keychains.dev

Keychains.dev Security Whitepaper

Architecture, Threat Model, and Cryptographic Design

Version 1.1 — February 2026
Author Interagentic, Inc.
Contact security@interagentic.inc
Web https://keychains.dev/security

1. Executive Summary

Keychains.dev is a credential delegation proxy purpose-built for AI agents. It solves a fundamental problem in agentic AI: how do you give an autonomous agent access to third-party APIs without exposing the underlying credentials?

Today, developers commonly pass API keys, OAuth tokens, and other secrets directly into agent environments — through environment variables, tool configurations, or system prompts. This is dangerous. Agent context windows are vulnerable to prompt injection, tool outputs can be manipulated, and any credential in the agent's memory can be exfiltrated by a sufficiently clever attack.

Keychains eliminates this attack surface entirely. Instead of giving the agent credentials, the agent authenticates itself to Keychains using an Ed25519 SSH keypair. When the agent needs to call a third-party API, it sends the request through the Keychains proxy, which injects the user's credentials server-side. The agent never sees, handles, or stores the actual secrets.

Core Security Properties

- **Credential isolation.** User secrets are never present in the agent's context window, tool outputs, or environment variables. They exist only in encrypted storage and in proxy memory during request execution.
- **Machine identity.** Each agent is identified by an Ed25519 keypair with challenge-response authentication. A rolling hash chain ratchet detects stolen key usage on the first unauthorized call.
- **Scoped delegation.** Users grant fine-grained, time-limited permissions. Agents cannot escalate beyond their delegated scope. The proxy enforces boundaries server-side.
- **Destination binding.** Each credential can only be forwarded to its designated provider API. Credential redirection attacks are architecturally impossible.
- **Tamper-evident audit.** Every proxied request is logged in a SHA-256 hash chain and archived to S3 WORM storage. Audit logs cannot be silently modified.

This document provides a comprehensive analysis of the security architecture, threat model, cryptographic primitives, trust boundaries, known limitations, and incident response procedures for the Keychains.dev platform.

2. Threat Model — STRIDE Analysis

We use the STRIDE framework (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to systematically analyze threats against the Keychains platform. For each category, we document the threat scenario, the mitigation implemented, and the residual risk we acknowledge.

S — Spoofing

THREAT SCENARIO

An attacker impersonates a legitimate machine to gain access to a user's delegated credentials. This could occur through SSH private key theft (filesystem exploit, backup exposure, or memory dump), man-in-the-middle interception of authentication tokens, or registration of a rogue machine under a compromised user account.

MITIGATION

Machine identity is established through Ed25519 SSH keypairs using challenge-response authentication. The server issues a random challenge; the machine signs it with its private key; the server verifies against the registered public key. No bearer tokens are used for machine authentication.

Critically, each authentication advances a rolling hash chain shared between client and server. The client stores its current chain state and sends it with each request. If an attacker copies the private key and authenticates, the chain state diverges — the very next request from either the legitimate machine or the attacker triggers a state mismatch, and the key is immediately invalidated. This means stolen keys are detected on first use, not after damage is done.

RESIDUAL RISK

If an attacker steals both the private key and the current chain state before the legitimate machine makes another request, they could make a limited number of calls before detection. The window is bounded by the time until the legitimate machine's next authentication attempt. In practice, active agents authenticate frequently, keeping this window small.

T — Tampering

THREAT SCENARIO

An attacker modifies audit log entries to conceal unauthorized credential usage, or tampers with proxy request/response data in transit to inject malicious payloads.

MITIGATION

Audit logs are structured as a SHA-256 hash chain. Each log entry includes the hash of the previous entry, creating a tamper-evident sequence. Any modification to a historical entry breaks the chain and is detectable through verification.

Audit logs are archived to AWS S3 with Object Lock (WORM — Write Once, Read Many), providing an

immutable secondary copy. Even if the primary database is compromised, the S3 archive preserves the original record.

All proxy communication uses TLS 1.2+ for transport security. Request integrity is ensured by the SSH signature on the original request.

RESIDUAL RISK

The hash chain is only as trustworthy as the initial seed. If an attacker gains write access to both the primary database and S3 archive simultaneously, they could theoretically reconstruct a valid chain. We mitigate this through separate access credentials for each storage layer.

