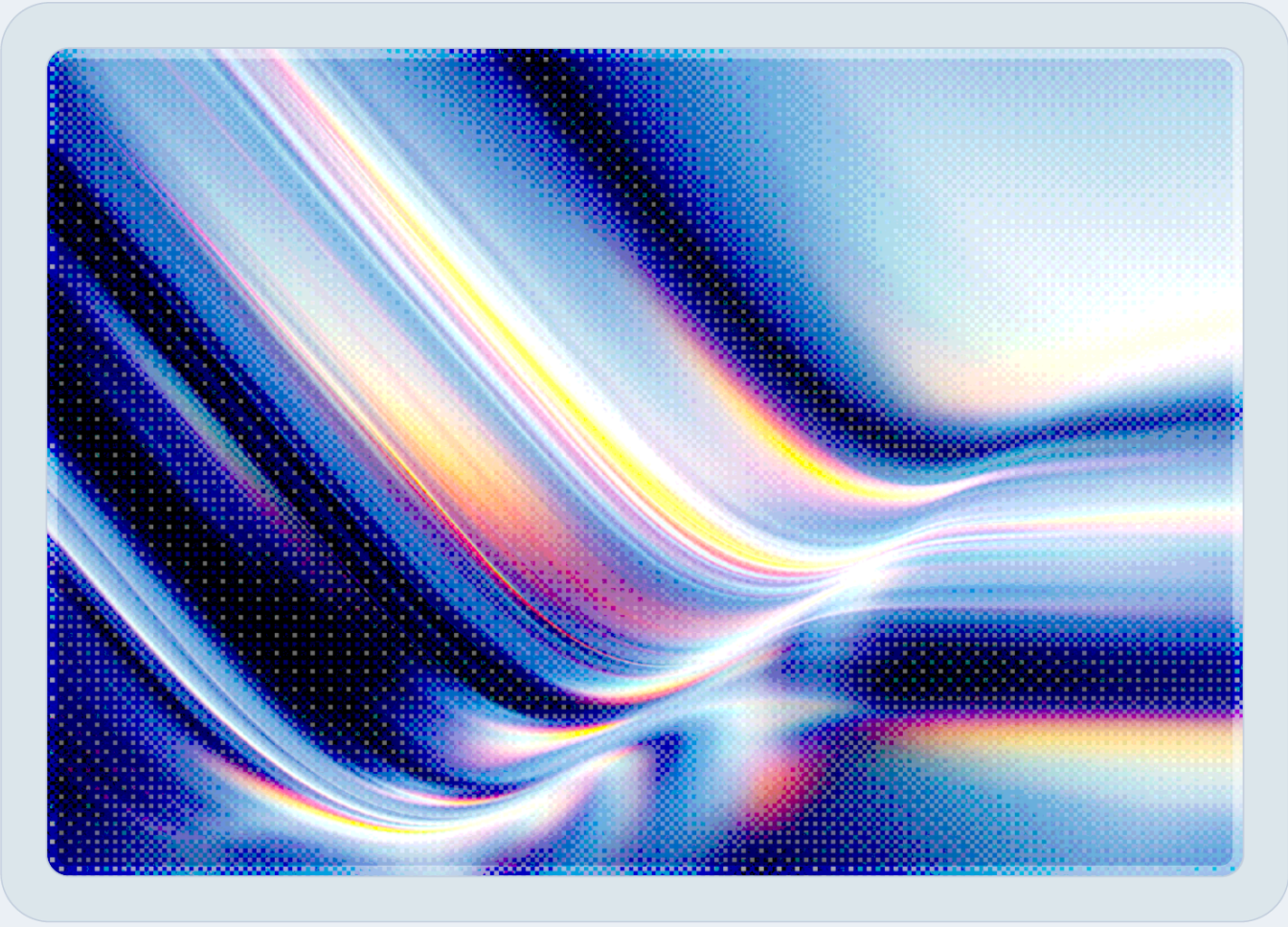
CSAI Foundation | Cloud Security Alliance

