impart

MCP Security and Runtime Protection

A practitioner's framework for securing MCP servers, tools, and agent workflows

Table of contents

Part 01

Where MCP Stands Today

MCP has already reached mainstream enterprise adoption. Less than two years after Anthropic introduced the model context protocol, there are more than 10,000 active MCP servers and more than 97 million monthly SDK downloads.

The value of MCP is that agents can discover productivity tools at runtime, autonomously invoke them, exchange data across systems, and feed the results into ongoing work and/or workflows.

Its flexibility and autonomy also creates a security problem. Before MCP, most agents operated from a relatively small, fixed toolset embedded in the application. With MCP, an agent can interact with any server it can connect to, and every server becomes a trust boundary.

10,000
active MCP servers

97M
monthly SDK downloads.

There is also a meaningful difference between securing an MCP server's infrastructure and protecting the systems connected through it. Most programs start with the first problem and assume it covers the second. In reality, the gap between them is where attacks propagate.

MCP security and MCP protection are not the same thing

The market uses the two terms interchangeably, but they describe different control layers. One vendor uses "MCP security" to describe access control and authentication. Another says it in the context of observability and audit logging. A third means runtime inspection or threat enforcement. The lack of standardized terminology complicates vendor evaluations as well as how enterprise security teams operationalize their programs to secure MCP use.

MCP security typically refers to the set of controls that govern which agents can access which tools, along with the logging and policy infrastructure around those interactions. For most security teams, this is the first step of evaluating how to secure MCP deployments. It covers a few well-understood areas:

- **Centralized authentication.** Agents authenticate once through an identity provider, and that identity is enforced consistently across connected servers rather than passing credentials directly to individual tools.
- **Tool registry and access control.** A curated catalog of approved servers with per-agent, per-user permissions enforced at a gateway. This is where shadow server risk gets addressed, by blocking unvetted servers before an agent can reach them.
- **Audit logging.** The forensic record of every tool call, identity, server, and actions, along with timestamps.
- **Static policy enforcement.** Allowlists, blocklists, and rate limits that prevent known-bad access patterns on request metadata.

[1] Figures come from industry and community reporting rather than independently audited data.

Without this layer, teams lack a consistent record of which agents connected to which tools and no reliable baseline for investigation or policy enforcement. Most MCP security layers operate primarily at the infrastructure and request layer: authentication events, tool invocation metadata, access policy decisions, and audit logging.

Some platforms can inspect deeper runtime context, but many tools still have limited visibility into the user's original prompt, the model's reasoning, the content of a tool response, or the sequence of actions across a session. The reality is, at this stage, many MCP risks emerge based on how authorized tools are used rather than if access was permitted in the first place.

