The MCP Trust Collapse

How the Model Context Protocol became the software supply chain's newest — and least-defended — attack surface.

Kymata Labs Research · An independent research institution studying how AI systems actually behave in production.

~16 min read

Tags · AI Security · Supply Chain · MCP · Prompt Injection

The Model Context Protocol has, in roughly eighteen months, become the default way to give AI agents access to tools and data — on one fateful assumption: that the descriptions a tool ships with are trustworthy. They are not. A tool description is text the model reads as instruction, supplied by a third party, mutable after approval, and never shown to the user in full, collapsing three distinct trust domains into a single instruction stream the model cannot tell apart. The result is not a bug to be patched but a class of attack — tool poisoning, rug pulls, and cross-server shadowing, now joined by remote-code-execution CVEs scoring 9.4–9.6. This paper argues MCP has reproduced, at agent speed and blast radius, the software-supply-chain problem npm took a decade to half-tame, and that the window to design integrity, provenance, and isolation into the protocol — rather than bolt them on afterward — is closing now.

The argument, in moves

1. **MCP's security rests on a single assumption: that a tool's description is trustworthy.** A tool ships with a natural-language description; the model reads it as instruction. But it is third-party text, mutable after you approve it, and in nearly every client never shown to you in full.
2. **That collapses three trust domains into one instruction stream.** User intent, the developer's prompt, and an untrusted tool description are concatenated into the same context window — and the model has no native means to tell them apart.
3. **The deskilling of the trust boundary is already weaponised.** Tool poisoning, rug pulls, cross-server shadowing, and data exfiltration are all publicly demonstrated; the most severe have been assigned CVEs and patched.
4. **It is not "just" prompt injection — it is prompt injection industrialised.** MCP takes a per-application hazard a careful developer might contain and supplies it with a registry, one-line installs, mutable artifacts, and no provenance.
5. **The fix does not require solving prompt injection.** Because the protocol is the vector, there is a great deal to do at the protocol, registry, and client layers — integrity, isolation, runtime defense — none of which waits on a model-level cure.

The one-liner: MCP's security rests on a single assumption — that a tool's description is trustworthy. It isn't. The result is not a bug to be patched. It is a class of attack.

What actually collapsed

Begin with the mechanism, because the whole argument rests on it. When an MCP client connects to a server, the server advertises its tools. Each tool ships with a description — ostensibly to help the model decide when to call it. That description is concatenated into the model's context window alongside your request and the developer's system prompt, and the model reasons over the whole, undifferentiated blob.

Here is the fatal property: the model reads the tool description as instruction, but you are shown — at most — a tool's name and a simplified argument summary. The full text the model acts on is invisible to the person who approved it. Invariant Labs' canonical demonstration is a tool that purports to add two numbers, with a hidden `<IMPORTANT>` block instructing the agent to read `~/.cursor/mcp.json` and `~/.ssh/id_rsa` and exfiltrate them through an unused `sidenote` parameter — then narrate an innocent lecture on arithmetic to hide the theft [1]. The approval dialog hides the exfiltrated key entirely.

It helps to name what has collapsed. A secure system keeps trust domains separated: data is not executed as code; content from the network is not obeyed as a command. MCP merges three such domains into one stream — user intent (trusted) and the developer's prompt (trusted) are glued to tool descriptions that are third-party, mutable, and model-visible (untrusted). This is the "confused deputy" problem, as Simon Willison frames it: the deputy — the agent — holds real authority (file access, API credentials, the ability to send mail or move money) and can be talked into misusing it by anyone who can get text into its context [2]. MCP's contribution is to *guarantee* that an attacker can: tool descriptions are, by specification, text in the context, and the ecosystem encourages you to install them from strangers.

The strongest evidence: from instructions to remote code execution

None of this is hypothetical. The most severe attacks have moved past manipulating what the agent *says* to executing code on the host outright. In July 2025, JFrog disclosed **CVE-2025-6514 (CVSS 9.6)** in `mcp-remote`, a widely used proxy: a malicious server need only return a crafted OAuth value, and when the client passes it to the operating system, it executes [3]. JFrog called it "the first time that full remote code execution is achieved in a real-world scenario on the client operating system when connecting to an untrusted remote MCP server."

