BMB NEXUS · GENESIS PLATFORM

FORTRESS v2: Data Protection Beyond the Managed Runtime

The second-generation FORTRESS architecture. Cryptographic operations moved out of the managed runtime into a native boundary, memory-resident secrets held in guarded regions with a microsecond exposure window, active deception against memory acquisition, and anti-instrumentation integrated into the key-derivation path itself. Includes an explicit correction to a claim made in the v1 specification.

Technical Specification · BMB Nexus
Updated 2026-08-04 · 9 pages
https://genesis.bmbnexus.ai/whitepapers/fortress-v2

Security · Cryptography · Memory Protection · Anti-Instrumentation · Key Derivation · Active Defence

FORTRESS v2: Data Protection Beyond the Managed Runtime

Technical Whitepaper v2.0

Supersedes "FORTRESS: A Quantum-Inspired Multi-Layer Security Architecture" (v1.0, January 2026), which remains published as a historical record.

BMB Nexus · Genesis Platform

Specification v2.0 · 2026-08-04 · genesis.bmbnexus.ai/whitepapers/fortress-v2

Security correspondence: care@bmbnexus.ai

Abstract

This document specifies the second-generation FORTRESS architecture, the data-protection layer of AETHERION. Where the v1 specification described a protection scheme implemented entirely within a managed JavaScript runtime, v2 describes a materially different system: cryptographic operations have been relocated to a **native boundary** outside the managed heap; secrets resident in memory are held in **guarded regions** that are unreadable to the process itself except during a microsecond-scale window; memory acquisition is met with **active deception** rather than passive encryption alone; and anti-instrumentation is integrated into the key-derivation path itself rather than bolted alongside it. We also state a **correction**: the v1 specification contained a claim about master-password compromise that was, on inspection, false, and this document supersedes it. Finally, v2 deliberately publishes *less* operational detail than v1, and §11 explains why that is a maturation rather than a retreat.

Keywords: Data at Rest, Key Derivation, Argon2id, Memory Protection, Guarded Regions, Anti-Instrumentation, Active Deception, Memory Acquisition, Zero-Knowledge Authentication, Defence in Depth

---

1. Why There Is a v2

The v1 specification was published in January 2026 and described the system accurately at that time. It is retained, unedited in substance, as a historical record — a security programme that quietly rewrites its own past claims is not one a reader should trust.

Three things happened between then and now.

1.1 The architecture changed rather than grew. v1 described protections implemented within a managed runtime. That approach has an upper bound: a managed heap is observable, its garbage collector relocates data outside the program's control, and its cryptographic primitives are reachable by anything else running in the same runtime. v2's central change is the introduction of a **native boundary** beneath the managed layer, which lifts that bound. This is not an optimisation of v1; it is a different structure.

1.2 The protection surface roughly doubled. v1 documented nine layers. The current system carries twelve, the additions being concerned with memory acquisition and process instrumentation — categories that passive encryption does not address.

1.3 We found an error in our own document. §12 states it explicitly rather than correcting it silently.

---

2. Threat Model

FORTRESS v2 protects user-owned data and runtime secrets against the following.

- **Device loss and offline analysis.** A stolen disk, a cloned image, or a recovered backup.
- **Memory acquisition.** An adversary obtaining a process or system memory image — by crash dump, hibernation artefact, a signed kernel driver, or a debugger. *New in v2; largely unaddressed in v1.*
- **Process instrumentation.** An adversary attaching to or observing the running process to recover material while it is legitimately in use. *Substantially expanded in v2.*
- **Hardware transplantation.** Moving protected artefacts to different hardware.
- **Offline password attack.** Recovering a passphrase from stored verification material.

2.1 The distinction v1 got wrong. Authorisation to *operate* the software is not authorisation to *dismantle* it. Holding the master password authorises the software to act on the user's behalf and to open the user's own data through supported interfaces. It does not, by itself, yield 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 separate promises**, and conflating them — as v1 did — understates the architecture while sounding appropriately humble.