MCP protection is the runtime enforcement layer

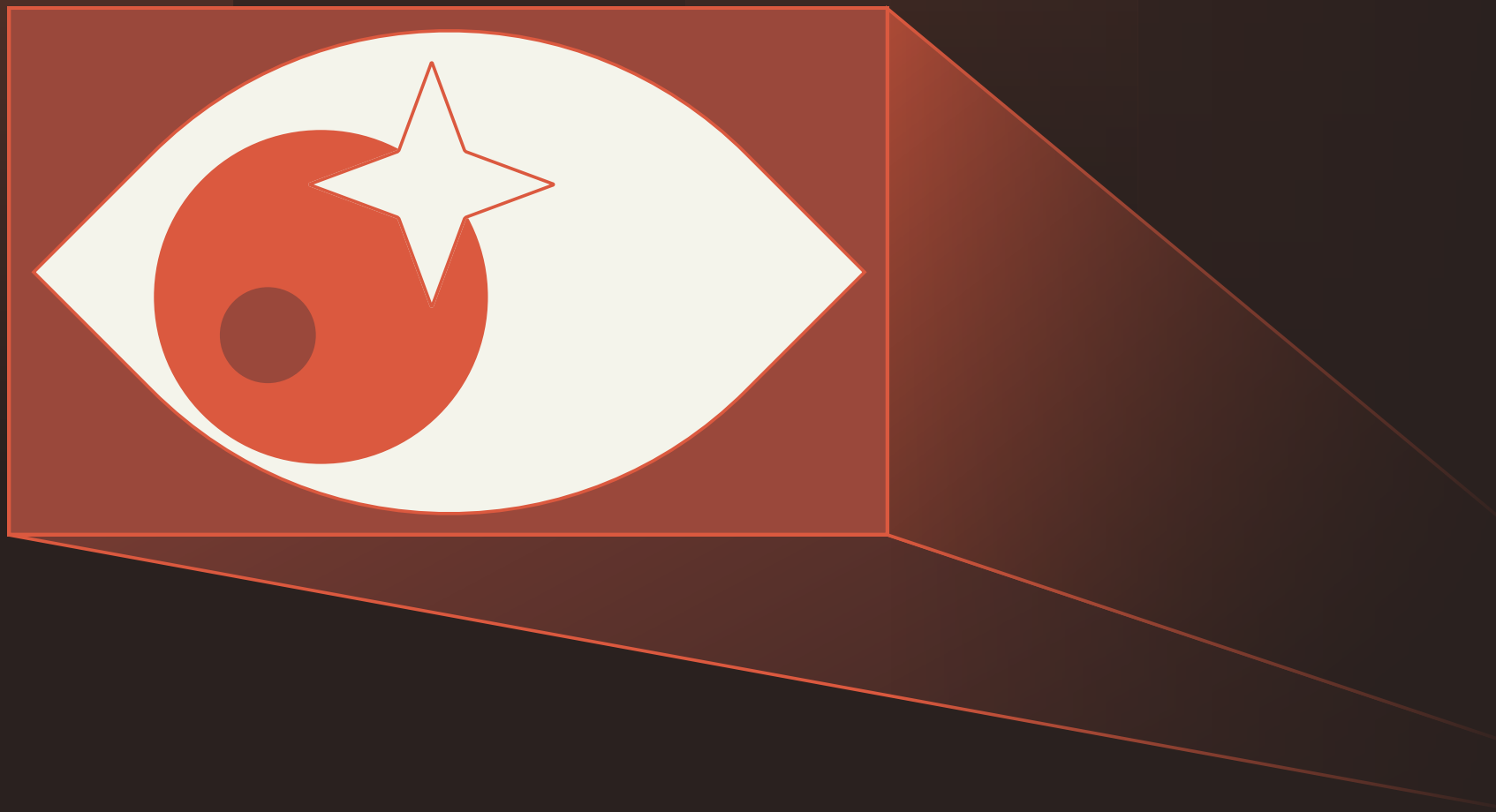
MCP protection is the active defense that inspects and, when necessary, blocks traffic between MCP servers, agents, and downstream tools while requests are executing. Where MCP security is about identity and policy, protection is about inline enforcement. It is the difference between firing an alert and stopping the action that triggered it.

Many agentic attacks begin with prompt injection delivered through a legitimate file server, using valid credentials and approved tooling. The request may also come from an authorized channel. It is not an access-control failure. The agent did exactly what it was permitted to do, against content that redirected it. Stopping that requires a control point in the request path itself.

This is where runtime protection extends beyond authentication and audit logging. It must evaluate behavioral analysis, session-level correlation, and active response. In multi-step agentic attacks, each individual call is authorized and appears normal in isolation. The attack only becomes visible when the sequence is analyzed as a whole.

Layer	Purpose
Governance	Access, ownership, inventory
Detection	Visibility and anomaly detection
Protection	Runtime prevention and enforcement

These are complementary, not competitive. MCP security establishes the governance and access boundaries. MCP protection adds runtime analysis, plus the ability to stop malicious behavior.



Part 02

Governance Controls for MCP Servers

Governance is the layer that decides whether the rest of the program is still working a year from now. Controls that are not owned drift. Controls that are not versioned diverge between environments.

Ownership before access

Every MCP server needs a named owner who is accountable for its posture, and a data classification based on what its tools can actually access. Those two attributes answer the prioritization question that everything else depends on. External exposure, broad tool permissions, and sensitive data access together define the highest-risk tier. An inventory that includes ownership becomes the basis for every decision about where to spend effort first.

Access scoped to the threat model

MCP authorization problems usually stem from overly broad access and poor visibility into chained tool behavior.

