OWASP | GenAI SECURITY PROJECT
TOP 10 FOR LLM AND GENERATIVE AI

A Practical Guide for Secure MCP Server Development

Version 1.0
February, 2026

The information provided in this document does not, and is not intended to, constitute legal advice. All information is for general informational purposes only. This document contains links to other third-party websites. Such links are only for convenience and OWASP does not recommend or endorse the contents of the third-party sites.

License and Usage

Table of Content

Introduction & Background

Organizations need to secure their MCP servers to safeguard AI integrations. These servers act as bridges between AI assistants and external tools or data sources, so any security gap could allow attackers to manipulate those AI assistants, steal sensitive information, or compromise downstream systems. Unlike traditional APIs, MCP servers often operate with delegated user permissions, dynamic tool-based architectures, and can chain multiple tool calls, amplifying the impact of any single vulnerability.

This guide provides best practices for designing and implementing secure MCP servers. While security practitioners will certainly benefit from this guide, it is intended for software architects, platform engineers, and development teams responsible for MCP server development. The content outlines essential security controls and design decisions to apply from initial architecture through production deployment.

Current Vulnerability Landscape

MCP servers expose a broad and unique attack surface, containing traditional API security vulnerabilities and some unique AI-oriented risks. Common vulnerabilities/risks include:

- **Tool Poisoning:** Adversaries manipulate the AI through crafted inputs in prompts or tool metadata. A maliciously designed tool description, for example, can include hidden instructions that trick the model into performing unintended actions or exfiltrating data.
- **Dynamic Tool Instability ("Rug Pulls"):** Given the lack of strict versioning for tool descriptions and the dynamic nature of loading them, there is a risk of "rug pulls." A previously trusted tool definition can be swapped or modified in real-time to introduce malicious behavior, bypassing initial security checks that occurred before the change.
- **Code Injection & Unsafe Execution**: The MCP server passes model-provided inputs directly into system commands, APIs, or database queries without validation. Incorrect, misguided, or malicious code might be executed without user awareness or approval.
- **Credential Leakage & Token Misuse:** MCP servers often handle API keys or OAuth tokens. If these secrets are improperly stored, logged in plaintext, or cached for too long, attackers can steal them to impersonate the client or access connected systems.
- **Excessive Permissions:** Over-privileged tools or broad access scopes dramatically amplify the impact of a breach. This violates the principle of least privilege and allows a single compromised tool to endanger many systems. Applicable if the MCP server is the policy enforcer, or if the LLM has a permissions mismatch when interacting with a user.
- **Insufficient Isolation (Session, Identity & Compute):** MCP servers often manage multiple concurrent sessions, share access privileges, or execute code in shared environments, creating multiple layers of isolation risks. Without protocol-enforced state separation, there is a distinct risk of cross-tenant data leakage where one user might accidentally access another's context. Furthermore, if the MCP server shares a single access identity (such as a service account) among multiple users, it creates an identity impersonation risk where individual user actions cannot be distinguished. Beyond logical separation, there is a critical need for compute isolation; when servers execute code or tools in a shared host environment without strict sandboxing (like containers or micro-VMs), malicious tenants can potentially exhaust shared resources or access the process memory of others.

1. Secure MCP Architecture

- **Local vs Remote Connections:** MCP exposes two communication channels, STDIO or Unix sockets for local network communications, and Streamable HTTP for remote connections. For each communication type, additional security controls apply:
    - **Local MCP Servers:** Prefer STDIO or Unix sockets over network sockets. If you must use local HTTP, bind only to 127.0.0.1, still utilize explicit authorization/authentication, and validate the Origin header. Run processes in isolated/sandboxed subprocesses or containers, with minimal or no access to external networks and minimal privileges.
    - **Remote MCP Servers:** Enforce TLS 1.2+ for all remote connections. Strictly validate all JSON-RPC messages against the MCP schema, rejecting any malformed or unrecognized data.
- **Enforce Trusted Connections and Authentication for MCP Clients:** To secure interactions between MCP Clients and Servers, use strong validation techniques such as allowlists, hardcoded connections, or mutual TLS (mTLS) for known static relationships. In dynamic environments where clients change, use strong authentication protocols (such as OAuth 2.1 or OIDC) to dynamically verify client identity and allow connections rather than depending exclusively on static network trust.
- **Isolate Users and Sessions:** In multi-tenant systems, strictly segregate execution contexts, memory, and temp storage for each user or agent. Strictly prohibit the use of global variables, class-level attributes, or shared singleton instances for storing user-specific data. Architect the server to instantiate separate objects per session or utilize a strictly session-keyed state store (e.g., Redis with session_id namespaces).
    - **Strict Lifecycle Management:** Implement deterministic cleanup routines. Ensure that when an MCP session terminates, disconnects, or times out, all associated file handles, temporary storage, in-memory contexts, and cached tokens are immediately flushed and destroyed to prevent residual data exposure.
    - **Per-Session Resource Quotas:** Enforce strict limits on memory, CPU, file system usage, and API rate limits keyed to the session ID or user identity.