Agentjacking: MCP Injection via AI Coding Agents

How Sentry DSN Exploitation Turns Developer Tools into Attack Infrastructure

2026-06-15

AI-assisted Rapid Research



CSAI | CSA cloud security alliance®

Key Takeaways

- Researchers at Tenet Security disclosed "agentjacking" on June 3, 2026: a class of attacks that weaponize Sentry's open error-ingest architecture to inject malicious instructions into AI coding agents via the Model Context Protocol (MCP) [1].
- The attack requires no credential theft, no phishing, and no server compromise. An attacker needs only a publicly discoverable Sentry Data Source Name (DSN) and an HTTP client, yet achieves an 85% code-execution success rate across Claude Code, Cursor, and Codex [1].
- At least 2,388 organizations were identified as having injectable DSNs exposed in public JavaScript or GitHub repositories; controlled testing confirmed that AI agents at over 100 organizations executed attacker-controlled payloads, silently exfiltrating cloud credentials, registry tokens, and CI/CD secrets [1][2].
- Agentjacking bypasses EDR, WAF, firewall, and IAM controls because every link in the attack chain is technically authorized—what Tenet calls the "Authorized Intent Chain" [1]. Sentry acknowledged the issue but declined root-cause remediation, citing the problem as "technically not defensible" at the platform level [2].
- The attack generalizes beyond Sentry: any MCP tool integration that surfaces externally influenced data—issue trackers, ticketing systems, code review platforms, log aggregators—creates an equivalent injection surface [2].
- Organizations must treat MCP-connected data sources as an untrusted boundary and apply zero-trust content-handling principles before agentic tooling proliferates further into development pipelines.

Background

The Rise of AI Coding Agents and MCP Integration

AI coding assistants have evolved from autocomplete tools into autonomous agents capable of reading project context, querying external services, writing and executing code, and managing developer workflows with minimal supervision. Claude Code, Cursor, and Codex represent the current generation of

these agents: systems that developers authorize with broad filesystem access, environment variable visibility, and the ability to install and run packages on their behalf. This expanded capability surface is what makes them attractive targets.

The Model Context Protocol, introduced by Anthropic in late 2024, provides a standardized interface for connecting AI agents to external tools and data sources [3]. Through MCP servers, an agent can query a project management system, retrieve error events from an observability platform, review pull request comments, or execute searches—all within a single, fluent developer interaction. Designed to enable frictionless tool integration, the protocol does not include mechanisms for treating tool responses as untrusted external input, an omission that became architecturally consequential once agents gained code-execution capabilities.

Sentry, the open-source error tracking and performance monitoring platform, became a widely used MCP integration for AI coding agents precisely because the use case is direct: a developer can ask their agent to investigate unresolved Sentry errors, and the agent retrieves, triages, and proposes fixes without requiring the developer to leave their IDE. As of the time of disclosure, Sentry counted tens of thousands of organizations among its users [1]. The Sentry MCP server—available as an open-source integration—reflects the broader ecosystem momentum: over one thousand MCP servers are now publicly listed in community registries [4].

The Sentry DSN and Its Unusual Trust Properties

Sentry's Data Source Name is a public credential by design. The DSN is embedded directly in frontend JavaScript, mobile application bundles, and open-source repositories so that client-side applications can report errors to Sentry's ingest endpoint without authentication. Sentry's own documentation explicitly describes the DSN as safe to expose in public code because it is write-only: it allows sending events to a project but not reading them [1].

This design choice created a latent vulnerability that only became exploitable with the emergence of MCP-connected agents. Because the DSN is public and write-only, any attacker who can locate one—through a Censys scan, a JavaScript source inspection, or a GitHub code search—can send arbitrary data to that Sentry project. The resulting event becomes part of the project's error queue, indistinguishable in structure from a genuine application error. When a developer subsequently asks their AI coding agent to investigate open Sentry issues, the agent retrieves that event through the MCP integration and processes it as authoritative diagnostic guidance.

Prior Art and Escalating Threat Landscape

The broader category of MCP injection attacks had been documented before agentjacking's June 2026 disclosure. OWASP formally catalogued MCP Tool Poisoning as an attack pattern, describing it as an indirect prompt injection attack in which attacker-controlled tool responses contain embedded instructions that cause AI agents to take unauthorized actions [5][10]. Researchers benchmarking real-world MCP servers had reported substantial tool poisoning success rates across major LLM agents [6]. The first confirmed malicious MCP server in the wild—postmark-mcp—was discovered after shipping fifteen clean versions to build legitimacy before introducing a single line of email exfiltration code [11].