2.2 Out of scope, stated precisely. An adversary with privileged execution inside the process while it is legitimately unlocked, at the moment secrets are in use, is inside the boundary. No data-protection layer resolves that. Resistance to code inspection and unauthorised modification is the responsibility of VEIL; entitlement binding is SERAPHUM's.

---

3. The Native Boundary

The defining change in v2. Cryptographic operations that previously executed inside the managed runtime now execute in a compiled native component.

3.1 Why the managed runtime is the wrong place for key material.

- A managed heap is **observable** — anything sharing the runtime can potentially reach it.
- A garbage collector **relocates and copies** data outside the program's control, so a "wiped" buffer may persist in an abandoned copy the program can no longer address.

- Managed strings are **immutable**, so a secret that passes through one cannot be reliably erased.
- Runtime-level instrumentation observes managed execution far more readily than compiled code.

3.2 What moved. Password-based key derivation, commitment creation and verification, and the sealed-artefact decrypt path now run natively. The decrypt path in particular is a single native operation that takes hardware-derived input and sealed material and returns plaintext, with **all intermediate key material zeroised before return** — no intermediate value is ever held in a managed object.

3.3 What this does and does not buy. It removes an entire category of exposure — managed-heap residue and GC-copied secrets — and it makes observation materially harder. It does not make memory inspection impossible for a sufficiently privileged adversary. §5 and §6 address what remains.

---

4. Key Derivation and Authentication

4.1 Memory-hard derivation. Primary derivation uses **Argon2id**, chosen because it imposes a memory cost that resists the parallel hardware advantage available to an offline attacker. Iterated-hash derivation (PBKDF2-HMAC-SHA512, at 600,000 iterations) is retained for compatibility with material created under earlier formats.

4.2 Per-install salting, introduced compatibly. Derivation salts are randomised per installation, so identical passphrases on two machines produce unrelated material and no precomputation transfers between installs. This was introduced *without invalidating existing installs*: the derivation interface accepts either a per-install salt or the legacy path, and byte-identical output for legacy inputs is enforced by parity tests against the original vectors. **A security improvement that locks existing customers out of their own data is not an improvement**, and in a perpetual-licence product it is unacceptable.

4.3 The password is never stored. Verification is by zero-knowledge commitment. Both the current and legacy commitment formats verify correctly, so upgrades never strand an existing user.

4.4 The anti-instrumentation principle worth publishing. Anti-debug logic is integrated into the derivation path itself rather than performed as a separate check. That integration raises a subtlety worth stating as a general lesson, because it is not obvious and getting it wrong is worse than omitting the defence entirely.

The naive design short-circuits: if instrumentation is detected, skip the expensive derivation and fail early. That is a mistake. Memory-hard derivation costs on the order of a hundred milliseconds; short-circuiting returns in microseconds. **The defence therefore creates a timing differential of roughly five orders of magnitude, trivially measurable, which tells an attacker precisely when they were detected** — converting a defence into an oracle that reports the state of the defence.

FORTRESS always performs the full derivation regardless of detection state, and decides what to return only afterwards. The cost is real and accepted. This is the same principle stated in the SERAPHUM specification for signature timing: **a security control must not be observable by its own performance.**

---

5. Guarded Memory Regions

Encryption at rest protects a powered-off disk. It does nothing for material that must exist in cleartext while in use. v2 addresses the resident-secret window directly.

Sensitive material — master key, key shares, database keys, communication keys — is held in **guarded regions** rather than ordinary allocations. A guarded region is mapped **inaccessible by default**: the process itself cannot read it. Legitimate use briefly transitions the region to readable, executes a bounded native routine with no allocation and no managed-runtime involvement, then returns it to inaccessible.

5.1 What this achieves. The window during which a secret is readable in the process address space is reduced from "the lifetime of the object" to **a few microseconds per legitimate access.** A memory image captured outside those windows contains inaccessible mappings rather than key material. It also converts stray reads into faults instead of silent successes, which turns an entire class of bug from a leak into a crash.