Server-level access is too coarse

An agent granted access to a server inherits access to every tool on it, even when the tools have very different risk profiles. Read operations, administrative functions, and write capabilities frequently coexist. Tool-level authorization limits blast radius by treating each capability independently.

Scope by agent identity, not role

Role-based authorization applied to agents tends to over-permission. Authorization by agent identity is more precise and auditable. It allows specific agents to invoke specific tools under specific constraints. Start from least privilege and expand based on observed legitimate usage rather than anticipated need.

Policy as a governed artifact

Security policy should live in version control with the same discipline as infrastructure code. Without it, temporary exceptions quietly become permanent, thresholds drift, and disabled controls remain disabled long after original reasoning applies.

Three governance practices prevent the predictable forms of decay.

- **Recalibration triggers.** A defined event, a tool update, a traffic-pattern shift, that initiates a threshold review rather than waiting for alert fatigue to force one.
- **Exception expiration.** Every exception carries a date and a review. An exception without an expiry is a permanent hole no one decided to keep.
- **Explicit ownership of every control.** Reviewed when team structures change, because the most common way a control dies is the quiet departure of the only person who understood it.

Measure coverage, not configuration. The operational health question is what percentage of the MCP surface has active controls monitored by server, tools, agent, and threat class. A known gap with a documented remediation date is a risk-management position you can defend.