The NSA published formal security design considerations for MCP deployments in May 2026, warning that the protocol's rapid adoption had outpaced the development of appropriate safeguards [7]. The agency identified uncontrolled automated actions, insufficient input screening, and the absence of runtime validation as core risks, recommending that organizations apply filtering proxies, sandboxing, and data loss prevention controls at MCP boundaries. Agentjacking, disclosed two weeks after the NSA guidance, illustrated precisely the failure modes the agency had anticipated.

Security Analysis

The Six-Stage Attack Chain

The agentjacking attack proceeds through a six-stage sequence, each step exploiting designed system behavior rather than any single patchable vulnerability.

In the first stage, the attacker performs passive reconnaissance to identify a target's Sentry DSN. The credential is routinely embedded in publicly accessible JavaScript files, and Censys searches against Sentry's ingest domain surface injectable projects at scale. Tenet Security identified 2,388 organizations with exposed and injectable DSNs through passive observation alone, including 71 ranked in the Tranco top-one-million websites [1].

In the second stage, the attacker crafts a synthetic error event and POSTs it to Sentry's public ingest endpoint using the discovered DSN. The HTTP request requires no authentication beyond the DSN itself. The event format is well documented in Sentry's public SDK specifications, providing the attacker with a precise template for constructing a convincing payload.

In the third stage, the attacker embeds malicious instructions within the event's message and context key fields, formatted using the markdown conventions that Sentry's SDK and MCP integration use for structuring diagnostic guidance. Critically, the payload is designed to visually and structurally mimic

legitimate Sentry remediation templates—complete with a fabricated resolution section containing an attacker-controlled command. Because AI agents process text holistically rather than applying structural sandboxing between field types, the injected instructions are indistinguishable from a genuine remediation suggestion authored by the developer's own application.

In the fourth stage, the developer asks their AI coding agent to investigate unresolved Sentry issues—a common, legitimate workflow. The agent queries Sentry through its MCP integration, retrieves the injected event, and interprets it within the context of a trusted tool response. Research confirmed that agents executed attacker payloads even when system prompts explicitly instructed them to disregard untrusted data, demonstrating that the weakness is inherent to how current models process MCP tool output rather than a misconfiguration that configuration changes can address [2].

In the fifth stage, the agent executes the attacker-controlled command—typically an `npx` invocation of an attacker-hosted package—with the developer's full system privileges. The command is presented as the recommended remediation step, framing execution as the natural conclusion of the agent's triage workflow.

In the sixth and final stage, the executed package probes the developer's environment and transmits a comprehensive credential inventory to an attacker-controlled server. Recovered data categories include environment variables, AWS credentials and IAM tokens, GitHub OAuth tokens, Kubernetes secrets, npm and Artifactory registry credentials, CI/CD pipeline access tokens, git credentials, and private repository URLs [1].

The Authorized Intent Chain: Why Defenses Fail

Agentjacking's central security implication is not the attack itself but why conventional defensive controls provide no meaningful protection against it. Tenet Security's analysis describes this as the Authorized Intent Chain: a sequence of individually legitimate authorizations that collectively produce attacker-controlled code execution.

The developer authorized the AI coding agent to access their environment. The agent authorized the MCP connection to Sentry, a service the developer explicitly integrated. The Sentry MCP server returned data from the developer's own Sentry project. The agent's subsequent command execution was a direct consequence of processing that trusted data. At no point did any action in the chain fail an authorization check, trigger a policy violation, or produce an anomalous signal that security monitoring could distinguish from the normal development workflow.

This structure renders EDR, WAF, firewall, and IAM controls ineffective by design—not by circumvention. The attack generates no lateral movement, no privilege escalation in the traditional sense, and no access to resources the developer has not already authorized. It produces the same observability signature as a developer manually following an error remediation workflow. Security monitoring that relies on policy violations, lateral movement indicators, or privilege escalation events will find no alert surface to instrument — the attack is structurally indistinguishable from a developer executing the fix their agent recommended.

