ChatGPT

Security Implications of the Model Context Protocol (MCP) and the Need for Robust Infrastructure

Introduction

The **Model Context Protocol (MCP)** has emerged as a powerful open standard that enables Large Language Models (LLMs) to interface with external tools and data sources in a uniform way [1]. By acting as a kind of "USB-C for AI" – a universal interface – MCP allows applications to dynamically **discover and invoke** capabilities (functions, APIs, services) using LLM reasoning [1]. This innovation addresses a key challenge in AI integration: how to let an LLM *talk to* other systems (file systems, code repositories, databases, web services, etc.) in a flexible yet standardized manner. In the same way that past technologies like WSDL (for SOAP) and OpenAPI/Swagger brought standardized schemas to web APIs, MCP introduces a schema and handshake for LLM-to-tool communications. Major AI platforms have rapidly embraced MCP – for example, Hugging Face's frameworks now expose MCP endpoints to let assistants invoke model functions [2] – and new MCP-based integrations are announced almost daily.

While MCP provides a **solid foundation for LLM tool integration**, its success also highlights a critical issue: **the security is only as strong as the surrounding infrastructure**. The protocol itself solves a connectivity problem, but it does not inherently solve the *safety* problem of connecting an advanced, non-deterministic AI agent to arbitrary powerful tools. As one security expert noted, "the problem is not MCP – the problem is the infrastructure [that] MCP is built on top [of]" (Cruz, voice memo). In other words, MCP's very strength – seamlessly bridging LLMs with real-world actions – can become a massive vulnerability if traditional security controls (identity, authentication, authorization, monitoring) are not up to par.

In this paper, we examine the **security implications of MCP**. We draw parallels to earlier computing paradigms, analyze known vulnerabilities and emerging attack vectors, and survey recent research and proposals to harden MCP-based systems. Ultimately, we argue that **MCP's arrival forces a rethinking of application security architecture**: to safely harness AI agents, we must overhaul identity and permission models, adopt fine-grained trust boundaries, and integrate continuous threat modeling into the development of AI-driven applications.

The Promise of MCP and Parallels to Past Integrations

MCP was created to fill a gap: previously there was no easy, standardized way for an LLM to invoke external functionality or query live data. Each integration was ad-hoc. With MCP, any tool that implements the protocol can advertise its capabilities (usually via a manifest) and accept text-based invocations from an AI agent. This has been likened to an **evolutionary step in software integration** – similar to how moving from bespoke integrations to RESTful APIs and Swagger documentation catalyzed a new ecosystem. In strategic terms, MCP represents the commoditization of LLM-to-tool interfaces, which in turn enables higher-level constructs and rapid innovation (a concept noted in Wardley Maps' **Innovate-Leverage-Commoditize** cycle). We are currently in the "Genesis and Custom

Build" phase of agents using MCP, with a flurry of innovation around new tools and agent behaviors built on this standard.

One especially intriguing aspect is how MCP fulfills a vision of *dynamic interoperability* that was imagined decades ago. In Brett Victor's **"Future of Programming"** parody talk, a 1970s engineer speculates that future systems would not rely on rigid APIs, but rather could *negotiate* to understand each other's capabilities. Traditional APIs did not reach that ideal – they are static contracts, and once an API is published and used, it becomes brittle (changes risk breaking clients). This rigidity led to fragile architectures (e.g. microservices where any change in one service's API reverberates across many consumers). By contrast, MCP-powered agents can introspect a tool's schema and flexibly figure out how to use it on the fly, guided by natural language descriptions. In effect, **MCP moves integration closer to a conversational query**: *"I (the agent) need to do X – what can you do and how can we accomplish this?"*. This dynamic flexibility is a major leap forward in capability.

However, with this power comes new risk. In prior generations of software, the boundaries between systems were static and well-defined (even if that led to inertia). With MCP, the boundaries become fluid and **driven by AI decisions at runtime**. An LLM agent can potentially rewire workflows or chain tools in ways the original developers might not have anticipated. If the proper **guardrails and trust mechanisms** aren't in place, the system may do *exactly what you asked* – and more. As we examine next, **the line between data and instructions has effectively disappeared** [3], meaning inputs from one system can be interpreted as executable plans by an AI, often with surprising (and dangerous) results.

