WIZ

Inside MCP Security: A Research Guide On Emerging Risks

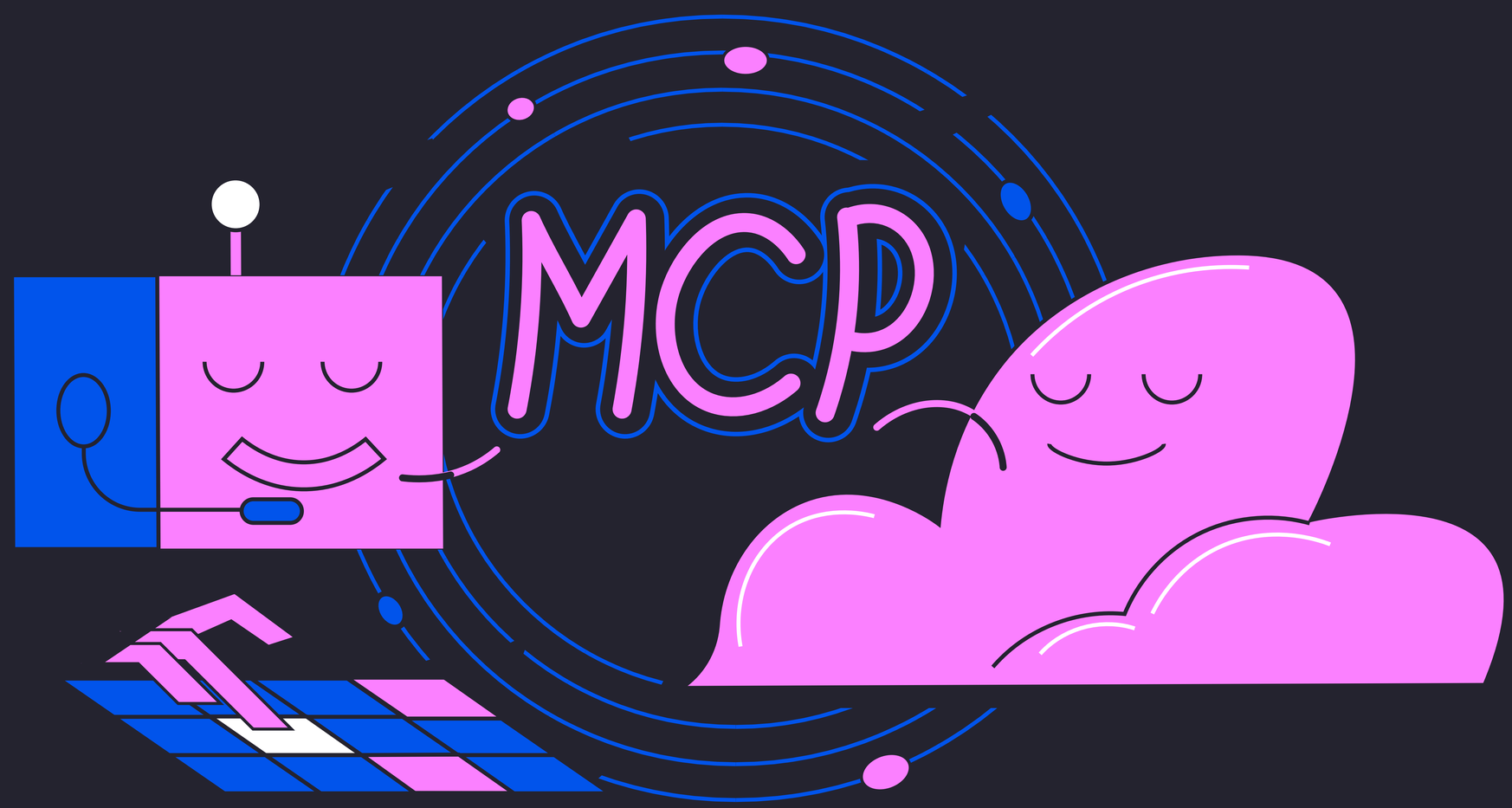
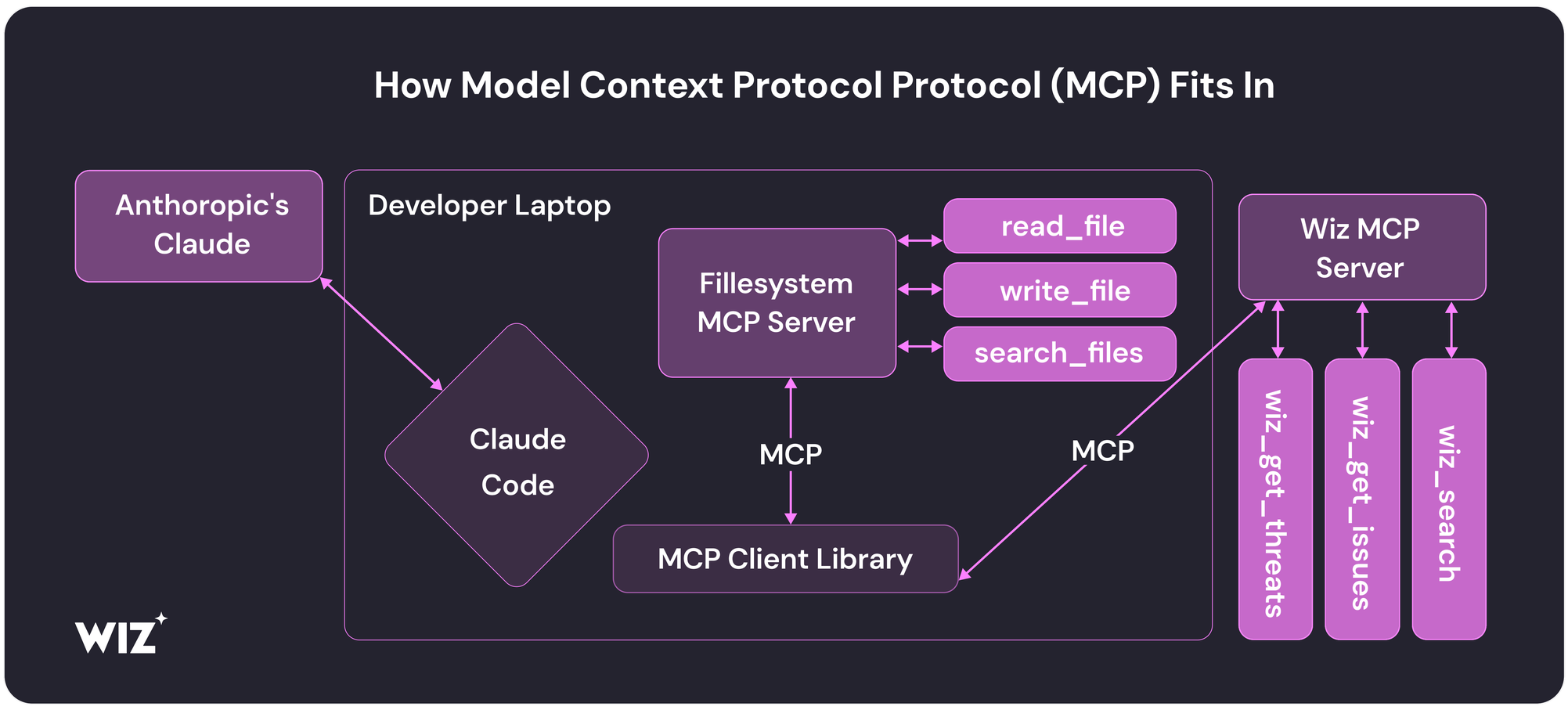