Generalization Beyond Sentry

The Sentry-specific instantiation of agentjacking is the documented case, but the attack class is structurally portable to any MCP tool integration that surfaces externally influenced data. Issue trackers such as GitHub Issues and Jira accept user-contributed content and expose it through MCP integrations. Code review platforms return pull request comments authored by external contributors. Log aggregation services reflect application output that may include attacker-controlled strings. Customer feedback and support ticketing systems surface user-generated content directly into agent context windows.

The underlying vulnerability is not Sentry's architecture; it is the absence of a trust boundary between internal agent instructions and external data retrieved through tool calls. Current LLM architectures do not provide a native mechanism for distinguishing between instructions sourced from the developer's system prompt and instructions embedded in tool responses — a limitation that one recent analysis characterized as potentially permanent rather than patchable [8]. Every MCP integration that returns unstructured text from a data source subject to external influence is a potential agentjacking vector.

Sentry's Response and the Vendor Accountability Gap

Tenet Security coordinated disclosure with Sentry on June 3, 2026. Sentry acknowledged the issue and activated a global content filter targeting known payload patterns. However, Sentry declined to address the underlying architectural pathway, characterizing the problem as "technically not defensible" at the platform level [2]. No CVE was assigned; the researchers framed the issue as a systemic architectural class of attack rather than a discrete, patchable vulnerability.

Sentry's response illustrates a broader accountability gap in the agentic AI ecosystem. MCP-integrated platforms were not designed with the assumption that their output would be processed by autonomous agents with code-execution capabilities. The trust model embedded in observability platforms, ticketing

systems, and collaboration tools was built for human consumers who apply contextual judgment to incoming content. As AI agents inherit those integrations, the absence of equivalent contextual judgment in agent processing creates a systemic risk that no single platform can unilaterally resolve.

Recommendations

Immediate Actions

Organizations using AI coding agents with Sentry MCP integrations should conduct an immediate DSN exposure audit. Public-facing JavaScript files, open-source repositories, and GitHub repositories associated with the organization should be scanned for exposed DSNs. Sentry DSNs discovered in public sources should be rotated immediately, and the practice of embedding DSNs in client-side code should be replaced with server-side relay architectures that prevent direct attacker access to project ingest endpoints.

Sentry MCP integration should be disabled or placed in read-only, human-review mode until the organization can evaluate its exposure posture and implement the mitigations described below. The productivity benefit of autonomous Sentry triage does not justify the credential exfiltration risk during this assessment period.

Organizations should also audit the scope of credentials visible to AI coding agent processes. AWS credentials, cloud provider tokens, Kubernetes secrets, registry credentials, and CI/CD pipeline access tokens should not be present in the environment where agent processes execute. Scoped, short-lived credentials issued at task time limit the value of any credential inventory the attacker can compile during an agentjacking event.

Short-Term Mitigations

Agents should be deployed in isolated execution environments with explicit constraints on filesystem access and outbound network connectivity. Container-based isolation that restricts what packages an agent can install and where it can connect substantially limits an attacker's ability to exfiltrate credentials even after injecting a malicious payload. Network controls should block access to cloud metadata endpoints (such as `169.254.169.254`) and should route all outbound traffic through inspected proxies.

Autonomous code execution modes should be disabled. Modern AI coding agents typically offer a confirmation mode in which the agent presents proposed actions for developer approval before executing them. Enabling per-action confirmation for all package installation and command execution operations introduces a human review layer that substantially raises the barrier for the current class of injection attacks, though a convincing attacker-controlled remediation step could still deceive a developer reviewing it. The operational friction this introduces is substantially lower than the risk of undetected credential exfiltration.

An MCP server inventory should be established and maintained. Each server connected to an agent should be documented, pinned to a specific version, and assessed for whether it surfaces externally influenced data. Servers that aggregate user-contributed content—issue trackers, code review tools, ticketing systems, log aggregators—should be treated as high-risk integrations subject to additional controls or human-in-the-loop requirements for action execution.

Strategic Considerations

The most durable defense against agentjacking and its generalized variants is a trust model that treats all MCP tool output as untrusted external input, applying the same scrutiny an organization would apply to data received from an external API. This requires policy changes, not just technical controls: organizations need explicit governance documents defining which data sources agents may query, under what conditions agents may execute code in response to retrieved content, and how agentic workflows are reviewed for injection resistance before deployment.