5.2 Compatible degradation. Where the native guard is unavailable, the system falls back to the prior-generation approach — scrambled representation in locked, non-dumpable pages. The fallback is weaker and is described as weaker.

---

6. Active Defence Against Memory Acquisition

The threat this addresses is an adversary who obtains a memory image despite everything above — via a signed kernel driver, a crash artefact, or hibernation state.

6.1 Decoy regions. The process maintains regions containing material shaped to resemble the real thing. An adversary searching an acquired image encounters candidate material that is structurally plausible and cryptographically useless. The intent is not to make acquisition impossible but to make **the results of acquisition unreliable**, and to impose analysis cost on every candidate. Passive encryption yields nothing to an attacker's search; deception yields something wrong, which is more expensive.

6.2 Dump corruption. Where a memory image is being produced, protected regions are structured so that the resulting artefact is degraded rather than faithful.

6.3 Process hardening. On platforms that support it, the process applies available operating-system mitigations at startup and relinquishes privileges it does not require, reducing the ways it can be attached to or manipulated. Specific mitigations are deliberately not enumerated (§11).

---

7. Liveness and Cascade Collapse

FORTRESS is not a static container. It maintains a continuous liveness check that establishes statistical baselines for its own behaviour, accumulates soft and hard failures with decay so that transient noise does not accumulate into a false positive, and verifies that internal state remains self-consistent.

On sustained anomaly, **cascade collapse** destroys resident key material. Timing manipulation and tampering therefore do not merely fail — they accumulate toward a terminal response.

7.1 The design constraint that matters. An aggressive collapse threshold destroys a legitimate user's session on a laptop that was merely suspended, a busy machine, or a slow disk. **A protection that fires on ordinary computing is not a protection; it is a defect with a security justification.** Thresholds are therefore derived from observed baselines rather than fixed, and the specific values are not published (§11).

---

8. Integrity Verification

Every database write is checksummed and every read verified; unverified records are filtered rather than served. Background verification is **prioritised by predicted risk** — derived from record criticality, observed write activity, and historical anomalies — rather than sweeping uniformly, so that verification effort concentrates where change actually occurred.

---

9. Recovery

A protection layer that can strand a user from their own data has failed at its purpose regardless of its cryptographic quality.

All protected databases can be exported to plain, portable formats using raw key material, independently of the FORTRESS runtime. If the application refuses to start, the data remains recoverable. **We regard this as a security feature, not a concession:** in a perpetual-ownership product, the ability to leave is part of what was sold.

---

10. Composition

FORTRESS protects data and secrets. It does not protect the code enforcing those protections — that is VEIL, and without it FORTRESS's controls would be advisory rather than durable. It does not bind entitlement — that is SERAPHUM. It does not protect material in transit — that is AETHER. FORTRESS additionally owns **runtime environment inspection** for the platform, including debugger and virtualisation detection, which is why the VEIL specification explicitly disclaims it.

---

11. Disclosure Policy — Why v2 Says Less

A reader comparing the two documents will notice that v2 omits material v1 published: exact intervals, specific thresholds, verification ordering, enumerated environment indicators, and implementation pseudocode. This is deliberate, and we consider v1's disclosure an error rather than a virtue.

The distinction is between design and coordinates. A cryptographic protocol is published in full because publication enables analysis — the security rests on the construction, not on the construction being unknown. An anti-tamper implementation map is different in kind: publishing exact thresholds and check ordering does not enable a reader to evaluate the design, it just tells an attacker where to begin and when they have been noticed.

v2 therefore publishes architecture, principles, and the reasoning behind each design decision — enough to evaluate the system — while withholding detection heuristics, thresholds, tolerances, verification ordering, region layouts, native symbol details, and deception structure. **This is not security through obscurity: the design is here, and it is intended to be argued with.**

We likewise do not publish specifics of any weakness identified in testing that is not yet remediated. Those are disclosed when they are fixed.