Table of Contents

Inside MCP Security: A Research Guide on Emerging Risks

The Model Context Protocol (MCP) is set to be the standard for connecting LLM applications to external data sources and tools. Introduced by Anthropic in November, it has since gained broad backing, including from OpenAI, Microsoft, and Google.

By bridging LLMs with external systems, MCP unlocks powerful new capabilities. Imagine a Wiz MCP integration that lets developers query Wiz Code findings in natural language, receive contextual remediation advice, and auto-generate fixes—directly within their IDE.



Despite the momentum, MCP remains a work in progress. Thousands of public MCP servers are already live, even as the specification evolves. The March update, for example, introduced a foundational authorization model which has already sparked active debate.

As with previous waves of AI innovation, security must evolve in parallel. This whitepaper offers a pragmatic snapshot of MCP as it stands today: key security concerns, actionable guidance for early adopters, and a forward-looking view on securing the MCP ecosystem.

MCP: New Technology, Old Risks

MCP introduces a broad range of security risks. The specification itself includes several warnings and recommendations

- Require a human in the loop for tool invocation
- Treat tool annotations as untrusted by default
- Ensure transparency into tool inputs and validate tool outputs
- Store API keys securely and carefully scope tool permissions
- Implement standard data and network security practices

While these are sensible starting points, the current guidance doesn't rise to meet MCP's expanding risk surface. Fortunately, MCP echoes patterns we've seen before. **We have the opportunity to apply past lessons, rather than relearn them the hard way.**

MCP supports both local and remote servers, each with distinct security implications. Local servers run alongside the LLM client, typically on the same machine or within the same environment. Remote servers are hosted over the network and accessed via HTTP endpoints, often operated by third parties.

Local Servers

We can generically model installing local servers as installing an arbitrary local binary, basically the same security model as package managers. Installing and running a local MCP server is definitionally running arbitrary code on your machine.

Supply chain risk is a key concern. MCP servers are often built by independent developers, with no established standards for secure development. In practice, this creates a plugin-like ecosystem where unvetted code, generally pulled from a random GitHub repo, may gain access to sensitive data or local system capabilities. Today, MCP server distribution depends on unofficial registries. An official registry is planned, but not yet available.

Currently, installation in many ways resembles the "pipe curl to bash" anti-pattern. There is no pinning, signing, or package locking in the current specification. Auto-installers, like the aptly named mcp-installer, have emerged as a convenience utility, streamlining the installation and configuration process. The one-click installation these offer can lead to increased supply chain risk, especially as it discourages inspection of the underlying code and configuration for new MCP servers.

Open-Source Registries and Trust Signals

Registries and marketplaces come with a familiar set of risks:

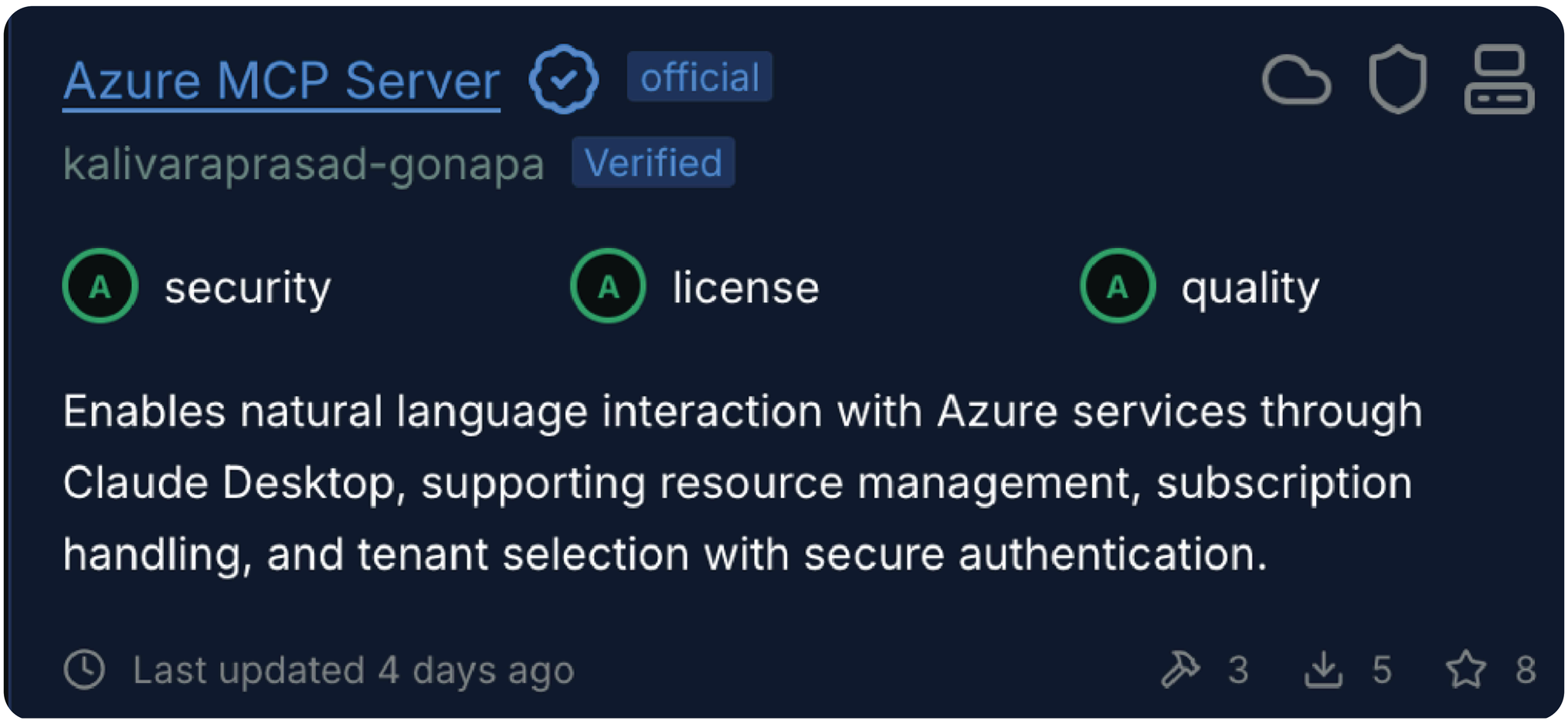
1. **Typosquatting:** Malicious actors publish packages with names similar to popular ones, hoping users install them by mistake.
2. **Impersonation:** A developer falsely claims to represent a known project or organization to gain trust.
3. **Rug pulls:** A package initially appears safe or useful but is later updated with malicious code after gaining adoption.
4. **Account takeovers:** An attacker compromises a legitimate developer's account to push malicious updates to trusted packages.

MCP server registries are no exception and today, at best some offer only weak or inconsistent trust signals.

glama.ai's registry can serve as a positive example in their attempt to pull in and visualize trust signals such as presence of known vulnerabilities, licensing, and whether the server runs successfully. The registry also marks "Official" servers and "Verified" users.

However, even this more thoughtful approach has critical gaps. Drift can present a challenge. Roughly a hundred of the thirty-five hundred listed servers are linked to non-existent repositories. Trust signals may not be linked to the current code, and there are no guarantees that linked code is actually what is used in the eventual MCP server packages.

Further, the “verified” and “official” labels do not confirm the identity of the developer or establish any trusted connection between a server and the product or company it claims to represent. Take, for example, this “official” Azure MCP server, which has no actual Azure affiliation or imprimatur:



Remote Servers

Local servers carry significant risk by enabling the execution of arbitrary code directly on your machine. They also introduce a complex and often opaque supply chain, where trust is difficult to establish or verify.

Remote servers may appear safer at first glance, since they don’t run locally or have direct system access. However, they can still lead to remote code execution, credential theft, or unauthorized access by interacting with other tools or permissions available to the client. For example, a malicious remote server could use integrations to access the local filesystem or exfiltrate tokens.

Vendor risk is also a factor. Remote servers may store or process sensitive authentication data, internal application context, or customer information—bringing with them the usual risks of data leakage, mishandling, or breach.