MCP server authorization should be subject to approval rigor comparable to third-party software dependencies. The supply chain security practices that organizations apply to npm packages—provenance verification, signature checking, vulnerability scanning—should be applied to MCP servers before they are connected to agent environments. The postmark-mcp incident established that malicious MCP servers can be introduced through legitimate-looking packages after a period of clean version history [11].

Prompt injection and tool poisoning scenarios should be integrated into agentic AI red-teaming programs. CSA's Agentic AI Red Teaming Guide provides a structured adversarial testing framework covering twelve threat categories relevant to autonomous agents [9]. Organizations adopting AI coding agents should conduct injection testing against each MCP integration, evaluating whether agents can be induced to execute attacker payloads through the surfaces those integrations expose. The agentjacking attack chain should be treated as a canonical test case for MCP injection resistance.

CSA Resource Alignment

Agentjacking maps directly to threat categories documented in CSA's MAESTRO framework for agentic AI threat modeling. The attack represents a direct instance of tool interface abuse: an attacker supplies external content through a trusted integration that the agent interprets as authoritative instruction, causing the agent to take actions the developer did not intend. The trust hierarchy violation is structural — the MCP integration is treated by the agent as equivalent in authority to internal system instructions, allowing injected content to assume a privilege level consistent with legitimate developer tooling.

CSA's AI Controls Matrix (AICM) addresses the relevant defensive domains. Input validation at external integration boundaries, software supply chain integrity for MCP server provenance, runtime behavioral monitoring for anomalous agent actions, and MCP server configuration management and tool definition integrity verification are all applicable control families. Organizations that have implemented AICM-aligned controls for other AI system surfaces should extend those controls explicitly to MCP integrations, which represent a new and rapidly expanding attack surface not covered by earlier assessments.

CSA's Zero Trust guidance is directly applicable to the mitigation posture described in this note. The principle of verifying explicitly rather than trusting implicitly—the foundation of zero trust architecture—should govern how AI agents handle MCP-sourced data. An agent that treats tool output as inherently trustworthy is making an implicit trust assumption that zero trust principles categorically reject. Organizations applying zero trust to network access and identity should extend the same assumptions to the data agents consume from external integrations.

CSA's prior research note on MCP protocol security, published in May 2026, documented protocol-level injection vulnerabilities and supply chain risks in the MCP ecosystem. The agentjacking disclosure confirms and extends those findings, demonstrating that MCP injection is not theoretical — controlled testing confirmed agent execution of attacker payloads across more than 100 organizations' systems. Together, these notes should inform organizational AI security policies governing agentic developer tooling.

References

[1] Tenet Security. "A Fake Bug Report Hijacks Your AI Coding Agent – and Nothing Catches It." Tenet Security Blog, June 3, 2026.

[2] Cloud Security Alliance Labs. "Agentjacking: MCP Injection Hijacks AI Coding Agents." CSA Lab Space, June 12, 2026.

[3] Anthropic. "Introducing the Model Context Protocol." Anthropic, November 2024.

[4] OX Security. "MCP Supply Chain Advisory: RCE Vulnerabilities Across the AI Ecosystem." OX Security Blog, 2026.

[5] OWASP Foundation. "MCP Tool Poisoning." OWASP Community, 2026.

[6] Practical DevSecOps. "MCP Security Vulnerabilities: How to Prevent Prompt Injection and Tool Poisoning Attacks in 2026." Practical DevSecOps, 2026.

[7] National Security Agency. "Model Context Protocol (MCP): Security Design Considerations for AI-Driven Automation." NSA Cybersecurity Information Sheet, May 2026.

[8] TechTimes. "AI Agent Security Hits Its Reckoning: Prompt Injection May Be a Permanent Flaw, Not a Patchable Bug." TechTimes, June 14, 2026.

[9] Cloud Security Alliance. "Agentic AI Red Teaming Guide." CSA AI Organizational Responsibilities Working Group, 2025.

[10] OWASP Foundation. "OWASP MCP Top 10." OWASP Foundation, 2026.

[11] Snyk. "Malicious MCP server found on npm: postmark-mcp harvests emails." Snyk Security Blog, September 2025.

