AI INFRASTRUCTURE · AUGUST 11, 2026

MCP Authentication Between Agentic Clients and Servers: Types and Common Implementations

How MCP connections are secured today: environment credentials for local stdio servers, static API keys for internal use, and OAuth 2.1 with PKCE as the mandated standard for remote Streamable HTTP servers, plus the client-credentials and EMA extensions for machine-to-machine auth.

7 min read · 12 sources

AI agents | MCP | OAuth | authentication | security

CONTENTS

Two transports, two threat models 1
The three authentication methods in practice 2
The OAuth 2.1 flow, step by step 2
Spec evolution and what OAuth 2.1 changed 4
Machine-to-machine: client credentials and EMA 4
Where implementations go wrong 5
Bottom line 5
Sources 5

Authentication in the Model Context Protocol is decided by transport, not by a single protocol-wide mechanism. Local servers over stdio trust the process environment; remote servers over Streamable HTTP are expected to implement OAuth 2.1, with static API keys still common in practice for internal deployments. The spec treats authorization as **optional but strongly recommended**, and the ecosystem has converged on three practical methods: environment-based credentials, static bearer tokens/API keys, and the full OAuth 2.1 authorization-code flow with PKCE[1][6].

Two transports, two threat models

MCP defines two standard transport bindings: **stdio** (newline-delimited JSON-RPC over the standard streams of a client-launched subprocess) and **Streamable HTTP** (each message an HTTP POST to a single MCP endpoint)[2]. Older SSE-based remote transport was deprecated in the 2025-06-18 spec revision[4].

- **stdio (local servers):** the client spawns the server as a subprocess, so there is no cross-network trust to establish. The authorization spec says stdio implementations **SHOULD NOT** use the OAuth flow and should instead retrieve credentials from the environment[1]. The MCP server

authenticates to *upstream* services (APIs, databases) using env vars such as `API_KEY`, and may use OS-level credential stores — none of that is defined by the MCP spec[4][6].

- **Streamable HTTP (remote servers):** no shared trust exists, so the spec prescribes OAuth 2.1 at the HTTP layer, with the MCP client acting as an OAuth client, the MCP server as an **OAuth 2.1 resource server only**, and a separate authorization server (typically an enterprise IdP) issuing tokens[1][4][7].

> *"MCP servers MUST NOT accept any tokens that were not explicitly issued for the MCP server." — MCP security best practices, quoted in*[4]

The three authentication methods in practice

METHOD	WHERE IT'S USED	STRENGTHS	LIMITATIONS
Environment-based credentials	stdio/local servers	Zero setup, no external flow[6]	Credentials readable by any process on the host; no per-user identity[6]
Static API keys / bearer tokens	Internal tools, dev, well-bounded networks	Simplest remote option[6]	No expiry, no user identity, no granular scopes; leaks via logs/Git are a top real-world breach path[6]
OAuth 2.1 with PKCE	Remote/production servers (spec standard)	Short-lived scoped tokens, per-user identity, rotation, audit[6][11]	Most moving parts: discovery, registration, token lifecycle[5]

TrueFoundry's summary of the same taxonomy: "static API keys for internal or development use cases, OAuth 2.1 with PKCE for all remote MCP server deployments as required by the MCP specification, and environment-based credentials" for stdio[6]. An unauthenticated remote server is viable only for read-only public services (e.g., the context7 documentation servers)[4].

The OAuth 2.1 flow, step by step

The canonical flow has five stages — discovery and registration happen once per client–server pair; token acquisition repeats as needed[5].

1. Server rejects and points to metadata. The client's first unauthenticated request gets `HTTP 401` with a `WWW-Authenticate: Bearer` header carrying a `resource_metadata` URL (RFC 9728 Protected Resource Metadata)[1][3]:

```http
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="mcp",
  resource_metadata="https://your-server.com/.well-known/oauth-protected-resource"
```

The PRM document lists the trusted authorization servers and `scopes_supported`. MCP servers **MUST** implement PRM; clients **MUST** use it for authorization-server discovery[1][7].

2. Authorization server discovery. The client fetches the AS's metadata from `/.well-known/oauth-authorization-server` (RFC 8414) or OpenID Connect Discovery; clients must support both. The metadata exposes `authorization_endpoint`, `token_endpoint`, `registration_endpoint`, and `code_challenge_methods_supported` — in a conformant server only `["S256"]` appears [1] [5].