2. Safe Tool Design

- **Cryptographic Tool Manifests:** Require every tool to have a signed manifest that includes its description, schema, version, and required permissions. Verify this signature and hash at load time.
- **Strict Onboarding and Approval:** Maintain a formal approval workflow for adding or updating any tool description or functionality. This must include code scanning (SAST), dynamic testing, dependency scanning (SCA), and manual security review.
    - **Validate Descriptions vs. Behavior:** Implement manual checks, tool pinning, and LLM scans to ensure a tool's advertised functionality (in its description) matches its actual code behavior during runtime. Flag any tool that attempts to perform actions (e.g., network writes) not mentioned in its description.
    - **Tool Structure Validation:** Maintain and audit all fields of each tool added to the policy. Only expose minimal, strictly necessary fields to the model, keeping internal metadata and sensitive fields outside of model context.

3. Data Validation & Resource Management

- **Resource Usage Limits:** Impose quotas and rate limits on tool invocations and data fetches per session. Use timeouts and isolated memory/compute budgets to prevent DoS attacks or runaway processes.
- **Strict Input/Output Validation:** Treat all data as untrusted. Define and enforce JSON Schemas for every tool's inputs (from the model) and outputs (back to the model). Reject any request that doesn't match the expected schema.
- **Strict Input/Output Sanitization and Encoding:** Filter and sanitize all inputs to the model and outputs into the tool. Strip or escape sequences which could lead to classic code injection attacks (e.g. XSS, SQL Injection, RCE). Enforce size limits on all outputs from the tools or model.

4. Prompt Injection Controls

- **Structured Tool Invocation:** Favor structured JSON tool calls over having the model generate free-form text commands. This funnels the model's intent through a formal, schema-validated interface.
- **Human-in-the-Loop (HITL):** For high-risk actions (e.g., deleting data, sending money, system-level changes), implement an approval checkpoint. Pause the action and require explicit confirmation (e.g., using MCP elicitations) from a human user on MCP Client before proceeding.
- **LLM-as-a-Judge:** For high-risk actions, run a dedicated approval check in a distinct context LLM session, using a policy prompt that defines which tool calls and parameters are allowed and which are blocked.
- **One Task, One Session:** Reset MCP sessions when an agent switches contexts or tasks. This "context compartmentalization" prevents hidden instructions from persisting in a long conversation history, and could minimize “context degradation" improving LLM output quality.

5. Authentication & Authorization

- **Enforce OAuth 2.1 / OIDC:** Treat authentication as mandatory for all remote MCP servers. Use OAuth 2.1/OIDC for client and user identity, validate iss, aud, exp, and signatures on every request.
    - **Token Delegation:** Use the OAuth token delegation flow to pass user context and permissions to an MCP server while limiting permissions and retaining distinct server non-human identity (NHI). (RFC 8693)
    - **Prohibit Token Passthrough**: Do not forward client tokens to downstream APIs; instead, use tokens explicitly issued to the MCP server or validated via "On-Behalf-Of" flows. Direct passthrough breaks audit trails and bypasses policies, creating a critical Confused Deputy risk. This vulnerability allows the MCP server to be tricked into misusing a user's privileges to execute unauthorized actions, effectively bypassing the server's intended security constraints.
- **Use Short-Lived, Scoped Tokens**: Issue access tokens with minimal lifetimes (minutes) and narrow scopes. Revalidate token signature, audience, and expiry on each call before executing any tool or resource access.
- **Treat Sessions as State, Not Identity**: Never rely on session IDs alone for authorization or authentication. Bind any session/stream/queue entry to validated user and client identity (OAuth), rotate/expire aggressively, and re-check authorization before performing sensitive actions.
- **Centralize Policy Enforcement:** Use a dedicated policy/gateway layer to enforce authentication, authorization, consent, tool filtering, and audit logging. Evaluate all requests centrally so policies remain consistent across agents, servers, and upstream services.

