11. Status and Future Work

11.1 Deployed. Every stage described in Sections 4 through 7 is in production and applied to every shipped release.

11.2 Verified at release. Each release is inspected by extracting the distributed package and confirming that the expected sealed artefacts are present and that no readable implementation escaped the transformation. This is done because build success is not evidence of correct packaging: we have shipped builds that passed continuous integration while a required runtime asset had been removed by an unrelated cleanup step. **A green build is a claim; an extracted package is evidence.**

11.3 Sought. Independent third-party review, for the reason in Section 10.4.

11.4 On how this document is written. Every capability described here was verified against the shipped implementation before it was written down. Where a capability existed in an adjacent layer rather than in VEIL, it is credited to that layer (Section 8). Where a protection is build-time rather than continuous, it is described as build-time (Section 10.5). Claims we could not verify were removed rather than softened.

Appendix: VEIL within AETHERION

AETHERION is the Genesis security shield: four layers, four responsibilities, four specifications. Each states its own boundary so that no layer makes vague claims on behalf of the others.

The thesis the four layers share: the system trusts the operator to use it. It does not trust any process — including the operator's — to dismantle it.

THE FOUR LAYERS OF AETHERION

FORTRESS — The Vault

Runtime secrets, protected state, and user-owned data at rest. Owns runtime environment inspection.

SERAPHUM — The Seal

Binds entitlement, installation, and machine cryptographically, with no server in the trust path.

AETHER — The Voice

Every message, call, and file that leaves the endpoint, sealed post-quantum.

VEIL — The Skin

Executable identity and integrity on hardware the operator fully controls. *This document.*

Companion reading: SERAPHUM specifies the cryptographic entitlement binding VEIL protects from modification. FORTRESS specifies the key hierarchy the sealed artefact is re-keyed into at first unlock, and owns the runtime environment checks explicitly excluded from Section 8.

Cite as: BMB Nexus. “VEIL: Code Integrity on Untrusted Hosts.” Genesis Platform Technical Specification v1.0, 4 August 2026. https://genesis.bmbnexus.ai/whitepapers/veil-integrity