3. Client registration. Clients must obtain a client ID through one of three mechanisms, in priority order: **Client ID Metadata Documents (CIMD)**, **pre-registration**, or **Dynamic Client Registration** (RFC 7591) [1] [4]. CIMD — where the client's ID *is* an HTTPS metadata URL the AS fetches at authorization time — became the recommended approach because DCR creates unbounded database growth, client-expiry "black holes," and an unauthenticated write endpoint that's a DoS vector [9]. DCR is retained for backward compatibility [1] [9].

4. Authorization code with PKCE. The client generates a random `code_verifier` (43–128 chars), derives an S256 `code_challenge`, and redirects the user to `/authorize` [5] [11]:

```
GET /authorize?response_type=code&client_id=...&redirect_uri=http://localhost:8400/callback
  &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
  &code_challenge_method=S256&state=xyz789&resource=https%3A%2F%2Fmcp.example.com
```

Two MCP-mandated details: the `resource` **parameter (RFC 8707)** must identify the MCP server in both authorization and token requests, and **PKCE is mandatory** (S256 only — plain challenges are not allowed) [1] [5]. Desktop clients like Claude Desktop and Cursor can't take web redirects, so they run a temporary local HTTP server (commonly `localhost:8400`) as the redirect URI [5].

5. Token exchange and usage. The client posts the code plus the original verifier to `/token` and receives a short-lived access token (JWTs of ~5–15 minutes are typical) and optionally a refresh token [5] [11]. Every subsequent MCP request carries it [1]:

```HTTP
POST /mcp HTTP/1.1
Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
```

Tokens **MUST NOT** go in the URI query string, and access tokens must be validated for issuer, signature, audience (the MCP server's URI), expiry, and scope [1] [4]. Refresh-token rotation (single-use refresh tokens) is required by OAuth 2.1, and refresh tokens are requested via the `offline_access` scope per the July 2026 revision [7] [11].

Error handling and step-up. Servers return `401` for missing/invalid tokens, `403` with `error="insufficient_scope"` plus the required scopes when a token lacks permissions, and `400` for malformed requests [1]. On a 403 scope challenge, clients re-run authorization requesting the **union** of previously granted and newly required scopes, then retry [1] [7].

Spec evolution and what OAuth 2.1 changed

Mar 2025 — **OAuth 2.1 adopted**
Defined as the standard for MCP authorization [6]

Jun 2025 — **Resource-server separation**
MCP servers become OAuth resource servers; token issuance moves to external IdPs [6] [7]

Nov 2025 — **PKCE mandatory, CIMD**
PKCE required for all clients; CIMD introduced as client-identity mechanism [6]

28 Jul 2026 — **Auth spec revision**
Issuer validation, application_type, credential binding, offline_access, scope-union step-up, clarified discovery [7]

The choice of OAuth 2.1 over 2.0 matters directly for agent clients, which are typically **public clients that cannot store secrets** [11]:

PROPERTY	OAUTH 2.0	OAUTH 2.1
PKCE	Optional	**Mandatory, all flows**
Implicit grant	Permitted	**Removed**
Resource-owner password grant	Permitted	**Removed**
Refresh token rotation	Optional	**Required, single-use**
Redirect URI matching	Partial allowed	**Exact only**
Token transport	Query params permitted	**Authorization header only**
Resource indicators	Optional	**Recommended (RFC 8707)**

Machine-to-machine: client credentials and EMA

The core spec is deliberately silent on server-to-server auth, but two official extensions in the `modelcontextprotocol/ext-auth` repository fill the gap [4] [10]:

- **Client Credentials (draft extension, added Oct 2025):** OAuth 2.1 client-credentials flow for agents acting without a user. It requires **pre-registered** credentials (DCR is not used) and supports JWT client authentication (RFC 7523, recommended) or client secret. Tokens are obtained from `/token` with `grant_type=client_credentials` plus `resource` and `scope`, then presented to the MCP server exactly like user-flow tokens [10].
- **Enterprise-Managed Authorization (EMA, stable extension):** lets an enterprise IdP govern which clients can reach which servers using the Identity Assertion JWT Authorization Grant (ID-JAG), minting scoped tokens without per-user consent screens [7] [10].