R — Repudiation

THREAT SCENARIO

A user or machine denies having performed a specific API call, or an attacker disputes the audit trail to evade accountability.

MITIGATION

Every proxied request is logged with the machine's public key fingerprint, the SSH signature of the request, the delegated permission token used, the target API endpoint, the timestamp, and the hash chain position. The SSH signature cryptographically binds the request to the machine's identity — only the holder of the private key could have produced it. Combined with the hash chain and WORM archival, this creates a non-repudiable audit trail.

RESIDUAL RISK

If a machine's private key is compromised, the attacker can produce valid signatures indistinguishable from the legitimate machine's. The rolling hash chain mitigates this by detecting the compromise, but actions taken before detection cannot be attributed with certainty.

I — Information Disclosure

THREAT SCENARIO

Credentials are leaked through the agent's context window (prompt injection), through API response headers containing auth tokens, through server logs, or through a breach of the credential store.

MITIGATION

The proxy architecture ensures credentials never enter the agent's environment. The agent sends requests to the Keychains proxy; credentials are injected server-side. The agent receives only the API response body.

Response headers are scrubbed before returning to the agent — authorization headers, set-cookie headers, and other sensitive headers are removed. Credentials at rest are encrypted with AES-256-GCM using per-user derived keys. Server-side logs never contain raw credential values.

RESIDUAL RISK

API response bodies may contain sensitive data (user PII, financial data, etc.). While these are not credentials, they are accessible to the agent and could be exfiltrated via prompt injection. This is inherent to the agent having functional API access and is outside Keychains' scope to prevent — the agent needs to see responses to do its work.

D — Denial of Service

THREAT SCENARIO

An attacker floods the proxy with requests to exhaust rate limits on the user's third-party API accounts, or overwhelms the Keychains infrastructure itself.

MITIGATION

Permission tokens are short-lived (configurable, default 5–15 minutes) and rate-limited per machine. The proxy enforces per-user and per-machine request quotas. Machine authentication via challenge-response is computationally inexpensive to reject (signature verification fails fast). Infrastructure runs on auto-scaling serverless compute (Vercel) with upstream DDoS protection.

RESIDUAL RISK

A compromised machine with a valid permission token could issue many requests within the token's TTL. Rate limits bound the damage, but some unauthorized API calls may succeed before revocation takes effect. Aggressive rate limiting can mitigate this at the cost of legitimate throughput.

E — Elevation of Privilege

THREAT SCENARIO

An agent or sub-agent attempts to access APIs or scopes beyond what the user explicitly delegated. A parent agent delegates more permissions to a child agent than it holds itself. An attacker manipulates delegation flows to escalate scope.

MITIGATION

Permission tokens encode the exact scopes granted by the user. The proxy validates every request against the token's scope before injecting credentials. Delegation is constrained by scope ceilings — a delegating agent can only pass a subset of its own permissions, never escalate beyond them. Scope validation is performed server-side; the agent cannot bypass it by modifying the token (tokens are signed with EdDSA using server-held keys).

Destination binding is enforced independently of scope: even if an agent constructs a request to an unauthorized endpoint, the proxy rejects it unless the endpoint matches the credential's registered provider.

RESIDUAL RISK

If scopes are defined too broadly by the credential provider (e.g., a single OAuth scope grants read and write access), Keychains cannot enforce finer granularity than the upstream API supports. Users should prefer credentials with the narrowest available scope.

3. Cryptographic Primitives

Keychains relies on a small set of well-understood, industry-standard cryptographic primitives. We deliberately avoid novel or experimental cryptography.

Ed25519 — Machine Identity

Machine authentication uses Ed25519 (Edwards-curve Digital Signature Algorithm over Curve25519). Ed25519 provides 128-bit security strength, produces compact 64-byte signatures, and is deterministic (no random nonce required, eliminating a class of implementation vulnerabilities that have historically affected ECDSA).

The authentication flow is challenge-response: the server generates a cryptographically random challenge, the machine signs it with its Ed25519 private key, and the server verifies the signature against the registered public key. This avoids bearer token semantics — intercepting a completed authentication exchange does not grant the attacker future access.