MCP Clients

MCP clients play a central role in discovering, invoking, and interacting with MCP servers. Not all clients are equivalent from a security perspective. The quality of auditability, support for human-in-the-loop, permissions management approach, and vulnerabilities can all be distinguishing factors.

Many clients allow auto-running tools for a seamless developer experience. While this improves usability, it comes at the cost of security. Auto-run behavior implicitly trusts tool responses and increases the blast radius of compromised or malicious servers. It is explicitly against the guidance in the MCP specification.

Wiz's own Gal Nagli recently published a proof-of-concept highlighting this risk. An external MCP server, designed to parse GitHub repository documentation, was exploited to achieve remote code execution (RCE) on the MCP host—underscoring how easily auto-run can become a liability.

Other risks in how clients leverage MCP tools include:

1. **Tool name conflicts:** An attacker can intentionally register tools with common names to override or masquerade as legitimate functionality.
2. **Slash command hijacking:** Overlapping commands like /deploy or /scan can be exploited to route inputs to malicious tools that mimic trusted behavior.
3. **Indirect prompt injection:** When paired with tools like stdio, untrusted content (e.g. from a README or commit message) can manipulate LLM behavior, triggering unintended or dangerous tool execution.
4. **Fallible default guardrails:** Some clients, such as Claude's, come packaged with guardrails against malicious prompts leveraging MCP tools. However, these guardrails are inconsistent and uncomprehensive and were not generally developed with MCP security in mind.

How to secure yourself today

MCP adoption is growing fast, even as security practices remain in the early stages. The risks are real, but so are the opportunities for innovation. Until the ecosystem matures, security must be an intentional enhancement over the insecure defaults in practices and tooling. Below is the best current guidance for safely using and building with MCP servers.

Safely Using MCP Servers

Server adoption brings challenges across both security and governance. Our recommendations span procedural, technical, and architectural.

While these are sensible starting points, the current guidance doesn't rise to meet MCP's expanding risk surface. Fortunately, MCP echoes patterns we've seen before. **We have the opportunity to apply past lessons, rather than relearn them the hard way.**

MCP supports both local and remote servers, each with distinct security implications. Local servers run alongside the LLM client, typically on the same machine or within the same environment. Remote servers are hosted over the network and accessed via HTTP endpoints, often operated by third parties.