6. Secure Deployment & Updates

- **Secrets Storage and Management:** Utilize secrets storage vaults for client credentials and API keys. Do not store secrets in environment variables, logs or plain-text in code. Never allow an LLM access to a secret, perform all secrets management functions in transparent middleware which is inaccessible to the LLM.
- **Containerize and Harden:** Deploy the MCP server in a minimal, hardened container. Run the container as a non-root user and drop all unnecessary container packages, dependencies, or Linux capabilities to limit residual attack surface.
- **Network Segmentation:** Place the server in a restricted network segment. Use firewall rules or Kubernetes NetworkPolicies to block all inbound and outbound traffic, except for what is explicitly required.
- **Supply Chain Controls:** Version pin and scan all dependencies at build time, use signed container images, monitor for new CVEs, and maintain AIBOMs for all builds to ensure provenance and detect tampering.
- **CI/CD Security Gates:** Integrate security checks and scanning directly into your pipeline as a gate. Fail the build if new code introduces a vulnerability, uses an unapproved dependency, or fails a policy check (e.g., using policy-as-code tools like OPA).
- **Safe Error Handling:** Do not return stack traces, tokens, filesystem paths, or tool internals in responses returned to the model/client.

7. Governance

- Cryptographic Integrity: Use cryptographic signing and version pinning for all tools, dependencies, and registry manifests to ensure their integrity and prevent tampering.
- Peer Review and Oversight: Establish a policy that no new tool or major code change goes live without a security-focused peer review.
- Audit Logs and Trails: Log every action, including tool invocations (with parameters), resource access, authentication/authorization events, and configuration changes (logging new and old values). Use field-level allowlists and redaction/hashing to prevent sensitive data from entering verbose logs. Store these logs securely and immutably for forensic analysis.
- Non-Human Identity Governance: Treat all automated agents, backend processes or MCP server systems as first-class identities with unique credentials and tightly scoped permissions. Continuously audit NHI systems for data access and tool usage.

8. Tools & Continuous Validation

- **Automated Code Scanning:** Use static analysis (SAST) tools (with custom MCP rules) and Invariant MCP-Scan in your CI/CD pipeline. Use Software Composition Analysis (SCA) tools (e.g., npm audit, pip audit, OSV-Scanner) to find vulnerabilities in dependencies. Break any build which exceeds your risk tolerance due to scan findings.
- **Runtime Protections:** Use runtime security tools to enforce isolation. This includes Docker's seccomp profiles, AppArmor, or specialized wrappers like context-protector. Tools like mcp-watch can monitor runtime behavior.
- **Continuous Monitoring:** Feed MCP server audit logs into a centralized monitoring system (SIEM). Configure real-time alerts for suspicious patterns, such as a spike in failed validations, high-frequency tool calls, or a tool suddenly accessing unusual files.
- **Supply Chain Vigilance:** Use frameworks like the OpenSSF Scorecard to evaluate your project's security posture. Monitor vulnerability databases like OSV for new issues in your dependencies.

MCP Security Minimum Bar (Review Checklist)

1. Strong Identity, Auth & Policy Enforcement
   - All remote MCP servers use OAuth 2.1/OIDC.
   - Tokens are short-lived, scoped, validated on every call.
   - No token passthrough; policy enforcement is centralized.

2. Strict Isolation & Lifecycle Control
   - Users, sessions, and execution contexts are fully isolated.
   - No shared state for user data.
   - Sessions have deterministic cleanup and enforced resource quotas.

3. Trusted, Controlled Tooling
   - Tools are cryptographically signed, version-pinned, and formally approved.
   - Tool descriptions are validated against runtime behavior.
   - Only minimal, necessary tool fields are exposed to the model.

4. Schema-Driven Validation Everywhere
   - All MCP messages, tool inputs, and outputs are schema-validated.
   - Inputs/outputs are sanitized, size-limited, and treated as untrusted.
   - Structured (JSON) tool invocation is required.