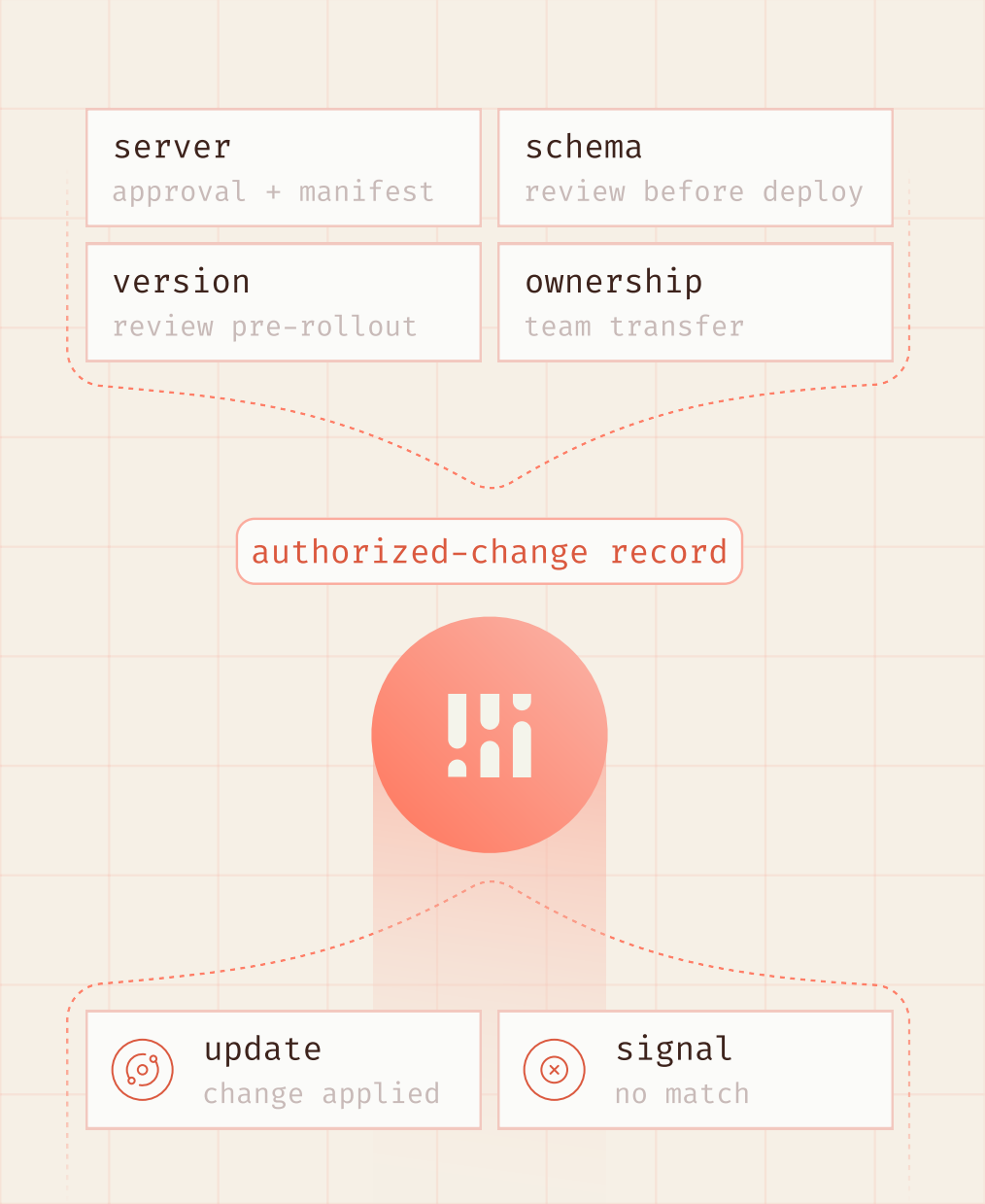
Governance of change

MCP environments change constantly. New servers come online, schemas evolve, versions ship, and ownership shifts between teams. Governance is what separates an authorized change from a suspicious one.

Every change should fall into one of two buckets: authorized or not. Four processes cover the surface.

- New MCP server approval and manifest
- Schema review before deployment
- Version review prior to rollout
- Ownership transfer during team changes

The connective tissue is the authorized-change record.
A change with a matching authorization is an update. A change without one is a signal. Everything the guardrails do later to catch tampering depends on this layer drawing that line cleanly.



Part 03

Behavioral Baselining for MCP Traffic

Static policy only protects against anticipated behavior when it was written. Behavioral protection covers the patterns that operate within permitted bounds, emerge after the policy was set, or are too dynamic for a static rule to capture. Many MCP attacks are sequence-based, rather than a single malicious request.

Agent traffic is not human traffic

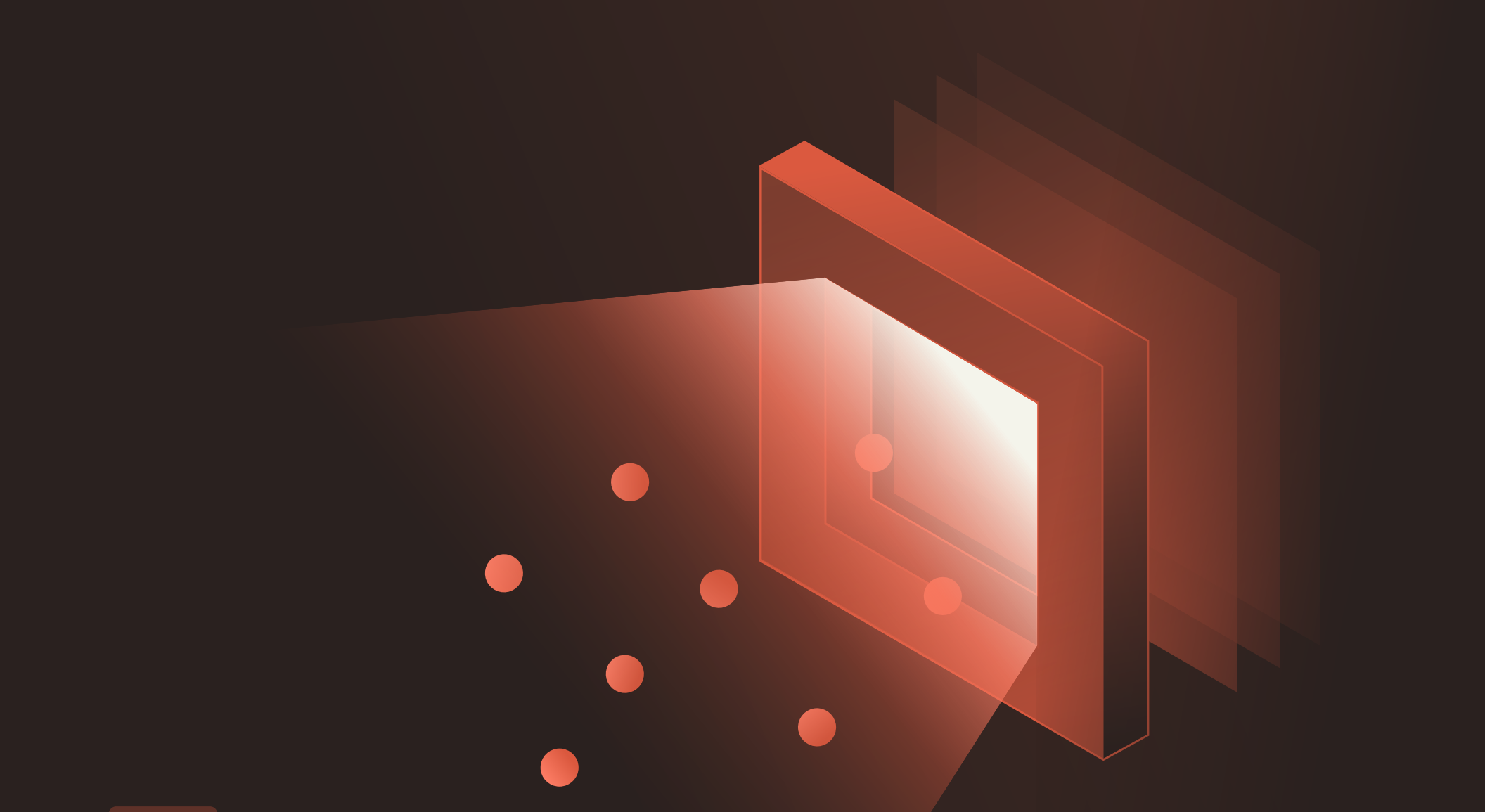
One of the more common baselining mistakes is calibrating against human-driven API patterns and applying the result to agents. Agentic workloads are naturally more variable, operate in bursts, change invocation rates across operational phases, and may call the same tool with very different argument characteristics depending on the input data being processed. A baseline built for human traffic flags all of that as anomalous, produces alert fatigue, and erodes confidence in inline enforcement.