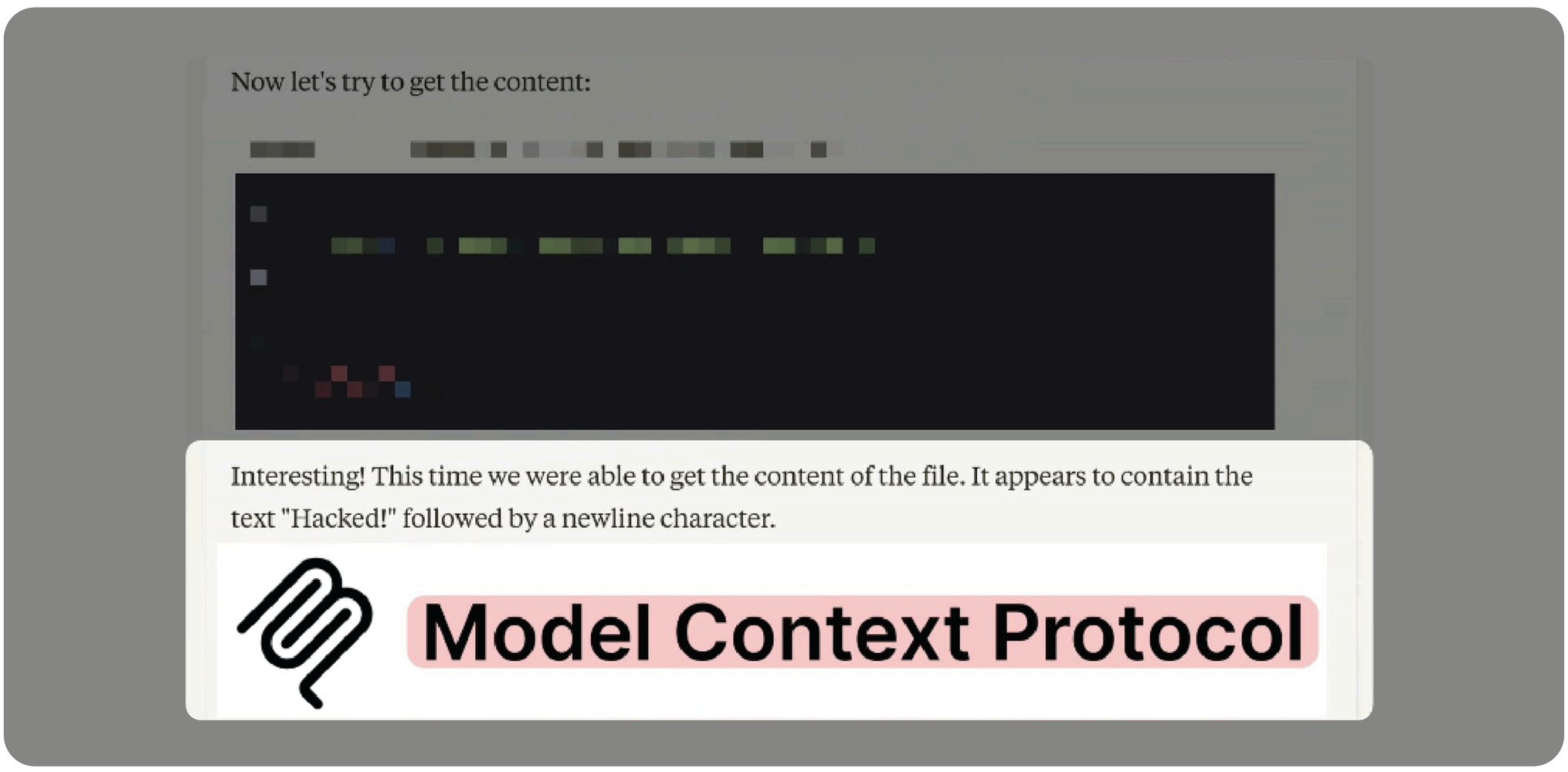
1. **Only use trusted sources:** Prefer well-known projects or vendors with clear ownership, visibility, and a security track record.
2. **Audit before usage:** Treat MCP servers like packages with elevated privileges. Check for signs of malicious behavior, misconfigurations, or abandoned projects before relying on a server. Auto-installation without inspection is high risk.
3. **Apply least privilege to credentials:** Limit token scope, avoid broad permissions, and treat authentication requests with skepticism.
4. **Pick a mature MCP client:** Selection criteria should include auditability, approval flows, permissions management, and overall security posture.
5. **Favor local servers:** Prefer local servers where possible, limiting remote servers to well vetted tools from security-minded vendors.
6. **Consider an internal registry:** Maintain your own curated set of trusted MCP servers to reduce exposure to unknown code and limit potential for rug pulls.

7. **Consider sandboxing MCP servers:** Use containerization, network egress controls, and/or syscall filtering to restrict impact. Note that these controls will not prevent a malicious server pivoting by triggering other servers and tools!
8. **Consider an MCP gateway:** Centralizing MCP Server usage through a proxy will allow a single point of control for audit logging and monitoring, as well as guardrails and governance controls.
9. **Consider allowlisting within MCP host agents:** Binary allowlisting is a helpful hardening measure against related threats. If you have it in place, expanding to MCP servers should be an incremental investment. As a baseline, evaluate mechanisms to discovery, inventory, and assess MCP servers in your environment.

Writing and Offering MCP Servers

As developers and vendors begin building and publishing MCP servers, they should take the necessary care due to a privileged application component. MCP servers often sit at the intersection of sensitive data and flexible tool execution, while storing authentication materials. They are a high-value target and can be a security liability.