EdDSA (Ed25519) — JWT Token Signing

Permission tokens and internal JWTs are signed using EdDSA (Ed25519) with server-held private keys. Public keys are published via a JWKS (JSON Web Key Set) endpoint, enabling token verification without sharing signing keys.

Key rotation is performed by adding a new key to the JWKS and deprecating the old one. Tokens signed with the old key remain valid until their natural expiry (short TTL), but no new tokens are issued with the deprecated key. This allows zero-downtime rotation without token invalidation races.

AES-256-GCM — Credential Encryption at Rest

User credentials are encrypted at rest using AES-256-GCM (Galois/Counter Mode). AES-256 provides 256-bit key strength, and GCM provides both confidentiality and authenticity — a tampered ciphertext will fail authentication during decryption.

Encryption keys are derived per-user, ensuring that a breach of one user's key material does not compromise other users' credentials. Key derivation uses standard KDF practices with unique salts per user.

SHA-256 Hash Chains — Audit Log Integrity

Audit logs are structured as a SHA-256 hash chain. Each entry contains: the log payload, a timestamp, and the SHA-256 hash of the previous entry. This creates an append-only, tamper-evident log structure. Verification is O(n) — any break in the chain is detectable by recomputing hashes from the genesis entry forward.

Rolling Hash Chain Ratchet — Stolen Key Detection

The machine authentication system maintains a rolling hash chain between client and server. After each successful authentication, both parties advance their state by computing H(current_state || session_data). This creates a synchronized ratchet: the state moves forward irreversibly.

If an attacker clones the private key and authenticates, they advance the server's state along a different path. The legitimate machine's next request presents a state that no longer matches the server's expectation — triggering immediate key invalidation and user notification. This mechanism detects stolen keys on first use by the attacker, regardless of whether the attacker's request succeeds.

4. Trust Boundaries

Understanding where credentials exist — and where they do not — is essential to evaluating the security posture of any credential management system. This section maps the trust boundaries in the Keychains architecture.

Credential Lifecycle

- **At rest:** Encrypted with AES-256-GCM in MongoDB Atlas. Per-user key derivation. The database never stores plaintext credentials.
- **In transit (user ’Keychains):** Credentials are submitted over TLS during the initial setup flow. They are encrypted before being written to the database.
- **In proxy memory:** During a proxied request, the credential is decrypted in the serverless function's memory, injected into the outgoing HTTP request, and discarded. The decrypted value exists in memory for the duration of a single request (typically milliseconds).
- **In transit (Keychains ’target API):** The credential is sent to the target API over TLS as part of the proxied request (typically as an Authorization header or API key parameter).

What the Agent Sees

- API response bodies from the target service (necessary for the agent to function).
- A short-lived permission token (opaque to the agent; contains no secrets).
- The proxy endpoint URL. The agent knows it is sending requests through a proxy.
- **Never:** Raw OAuth tokens, API keys, passwords, or any credential material.

Infrastructure Dependencies

Keychains runs on managed cloud infrastructure. We are transparent about the trust we place in each provider:

- **Vercel (compute):** Serverless functions execute proxy logic. We trust Vercel's function isolation model — that one invocation's memory is not accessible to another. Vercel's SOC 2 Type II compliance provides assurance here.
- **MongoDB Atlas (credential store):** Encrypted at rest (MongoDB's built-in encryption) plus our application-layer AES-256-GCM encryption. Even a full database dump yields only ciphertext.
- **Upstash Redis (session state):** Stores rolling hash chain state and short-lived session data. TLS-encrypted connections. Data is ephemeral by design.

- **AWS S3 (audit archive):** WORM-locked audit log archives. Separate AWS credentials from other infrastructure. Object Lock prevents deletion or modification for the retention period.

Third-Party Trust

We trust the serverless runtime environment not to inspect function memory during execution. We trust TLS to protect data in transit. We trust the cryptographic primitives (Ed25519, AES-256-GCM, SHA-256) to be computationally secure against current attack capabilities. These are standard assumptions shared by virtually all cloud-based security products.