The signals worth modeling

Some behavioral patterns are consistently useful:

- Broad tool discovery with little follow-on activity can indicate catalog scraping or reconnaissance.
- Unusual invocation sequences that deviate from established behaviors or resemble known attack patterns across multiple tool calls.
- Argument-level deviations, including long inputs, encoded payloads, or instruction-like language in fields that normally carry structured values.

Importantly, signals are not verdicts, and no one of these signals is definitive on its own. Behavioral baselining gives teams enough operational context to distinguish between authentic and malicious, and allows them to feel confident in enforcement decisions.



Part 04

Why the Enforcement Has to Be at Runtime

The MCP threat model is dominated by short-lived executions. The window between an invocation being initiated and a tool call completing is narrower than any detect-and-respond loop. Detection provides an audit trail. It does not offer protection. Enforcement requires guardrails in the execution path, with the authority to block a bad request before it completes.

Discovery and inventory come first

MCPs don't have a native registry, and deployment is typically developer-led, so inventory debt accrues by default. Surface mapping is continuous, and must encompass various discovery methods to ensure aggregate findings are not blind to any one tool's deficiencies.

- Network scanning covers broad surface to find endpoints but lacks context.
- Agent instrumentation shows which servers agents are actively calling but misses unseen or unused servers.
- CI/CD pipeline inspection catches deployed assets but misses out-of-band deployments.
- Infrastructure-as-code analysis is accurate for declared infrastructure but not runtime state.

Three catalogs, one surface area

The inventory is not a single list; it is a set of linked catalogues:

The server catalog records presence, ownership, transport,authentication configuration, connected agents and clients, and a data classification. This is the prioritization layer.

The tool catalog records function-level detail including schemas, parameter types, required fields, value constraints, and data access. Schema enforcement, drift detection, and behavioral baselining all read from this catalog.

The skills catalog records multi-step, prompt-driven behaviors from tool calls. This ensures risk is visible even if it emerges from combinations of calls.

Track catalog state over time

A snapshot is only useful against a baseline. Tool additions, schema modifications, and description changes after the initial review are a threat class, not configuration noise. Snapshot the full manifest at registration and on every detected change, with timestamps, because incident investigation routinely requires reconstructing what a server was advertising at a specific moment.