It was not isolated, and the protocol's steward was not exempt. **CVE-2025-49596 (CVSS 9.4)** is a critical RCE in Anthropic's own MCP Inspector [5]. In January 2026, researchers at Cyata disclosed three further flaws in Anthropic's *official* Git MCP server [6]. When the reference implementations shipped by the protocol's own authors carry critical CVEs, "use a trusted server" stops being a sufficient answer.

These RCE flaws are a *different class* from tool poisoning, and the distinction matters. Tool poisoning exploits the *model* as the vulnerable interpreter. The RCE CVEs exploit the *client and server code* — a server-controlled field reaching an OS shell through ordinary software carelessness. What unites them is a single root cause: MCP invites the client to treat an untrusted, third-party server as trustworthy, and that one misplaced trust fails at both layers at once — the reasoning layer and the code layer.

The pattern repeats: a ranked attack surface

What makes the thesis hard to dismiss is that the failure recurs in form after form, each one publicly documented and the most severe assigned CVEs. They form a progression — from manipulating what the agent *says*, to controlling what it *does*, to executing code on the host outright. Severity rises top to bottom.

Attack	Mechanism	Evidence

Tool poisoning	Hidden instructions in a tool description the model obeys but the user never sees	Invariant Labs PoC · OWASP MCP03
Rug pull	Tool description silently mutated after the user approves it	Cross / Invariant · OWASP MCP03
Cross-server shadowing	A malicious server rewrites the agent's behaviour toward a trusted server	Invariant email-redirect PoC
Data exfiltration	Poisoned tool siphons credentials or chat history, hidden off-screen in the UI	WhatsApp history exfiltration
Remote code execution	Crafted server field reaches an OS shell or URL handler	CVE-2025-6514 · 9.6 / CVE-2025-49596 · 9.4
Command injection	Unsanitized input passed to a server-side shell call	CVE-2025-53107 (Git MCP)

Table 1 — The demonstrated MCP attack surface, 2025–26. Severity rises top to bottom; every row is publicly documented. By scale, the surface is large and growing: more than 10,000 public MCP servers and tens of millions of monthly SDK downloads, and one 2026 survey of over 7,000 public servers found 36.7% vulnerable to server-side request forgery alone [4]. In roughly a year from launch, MCP has earned an OWASP Top 10 [4], a CoSAI taxonomy [8], and NSA design guidance [9].

The lethal trifecta, and why MCP detonates it

Willison gave the danger its sharpest name. An agent becomes exploitable precisely when it combines three capabilities — and MCP is a machine for assembling all three [2]. **Private data:** the tools you install to be useful are the ones that reach your files, mail, repos, and keys. **Untrusted content:** "a tool that can access your email is a perfect source of untrusted content — an attacker can literally email your LLM and tell it what to do." **External communication:** the same toolset can issue the outbound request that ships the stolen data back out.

"LLMs are unable to reliably distinguish the importance of instructions based on where they came from," Willison writes. "Everything eventually gets glued together into a sequence of tokens and fed to the model" [2]. The protocol does not merely permit the lethal trifecta. It is the most efficient distribution mechanism for it yet built.

"But it's just prompt injection"

The most serious counterargument comes from Willison himself: these vulnerabilities are "not inherent to the MCP protocol" — they appear any time you give an LLM tools and untrusted input [2]. He is right about the root cause and wrong about the implication. Buffer overflows are not inherent to any package manager; they are a property of unsafe memory handling in C. Yet no one concludes that npm therefore bears no responsibility for what it distributes. A distribution channel's job is not to cure the underlying vulnerability class — it cannot — but to govern the conditions under which that class can be exploited at scale: who may publish, whether artifacts are signed, whether they can change after review, whether the consumer can see what they are running. MCP, today, governs none of these.