For workloads running in cloud environments, a widely recommended pattern is **infrastructure-asserted identity** — authenticating the agent to the AS via AWS IAM roles, Kubernetes service-

account tokens, or Azure instance metadata, with conditional-access policies (host posture, environment, time) deciding whether tokens are issued [8].

Where implementations go wrong

- **mcp-remote RCE (CVE-2025-6514):** the popular `mcp-remote` client was vulnerable to OS command injection from crafted `authorization_endpoint` URLs in the OAuth handshake — **CVSS 9.6 critical**, affecting versions 0.0.5 through 0.1.15 — a reminder that OAuth discovery input from remote servers must be treated as untrusted [12].
- **Static tokens committed to repos or `mcp.json`/`.env` files**, over-permissioned scopes, and long-lived tokens with no rotation remain the top practical failures; most are resolved by moving to OAuth 2.1 [6].
- **Token passthrough / confused deputy:** servers calling upstream APIs must obtain their own tokens (e.g., via RFC 8693 token exchange) and must never forward the client's token [4] [7].
- **Client impersonation:** desktop apps can't hold secrets, so `localhost` impersonation is an unsolved trust problem; the roadmap is signed **software statements** and eventually platform-level attestation [9].

Bottom line

If you're building or consuming MCP today: local stdio servers should keep using environment credentials, and any remote server should implement the OAuth 2.1 authorization-code flow with mandatory PKCE, RFC 9728 protected-resource metadata, RFC 8707 resource indicators, and audience-bound token validation — using an existing IdP (Keycloak, Okta, Azure AD) rather than writing an authorization server. For agent-to-agent or cron-style workloads, use the client-credentials extension with pre-registered clients, and treat every piece of metadata fetched during discovery as attacker-controlled input — the mcp-remote CVE shows the cost of not doing so. The July 2026 revision keeps the same architecture but tightens issuer validation, client registration, and refresh-token handling, so pin your SDKs to a spec revision and check which one your clients support [4] [7].

SOURCES

1 MCP Authorization Specification (draft) — modelcontextprotocol.io
https://modelcontextprotocol.io/specification/draft/basic/authorization

2 MCP Transports documentation — modelcontextprotocol.io
https://modelcontextprotocol.io/docs/concepts/transports

3 Understanding Authorization in MCP (official tutorial, 2026-07-28)
https://modelcontextprotocol.io/docs/2026-07-28/tutorials/security/authorization

4 Is that allowed? Authentication and authorization in Model Context Protocol — Stack Overflow Blog
https://stackoverflow.blog/2026/01/21/is-that-allowed-authentication-and-authorization-in-model-context-protocol/

5 MCP OAuth: How OAuth 2.1 Works in the Model Context Protocol — Prefect
https://www.prefect.io/resources/mcp-oauth

6 Understanding What is MCP Authentication and How It Works — TrueFoundry
https://www.truefoundry.com/blog/mcp-authentication

7 Diving Into the MCP Authorization Specification — Descope
https://www.descope.com/blog/post/mcp-auth-spec

8 MCP, OAuth 2.1, PKCE, and the Future of AI Authorization — Aembit
https://aembit.io/blog/mcp-oauth-2-1-pkce-and-the-future-of-ai-authorization/

9 Evolving OAuth Client Registration in the Model Context Protocol — MCP Blog
https://blog.modelcontextprotocol.io/posts/client_registration/

10 modelcontextprotocol/ext-auth — MCP Authorization Extensions (GitHub)
https://github.com/modelcontextprotocol/ext-auth

11 MCP OAuth 2.1: PKCE, Scopes & Token Management — Practical DevSecOps
https://www.practical-devsecops.com/mcp-oauth-2-1-implementation/

12 CVE-2025-6514 mcp-remote OS Command Injection — NVD
https://nvd.nist.gov/vuln/detail/CVE-2025-6514

Researched and written with deepseek/deepseek-v4-flash-0731, from the sources listed above. Figures are as reported by those sources. Exported November 29, 2026 from https://intel.innerpulse.net/doc/mcp-authentication-between-agentic-clients-and-servers-types-and-common, where the current version lives.