Guardrails in the request path

With an inventory and manifest in place, the next layer in inline enforcement. These controls sit between the agent and the MCP server, evaluating invocations before execution finalizes. Without this placement in the sequence, protection is retrospective and therefore less effective.

Schema validation, integrity, and drift detection

Every MCP tool exposes an argument schema. Enforcing it means validating every inbound invocation before the tool executes. Validate requests against the tools' declared schema: required fields, expected argument types, value ranges, and unexpected additional fields. The gap between what a tool claims to accept and what it actually receives is exactly where type confusion, argument injection, and schema boundary exploitation live.

The schema is also a baseline for drift detection. Catalog integrity starts with the manifest recorded at the time of security review and depends on comparing the live manifest against that baseline continuously, on a regular interval and on every new server connection. Classify deviations by severity rather than treating them as uniform noise.

- New tools expand the attack surface.
- Argument schema changes are high severity because they can alter agent behavior.
- Instruction-shaped content in tool descriptions are critical because they may indicate tempering or supply chain compromise. Critical. This is the signature of a rugpull or supply-chain compromise.

Schema comparison alone is not sufficient because behavior can change without structural changes to the manifest. A tool that previously returned structured data may now return instruction-shaped content instead. Detection therefore needs both manifest comparison and behavioral analysis of tool responses over time.

Protocol enforcement

MCP commonly operates over JSON-RPC 2.0, which makes malformed or non-conforming requests part of the attack surface. The enforcement layer should reject malformed JSON, invalid method calls, oversized payloads, and requests that do not match the server's declared interface before they reach application logic.

The gateway is the single enforcement point

Guardrails need a place to live, and that place is an inline gateway. An inline control sits in the request path before the tool executes, evaluates it against policy, and returns a pass, block, or modify decision before execution completes. Without this placement, detection happens after the action has completed.

Existing Tool	Limit of Authority
API gateways	Enforce transport controls like TLS, authentication, and rate limiting, but generally do not inspect MCP tool semantics or response content.
SIEMs	Provide correlation and investigation after execution has already occurred.
LLM observability	Operates at the model boundary, after tool responses have entered context.
CNAPP	Protect cloud infrastructure around MCP systems, but not MCP interactions.

Each tool generates visibility. None of them stop attacks in progress. Only runtime enforcement is capable of stopping prompt injection, token-replay with valid stolen credentials, or manifest drift. Importantly, runtime enforcement closes the window of attack without replacing other platforms.

Part 05

Detecting and Blocking at Runtime, Safely

A control that blocks legitimate traffic gets disabled by the developer within a week, which returns the process to detect-only mode. Bypassed controls protect nothing.

This makes continuous testing operationally necessary. Teams need to validate regularly that controls block malicious behavior without disrupting legitimate traffic.

AI-assisted blue teaming is becoming part of this equation. Blue teaming has always been a crucial process, but with attack techniques moving at AI speed, attackers can't be the only ones to capitalize on their efficiencies. Defensive methods need to generate, test, and refine rules at the same rate as attackers.

Policy as code

Policy belongs in version control, not a settings panel. Expressed as code, an enforcement rule is reviewed in a pull request, deployed through a pipeline, diffed, reverted, and owned like any other infrastructure. Expressed in a console state, it rarely receives the same scrutiny: A programmable policy engine becomes part of the infrastructure workflow. A checkbox console is circumvented.

Policy as code is also what makes the blue-team testing operationally useful. When a test reveals a gap, the fix becomes a tracked artifact, not an undocumented edit the next person cannot find. It keeps environments from diverging, gives every rule an owner and a history, and lets teams answer exactly what the enforcement layer was configured to do at any given time.

The discipline is the one that already governs infrastructure. MCP policy does not deserve a lower standard.

The risks that warrant inline blocking

The OWASP MCP Top 10 is the shared taxonomy. This section covers the classes where runtime enforcement is the deciding control, and grounds each in an incident that illustrates how adversaries are probing the surface right now.