The sharpened thesis: MCP does not invent prompt injection. It *industrializes* it. It converts a per-application hazard a careful developer might contain into an ecosystem-scale supply chain — with distribution, low friction, post-approval mutability, and no provenance — for delivering that hazard to agents that hold the user's credentials. The protocol is not the disease; it is the vector that makes the disease epidemic. This reframing matters because it determines what to fix. If MCP's troubles were "just prompt injection," the

only honest response would be to wait for a model-level solution that has not arrived in two and a half years. But if the protocol is the *vector*, then there is a great deal to do at the protocol, registry, and client layers — none of which requires first solving prompt injection. That is the good news buried inside the bad.

The npm parallel: a decade already paid for

If MCP is a supply chain, the most useful thing we can do is read ahead in a book the industry has already written. The canonical incident is **event-stream** (2018): a popular, dormant package handed off to a new "maintainer" who shipped an obfuscated payload weeks later, after trust was established. Every structural feature has an exact MCP analogue today — a trusted-by-reputation artifact, a quiet handoff, a malicious update delivered *after* approval, a payload engineered to stay invisible. event-stream is a rug pull. The MCP ecosystem is currently reproducing 2018.

What npm did next is the part worth studying. Over years it accreted defenses: `npm audit`, mandatory 2FA for high-impact maintainers, lockfiles that pin exact hashes, and, by 2023, build *provenance* — cryptographically attested links between a package and the source that produced it. These are precisely the controls MCP lacks: there is no `mcp audit`, no required signing of tool manifests, no registry-level review.

Even npm's mature defenses are not sufficient. Analyzing a 2025 compromise, Palo Alto's Unit 42 found the registry "received 84 valid, signed, provenance-attested package publishes anyway" [7]. Provenance proves *where* an artifact came from; it does not prove the artifact is *benign*. Integrity must be paired with isolation — so that a signed-but-malicious tool still cannot reach the SSH key. On the controls that took npm a decade to build, MCP today has *none* in the protocol itself. It is, generously, at npm's 2015 moment. The difference is the payload: an npm package runs code inside whatever sandbox the host provides; a poisoned MCP tool issues *instructions* to a reasoning agent that already holds the user's credentials. Same supply chain, larger blast radius, and — because the attack can avoid the interaction log — worse observability.

What a trustworthy MCP would require

The answer is not exotic; it is the standard supply-chain security stack, adapted to an instruction-carrying artifact, in three layers. **Integrity:** pin and hash tool definitions so a change re-prompts, and sign manifests with attested provenance — the floor npm reached years ago. **Isolation:** treat tool descriptions as data, not commands; enforce least privilege so a tool that adds numbers cannot read `~/.ssh`; and control egress, which turns a silent theft into a blockable, logged event. **Runtime defense:** make the specification's security "SHOULD"s into "MUST"s, and run independent guardrails — with no illusions that they are the answer rather than a backstop.

The operator's minimum, today

Five controls an organization can adopt now, none of which require the protocol's cooperation. We hold them as mandatory in our own deployments, where the protocol leaves them optional.

1. Pin and diff tool descriptions; a silent change must re-prompt, not pass.
2. Allow-list and audit all outbound network egress.
3. Scope server capabilities to least privilege; deny filesystem and secret access by default.
4. Render the full tool description and require explicit confirmation for sensitive actions.
5. Run an independent guardrail that logs every tool invocation — and weight isolation over detection.

The last clause is the load-bearing one: weight isolation over detection. The defenders who have looked hardest at guardrails are the least confident in them. Integrity proves *where* a tool came from; only isolation guarantees that a signed-but-malicious tool still cannot reach the key.

Frequently asked questions

What is MCP tool poisoning?

Tool poisoning hides malicious instructions inside an MCP tool's natural-language description. The model reads that description as instruction and acts on it, while the user sees only the tool's name and a simplified argument summary. Invariant Labs demonstrated a tool that claimed to "add two numbers" but instructed the agent to read the user's SSH private key and exfiltrate it through an unused parameter — then narrate an innocent explanation to hide the theft.

What is the "lethal trifecta"?