---

12. Corrections to v1

12.1 The master-password claim. The v1 specification states, in its limitations section: *"No protection against compromised master password: If the user's password is known, all protections are bypassed."*

That statement is incorrect, and it was incorrect when written. Knowledge of the master password permits the software to open the user's own data through supported interfaces. It does not by itself yield raw key material, disable integrity verification, expose protected implementation, or authorise modification of the security boundary — those are governed by protections the password does not unlock. v1 conceded a defence the architecture actually provides.

The corrected statement is §2.1. We note this prominently for two reasons. It is the kind of error that propagates — a limitation stated by a vendor is quoted by evaluators as authoritative. And a specification that silently improves its own past claims is not a document a reader can rely on. **The correction is more useful published than fixed quietly.**

12.2 Scope of the v1 companion validation. The penetration-test report accompanying v1 documents adversarial testing against the *v1* architecture. Its findings are not evidence about the system described here. The native boundary, guarded regions, and acquisition defences in this document did not exist when that testing was performed and have not been subjected to equivalent external adversarial review. **A validation result does not transfer across an architectural change**, and treating it as though it does would be the same class of error as §12.1.

---

13. What FORTRESS Does Not Claim

13.1 Not impossibility. A sufficiently privileged, patient, expert adversary with full control of the machine is not provably excluded. FORTRESS raises cost, narrows windows, and converts silent compromise into detected or degraded compromise. Any data-protection layer claiming more on a machine it does not control is misrepresenting the problem.

13.2 Not protection during legitimate use. An adversary executing privileged code inside the unlocked process is inside the boundary (§2.2).

13.3 Not protection against the user's own decisions. Disclosed passphrases and authorised-but-misunderstood actions defeat the layer legitimately.

13.4 Not a guarantee against every acquisition technique. §6 imposes cost and unreliability on memory acquisition. It does not claim to defeat all methods, and an adversary capable of arbitrary kernel-level access at an arbitrary moment may succeed.

13.5 Not independently audited. The v2 architecture has not undergone third-party review. Individual primitives follow published standards and are tested internally; that is not equivalent, and §12.2 explains why the v1 validation does not close this gap. We consider it the most significant open item in this document.

13.6 One known limitation, disclosed. A master-password change re-keys FORTRESS but does not currently re-key subsidiary domain databases. The user-facing control is disabled pending correction. We publish this because a security document that lists only resolved issues is a marketing document, and because a reader is better served knowing which control is unavailable and why.

---

14. Status

Deployed. Every mechanism in §3 through §9 is in production and applied to every shipped release.

Superseded. This document supersedes the v1 specification, which remains published as a historical record. Where the two conflict, this document is correct.

Open. Independent third-party review of the v2 architecture (§13.5); re-validation to replace the v1-era penetration test (§12.2); remediation of the limitation in §13.6.

On method. Every claim here was checked against the implementation before it was written. Mechanisms belonging to adjacent layers are credited to those layers. Claims that could not be verified were removed rather than softened — and one claim from the previous specification was removed because verification showed it to be false.

Appendix: FORTRESS within AETHERION

AETHERION is four layers with four responsibilities. Each states its own boundary so that no layer makes vague claims on behalf of the others.

THE FOUR LAYERS OF AETHERION

FORTRESS — The Vault

Runtime secrets, protected state, and user-owned data at rest. Owns runtime environment inspection. *This document (v2).*

VEIL — The Skin

Executable identity and integrity on hardware the operator fully controls.

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.

Historical: the v1 specification (January 2026) and its companion penetration-test report remain published. Both describe the v1 architecture; see §12 for the corrections that apply to them.

Cite as: BMB Nexus. “FORTRESS v2: Data Protection Beyond the Managed Runtime.” Genesis Platform Technical Specification v2.0, 4 August 2026. https://genesis.bmbnexus.ai/whitepapers/fortress-v2