New Attack Surface: When AI Agents Meet Unfettered Privileges

By extending an LLM's reach into external tools, MCP effectively gives the model **actions in the world** – from reading files, to controlling software, to calling APIs. This creates an unprecedented attack surface. A malicious prompt or input to the LLM is no longer just a quirky conversation; it could trigger **real operations with real consequences**. As security researchers recently put it, *"Data is now code"* [4]. Any data the agent processes (e.g. a ticket in an issue tracker, an email, a document) might contain hidden instructions that cause the agent to misbehave. If that agent has high-level privileges via MCP, the stakes are high.

A concrete example is **the GitHub MCP exploit** discovered in May 2025 [5]. In that scenario, a developer had connected an AI coding assistant (Claude) to GitHub via an MCP integration (14k-star open-source project) [5]. The developer had two repositories: one public and one private. An attacker simply created a maliciously crafted issue on the **public repo** – essentially a prompt injection hiding in plain text. When the developer later asked the AI agent to "summarize open issues" across projects, the agent dutifully read the attacker's issue. The prompt injection *coerced the agent into exfiltrating data*: it made the agent draft a pull request that took content from the private repo and posted it publicly [6] [3]. In effect, the AI was tricked into leaking proprietary code. Notably, this **was not a bug in GitHub's MCP server** logic – rather, it was a *fundamental design flaw* of the agent having too-broad access with too little oversight [7]. The incident highlights how **prompt-level attacks (injections)** can combine with broad tool privileges to produce serious breaches.

Another class of threat comes from **malicious or compromised tools** in the MCP ecosystem. Because MCP makes it trivially easy to plug in a new tool (by URL or manifest), one must assume attackers will create **trojan tools or "tool squatting" attacks**. For instance, an agent might be instructed to connect to what appears to be a useful public MCP service, but that endpoint could be designed to steal data or issue dangerous commands. Researchers have indeed identified *"Tool Poisoning"* and *"Rug Pull"* attacks in the MCP specification [8]. *Tool poisoning* refers to a scenario where a tool's definition or responses

are maliciously crafted to mislead or exploit the agent. A *rug pull* is when a tool that may have behaved benignly (or claimed a certain interface) later changes behavior or version to perform malicious actions, after users have come to trust it. The standard MCP lacks robust authentication or integrity checks for tools, making such attacks easier. An agent might unknowingly load a harmful tool that, say, claims to convert images but in fact also siphons user data when called. This risk is amplified by the **"mix-and-match" nature of MCP**, where an enterprise agent might connect to both internal (trusted) and external (third-party) MCP endpoints. A compromised external tool can become a bridge into internal systems if the agent isn't carefully restricted.

Underlying all these attack vectors is a **common root issue**: **unrestrained privileges and implicit trust**. Today's identity and authorization frameworks are not granular or dynamic enough to cope with AI agents. If you give an agent access to your file system via MCP, it likely has *access to everything* your user account can see – which in a modern environment includes a trove of sensitive data, API keys, and services. Unlike a human user, the LLM might expose that data unintentionally or be tricked into doing so. And importantly, we cannot fully "lock down" the LLM's logic or outputs through alignment alone. Even the most aligned models (e.g. Anthropic's Claude v4, known for safety) can be manipulated in certain contexts. In a recent exploit, Claude 4 (a state-of-the-art aligned model) was still *"susceptible to manipulation through relatively simplistic prompt injections"*, underscoring that **model-level guardrails cannot anticipate every context or malicious input** [9]. The **security of agent systems is fundamentally contextual and environment-dependent** [10], meaning that *system-level defenses* must complement the AI model's built-in safety.

In summary, **MCP has effectively collapsed the distance between an attacker's input and the system's powerful actions**. The attack surface is no longer just the model's prompt; it extends to any tool the model can call and any data those tools handle. This demands a reevaluation of trust: Which actions should we allow an AI agent to take autonomously? How do we verify and constrain those actions? And how do we prevent untrusted inputs from piggybacking on trusted connections? To answer these, we need to fix the *infrastructure* around MCP.