5. Hardened Deployment & Continuous Oversight
   - Server runs containerized, non-root, network-restricted.
   - Secrets are stored in vaults and never exposed to the LLM.
   - CI/CD security gates, audit logs, and continuous monitoring are mandatory.

Acknowledgements

Contributors

Name	Company
Idan Habler, PhD	CISCO, OWASP
Tomer Elias	OWASP
Joshua Beck	SAS
Venkata Sai Kishore Modalavalasa	OWASP
Manish Bhatt	OWASP/Amazon Leo
Vineeth Sai Narajala	Cisco, OWASP
Yuval Sarel	Cyera
Tomer Wetzler	Zenity
Rico Komenda	adesso SE & OWASP
Hisham Abdulhalim	Payoneer
Guy Shtar	intuit
Sagiv Antebi	BGU
John Cotter	Bentley Systems
Dipen Shah	Affirm
Aamiruddin Syed	OWASP
Roy Barkay	Nebius
Almog Langleben	Sunbit
Riggs Goodman III	AWS
Roi Vanunu	Jazz

OWASP GenAI Security Project Sponsors

We appreciate our Project Sponsors, funding contributions to help support the objectives of the project and help to cover operational and outreach costs augmenting the resources provided by the OWASP.org foundation. The OWASP GenAI Security Project continues to maintain a vendor neutral and unbiased approach. Sponsors do not receive special governance considerations as part of their support.

Sponsors do receive recognition for their contributions in our materials and web properties. All materials the project generates are community developed, driven and released under open source and creative commons licenses. For more information on becoming a sponsor, visit the Sponsorship Section on our Website to learn more about helping to sustain the project through sponsorship.

Project Sponsors:

GenAI Security Project – Gold Project Sponsor – Top 10 for LLM and Generative AI

ByteDance · HiddenLayer · Mend.io · Straiker · snyk · paloalto Networks · TREND micro

GenAI Security Project – Silver Project Sponsor – Top 10 for LLM and Generative AI

SPLX · Calypso AI · Acuvity · pangea · Cobalt · LAKERA · securiti · ActiveFence · TROJ.AI · LASSO · PROTECTO · PROMPTARMOR · Synack · Starseer · NeuralTrust · Prompt: · akto · Capsule

Sponsors list, as of publication date. Find the full sponsor list here.

Project Supporters

Project supporters lend their resources and expertise to support the goals of the project.

Accenture
AddValueMachine Inc
Aeye Security Lab Inc.
AI informatics GmbH
AI Village
aigos
Aon
Aqua Security
Astra Security
AVID
AWARE7 GmbH
AWS
BBVA
Bearer
BeDisruptive
Bit79
Blue Yonder
BroadBand Security, Inc.
BuddoBot
Bugcrowd
Cadea
Check Point
Cisco
Cloud Security Podcast
Cloudflare
Cloudsec.ai
Coalfire
Cobalt
Cohere
Comcast
Complex Technologies
Credal.ai
Databook
DistributedApps.ai
DreadNode
DSI
EPAM
Exabeam
EY Italy
F5
FedEx
Forescout
GE HealthCare
Giskard
GitHub
Google
GuidePoint Security
HackerOne
HADESS
IBM
iFood
IriusRisk
IronCore Labs
IT University Copenhagen
Kainos
KLAVAN
Klavan Security Group
KPMG Germany FS
Kudelski Security
Lakera
Lasso Security
Layerup
Legato
Linkfire
LLM Guard
LOGIC PLUS
MaibornWolff
Mend.io
Microsoft
Modus Create
Nexus
Nightfall AI
Nordic Venture Family
Normalyze
NuBinary
Palo Alto Networks
Palosade
Praetorian
Preamble
Precize
Prompt Security
PromptArmor
Pynt
Quiq
Red Hat
RHITE
SAFE Security
Salesforce
SAP
Securiti
See-Docs & Thenavigo
ServiceTitan
SHI
Smiling Prophet
Snyk
Sourcetoad
Sprinklr
stackArmor
Tietoevry
Trellix
Trustwave SpiderLabs
U Washington
University of Illinois
VE3
WhyLabs
Yahoo
Zenity

Supporters list, as of publication date. Find the full supporter list here.