Coined by Simon Willison, it is the combination of three capabilities that makes any AI agent exploitable: access to private data, exposure to untrusted content, and the ability to communicate externally. An agent with all three can be instructed by an attacker to read private data and send it out. MCP is the most efficient mechanism yet built for assembling that trifecta, because the tools users install to reach private data are frequently the same ones that expose the agent to untrusted content and let it issue the outbound request that exfiltrates it.

Is MCP fundamentally insecure, or is this "just" prompt injection?

Both, and the distinction is the whole argument. Prompt injection is the underlying, unsolved problem in language models, and no protocol change cures it. But MCP industrializes it: it takes a per-application hazard a careful developer might contain and supplies it with a registry, one-line installs, mutable artifacts, and no provenance. The protocol is not the disease; it is the vector that makes the disease epidemic — and unlike prompt injection, the supply-chain problem is tractable.

Has MCP actually been exploited, or is this theoretical?

It is not theoretical. CVE-2025-6514 (CVSS 9.6) achieves full remote code execution simply by connecting a client to a malicious server; CVE-2025-49596 (CVSS 9.4) is a critical RCE in Anthropic's own MCP Inspector; and in January 2026 researchers at Cyata disclosed three further CVEs in Anthropic's official Git MCP server. One 2026 survey of more than seven thousand public MCP servers found 36.7% vulnerable to server-side request forgery alone.

What can I do today to deploy MCP safely?

Five controls, none of which require the protocol's cooperation: (1) pin and diff tool descriptions so a silent change re-prompts; (2) allow-list and audit all outbound network egress; (3) scope server capabilities to least privilege and deny filesystem and secret access by default; (4) render the full tool description and require explicit confirmation for sensitive actions; (5) run an independent guardrail that logs every tool invocation. Weight isolation over detection — the defenders who have looked hardest at guardrails are the least confident in them.

How is MCP like the npm supply chain?

Almost exactly. npm spent a decade learning what happens when low-friction distribution meets untrusted publishers — the 2018 event-stream attack was a textbook rug pull. It answered with audit tooling, mandatory 2FA, lockfile pinning, and signed provenance. MCP has none of these in the protocol itself; it sits, generously, at npm's 2015 moment. The difference is the payload: an npm package runs code inside a sandbox, while a poisoned MCP tool issues instructions to an agent that already holds your credentials and the authority to act.

References

1 Invariant Labs (April 2025). "MCP Security Notification: Tool Poisoning Attacks." https://invariantlabs.ai/blog/mcp-security-notification-tool-poisoning-attacks

2 Willison, S. (June 2025). "The lethal trifecta for AI agents: private data, untrusted content, and external communication." https://simonwillison.net/2025/Jun/16/the-lethal-trifecta/

3 JFrog Security Research (July 2025). "CVE-2025-6514: Critical RCE in mcp-remote (CVSS 9.6)." https://jfrog.com/blog/2025-6514-critical-mcp-remote-rce-vulnerability/

4 OWASP Foundation. "OWASP Top 10 for Model Context Protocol." https://owasp.org/www-project-mcp-top-10/

5 Tenable Research (2025). "How Tenable Research Discovered a Critical RCE in Anthropic's MCP Inspector (CVE-2025-49596, CVSS 9.4)." https://www.tenable.com/blog/how-tenable-research-discovered-a-critical-remote-code-execution-vulnerability-on-anthropic

6 Cyata Security Research (January 2026). Three disclosed flaws in Anthropic's official Git MCP server (see also CVE-2025-53107). https://www.cyata.ai

7 Palo Alto Networks, Unit 42 (2025). Analysis of a software-supply-chain compromise: 84 valid, signed, provenance-attested malicious npm publishes. https://unit42.paloaltonetworks.com

8 Coalition for Secure AI (CoSAI). "Securing the AI Agent Revolution: A Practical Guide to MCP Security." https://www.coalitionforsecureai.org/securing-the-ai-agent-revolution-a-practical-guide-to-mcp-security

9 U.S. National Security Agency (May 2026). "Model Context Protocol: Security Design Considerations." https://www.nsa.gov/Portals/75/documents/Cybersecurity/CSI_MCP_SECURITY.pdf