MCP06 and MCP01: intent flow subversion and secret exposure through injection

Prompt injection is the most operationally significant class on the list, and the one most often misframed as a model problem. Injected instructions do not need to bypass the model's safety training. They only need to appear in context before the model makes its next decision.

A tool response containing an instruction to ignore prior instructions and call a deletion tool with elevated credentials is a protocol-layer attack. It succeeds because the instruction arrived in context at the decision point. System-prompt hardening and output filters operate at a different layer and address a different failure mode. Blocking this requires a inspection in the response path, between the tool output and the consuming agent.

EchoLeak
In June 2025, researchers at Aim Labs disclosed EchoLeak (CVE-2025-32711, CVSS 9.3), a zero-click indirect prompt injection in Microsoft 365 Copilot. A single crafted email, with no user interaction, caused Copilot to reach internal data and exfiltrate it to an attacker-controlled destination. The chain bypassed Microsoft's cross-prompt-injection classifier, link redaction, and content-security policy. Microsoft patched it server-side and reported no exploitation in the wild.

Why it matters here
EchoLeak is the first documented case of prompt injection weaponized for concrete data exfiltration in a production AI system. The instruction lived in ordinary business content and entered through an authorized retrieval path. No access control was violated. The defense is scoping and inline inspection of content before the model acts on it, not another model-layer filter.

Inspection also must cover structured data in addition to free text. A JSON field, a filename, a database record, any string value that lands in model context is an injection surface. Inspect for instruction-shaped text in fields meant for structured values, role or context override attempts, embedded tool-invocation directives, encoding inconsistencies such as base64 or excessive escaping, and self-referential content that names the agent's own context or configuration.

Inspection depth should vary by trust level, since deep inspection across every response is not sustainable at scale. External sources, user-generated content, and third-party APIs get the deeper inspection than internal systems with schema-constraints.

The same channel can also expose secrets. Hard-coded credentials, long-lived tokens, and sensitive data sitting in model context or protocol logs can be extracted through injection, which is why short-lived, scoped credentials and keeping secrets out of agent context are the foundational mitigations for MCP01.

MCP03: Tool poisoning, rugpulls, schema poisoning, and tool shadowing

Tool descriptions are instructions that directly shape model behavior. A tool that appears benign can carry hidden instructions to exfiltrate credentials before performing its stated function. Detecting it requires analyzing descriptor content at the semantic level.

Tool poisoning has three common attack patterns:

- A rugpull is a malicious update to a previously trusted tool.
- Schema poisoning corrupts the interface definition to mislead the model.
- Tool shadowing introduces a fake or duplicate tool to intercept or alter interactions.

All three defeat a one-time review because malicious change happens after approval.

The Postmark Rugpull
In September 2025, a malicious npm package, postmark-mcp, impersonated the legitimate Postmark email connector. It behaved correctly across fifteen releases, building trust, then version 1.0.16 added a single line that silently BCC'd every outbound email to an attacker domain. It was downloaded roughly 1,500 times before removal. The maintainer of the real service confirmed no affiliation. Researchers called it the first malicious MCP server caught in the wild.

Why it matters here
The schema never changed. The behavioral contract did. A pure manifest comparison would have passed this every time. The agent saw a working email tool, success after success, while every message was exfiltrated. This is the case version pinning, third-party verification on every connection, and behavioral drift detection exist to catch, and the case that argues against trusting any cached manifest for a server you do not own.

MCP05: Command injection in agentic workflows

Command injection in an agentic workflow is the classic pattern applied to a new interpreter. An agent constructs and executes a system command, shell script, API call, or code snippet from untrusted input without validation or sanitization. The agent's autonomy is what makes it dangerous; without a human checking the constructed command before it runs, the agent executes directly.

MCP-Remote RCE
In July 2025, JFrog disclosed CVE-2025-6514 (CVSS 9.6) in mcp-remote, a widely used proxy, with over 400,000 downloads, that lets local MCP clients such as Claude Desktop, Cursor, and Windsurf reach remote servers. A malicious server could return a crafted authorization_endpoint value during the OAuth flow that mcp-remote passed to an OS command without sanitizing it, yielding remote code execution and full system compromise. It affected versions 0.0.5 through 0.1.15 and was fixed in 0.1.16.