Satellite Proxy — Self-Hosted Data Isolation

For organizations that require their API request bodies and responses not to flow through Keychains.dev infrastructure, we provide an open-source satellite proxy (MIT licensed) that can be deployed on the user's own infrastructure. When a satellite proxy is configured, the data flow changes as follows:

- **Without satellite proxy:** Agent ’ Keychains.dev proxy (credentials injected here) ’ Target API. Both the credential metadata and the API traffic flow through Keychains.dev servers.
- **With satellite proxy:** Agent ’ User's satellite proxy ’ Target API. The satellite proxy calls Keychains.dev only to resolve credentials (metadata exchange). The actual API request bodies, headers with injected credentials, and API responses never leave the user's infrastructure.

The satellite proxy is intentionally simple (~200 lines of TypeScript). It extracts placeholder names from the incoming request, sends only the target URL, HTTP method, and placeholder names to Keychains.dev via a single API call, receives resolved credential values, performs string replacement, and forwards the request to the target API. Keychains.dev remains the sole authority over which credentials are released — the proxy cannot override provider resolution, inflate scopes, or obtain refresh tokens (which are never returned to satellite proxies).

A built-in end-to-end test verifies the full placeholder resolution pipeline before the proxy is activated, ensuring correct operation before any real credentials are sent to the proxy. The proxy source code is available at github.com/interagentic/keychains.dev_proxy.

5. Known Limitations

No security system is perfect. We believe transparency about limitations is essential to building trust with the security teams evaluating our platform. The following are known constraints of the current architecture.

Within-Scope Manipulation via Prompt Injection

Keychains prevents credential theft, but it cannot prevent an agent from misusing credentials within its delegated scope. If a user grants an agent write access to a GitHub repository and the agent is subsequently manipulated via prompt injection, the agent could make destructive API calls that are technically within its authorized scope. Keychains' audit trail enables detection and forensic analysis, but the damage may already be done. Users should follow the principle of least privilege when delegating scopes.

Response Data Sensitivity

API responses returned through the proxy may contain sensitive data — personally identifiable information, financial records, private repository contents, etc. This data is visible to the agent and could be exfiltrated through prompt injection. Keychains protects credentials, not the data those credentials access. This is a fundamental limitation of any credential proxy model. Response filtering is on our roadmap but introduces significant complexity.

Approval Fatigue

Agents that require many different API scopes generate many consent prompts for the user. This can lead to "click-through" behavior where users approve requests without careful review. We are exploring grouped consent flows and policy-based auto-approval for trusted agents to mitigate this.

Shared Cloud Runtime

Keychains runs on Vercel's serverless infrastructure, which is a shared multi-tenant environment. While Vercel provides strong isolation guarantees (SOC 2 Type II certified), we do not control the underlying hypervisor or hardware. Organizations that prefer their API request bodies and responses not to flow through our infrastructure can deploy the open-source satellite proxy on their own Vercel account or any Node.js host (see Section 4: Satellite Proxy).

No Formal Security Audit

As of this writing, Keychains has not undergone a formal third-party security audit or penetration test. This is planned and will be prioritized as the platform matures. We actively welcome responsible disclosure at security@keychains.dev.

6. Comparison with Alternatives

Keychains occupies a specific niche: credential delegation for AI agents. Several existing tools address adjacent problems. We view most as complementary rather than competitive.

Alternative	Comparison
HashiCorp Vault + API Gateway	Vault excels at secret storage and access control for services and CI/CD pipelines. However, it was not designed for agentic workflows: there is no concept of delegated consent, machine identity for AI agents, or proxy-based credential injection. Keychains can work alongside Vault — using Vault as a backend secret store while providing the agent-facing delegation layer.
AWS IAM Roles	IAM provides robust access control for AWS services but is inherently scoped to the AWS ecosystem. It has no support for arbitrary third-party APIs (Stripe, GitHub, Slack, etc.), no user consent flows, and no mechanism for agent-to-agent delegation. Keychains is API-agnostic and designed for the heterogeneous tool landscape that AI agents navigate.
OAuth 2.0 DPoP	Demonstration of Proof of Possession (DPoP) binds OAuth tokens to a specific key, preventing token theft. This is a valuable primitive, but it operates at the OAuth layer — it requires the target API to support DPoP, and it does not provide a proxy, audit logging, delegation hierarchies, or agent identity. Keychains provides a full proxy layer that works with any API, regardless of its OAuth support.
Environment Variables / .env files	The most common approach today: passing API keys directly to agent environments. This is fundamentally insecure for agents that process untrusted input. Any prompt injection attack can exfiltrate environment variables. Keychains exists specifically to replace this pattern.