Why Current Security Infrastructure Falls Short

Modern computing platforms do offer identity and permission systems – cloud IAM roles, OAuth scopes, UNIX permissions, etc. However, these were built for relatively static scenarios (users or services with well-defined roles performing known tasks). **AI agents break those assumptions**. An agent's behavior is dynamic, emergent, and context-driven; it does not neatly fit into a static role. Moreover, as discussed, an agent effectively inherits the union of all permissions it's been granted across MCP tools. Several specific shortcomings in today's security infrastructure become apparent:

- **Coarse Authorization and Over-Privilege:** Both humans and applications today tend to be over-provisioned with access rights. In cloud environments, it's well documented that IAM roles often accumulate far more privileges than necessary, simply because it's difficult to predict and least-privilege policies are hard to write. This problem is magnified for MCP. If an agent is given a token that grants read/write to an entire GitHub account (a common practice with current LLM coding assistants), then *any* repo in that account – public or private – is in scope. Indeed, popular LLM integrations often ask for excessive scopes (e.g. full repository control) because fine-grained alternatives (like read-only or PR-only access) are either not available or not used. A key example: OpenAI's and Anthropic's coding copilots have required broad GitHub access to function, where a safer design would be to allow only specific operations like creating pull requests [11] [12]. The infrastructure (GitHub's OAuth in this case) did not easily support issuing a narrowly-scoped

token just for the assistant's needs – it was all or nothing. As a result, **agents are routinely being over-privileged**, violating the principle of least privilege.

- **Lack of Agent-Specific Identities:** Many systems do not yet support the idea of a "bot" or agent identity separate from a human user. In practice, this forces developers to share their own identity with the LLM agent. For example, one might plug an LLM into a corporate system by giving it an API key or personal token. This is problematic – any action the agent takes is attributed to the user, and there is no native way to apply a distinct permission set to the agent. Ideally, we would create **ephemeral service accounts or fine-tuned roles** for AI agents (e.g. a limited GitHub bot user with access only to one repo). But as practitioners have discovered, platforms like GitHub have historically made it difficult to create such secondary accounts or to subdivide permissions cleanly for this purpose (outside of full "GitHub Apps" which are still quite coarse). The **inability to easily spawn constrained identities for AI usage** leads developers to hand over broad credentials, which is a dangerous workaround. If the agent misuses those credentials, there is no built-in containment.

- **Authentication and Trust Gaps in MCP:** By default, the MCP protocol itself **does not mandate authentication or encryption** for tool endpoints [11]. It's simply a protocol – you can run an MCP server on localhost or expose one on the web, and unless you add your own auth, any agent can call it. This openness was by design (to encourage rapid adoption and ease of use), but it clashes with enterprise security needs. An organization cannot safely allow arbitrary tools that don't authenticate clients or that don't verify who they're talking to. The lack of a built-in authentication handshake means developers must bolt on their own (e.g. network whitelisting, API keys, or OAuth). Until standardized approaches (like an `.well-known/mcp-auth` mechanism to declare auth requirements) are adopted [13], there's a high risk of misconfiguration. Unauthenticated MCP endpoints could allow **anyone to invoke internal tools** if an endpoint is exposed – an obvious security hole. Similarly, without authentication, an AI agent might connect to a rogue tool impersonating a legitimate one (since the agent cannot verify the tool's identity or integrity by default).

- **Uniform Trust for All Tools:** In the current MCP design, **every tool integration is treated equally by the agent** – there is no native concept of "this tool is high-risk" vs "this tool is benign" [11]. For example, a tool that just formats text and a tool that deletes files are both just tools from the agent's perspective. This uniform trust model is too naive. In traditional systems, we classify and segregate actions by sensitivity (think of Linux capabilities, or the difference between reading a file and installing a driver). Agents lack that nuanced view. The result is that an agent may execute a highly sensitive operation as readily as a harmless one. Without an internal risk model, an LLM agent won't, on its own, pause and double-check a dangerous action – unless explicitly programmed to. The **MCP manifest schema today does not include a "danger level" or required confirmation step** for tools [14]. It probably should. This is an area where security policy must augment the agent's decision-making (e.g. require user confirmation or additional checks before certain high-impact tools are run).

