kovra

WHITEPAPER · V1.0 · JUNE 2026

Secrets in the age of AI Agents

kovra's whitepaper

A product of Kaeus Inc

kovra.sh

> **What this is.** This is **kovra's whitepaper** — a design-rationale article, not a research paper. It makes no empirical claims and reports no experiments; it argues from a real, observable problem and from established security principles toward the specific choices kovra makes — and it is candid about what those choices do **not** buy you. Every design claim below maps to a behaviour the tool actually enforces.

01 The problem

A secret is only useful when something uses it. So secrets spend their lives in motion: typed into terminals, exported into shells, written into `.env` files, pasted between a password manager and a config, copied into a dozen `export` lines. Each step leaves a residue — in shell history, in process listings, in logs, in a file that outlives its purpose. The industry's usual answer is a checklist: don't paste that there, rotate this, scrub those logs. Checklists lose to convenience, reliably, because the insecure path is the easy one and the secure path is friction.

Then a new actor walks into this picture: the **AI coding agent**. You point it at your repository to move faster, and in doing so you grant it the same reach you have. It can open every `.env`, read every config, scroll your shell history. The secrets that were merely sprawled are now legible to an automated reader that acts on what it reads.

This is the problem kovra exists to address: a developer needs their tools — and now their agents — to use secrets, while as little as possible ever sees them in cleartext, and while the easy path is also the safe one.

02 The tensions

The problem is hard because it is a knot of genuine tensions, not a single missing feature. Naming them honestly is the only way to reason about a solution.

Use versus exposure. A secret must be in cleartext somewhere at the moment of use — a database driver needs the actual password. You cannot both use a value and guarantee nothing ever sees it. The realistic goal is to shrink the set of things that see it, and the duration — not to reach zero.

Convenience versus control. Every control you add (a prompt, an allowlist, a confirmation) is friction, and friction is precisely what drives people back to the plaintext `.env`. A control that is too heavy doesn't get used; security that isn't used isn't security.

Agent usefulness versus agent containment. An agent is valuable because it can run your commands and touch your systems. The same capability is the risk. Lock it out of everything and it's useless; let it read everything and it's dangerous.

A trusted principal that can be manipulated. Classic threat models assume a principal who is trusted by default and occasionally betrays. An LLM agent is different in kind: it is **manipulable by the content it reads**. A poisoned README, a crafted error message, a malicious dependency's docstring can redirect it. The theoretical limit on what it could leak is the same as for a human; the expected frequency of an attempt is higher, and the trigger can be data, not intent.

03 The implications

Take those tensions seriously and several conclusions follow before any code is written.

1. **Containment, not prevention.** Since a value must be cleartext at the point of use, the design target is to reduce surface and to keep the most dangerous values away from the least trusted readers — not to promise the impossible.
2. **Default to safe, make safe convenient.** If the secure path is harder than pasting a secret, the secure path loses. The tool has to make using a secret correctly at least as easy as using it carelessly — otherwise its own controls select for being bypassed.
3. **Metadata is not plaintext.** An agent can be enormously useful knowing only that a secret exists, what it's called, and how sensitive it is — without ever seeing its value. The right unit to give an agent is metadata plus the ability to run things, not the value.
4. **The boundary belongs in one place.** If each interface re-implements "what's allowed," they will drift, and the weakest implementation becomes the de-facto policy. The rule has to live in one core that every interface consumes.
5. **Some risk is the human's to accept, deliberately.** There are moments a person genuinely needs a value on screen. The answer isn't to forbid it but to make it a deliberate, attended, audited act — never a default, never something an agent can trigger on its own.

04 The solution

kovra's model is a direct response to those implications. Its shape is: let things use secrets without seeing them, and put every exception behind a deliberate human act.

Use, don't see

Tools and agents get values through **injection**: kovra resolves a secret and places it directly into a child process's environment, never onto disk, into `argv`, or into shell history. The process uses the value; nothing in your workflow displays it. A committable `.env.refs` file maps variable names to **coordinates** — addresses, not values — so the wiring is shareable while the secrets stay in the vault.