Key differentiator: Keychains is built for the specific trust model of AI agents — where the "client" cannot be trusted with secrets, where actions require user consent, and where delegation hierarchies (agent ’ sub-agent) need cryptographic enforcement. These are not features that existing secret management or IAM tools were designed to provide.

7. Incident Response

The following outlines the expected impact and response procedure for each component compromise scenario.

SSH Private Key Compromised

Detection: The rolling hash chain ratchet detects state divergence on the attacker's first authentication or the legitimate machine's next authentication — whichever comes first. The key is automatically invalidated.
Response: The machine is revoked. The user is notified. A new keypair must be generated and registered. All permission tokens issued to the compromised machine are invalidated.
Blast radius: Limited to actions taken between key theft and detection. Audit logs preserve a complete record for forensic analysis.

Permission Token Stolen

Detection: Anomalous usage patterns, requests from unexpected IPs, or user report.
Response: Tokens are short-lived (5–15 minute TTL by default). Instant revocation is available through the dashboard. The underlying credentials are not exposed — only the delegated access scope is at risk.
Blast radius: Constrained to the token's scope and TTL. No credential exposure.

JWT Signing Key Compromised

Detection: Detection relies on monitoring for anomalous token issuance or external report.
Response: Rotate the signing key via the JWKS endpoint. Remove the compromised key from the JWKS. All tokens signed with the old key become unverifiable and are rejected. New tokens are issued with the new key.
Blast radius: An attacker with the signing key could forge permission tokens for any user. This is a critical severity incident. Time to detection and rotation is the primary risk factor.

Credential Store Breach

Detection: Database monitoring, access logs, or anomalous query patterns.
Response: All credentials are encrypted with AES-256-GCM using per-user derived keys. A database dump yields only ciphertext. If key material is also compromised, affected users' credentials must be rotated with their respective providers.
Blast radius: If only ciphertext is obtained: none (encryption holds). If encryption keys are also compromised: full credential exposure for affected users.

Audit Log Tampering

Detection: Hash chain verification fails during routine integrity checks.
Response: Compare primary audit log against S3 WORM archive. The archive is immutable for the retention

period — any discrepancy indicates tampering in the primary store. Restore from the archive and investigate the breach vector.

Blast radius: Audit integrity is compromised in the primary store, but the S3 archive provides a trustworthy secondary record.

8. Compliance and Audit Status

Keychains.dev is an early-stage product. We are transparent about our current compliance posture and our plans.

SOC 2

Keychains has not yet completed a SOC 2 Type I or Type II examination. SOC 2 Type II certification is planned for 2027 as the platform matures and customer volume justifies the audit investment. Our infrastructure providers (Vercel, MongoDB Atlas, AWS) are individually SOC 2 Type II compliant.

Third-Party Penetration Testing

No formal penetration test has been conducted by an external firm. This is planned and will be disclosed upon completion. We conduct internal security reviews and automated vulnerability scanning as part of our development process.

GDPR and CCPA

Keychains processes data on behalf of users in accordance with our privacy policy (available at key-chains.dev/privacy). We collect minimal personal data — primarily authentication identifiers and usage metadata. Credentials are encrypted and accessible only to their owner. Users can request data export or deletion through standard channels. Data processing is described in detail in our privacy policy and terms of service.

Responsible Disclosure

We welcome security reports at security@keychains.dev. We do not currently operate a formal bug bounty program with monetary rewards, but we commit to acknowledging reports within 48 hours, providing regular updates, and crediting researchers (with consent) upon fix.

Document History

v1.1 — February 2026: Added satellite proxy (self-hosted data isolation) to trust boundaries and updated known limitations to reflect availability of user-deployable proxy.

v1.0 — February 2026: Initial publication. Covers architecture, STRIDE analysis, cryptographic primitives, trust boundaries, known limitations, alternative comparison, incident response, and compliance status.

For questions about the contents of this whitepaper, contact security@keychains.dev.

For the latest version of this document, visit keychains.dev/api/whitepaper.