Why it matters here
This was the first real-world full RCE on a client OS triggered simply by connecting to an untrusted server. The injection rode in on the authentication handshake, before any tool was deliberately invoked. Strict input validation, never concatenating untrusted input into a command, and sandboxing tool execution are the controls. At runtime, the unexpected outbound connection and the anomalous command construction are both blockable signals.

Part 06

MCP Protection Readiness Assessment

A mature program is more than the sophistication of its tooling. Ultimately, it's defined by the questions it can answer.

Treat the following as a self-assessment. If any answer is "no," you have found your next priority.

01

Do you know what MCP servers are running, what tools they expose, and who is accountable for each?

02

Do you have active detection for MCP servers operating in your environment that were not explicitly registered or approved?

03

Does each tool connected to each server have a defined, minimum-permission scope, and is scope violation detectable?

04

Which OWASP MCP Top 10 attack classes have active controls and which have documented gaps with remediation dates?

05

What did the enforcement layer decide on every invocation in the last 90 days, and why?

06

Can you demonstrate a block, not just an alert, against a test injection, a replayed token, and a manifest change?

07

What configuration of your servers, tools, and agents run under explicit authorization rather than implicit server-level access?

Part 07

MCP Protection Buyer's Checklist

Most MCP protection evaluations fail because they focus on product categories rather than security outcomes. This short, concrete list should help with your vendor evaluation. The test throughout is whether a control can enforce protective controls during an attack or simply record the incident after the fact. Inline blocking authority is non-negotiable.

Enforcement architecture

- At least one inline control with blocking authority, in the request and response paths before execution completes.
- Inspection of tool response content before it returns to the agent, covering structured and unstructured data.
- Iterative protections that can enforce at any change, rather than reliance on post-execution tooling, SIEM, CNAPP, or observability, to block.

Discovery and integrity

- Layered discovery across network, agent instrumentation, CI/CD, and IaC, including ephemeral and embedded servers.
- Server, tool, and skills catalogs with ownership and data classification on every entry.
- Continuous manifest comparison against a reviewed baseline, with severity-classified drift and rugpull signals.
- Version pinning and verification on every connection for third-party and embedded servers.

Conformance and authorization

- JSON-RPC 2.0 policy-aligned enforcement with deliberate, logged failure modes: block, flag, or quarantine.
- Schema validation as an enforced contract on every inbound invocation.
- Tool-level authorization scoped to agent identity, not server-level grants or broad roles.
- Per-caller and per-tool quotas with blocking enforcement, plus cost quotas for expensive tools.

Detection and operations

- Behavioral baselining per agent, tool, and server, calibrated for agent traffic patterns.
- Dedicated detection for catalog scraping, unusual sequences, and argument-level anomalies.
- Session-level correlation that connects individually authorized calls into a sequence.
- Policy as code under version control, with recalibration triggers and exception expiration.
- An MCP-specific incident response runbook: session containment, chain reconstruction, per-type remediation.
- Coverage metrics by server, tool, agent, and threat class, with documented timelines on every known gap.

Questions to press on:

01 How is normal behavior defined?

02 How long is the learning period?

03 Is baselining performed per agent, per tool, and per server?

04 Does the baseline adapt automatically as workloads change?

05 How are legitimate operational changes distinguished from attack activity?

06 How are false positives measured and reduced?

07 How are false positives measured and reduced?

The Final Evaluation Test

The strongest MCP protection platforms combine governance, behavioral detection, and runtime enforcement into a single operational model.

They can answer three questions with evidence:

01 What is running?

02 What is abnormal?

03 What can be stopped before execution completes?

A platform that can answer only the first question provides governance. A platform that can answer the first two provides visibility. A platform that can answer all three provides protection.

The ubiquity of MCP server use in the enterprise is new enough that MCP security and MCP protection are not uniformly defined across vendor platforms or even taxonomy. This creates a knowledge gap for operators, and delays up-to-date controls and enforcement.

However, as defenders hastily implement guardrails to ensure MCP use is secure, attackers are forging ahead with attacks against MCP, necessitating better solutions quicker.

The steps in this guide are meant to assist teams and provide proven runtime security processes and solutions that prevent attacks against MCP from progressing. Let the attack start. It won't finish.