Metadata for agents, plaintext withheld

An agent connects over an MCP server under a **scope** — a capability that says what it may address and do. It reads metadata freely and injects secrets into the commands it runs, so those commands work — but the plaintext of your sensitive secrets never lands in the model's context window, which is the one place a prompt-injection attack could exfiltrate it.

Sensitivity decides delivery; the environment adds a floor

Each secret carries a sensitivity tier. `low` and `medium` flow directly; `high` requires an **attended biometric confirmation** before any delivery; `inject-only` is never revealed at all. The `prod` environment adds a structural floor on top — a `prod` secret is born `high`, can't be packaged for offline sharing, and its plaintext can reach an agent's context only through a human-initiated, confirmed reveal.

Keep the executor outside the agent's control

For the most dangerous case — injecting a `high`/`prod` secret — kovra adds an **executor allowlist**: the value may only be injected into a reviewed, allowlisted executable, not an ad-hoc script the agent just wrote. This is the crux. A process the agent authored can print its own environment; injection alone contains nothing from an executor the agent controls. Containment comes from the executable being outside that control.

One core, authoritative prompts

The policy lives in the **core**; the CLI, wrapper, web UI, and MCP server consume its decisions and never re-derive them. When a confirmation is required, the prompt text is built by the core from observed facts — the resolved command, the coordinate, the sensitivity — and is never supplied by the caller, so an attacker can't forge a reassuring prompt.

05 The cryptography

kovra deliberately uses a **small set of modern, well-reviewed primitives** from the Rust cryptography ecosystem, in standard ways. There is no homegrown cryptography here — the interesting, kovra-specific work lives in policy, not in inventing ciphers. Each choice maps to one job.

Primitive	Where it's used	Why this one
ChaCha20-Poly1305	Encryption at rest (every vault entry)	Authenticated, constant-time in software
Argon2id	Deriving a key from a passphrase	Memory-hard against brute force

Primitive	Where it's used	Why this one
BLAKE3	Secret fingerprints	Fast, modern; stored truncated (I12)
ed25519 (RSA for compat)	Keypair credentials, signing, sealing	Small, fast, hard to misuse
age (X25519 + ChaCha20-Poly1305)	Offline packages, master-key backup	Recipient-based, audited, no knobs
secrecy / zeroize	In-memory handling	Shrinks the plaintext window

Encryption at rest — ChaCha20-Poly1305

Every entry in the vault is sealed with the **ChaCha20-Poly1305** AEAD. An AEAD gives confidentiality and integrity in one step: a tampered ciphertext fails to authenticate rather than decrypting to plausible garbage. We choose it over AES-GCM because it is **constant-time in pure software** — it doesn't depend on AES hardware acceleration to avoid cache-timing side channels — so it behaves identically and safely on any machine kovra runs on.

Key derivation from a passphrase — Argon2id

When a vault is protected by a passphrase instead of the OS keychain, the encryption key is derived with **Argon2id** — the current password-hashing standard. It is **memory-hard**, which makes GPU and ASIC brute force expensive, and the `id` variant resists both side-channel and time-memory-tradeoff attacks. A human passphrase is low-entropy; a memory-hard KDF is what makes it safe to use as a key at all.

Identity & fingerprints — BLAKE3

Secrets are fingerprinted with **BLAKE3**, giving a stable, collision-resistant identity for a value without revealing it. kovra only ever stores and shows a **truncated** fingerprint — never one long enough to let someone confirm a guessed value by matching its hash (I12). The truncation is a deliberate anti-brute-force measure, not a shortcut.

Asymmetric keys — ed25519 (RSA for compatibility)

Keypair credentials default to **ed25519** (EdDSA): small keys, fast deterministic signatures, and no parameters to get wrong. **RSA** is supported but scoped to signing/verification and SSH compatibility — never asymmetric encryption, because RSA encryption invites padding-oracle footguns. Keys are generated and stored in the **OpenSSH format** (via `ssh-key`), so they interoperate cleanly with the ssh-agent and standard tooling. Asymmetric encryption is ed25519-only.

Sealed sharing & key backup — age

Offline packages and the encrypted master-key backup are sealed with **age** (X25519 key agreement + ChaCha20-Poly1305, ASCII-armored). age is a small, audited, opinionated format with **no configuration knobs to misuse**, and it is **recipient-based** — a package is sealed to who the recipient is (their public key), which is exactly the property kovra wants: authorization anchored to identity, not to possession of a file (I8). In passphrase mode the same format backs the master-key export, so a backup can be recovered with any age implementation in a disaster.

Memory hygiene — secrecy & zeroize

Not algorithms, but part of the same discipline: secret-bearing values are wrapped so they never land in logs or debug output, and their memory is **zeroized** when dropped — shrinking the window in which a plaintext lives in process memory. It doesn't change the last-mile limit, but it narrows it.

06 The risks

A security tool introduces its own risks; pretending otherwise would be the opposite of this article's intent.

The master key is a single root of trust. One per-vault key encrypts everything. Lose it and the vault is unrecoverable; leak it and the encryption at rest is moot. kovra custodies it in the OS keychain and offers an encrypted, passphrase-protected backup — but the concentration of trust is real, and key hygiene is now your most important habit.

The tool is part of your supply chain. kovra runs on your machine with access to your secrets. A compromise of the binary, its dependencies, or its build is a compromise of everything it guards. This is inherent to any secrets manager and is the reason for a small dependency surface and a conservative posture — not a risk that disappears.

Confirmation fatigue. Prompts are a control only while they're read. Ask too often and people approve reflexively, which is why kovra gates on sensitivity rather than prompting for everything — but a poorly-tiered vault can still train you to click "approve" without looking.

A convincing prompt is still a human decision. Authoritative prompt text raises the bar against forged prompts, but the human can still approve a legitimate-looking, genuinely-bad action. The tool informs the decision; it does not make it.

07 The limitations

These are not gaps to be closed in a later version. They are properties of the problem, and naming them is what keeps the rest of the article honest.

The last mile is inevitable. At the instant of use, the plaintext lives in a process's memory, and whoever controls that process can read it. No tool can hand a value to your application while preventing the application from reading it. Like every serious secrets manager — Vault, AWS Secrets Manager, Azure Key Vault, Doppler — kovra does **not** try to prevent the authorized principal from reading the secret. It invests in encryption, access control, audit, and surface reduction: "assume breach" mitigations, all probabilistic.

For a truly critical secret, containment lives in how the tool is used. The robust protection for a critical `prod` value is that the agent does not control the executable that receives it — reviewed deploy artifacts, not ad-hoc agent scripts. The vault enables that discipline; it cannot enforce it for you.

kovra governs the authentication event, not the session it opens. When kovra signs an SSH challenge or injects a database password, it governs that moment. The session that opens afterward is outside its reach; kovra is not a network proxy or a runtime sandbox.

A compromised host is out of scope. kovra defends against secret sprawl and against an agent reading what it shouldn't. It is not a defense against malware with your privileges, a kernel-level keylogger, or an attacker who already owns your machine.

The agent threat is reduced, not eliminated. Keeping plaintext out of the model's context closes the exfiltration-by-prompt-injection path for sensitive secrets. It does not make an agent trustworthy, and it does not stop an agent from misusing a value it was legitimately allowed to use.

08 In sum

kovra doesn't claim to solve secret management; that problem has a proven floor and this article has named it. What it does is move the easy path and the safe path into alignment, shrink what sees a secret in cleartext, and place the AI agent on the right side of a metadata-versus-plaintext line — with every exception turned into a deliberate, attended, audited human act. That is a meaningful, honest improvement in a setting where the reader of your secrets is now automated and manipulable. It is not, and does not pretend to be, the abolition of the last mile.