- **Limited Observability and Control:** Traditional AppSec tools like static analysis, fuzzing, and even runtime monitoring are challenged by the fluid, interpretive nature of AI agents. The behavior is not explicitly coded in one place; it emerges from model prompts and the interplay of tool responses. This makes it hard for security teams to **reason about what could happen** or to trace cause and effect. For example, a static analysis tool could normally detect if code might call a dangerous function, but in an AI agent, the "code" is the model's chain-of-thought, which is opaque. There is a nascent effort to apply dataflow tracking to AI agents – for instance, Invariant Labs built a security analyzer to detect *"toxic agent flows"* that could lead to data leaks [15]. Using

such tools, one can catch scenarios like "data from a private repo flows into a public output" as a policy violation. But these tools are not yet mainstream. The net effect is that **developers often lack feedback or alerts** when they wire up an agent with MCP. An insecure configuration (like connecting to an untrusted external tool, or giving an agent a powerful token) might not trigger any warning until an incident occurs. This is analogous to the early days of web services, when developers might inadvertently expose a dangerous API (e.g. an endpoint that executes SQL given a query string) and no automated tool flagged it. In fact, MCP's trajectory has been compared to "the early days of insecure-by-design APIs…anybody remember those WSDL endpoints?" [16] – a Wild West period where functionality took precedence over security and it took years for security practices to catch up.

In summary, our current security infrastructure was not designed for autonomous, general-purpose agents. It lacks the *finesse and dynamic responsiveness* needed to contain AI-driven actions. The concept of **least privilege** – granting the minimum permissions necessary – is difficult to enforce when we ourselves don't know what the AI will attempt to do. And the concept of **zero trust** – verify everything, assume nothing – must be reimagined at the AI-tool boundary (each tool invocation might need verification of both the tool's trustworthiness and the agent's intent). These gaps underscore an urgent need to evolve our security architecture alongside MCP's adoption.

Toward a Secure MCP Ecosystem: Best Practices and Research Directions

To safely realize MCP's potential, we need a multi-layered approach. Here we outline emerging best practices and research directions aimed at **fortifying MCP integrations**:

1. Fine-Grained Permissions & Sandboxing: A consensus is building that agents should operate under *tightly scoped privileges*, and that not all tools are equal. Each MCP tool should declare the nature of actions it performs and the resources it needs. For instance, a tool could specify permissions like **"read-only"**, **"write"**, **"exec"**, or **"dangerous"** in its manifest [14]. The agent runtime can then enforce rules: e.g., require explicit user consent before a dangerous action, or disallow combining certain tools in one session. Researchers from Invariant demonstrated a policy that **prevents cross-domain data leaks** – for example, an agent could be restricted to interacting with only one Git repository per session [17] [18]. In code, their guardrail policy checks that if an agent has used a tool on `RepoA`, it cannot immediately use a tool on `RepoB` [17], stopping the scenario of copying private RepoA's data into public RepoB. This is an illustration of enforcing *contextual least privilege*: even if the agent has theoretical access to two resources, the system can dynamically restrict combining them. Additionally, isolating tool execution can limit blast radius. Sensitive actions should be run in sandboxed environments (e.g. a Docker container or a restricted subprocess) [19]. If an agent's tool tries something fishy (deleting files, making external calls), the sandbox can intercept or constrain it. In practice, implementing such sandboxes for AI tools might involve OS-level controls (seccomp profiles, chroot jails) or language-level controls (running code with limited APIs). At minimum, **dangerous file system or network operations should be off-limits by default** and only enabled case-by-case with user approval.

2. Strong Authentication and Tool Verification: To counter the risk of malicious or rogue tools, the community is proposing extensions to MCP for authenticating tool identities. One such proposal is the **Enhanced Tool Definition Interface (ETDI)**, which adds *cryptographic identity verification* and *immutable versioned tool definitions* [20]. In essence, tools would have signed manifests (similar to how apps have signed certificates), and agents would verify those signatures against trusted authorities or fingerprints. This could prevent "tool spoofing" – an attacker shouldn't be able to trick an agent into loading a fake "calculator" tool without a valid signature from the real provider. Similarly, immutable versioning means

once a tool is registered at version X, it cannot silently change its behavior; any update would be a new version that goes through a trust process. Alongside identity, **OAuth2-based authorization** for tools is recommended [21]. Instead of giving an agent a blanket API key, each tool invocation could carry an OAuth token with narrowly scoped permissions (and possibly an interactive user consent step for first-time use). Some security researchers suggest having a standard `.well-known/mcp-auth` endpoint or manifest field where a tool declares its required auth scopes [13]. The agent (or the platform hosting the agent) would then handle obtaining a token from the user or an identity provider, constrained to exactly those scopes. For example, a code-generation tool might declare it needs "repo:read" access on GitHub; the user, upon connecting that tool, would grant a GitHub OAuth token limited to reading a specific repo. If the tool (or agent) later tries a write operation, the token simply won't allow it. This model brings the benefits of OAuth's consent and scoping to MCP's dynamic tool world.

3. Runtime Monitoring and "Toxic Flow" Detection: Even with permissions and auth in place, we must assume some attacks will slip through (e.g. cleverly crafted prompts might abuse legitimate permissions). Therefore, continuous **monitoring of agent-tool interactions** is vital. Security teams should treat the AI agent like a new type of application that **requires logging, auditing, and anomaly detection**. For instance, all MCP tool calls could be logged with parameters and results, enabling post-mortem analysis and real-time alerts. Researchers have built specialized **MCP traffic scanners** that function as a proxy, inspecting the data going in and out of tools [22]. These can detect patterns like large data exfiltration (e.g. an unusually big response being returned to an agent) or unusual sequences of calls (e.g. the agent reading a password file right after reading a user prompt – suggesting a prompt injection led it astray). Invariant Labs' tool **MCP-scan**, for example, can sit between an agent and tools to flag potential security policy violations in real time [22]. The earlier GitHub exploit was detected by an **automated analyzer** that recognized the cross-repo data flow as toxic [5]. Organizations deploying MCP should integrate such scanning into their pipelines – akin to how web application firewalls (WAFs) monitor HTTP traffic. Additionally, making use of **LLMs for security** is a promising angle: ironically, the same technology causing the issue can help solve it. We can have separate guardrail models or scripts that watch the agent's decisions and either warn or veto potentially dangerous actions (like a second pair of eyes). This concept is sometimes called an "AI observer" or a safety agent monitoring the primary agent.

4. Human-in-the-Loop and Confirmation Gates: For high-stakes operations, it is wise not to give full autonomy to the AI agent. **Human-in-the-loop** approaches, where the agent must seek user confirmation or review for certain actions, add friction but can prevent disasters. For example, an agent could be allowed to draft code changes (pull requests) but not directly merge them – a human must inspect the PR. Or if an agent wants to send data outside a protected network, the action could be paused pending approval. MCP implementations can facilitate this by supporting **"confirmation required" flags** on tool actions. Some in the community suggest a simple manifest tag for tools or endpoints that are sensitive, which would cause the agent UI to prompt the user "Allow this action? [Yes/No]" when invoked. While not foolproof (a negligent user might click yes without understanding), it at least raises awareness of potentially risky operations. Over time, one could imagine more sophisticated **policy engines** that incorporate user preferences, context, and risk level to decide when to auto-allow vs. require approval.

5. Graph-Based Threat Modeling and Continuous Analysis: A more strategic recommendation – drawn from recent research – is to apply **semantic knowledge graphs** and threat modeling practices to the AI agent domain. Instead of treating an agent integration as a black box, we can formally model the assets, relationships, and trust boundaries involved. For example, one can build a **graph of all tools an agent has access to**, the data those tools can touch, and the connections between them. Every node (tool, data store, account) and edge (access, data flow) in this graph represents a potential avenue for attack or misuse. By encoding this in a living knowledge graph, security teams can query it to find

dangerous paths (e.g., "Can data from node X end up in external node Y via some sequence of calls?"). In fact, **advances in threat modeling propose representing assets, threats, and mitigations as nodes in a graph for continuous analysis** [23] [24]. LLMs themselves can assist in this process: they can ingest system documentation or observe agent behavior and help populate the graph with facts (such as "Tool A can access Database B") [25]. These facts can then be reviewed by humans and enriched with known threats (e.g. "Database B contains PII, which is sensitive"). By leveraging a graph-based approach, organizations embed *security by design*: you treat the MCP ecosystem not as a set of ad-hoc scripts, but as an architecture that can be analyzed for weaknesses. As Dinis Cruz et al. describe, such an approach enables **"continuous threat modeling"** – the graph is updated as the system evolves, and queries or even natural language questions to an LLM can reveal new threat scenarios in real-time [24]. For instance, one might ask, "Show me if any external MCP tool can influence transactions in the finance database," and if the graph knows of a path (maybe through a chain of tool calls), it can highlight that risk. This approach is still cutting-edge, but it points to a future where **AI and security are integrated at a fundamental level**: AI agents help maintain their own threat models, and security becomes a dynamic, collaborative process between humans and AI.

6. Improved Defaults and Frameworks: Finally, it's worth noting that many of MCP's issues can be mitigated by improving defaults in the protocol and tools. The community is actively discussing **enterprise-ready enhancements** to MCP's core specifications [26] [27]. Some proposals include: requiring TLS/mutual-TLS for any networked MCP connection, standardizing an auth mechanism as noted earlier, adding a permissions section in the MCP tool manifest, and providing reference implementations that have security baked-in (so developers don't roll their own poorly). Developer tooling can also help prevent mistakes – e.g., a `create-mcp-tool` scaffolding command could generate a new tool with a secure template (complete with schema validation, auth handling, and logging) [28]. If the tooling makes it easy to "do the right thing," then more developers will configure agents securely by default. Another aspect is **testing and CI**: including security checks in continuous integration. For example, automated tests could simulate an agent receiving malicious input and verify it doesn't leak data, or linting tools could flag if a tool manifest lacks a schema or auth. These processes mirror what we already do for traditional software (like running unit tests or static analyzers), but applied to AI agent behavior. In short, **the ecosystem around MCP is maturing**, and with the right community effort, we can establish *secure-by-default patterns*.

Conclusion

The **Model Context Protocol** is a pivotal development in connecting AI with the wider software world. It delivers immense power – LLMs can not only converse about data but can act on it through tools, heralding a new era of AI-driven automation. Yet, as we have argued, **this power comes with equally immense responsibility to secure it**. The early experiences with MCP have revealed cracks in our current security infrastructure: from prompt injection data leaks to over-scoped permissions and unvetted third-party tools. These are not mere implementation bugs; they are *architectural challenges* stemming from the way we integrate AI agents into systems. In essence, MCP is shining a spotlight on longstanding security issues (identity management, least privilege, trust boundaries) that now must be addressed with renewed urgency.

The encouraging news is that the community – AI developers and security researchers alike – is responding. A flurry of research is underway to bolster MCP security, ranging from pragmatic hardening guides [29] [13] to advanced proposals like ETDI for cryptographic tool identity [20]. Innovative security tools have already demonstrated the ability to catch agent misbehavior (e.g. detecting toxic flows [5]), and companies are starting to incorporate these learnings into product roadmaps (for example, providing official support for safer permission scopes or "AI agent" user roles). Importantly,

there is a growing recognition that **relying on AI model alignment alone is not sufficient** [10]. Robust security must be engineered at the *system level*, with layers of defense much like any critical software.

For developers of MCP-based agents, the takeaway is clear: **treat security as a first-class concern from the start**. Implement authentication for your tool endpoints; grant the minimal privileges possible; validate inputs and outputs; log everything; and remain vigilant for new threats. For security professionals, the advent of AI agents is a call to "**bring AppSec to AI**" – many of the skills and principles (threat modeling, code review, abuse case testing) are directly applicable, even if the medium (natural language prompts and tool manifests) is new [30]. The silver lining is that by confronting these challenges, we may push the envelope of security innovation. If we can design identity systems and permission frameworks that handle the complexity of AI agents, those solutions will likely benefit all of software security.

In conclusion, MCP is not "the problem" – it is an inflection point that exposes where our **infrastructure** must evolve. We should embrace MCP's potential to revolutionize how software components interact, but do so with eyes open to the risks. By combining principled security engineering with novel AI-assisted methods (like semantic graphs and automated policy enforcement), we can build a future where **AI agents are powerful yet safe collaborators**. The work starts now: as MCP commoditizes LLM integrations, security must be woven into that commodity. The next generation of MCP developers will hopefully say that the lessons of the early days were learned well – and that secure-by-design AI tooling became the norm, not the exception.

References

- Cruz, D. (2025). *Voice Memo on MCP Security* (transcribed content). – *Insight on the importance of infrastructure and identity in MCP security.*
- Salim A. B. (2025). "MCP (Model Context Protocol) with GO." *Medium. – Overview of MCP as an open protocol standardizing context exchange with LLMs* [1].
- Liang, W. (2023). "Building a Modular Image Processing Server with Gradio and MCP." *HuggingFace Blog. – Demonstrates enabling an MCP endpoint in Gradio, highlighting widespread adoption* [2].
- Cohen, R. (2025). LinkedIn post – "After working hands-on with MCP… here's what's broken and how to fix it." – *Identifies security issues in MCP: lack of default auth, no permission distinctions, need for sandboxing* [12] [11].
- Invariant Labs (2025). **"GitHub MCP Exploited: Accessing private repositories via MCP."** – *Blog post detailing a prompt injection attack that hijacked an MCP-driven agent to leak private data* [5] [3].
- Invariant Labs (2025). *"Why Model Alignment Is Not Enough"* (section in blog post above). – *Explains that even aligned LLMs (Claude 4) remained vulnerable to contextual attacks, necessitating system-level security measures* [9] [10].
- Bhatt, M. et al. (2025). **"ETDI: Mitigating Tool Squatting and Rug Pull Attacks in MCP via OAuth-Enhanced Tool Definitions and Policy-Based Access Control."** *arXiv preprint 2506.01333. – Proposes strengthening MCP with cryptographic tool identities, versioned definitions, explicit OAuth2 permission scopes, and a policy engine for fine-grained access control* [20] [21].
- Cruz, D. (2025). **"Advancing Threat Modeling with Semantic Knowledge Graphs."** – *White paper advocating for representing assets/threats in a knowledge graph to enable continuous, AI-assisted threat modeling* [23] [24].

1 MCP (AI-Model Context Protocol) with GO | by Salim Amine Bou Aram | Medium
https://medium.com/@salimbouaram12/mcp-ai-model-context-protocol-with-go-61a4ad490dc0

2 Building a Modular Image Processing Server with Gradio and MCP
https://huggingface.co/blog/liangwen12year/gradio-mcp-image-processing

3 4 6 7 16 yap, MCP is like going back to the early days of insecure-by-design APIs... | Dinis Cruz
https://www.linkedin.com/posts/diniscruz_yap-mcp-is-like-going-back-to-the-early-activity-7334646331539398656-XuKA

5 9 10 15 17 18 22 GitHub MCP Exploited: Accessing private repositories via MCP
https://invariantlabs.ai/blog/mcp-github-vulnerability

8 20 21 [2506.01333] ETDI: Mitigating Tool Squatting and Rug Pull Attacks in Model Context Protocol (MCP) by using OAuth-Enhanced Tool Definitions and Policy-Based Access Control
https://arxiv.org/abs/2506.01333

11 12 13 14 19 26 27 28 29 30 where's AppSec when we need it? Great analysis from Reuven Cohen on... | Dinis Cruz
https://www.linkedin.com/posts/diniscruz_wheres-appsec-when-we-need-it-great-activity-7325141437006700547-eHf0

23 24 25 How to use Semantic Knowledge Graphs for Threat Modeling | Dinis Cruz posted on the topic | LinkedIn
https://www.linkedin.com/posts/diniscruz_advancing-threat-modeling-with-semantic-knowledge-activity-7334009715133177858-5uN3