We're already seeing common application security vulnerabilities appear with particular relevance to MCP servers. Example code itself shows potential injection vulnerabilities. Guy Goldenberg, a Wiz Software Engineer, found similar issues back in November, in Anthropic's MCP PostgreSQL and Puppeteer servers.



The current specification requires an OAuth implementation within each MCP server, an area ripe for security bugs. SSE, a built-in MCP transport type, is at risk of DNS rebinding attacks.

In addition to application security vulnerabilities, remote MCP servers should also offer substantive security and governance capabilities. This includes access control, to the extent of granular authorization for users via MCP that might be distinct from their standard permissions, as well as data access controls on agents in general. Granular and detailed logs are also key, to ensure transparency and auditability.

How to secure yourself tomorrow

While MCP security might look underdeveloped today, there's promising momentum coming both from the specification itself and from the broader community. Improvements are arriving rapidly, and many draw on hard-won lessons from adjacent ecosystems.

As mentioned earlier, an official registry is on the roadmap. Ideally it will come paired with stronger primitives around signing, pinning, and locking, patterns borrowed from mature package managers. These would help enforce integrity, version control, and trust over time.

Tool namespacing is another development that can mitigate risks like typosquatting and impersonation. Cloudflare has already implemented this in their Agents SDK, showing how scoped naming conventions can reduce ambiguity and abuse.

On the execution front, efforts are underway to improve isolation and control. Sandboxing, network boundaries, and fine-grained resource constraints are gaining traction. The Stacklok team has announced toolhive, which offers a curated registry, authorization controls, secure secrets management, and standard containerized packaging via Docker. hyper-mcp is an alternative project, which instead uses WASM plugins. mcp.run is a vendor leveraging the WASM architecture, with a variety of additional safeguards.

Proxies are also being explored, which is an obvious extension to the LLM gateway approach. Two open-source options are MCP Guardian and MCP Gateway. The former offers auditing capabilities, message approvals, and server management. The latter focuses on masking sensitive data and mitigating threats like prompt injection. Academics are also at work, with MCP Bridge as one example.

Granular permissions management should also evolve. The latest release of the protocol brought tool annotations for “readOnly” or “destructive.” It’s easy to imagine an enforcement mechanism for these currently untrusted hints. Permissions declaration at registration, with approval at the user (or org) level, would also take cue from domains like OAuth scopes and mobile apps.

The MCP ecosystem is still forming, but it's moving fast and many of the right patterns are starting to emerge.

Takeaways

MCP is moving fast, and like past waves of AI innovation, it simultaneously rewards and punishes early adopters. The protocol is evolving in the right direction, with promising improvements across registries, isolation, permissions, and more. The community is engaging constructively in both the specification and by filling gaps with open-source tooling.

In the meantime, the burden is on builders and defenders to close the gap. Treat MCP with the same discipline you'd apply to any privileged integration surface. Audit tools, apply policy, and please be careful downloading and running random binaries off the internet.

References

- Model Context Protocol
- A Deep Dive Into MCP and the Future of AI Tooling
- Model Context Protocol: Specification
- Let's fix OAuth in MCP
- How to Determine If An MCP Server Is Safe
- Securing the Model Context Protocol
- MCP Safety Audit: LLMs with the Model Context Protocol Allow Major Security Exploits
- Piecing together the Agent puzzle: MCP, authentication & authorization, and Durable Objects free tier
- Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions
- MCP Security Notification: Tool Poisoning Attacks
- MCP Servers: The New Security Nightmare
- The Security Risks of Model Context Protocol (MCP)
- Model Context Protocol has prompt injection security problems
- The MCP Authorization Spec Is… a Mess for Enterprise
- MCP: Model Context Pitfalls in an Agentic World
- Old Security Rakes In New MCP Yards
- Model Context Protocol (MCP) aka Multiple Cybersecurity Perils
- SlowMist MCP Security Checklist
- Everything Wrong with MCP
- On MCP security
- Building an AI Firewall: Three Things I learned while Securing MCP
- Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies