Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2026

R. Sharif
CyberSecAI Ltd

March 14, 2026

# MCPS: Cryptographic Security Layer for the Model Context Protocol
draft-sharif-mcps-secure-mcp-00

## Abstract

This document specifies MCPS (MCP Secure), a cryptographic security layer for the Model Context Protocol (MCP). MCPS adds agent identity verification, per-message signing, tool definition integrity, and replay protection to MCP communications without modifying the core protocol.

MCPS operates as an envelope around existing JSON-RPC messages. It introduces four primitives: (1) Agent Passports for cryptographic identity bound to a specific origin, (2) signed message envelopes for integrity and non-repudiation, (3) tool definition signatures covering the full tool object for detecting poisoning and tampering, and (4) nonce-plus-timestamp replay protection with transcript binding to prevent downgrade attacks.

The design is fully backward-compatible. MCPS-unaware clients and servers continue to function normally. MCPS-aware endpoints progressively negotiate security capabilities through trust levels L0 (no verification) through L4 (full mutual authentication with revocation checking).

All cryptographic operations use ECDSA P-256 (NIST FIPS 186-5). Signatures use IEEE P1363 fixed-length r||s encoding per RFC 7518 Section 3.4 with low-S normalization to prevent signature malleability. Canonical serialization uses JSON Canonicalization Scheme (JCS) per RFC 8785. The Trust Authority component is self-hostable with no external service dependency.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2026.

## Copyright Notice

Table of Contents

## 1. Introduction

The Model Context Protocol (MCP) defines a standard interface for connecting AI agents to external tools and data sources. MCP uses JSON-RPC 2.0 as its message format and supports multiple transport mechanisms including standard input/output (stdio), HTTP with Server-Sent Events (SSE), and WebSocket.

While MCP provides a robust framework for agent-tool interaction, the current specification lacks cryptographic guarantees for three critical security properties: identity, integrity, and tool authenticity.

### 1.1. The Problem

MCP in its current form has no built-in mechanism for:

Identity: There is no way for an MCP client or server to cryptographically verify the identity of its counterpart and bind that identity to a specific origin. A malicious actor can impersonate a legitimate MCP server, serving modified tool definitions or intercepting sensitive data.

Integrity: JSON-RPC messages between client and server are unsigned. There is no way for a recipient to verify that a message has not been tampered with in transit, or to prove after the fact that a particular message was sent by a particular party (non-repudiation).

Tool Authenticity: Tool definitions (name, description, input schema) are served unsigned. A compromised or malicious server can modify tool descriptions to inject instructions into agent prompts (tool poisoning), or silently change tool behavior between sessions.

These are not theoretical concerns:

- 41% of MCP servers have zero authentication (TapAuth research, scanning 518 production MCP servers).
- An independent scan of 39 AI agent frameworks against the OWASP Top 10 for Agentic Applications found that 13 frameworks had

no MCP security controls, 17 had partial controls, and only 9 implemented adequate protections.

- Tool poisoning attacks via description injection have been demonstrated in research contexts, where modified tool descriptions cause agents to execute unintended actions.

### 1.2. Scope and Limitations

This document addresses identity, integrity, and replay protection. It does NOT address:

Tool Safety Analysis: Signing a tool proves WHO authored it (provenance), not that the tool is safe to execute. A malicious server operator can sign their own poisoned tool definitions with a valid signature. Tool safety analysis (detecting malicious descriptions, sandboxing execution) is a separate concern that complements this work.

Authorization: This document does not replace OAuth-based access control. It provides a cryptographic identity layer that operates alongside authorization.

Confidentiality: Message encryption is out of scope. MCP-over-HTTP already provides transport-level confidentiality via TLS.

### 1.3. Relationship to Existing MCP Authorization

Existing MCP authorization work (OAuth 2.1-based) addresses session-level authentication, establishing that a client is authorized to connect to a server. This is necessary but not sufficient.

MCPS addresses a different layer: message-level and artifact-level cryptographic guarantees. The distinction is analogous to:

| Layer | HTTP Analogy | Addressed By |
|---|---|---|
| Session auth | OAuth bearer | Existing MCP auth |
| Transport | TLS | Existing (transport) |
| Message integrity | HTTP Signatures | MCPS |
| Artifact signing | Code signing | MCPS |
| Agent identity | Client certs | MCPS |

Note: MCP-over-HTTP already uses TLS for transport security. MCPS operates at the application layer above TLS, providing per-message non-repudiation and artifact integrity that TLS does not offer (TLS terminates at the connection, not the message).

MCPS complements existing OAuth-based authorization. A server MAY require both OAuth credentials (for access control) and MCPS signatures (for integrity and non-repudiation).

### 1.4. Relationship to DPoP (RFC 9449)

DPoP (Demonstrating Proof-of-Possession) [RFC9449] provides proof that the presenter of an OAuth access token possesses a particular key. MCPS's message signing serves a similar proof-of-possession function but extends beyond OAuth token binding:

1. Per-message signing (not per-session): Every JSON-RPC message is individually signed, enabling non-repudiation of specific tool calls and responses.

2. Tool definition signing: DPoP does not address artifact integrity.

3. Trust level negotiation: DPoP does not define progressive trust tiers.

Implementations MAY compose MCPS with DPoP. Specifically, the Agent Passport's public key MAY be the same key used in DPoP proofs, providing a unified identity across both OAuth authorization and message-level integrity.

### 1.5. Design Goals

The design of MCPS is guided by the following goals:

1. Backward compatibility: Non-MCPS endpoints MUST continue to function without modification.

2. Transport independence: MCPS MUST work over any MCP transport (stdio, HTTP SSE, WebSocket).

3. Progressive adoption: Organizations MUST be able to adopt MCPS incrementally through trust levels.

4. Self-sovereignty: No dependency on any centralized service MUST be required.

5. Standards alignment: Cryptographic operations MUST use established NIST and IETF standards.

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document:

Agent: A software entity that communicates over MCP, acting as either a client or server.

Agent Passport: A signed credential binding an ECDSA P-256 public key to an agent identity and a specific origin, issued by a Trust Authority or self-signed.

Trust Authority (TA): A service that issues, manages, and optionally revokes Agent Passports. Analogous to a Certificate Authority in the TLS/PKI ecosystem.

Trust Level: An integer (0-4) indicating the degree of cryptographic verification applied to an MCPS connection.

Signed Envelope: A JSON object wrapping a JSON-RPC message with an MCPS header containing a cryptographic signature, nonce, and timestamp.

Tool Signature: An ECDSA signature over a tool definition, binding the tool's name, description, and input schema to its author.

Canonical Serialization: JSON Canonicalization Scheme (JCS) per RFC 8785, used as the deterministic input to all signing operations.

Tool Hash: The SHA-256 digest of the JCS-canonicalized tool signing object (including name, description, input schema, and author origin), used for efficient change detection.

Pin Store: A persistent mapping from (server_origin, tool_name) tuples to tool hashes, maintained by clients for detecting tool definition changes.

Nonce Store: A data structure maintaining previously observed nonce values for replay detection.

Transcript Binding: A signed hash of the capability negotiation exchange, using ECDSA signatures (asymmetric), used to detect downgrade attacks. Note: Previous drafts used the term "Transcript MAC"; this has been corrected because MCPS uses asymmetric signatures, not symmetric MACs.

## 3. Protocol Overview

### 3.1. Architecture

MCPS operates as a security layer between the application logic and the MCP transport. The architecture consists of four components:

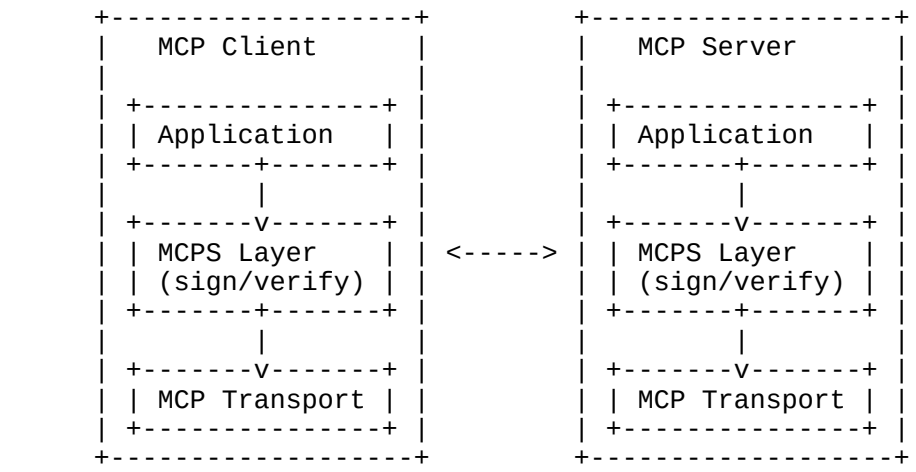
1. Agent Passports: Cryptographic identity credentials bound to an origin.

2. Message Signing: Per-message ECDSA signatures in an envelope.

3. Tool Signing: Signatures over complete tool definitions.

4. Trust Authority: Passport issuance, chain validation, and revocation.

The following diagram illustrates the message flow:




### 3.2. Envelope Model

MCPS wraps existing JSON-RPC messages in a signed envelope rather than modifying the JSON-RPC schema. The MCPS header is added as a top-level "mcps" field alongside the standard JSON-RPC fields. Non-MCPS implementations ignore the "mcps" field per standard JSON-RPC behavior.

This approach follows the precedent of HTTP Signatures [RFC9421], which add signatures alongside existing HTTP headers rather than modifying the HTTP message format.

## 3.3. Trust Levels

MCPS defines five trust levels for progressive security adoption:

| Level | Name | Requirements |
|---|---|---|
| L0 | None | No MCPS verification. Equivalent to current MCP behavior. Self-signed passports. |
| L1 | Signed | Messages are signed. Passport signed by ANY Trust Authority. |
| L2 | Verified | Messages are signed. Passport signed by a TA whose root key is in the verifier's trust store. |
| L3 | Strict | L2 plus tool definition signatures MUST be present and valid. TA has verified origin ownership. |
| L4 | Full | L3 plus mutual authentication and real-time revocation checking. TA has performed security audit. |

Servers SHOULD declare their minimum required trust level during capability negotiation (Section 9). Clients connecting at a trust level below the server minimum MUST be rejected with error code -33009 (MCPS_TRUST_LEVEL_INSUFFICIENT).

## 3.4. Trust Level Assignment Rules

Trust levels are NOT self-asserted by passport holders. They are constrained by the issuance process:

1. Self-signed passports are ALWAYS L0, regardless of any "trust_level" field value. Implementations MUST cap self-signed passports at L0.

2. L1-L2 require the passport to be signed by a Trust Authority whose public key can be verified through the issuer chain (Section 8.3).

3. L3 additionally requires the TA to have verified that the passport holder controls the "origin" URI (e.g., via HTTP-01 challenge, DNS-01 challenge, or documented manual verification).

4. L4 additionally requires the TA to have performed a security audit of the agent and to provide real-time revocation checking (Section 8.7).

An implementation receiving a passport claiming L4 but signed by an unknown TA MUST treat it as L0.

## 3.5. Cryptographic Primitives

All MCPS cryptographic operations use the following algorithms:

Key Algorithm: ECDSA with the P-256 curve (NIST FIPS 186-5 [FIPS186-5], deterministic signatures per [RFC6979]).

Hash Algorithm: SHA-256 (NIST FIPS 180-4).

Signature Format: IEEE P1363 fixed-length r||s encoding (exactly 64 bytes for P-256), per [RFC7518] Section 3.4. All signatures MUST use low-S normalization (see Section 3.7).

Signature Encoding: Base64 without padding [RFC4648].

Key Format: JSON Web Key (JWK) [RFC7517].

Canonical Serialization: JSON Canonicalization Scheme (JCS) [RFC8785].

Why JCS (RFC 8785): Canonical JSON serialization is critical for cross-implementation signature verification. Different language runtimes produce different byte representations for the same logical JSON value (e.g., Node.js JSON.stringify({a:1.0}) produces {"a":1} while Python's json.dumps({a:1.0}) produces {"a": 1.0}). RFC 8785 defines a deterministic serialization that guarantees identical bytes across all implementations.

The choice of P-256 provides 128-bit security strength, broad platform support (Web Crypto API, OpenSSL, Java KeyStore), and hardware security module (HSM) compatibility. FIPS 186-5 is referenced (not the superseded FIPS 186-4).

Note: RFC 6979 specifies deterministic ECDSA nonce generation. Implementations using HSMs that employ internal random number generation for nonce generation are compliant provided the HSM generates nonces with sufficient entropy per FIPS 186-5.

Implementations MAY support additional algorithms in future versions via algorithm negotiation in the "mcps" capability.

### 3.6. Signature Format Details

Signatures use the IEEE P1363 fixed-length format: the 32-byte big-endian encoding of r concatenated with the 32-byte big-endian encoding of s, for a total of exactly 64 bytes. This format is unambiguous (unlike variable-length DER encoding) and mandated by RFC 7518 Section 3.4 for JWA/JWS compatibility.

DER encoding produces variable-length signatures (typically 70-72 bytes for P-256). IEEE P1363 produces fixed-length signatures (exactly 64 bytes for P-256). Fixed-length format:

1. Eliminates ambiguity in signature parsing across implementations.

2. Is required by RFC 7518 Section 3.4 (JWA) and widely supported.

3. Simplifies cross-platform interoperability (no DER parsing needed).

### 3.7. Low-S Normalization

Given a valid ECDSA signature (r, s), the signature (r, n-s) is also valid for the same message and key. This malleability can be exploited to bypass nonce-based replay detection if the nonce store keys on full message bytes rather than the nonce string.

To eliminate this attack:

1. All signatures MUST satisfy s <= n/2, where n is the order

of the P-256 curve.

2. If the raw ECDSA signing operation produces s > n/2, the signer MUST replace s with n - s before encoding.

3. Verifiers SHOULD normalize incoming signatures (replace s with n - s if s > n/2) before verification for interoperability with implementations that do not perform low-S normalization on signing.

The P-256 curve order n is:

```
n = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84
    F3B9CAC2FC632551
```

The half-order n/2 is:

```
n/2 = 0x7FFFFFFF800000007FFFFFFFFFFFFFFFDE737D56D38BCF42
      79DCE5617E3192A8
```

Nonce stores MUST key on the nonce string value, NOT on full message bytes, to prevent signature malleability from bypassing replay detection (see Section 7.3).

# 4. Agent Passports

An Agent Passport is a signed credential that binds an ECDSA P-256 key pair to an agent identity and a specific origin.

## 4.1. Passport Structure

A passport is represented as a JSON object with the following top-level structure:

```
{
  "mcps_version": "1.0",
  "passport": {
    "id": "ap_<uuid-v4>",
    "agent_name": "<string>",
    "agent_version": "<string (semver)>",
    "issuer": "<string>",
    "origin": "<string (URI)>",
    "issued_at": "<string (ISO 8601 UTC)>",
    "expires_at": "<string (ISO 8601 UTC)>",
    "public_key": {
      "kty": "EC",
      "crv": "P-256",
      "x": "<string (base64url)>",
      "y": "<string (base64url)>"
    },
    "capabilities": ["<string>"],
    "trust_level": 0,
    "issuer_chain": [
      "<string (base64-encoded JSON)>"
    ],
    "key_rotation": {
      "previous_key_hash": "<string (hex SHA-256)>",
      "rotated_at": "<string (ISO 8601 UTC)>"
    }
  },
  "signature": "<string (base64, IEEE P1363)>"
}
```

## 4.2. Passport Fields

The following fields are defined for Agent Passports:

mcps_version: REQUIRED. String. The MCPS protocol version. MUST be "1.0" for this specification.

passport.id: REQUIRED. String. A unique identifier for this passport. MUST begin with the prefix "ap_" followed by a UUID v4 value.

passport.agent_name: REQUIRED. String. A human-readable name for the agent.

passport.agent_version: REQUIRED. String. The semantic version [SEMVER] of the agent software.

passport.issuer: REQUIRED. String. An identifier for the Trust Authority that issued this passport. For self-signed passports, this MUST be the string "self".

passport.origin: REQUIRED. String. The authorized origin URI this passport is bound to (e.g., "https://api.example.com:443"). Verifiers MUST reject passports presented to a different origin. See Section 4.4.

passport.issued_at: REQUIRED. String. The issuance timestamp in ISO 8601 UTC format (e.g., "2026-03-13T12:00:00Z").

passport.expires_at: REQUIRED. String. The expiry timestamp in ISO 8601 UTC format.

passport.public_key: REQUIRED. Object. The agent's public key in JWK format. See Section 4.7 for requirements.

passport.capabilities: OPTIONAL. Array of strings. A list of capability identifiers granted to this agent by the Trust Authority. Maximum 64 entries (see Section 4.3).

passport.trust_level: OPTIONAL. Integer (0-4). The trust level assigned by the issuing Trust Authority. Default value is 0. See Section 3.4 for assignment rules.

passport.issuer_chain: OPTIONAL. Array of strings. Chain of base64-encoded JSON intermediate Trust Authority passports. Each entry is a complete signed intermediate TA passport. Maximum depth: 5 entries (see Section 4.3). See Section 8.3 for format and verification.

passport.key_rotation: OPTIONAL. Object. Key rotation metadata for compromise recovery. See Section 4.5.

signature: REQUIRED. String. A base64-encoded IEEE P1363 ECDSA signature (exactly 64 bytes before encoding) computed over the JCS-canonicalized (RFC 8785) "passport" object, with low-S normalization (Section 3.7). For Trust Authority-issued passports, this signature is produced using the Trust Authority's private key. For self-signed passports, this signature is produced using the agent's own private key.

### 4.3. Passport Size Limits

To prevent denial-of-service via oversized passports, the following limits are defined:

MAX_PASSPORT_BYTES: 8192. The maximum size in bytes of a JCS-canonicalized passport. Implementations MUST reject passports exceeding this limit with error code -33013 (MCPS_PASSPORT_TOO_LARGE).

MAX_ISSUER_CHAIN_DEPTH: 5. The maximum number of entries in the "issuer_chain" array. Implementations MUST reject passports with chains exceeding this depth with error code -33014 (MCPS_CHAIN_TOO_DEEP).

MAX_CAPABILITIES: 64. The maximum number of entries in the "capabilities" array.

Implementations MUST enforce all three limits during passport validation.

### 4.4. Origin Binding

The "origin" field binds a passport to a specific server URI. This prevents passport reuse across domains:

1. When a server presents a passport, the client MUST verify that the passport's "origin" matches the URI it connected to.

2. When a client presents a passport, the server MUST verify that the passport's "origin" matches its own published URI.

3. Origin comparison MUST use scheme + authority (host + port) per RFC 6454 [RFC6454].

This prevents an attacker from obtaining a valid passport for their own server and presenting it when impersonating a different server.

If origin verification fails, the implementation MUST reject the passport with error code -33011 (MCPS_ORIGIN_MISMATCH).

### 4.5. Key Rotation and Compromise Recovery

Passports support key rotation for both routine rotation and compromise recovery:

Routine Rotation: An agent generates a new key pair, requests a new passport from the Trust Authority, and the old passport is revoked after confirmation that the new passport is in active use.

Compromise Recovery: When a key is suspected compromised, the agent generates a new key pair and includes a "key_rotation" field linking to the previous key via its SHA-256 hash. This enables Trust Authorities to verify the rotation request and revoke all passports associated with the compromised key.

The "key_rotation" object contains:

previous_key_hash: REQUIRED (within key_rotation). String. SHA-256 hash (hex-encoded) of the PEM-encoded previous public key.

rotated_at: REQUIRED (within key_rotation). String. ISO 8601 UTC timestamp indicating when the rotation occurred.

Trust Authorities SHOULD revoke the old passport upon issuing a new one with "key_rotation". Implementations SHOULD support a configurable grace period where both old and new passports are accepted to allow rolling deployments.

### 4.6. Passport Lifecycle

The lifecycle of an Agent Passport consists of the following

stages:

Generation: The agent generates an ECDSA P-256 key pair locally. The private key MUST be stored securely and MUST NOT be transmitted.

Issuance: The agent submits its public key and requested origin to a Trust Authority. The Trust Authority validates the agent's ownership of the origin (e.g., via HTTP challenge, DNS TXT record, or manual verification), assigns capabilities and a trust level, and signs the passport with the Trust Authority's private key.

Self-Signing: An agent MAY self-sign its passport. Self-signed passports are ALWAYS Trust Level L0 regardless of any "trust_level" field value. Implementations MUST enforce this. Self-signed passports MUST set the "issuer" field to "self".

Verification: A receiving party verifies the passport signature against the Trust Authority's public key (for TA-issued passports) or against the agent's own public key embedded in the passport (for self-signed passports). For self-signed passports, the trust level is capped at L0.

Expiry: Passports MUST be rejected after the "expires_at" timestamp. Implementations SHOULD allow a configurable clock skew tolerance. The default clock skew tolerance SHOULD be 60 seconds.

Key Rotation: An agent rotates its key by generating a new key pair and requesting a new passport from the Trust Authority. For compromise recovery, the "key_rotation" field (Section 4.5) links the new passport to the old key. The old passport continues to be valid until its "expires_at" or until explicitly revoked.

Revocation: Trust Authorities MAY publish revocation information for passports they have issued. See Section 8.7.

### 4.7. JWK Key Requirements

The "public_key" field of an Agent Passport MUST conform to the following requirements:

- The "kty" field MUST be "EC".
- The "crv" field MUST be "P-256".
- The "x" and "y" fields MUST contain the base64url-encoded coordinates of the public key point on the P-256 curve, without padding.
- The "d" (private key) parameter MUST NOT be present.
- Implementations MUST validate that the public key point lies on the P-256 curve before using it for signature verification.

## 5. Message Signing

Every JSON-RPC message exchanged between MCPS-aware endpoints is wrapped in a signed envelope. The envelope adds an "mcps" field to the top level of the JSON-RPC message.

### 5.1. Envelope Structure

A signed message envelope has the following structure:

```
{
  "mcps": {
    "version": "1.0",
    "passport_id": "ap_<uuid>",
    "timestamp": "<string (ISO 8601 UTC)>",
    "nonce": "<string (hex, 16 random bytes)>",
    "signature": "<string (base64, IEEE P1363)>"
  },
  "jsonrpc": "2.0",
  "method": "<string>",
  "params": { ... },
  "id": 1
}
```

The "mcps" field contains the following sub-fields:

version: REQUIRED. String. The MCPS protocol version. MUST be "1.0".

passport_id: REQUIRED. String. The identifier of the Agent Passport used to sign this message.

timestamp: REQUIRED. String. The current UTC time in ISO 8601 format at which the message was signed.

nonce: REQUIRED. String. A hex-encoded string of 16 cryptographically random bytes, unique to this message, used for replay protection.

signature: REQUIRED. String. A base64-encoded IEEE P1363 ECDSA signature (exactly 64 bytes before encoding, with low-S normalization) over the signing payload (see Section 5.2).

### 5.2. Signing Process

To sign a JSON-RPC message, an MCPS implementation MUST perform the following steps in order:

1. Construct the JSON-RPC message without the "mcps" field.

2. Generate 16 cryptographically random bytes, hex-encode them as the nonce.

3. Record the current UTC timestamp in ISO 8601 format.

4. Compute the message hash:

   ```
   message_hash = hex(SHA-256(JCS(JSON-RPC message)))
   ```

   where JCS is the RFC 8785 canonicalization of the complete JSON-RPC message (without the "mcps" field).

5. Construct the signing payload by JCS-canonicalizing [RFC8785] the following JSON object:

   ```
   {
     "message_hash": "<hex SHA-256>",
     "nonce": "<nonce>",
     "passport_id": "<passport_id>",
     "timestamp": "<timestamp>"
   }
   ```

   The keys in this object are sorted lexicographically per RFC 8785. The "message_hash" value is the hexadecimal-

encoded SHA-256 hash of the JCS-canonicalized JSON-RPC message.

6. Sign the UTF-8 encoding of the JCS-canonicalized signing payload using ECDSA P-256 with the agent's private key per [RFC6979] (deterministic signature generation), producing an IEEE P1363 signature with low-S normalization (Section 3.7).

7. Base64-encode the 64-byte IEEE P1363 signature without padding per [RFC4648].

8. Attach the "mcps" object to the JSON-RPC message.

Design note: The signing payload uses "message_hash" (the SHA-256 hash of the JCS-canonicalized message) rather than embedding the JCS string directly. Earlier drafts used "jsonrpc_message: JCS(message)" in the signing payload, nesting one JCS string inside another JCS object. This creates fragility: the inner JCS string must be JSON-escaped when embedded in the outer object, and different JSON libraries handle this escaping differently. Using a hash eliminates this ambiguity, is more robust, and is more efficient.

### 5.3. TLS Channel Binding

Implementations MAY include a TLS channel binding token (per [RFC9266], tls-exporter type) in the signing payload to bind MCPS signatures to the TLS session:

```
{
  "channel_binding": "<base64 tls-exporter binding>",
  "message_hash": "<hex SHA-256>",
  "nonce": "<nonce>",
  "passport_id": "<passport_id>",
  "timestamp": "<timestamp>"
}
```

When present, the verifier MUST verify that the "channel_binding" value matches the tls-exporter binding of the current TLS session. This prevents relay attacks where a man-in-the-middle with a valid TLS certificate for a different domain forwards signed messages between the legitimate client and server.

The "channel_binding" field is OPTIONAL. When absent, the signing payload uses the four-field format defined in Section 5.2.

### 5.4. Verification Process

To verify a signed message, an MCPS implementation MUST perform the following steps in order:

1. Extract the "mcps" field from the received message.

2. Verify that all required "mcps" sub-fields are present.

3. Verify that the "timestamp" is within the acceptable window (see Section 7.2). If the timestamp is outside the window, reject the message with error code -33006.

4. Verify that the "nonce" has not been previously observed (see Section 7.1). If the nonce is a duplicate, reject the message with error code -33005.

5. Look up the Agent Passport by "passport_id". Verify that the passport is valid, has not expired, is bound to the correct

origin (Section 4.4), and (for L4) has not been revoked. If the passport format is invalid, reject with error code -33001. If expired, reject with -33002. If revoked, reject with -33003. If origin mismatches, reject with -33011.

6. Determine the effective trust level using the rules in Section 3.4. If the effective trust level is below the server's minimum, reject with -33009.

7. Reconstruct the signing payload: compute the message hash from the received JSON-RPC message (with the "mcps" field removed), combine with received nonce, passport_id, and timestamp. If "channel_binding" is present, include it.

8. JCS-canonicalize the reconstructed signing payload.

9. Verify the IEEE P1363 ECDSA signature against the public key from the Agent Passport. Normalize s to low-S before verification (Section 3.7). If verification fails, reject with error code -33004.

10. Store the nonce (keyed on the nonce string) in the nonce store to prevent future replay.

If any step fails, the message MUST be rejected with the appropriate error code defined in Section 10.

### 5.5. Canonical Serialization (RFC 8785)

All MCPS signing operations use the JSON Canonicalization Scheme (JCS) defined in RFC 8785 [RFC8785] for deterministic serialization. This ensures that implementations in different programming languages produce identical byte sequences for the same logical JSON value.

Key JCS requirements relevant to MCPS:

1. Object keys MUST be sorted lexicographically by their Unicode code point values.

2. No extraneous whitespace (spaces, tabs, newlines) SHALL be present between tokens.

3. String values MUST use the shortest possible JSON escape sequences.

4. Numbers MUST be serialized according to ECMAScript number serialization: no unnecessary leading zeros, no trailing decimal zeros, integer-valued floats serialized as integers (1.0 becomes 1).

5. Negative zero (-0) MUST be serialized as 0.

Implementers MUST test their JCS implementation against the RFC 8785 test vectors to ensure cross-platform compatibility. In particular, number serialization differences between JavaScript and Python runtimes are a common source of interoperability failures.

## 6. Tool Definition Signing

Tool definitions MAY be signed by their author to prevent poisoning attacks and to detect unauthorized modifications.

### 6.1. Signed Tool Structure

A signed tool definition has the following structure:

```
{
  "tool": {
    "name": "read_file",
    "description": "Read contents of a file at the
                    given path",
    "inputSchema": {
      "type": "object",
      "properties": {
        "path": {
          "type": "string",
          "description": "File path to read"
        }
      },
      "required": ["path"]
    }
  },
  "tool_signature": {
    "author_passport_id": "ap_<uuid>",
    "author_origin": "https://author.example.com",
    "signed_at": "<string (ISO 8601 UTC)>",
    "signature": "<string (base64, IEEE P1363)>",
    "tool_hash": "<string (hex-encoded SHA-256)>"
  }
}
```

The "tool_signature" field contains:

author_passport_id: REQUIRED. String. The passport identifier of the tool author.

author_origin: OPTIONAL. String. The origin URI of the tool author. When present, enables tool-server origin binding (see Section 6.2).

signed_at: REQUIRED. String. ISO 8601 UTC timestamp of when the tool definition was signed.

signature: REQUIRED. String. Base64-encoded IEEE P1363 ECDSA signature (with low-S normalization) over the JCS-canonicalized tool signing object (see Section 6.3).

tool_hash: REQUIRED. String. Hexadecimal-encoded SHA-256 digest of the JCS-canonicalized tool signing object. This enables efficient change detection without re-verifying the full signature.

### 6.2. Author Origin Binding

The "author_origin" field in the tool signature binds the signed tool to a specific serving origin. This prevents tool relaying attacks where a validly signed tool is served from an unauthorized server.

When "author_origin" is present in the tool signature:

1. Clients SHOULD verify that the "author_origin" matches the origin of the server currently serving the tool.

2. If the origin does not match, the client SHOULD alert the user or reject the tool according to its configured policy.

When "author_origin" is null or absent, origin binding is not enforced. This supports cases where a tool author intentionally distributes tools for use on third-party servers.

### 6.3. Signing Process

To sign a tool definition:

1. Construct the signing object:

```
{
  "author_origin": "<author's origin URI or null>",
  "description": "<tool description>",
  "inputSchema": { ... },
  "name": "<tool name>"
}
```

   Note: The signing object keys are ordered lexicographically per RFC 8785 conventions.

2. JCS-canonicalize [RFC8785] the signing object.

3. Compute the SHA-256 hash of the JCS-canonicalized output. Record this as the "tool_hash" field (hex-encoded).

4. Sign the JCS-canonicalized output with the tool author's private key using ECDSA P-256, producing an IEEE P1363 signature with low-S normalization (Section 3.7).

Important: The hash covers the entire signing object, including the "description" field. This is critical because tool poisoning attacks primarily target the description field (injecting instructions into agent prompts), not the inputSchema. A hash covering only the schema would not detect description-based poisoning attacks.

### 6.4. Verification and Pinning

Clients SHOULD maintain a persistent pin store mapping (server_origin, tool_name) tuples to the last known tool_hash value. On each connection:

1. Verify the tool signature against the author's Agent Passport public key.

2. Verify the author's passport is bound to the correct origin (Section 4.4).

3. If "author_origin" is present in the tool signature, verify it matches the origin of the server serving the tool (Section 6.2).

4. Compare the received "tool_hash" against the pinned value in the local pin store.

5. If no pinned value exists, store the current tool_hash (trust on first use).

6. If the hash has changed since the last pinned value, the client MUST take action according to its configured policy (see Section 6.5).

### 6.5. Rug Pull Detection

When a tool's tool_hash changes between sessions, a "rug pull" may have occurred: the tool's name, description, or schema has been modified without the user's knowledge.

Implementations MUST support at least one of the following

policies for handling tool hash changes:

Alert: Notify the user that the tool definition has changed and request confirmation before proceeding.

Reject: Reject the tool and return error code -33008.

Accept: Accept the new definition and update the pin store. This policy is NOT RECOMMENDED for Trust Levels L3 and L4.

The default policy SHOULD be "alert" for Trust Levels L1 and L2, and "reject" for Trust Levels L3 and L4.

## 7. Replay Protection

MCPS uses a dual mechanism for replay protection combining nonce uniqueness with timestamp windowing.

### 7.1. Nonce Requirements

Each signed message MUST include a nonce in the "mcps" header. The nonce MUST be generated as 16 cryptographically random bytes, hex-encoded (producing a 32-character hexadecimal string).

Recipients MUST maintain a nonce store and MUST reject any message whose nonce has been previously observed. A duplicate nonce MUST result in error code -33005.

### 7.2. Timestamp Window

Messages with timestamps older than the configurable timestamp window MUST be rejected with error code -33006. The default timestamp window is 300 seconds (5 minutes).

Implementations SHOULD allow the timestamp window to be configured. The minimum permitted timestamp window is 30 seconds. The maximum permitted timestamp window is 3600 seconds (1 hour).

Implementations SHOULD allow a configurable clock skew tolerance to account for imprecise system clocks. The default clock skew tolerance is 60 seconds. The effective acceptance window is therefore (timestamp_window + clock_skew_tolerance).

### 7.3. Nonce Store Requirements

Implementations MUST store nonces for at least the duration of the timestamp window plus the clock skew tolerance.

Nonce stores MUST key on the nonce string value, NOT on full message bytes. This is critical for preventing ECDSA signature malleability from bypassing replay detection: given a valid signature (r, s), the signature (r, n-s) is also valid for the same message, producing different message bytes but containing the same nonce. Keying on the nonce string ensures that both the original and the malleable variant are detected as replays.

Implementations SHOULD periodically garbage-collect nonces whose associated timestamps fall outside the acceptance window.

The nonce store MAY be implemented as:

- An in-memory data structure for single-process deployments.
- A shared data store (e.g., Redis, database table) for distributed deployments.

- A Bloom filter for memory-constrained environments, accepting a configurable false-positive rate (which results in legitimate messages being incorrectly rejected as replays).

## 8. Trust Authority

A Trust Authority (TA) is a service that issues and manages Agent Passports. The TA role is analogous to a Certificate Authority in the TLS/PKI ecosystem.

### 8.1. Self-Hosting

Any organization MAY operate its own Trust Authority. A Trust Authority is defined by:

1. An ECDSA P-256 key pair (the TA's signing key).
2. A public key distribution mechanism (see Section 8.2).
3. For L3+ passports: a documented origin verification process.
4. For L4 passports: a revocation endpoint and audit process.

There is no requirement to use any specific Trust Authority service. Implementations MUST support configuring one or more custom TA public keys as trust anchors.

### 8.2. Public Key Distribution

Trust Authority public keys MAY be distributed via any of the following mechanisms:

- An HTTPS endpoint serving the TA's public key in JWK format.
- A JWK Set endpoint conforming to [RFC7517].
- Static configuration in the client or server application.

Implementations MUST support at least one of these mechanisms. HTTPS-based distribution SHOULD use TLS with certificate validation.

### 8.3. Issuer Chains

Trust delegation follows X.509-style chains:

1. A root TA signs intermediate TA passports.
2. Intermediate TAs sign agent passports.
3. Verifiers walk the chain from agent passport to a root key in their configured trust store.
4. If no chain leads to a trusted root, the passport MUST be treated as L0.

The "issuer_chain" field in the passport contains a chain of base64-encoded JSON intermediate Trust Authority passports, ordered from the agent's immediate issuer to the root. Each entry is a complete signed intermediate TA passport. The chain MUST NOT exceed MAX_ISSUER_CHAIN_DEPTH (5) entries.

### 8.4. Issuer Chain Entry Format

Each entry in the "issuer_chain" array is a base64-encoded JSON string representing a complete signed intermediate Trust Authority

passport. When decoded from base64, each entry contains:

```
{
  "mcps_version": "1.0",
  "passport_id": "ap_<intermediate-uuid>",
  "agent": {
    "name": "Intermediate TA Name",
    "version": "1.0.0",
    "capabilities": []
  },
  "public_key": {
    "kty": "EC",
    "crv": "P-256",
    "x": "<base64url>",
    "y": "<base64url>"
  },
  "origin": "https://intermediate-ta.example.com",
  "trust_level": 2,
  "issued_at": "<ISO 8601 UTC>",
  "expires_at": "<ISO 8601 UTC>",
  "issuer": "<parent-ta-id>",
  "issuer_chain": [],
  "signature": "<base64, IEEE P1363>"
}
```

The intermediate passport contains its own signature, public key, and issuer fields. The "signature" is produced by the parent TA (the TA that issued this intermediate), and the "public_key" is the intermediate TA's own key which was used to sign the next passport down the chain.

### 8.5. Chain Verification Algorithm

Verifiers MUST implement the following algorithm to validate an issuer chain:

1. Let P be the agent passport to verify.

2. Check that the length of P.issuer_chain does not exceed MAX_ISSUER_CHAIN_DEPTH (5). If it does, reject with error code -33014 (MCPS_CHAIN_TOO_DEEP).

3. If P.issuer is directly present in the verifier's trust store, verify P.signature against that TA's public key. If verification succeeds, the chain is valid. Done.

4. If P.issuer_chain is empty or absent, treat the passport as L0. Done.

5. Set i = 0 (index into issuer_chain).

6. Decode issuer_chain[i] from base64, parse as JSON. Let IC = the decoded intermediate passport.

7. Verify P.signature against IC.public_key (the intermediate TA's public key). If verification fails, treat the passport as L0.

8. Verify that IC has not expired (check IC.expires_at).

9. If IC.issuer is in the verifier's trust store, verify IC.signature against the trust store's public key for that issuer. If verification succeeds, the chain is valid. Done.

10. If i+1 < length of issuer_chain, set P = IC, increment i,

and go to step 6.

11. If the chain is exhausted without reaching a trusted root, treat the passport as L0.

At each step, signature verification MUST use IEEE P1363 format with low-S normalization (Section 3.7).

### 8.6. Multi-TA Deployments

Implementations SHOULD support multiple Trust Authorities simultaneously:

1. The trust store MAY contain public keys from multiple TAs.

2. When verifying an issuer chain, the verifier checks each configured TA in the trust store for a matching issuer identifier.

3. When checking revocation, implementations SHOULD try each configured TA that could have issued the passport until a reachable one responds.

4. Cross-TA trust (where TA-A trusts passports issued by TA-B) is established explicitly by including TA-B's signing key in TA-A's trust store. There is no implicit cross-TA trust.

5. Implementations MUST NOT assume any trust relationship between TAs unless explicitly configured.

### 8.7. Revocation

Trust Authorities providing L4 passports MUST publish signed revocation information using one or both of the following mechanisms:

#### 8.7.1. Revocation List (CRL-style)

An HTTP GET endpoint returning a signed JSON object:

```
{
  "revoked": ["ap_<uuid>", "ap_<uuid>"],
  "updated_at": "<string (ISO 8601 UTC)>",
  "signature": "<string (base64, IEEE P1363)>"
}
```

The "revoked" array contains the passport identifiers of all revoked passports. The "updated_at" field indicates when the list was last modified.

The "signature" field MUST contain a base64-encoded IEEE P1363 ECDSA signature (with low-S normalization) computed over the JCS-canonicalized JSON object containing only the "revoked" and "updated_at" fields. This signature MUST be produced by the issuing Trust Authority's private key.

Clients SHOULD cache the revocation list and refresh it periodically. The refresh interval SHOULD be configurable with a default of 300 seconds.

#### 8.7.2. Per-Passport Status Check (OCSP-style)

An HTTP GET endpoint at the path "/{passport_id}/status" returning a signed JSON object:

```
{
```

```
  "passport_id": "ap_<uuid>",
  "status": "<string>",
  "checked_at": "<string (ISO 8601 UTC)>",
  "signature": "<string (base64, IEEE P1363)>"
}
```

The "status" field MUST be one of:

- "active" - The passport is valid and has not been revoked.
- "revoked" - The passport has been revoked by the Trust Authority.
- "expired" - The passport has passed its "expires_at" timestamp.
- "unknown" - The passport identifier is not recognized by this Trust Authority.

The "signature" field MUST contain a base64-encoded IEEE P1363 ECDSA signature (with low-S normalization) computed over the JCS-canonicalized JSON object containing only the "passport_id", "status", and "checked_at" fields. This signature MUST be produced by the issuing Trust Authority's private key.

Revocation responses MUST be signed by the issuing Trust Authority. Unsigned revocation responses MUST be rejected.

### 8.8. Revocation Endpoint Discovery

The revocation endpoint URL MUST NOT be supplied by the party being verified. Instead:

1. The revocation endpoint is discovered from the Trust Authority's published metadata (JWK Set document or well-known endpoint).
2. Alternatively, the verifier configures the revocation endpoint out-of-band when adding the TA to its trust store.

This prevents an attacker from directing revocation checks to a server they control, which could return fraudulent "active" responses for revoked passports.

### 8.9. Revocation Failure Policy

If the revocation endpoint is unreachable, implementations MUST behave as follows:

For Trust Level L4: Implementations MUST fail-closed (reject the passport). Revocation checking is mandatory at L4; proceeding without it violates the L4 guarantee.

For L1-L3 with optional revocation configured: If a deployment enables optional revocation checking at L1-L3 and the endpoint is unreachable, implementations MUST fail-closed (reject the passport). If the operator wants fail-open behavior, they SHOULD disable revocation checking rather than relying on silent fallback.

For L1-L3 without revocation configured: Revocation checking is skipped entirely (default for L1-L3).

Implementations SHOULD cache the last successful revocation response and apply a configurable maximum cache age (default: 300 seconds). A valid cached response MAY be used when the endpoint is temporarily unreachable, provided the cache has not

expired.

## 9. Capability Negotiation

MCPS capabilities are negotiated during the MCP "initialize" handshake, using the standard MCP capability extension mechanism.

### 9.1. Version Negotiation

The "mcps" capability includes a "version" field for protocol version negotiation. Implementations MUST support version negotiation:

1. The client announces its supported MCPS version(s) in the "mcps.version" field. The value MAY be a single version string (e.g., "1.0") or an array of version strings (e.g., ["1.0", "2.0"]) for forward compatibility.

2. The server responds with the highest mutually supported version in its "mcps.version" field (always a single string).

3. If no mutually supported version exists, the server MUST reject the connection with error code -33015 (MCPS_VERSION_MISMATCH).

4. After version negotiation, all subsequent MCPS operations MUST use the negotiated version.

This mechanism enables forward compatibility as new MCPS protocol versions are introduced.

### 9.2. Client Announcement

An MCPS-capable client announces its support by including an "mcps" object within the "capabilities" field of the "initialize" request:

```
{
  "jsonrpc": "2.0",
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "mcps": {
        "version": "1.0",
        "trust_level": 2,
        "passport": { ... }
      }
    },
    "clientInfo": {
      "name": "my-agent",
      "version": "1.0.0"
    }
  },
  "id": 1
}
```

The "mcps" capability object contains:

version: REQUIRED. The MCPS protocol version(s) supported. String or array of strings.

trust_level: REQUIRED. The maximum trust level the client supports.

passport: REQUIRED. The client’s Agent Passport.

### 9.3. Server Response

An MCPS-capable server responds with its own MCPS capabilities:

```
{
  "jsonrpc": "2.0",
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "mcps": {
        "version": "1.0",
        "min_trust_level": 2,
        "passport": { ... }
      }
    },
    "serverInfo": {
      "name": "secure-server",
      "version": "2.0.0"
    }
  },
  "id": 1
}
```

The server's "mcps" capability object contains:

version: REQUIRED. The negotiated MCPS protocol version (single string).

min_trust_level: REQUIRED. The minimum trust level required for connections to this server.

passport: REQUIRED. The server's Agent Passport.

Note: The server's capability response MUST NOT include a revocation_endpoint field. Revocation endpoints are discovered via Trust Authority metadata (Section 8.8), not supplied by the party being verified.

### 9.4. Negotiation Failure

If the client's trust level is below the server's "min_trust_level", the server MUST reject the connection with JSON-RPC error code -33009 (MCPS_TRUST_LEVEL_INSUFFICIENT).

If the MCPS versions are incompatible (no mutual version), the server MUST reject the connection with error code -33015 (MCPS_VERSION_MISMATCH).

If the server does not include an "mcps" capability in its response, the client MUST operate without MCPS verification (Trust Level L0) or disconnect, depending on its configured minimum trust level requirement.

### 9.5. Transcript Binding (Anti-Downgrade)

After capability negotiation, both parties MUST exchange a transcript binding to confirm they agree on the negotiated security parameters. This prevents an active attacker from stripping the "mcps" capability during the handshake (downgrade attack).

Note: This mechanism uses ECDSA signatures (asymmetric), not MAC (symmetric). Previous drafts used the term "transcript MAC"; this has been corrected to "transcript binding" for accuracy.

Transcript scope: The transcript covers the "params" field from the client's initialize request and the "result" field from the server's initialize response -- NOT the full JSON-RPC envelope (which includes the jsonrpc version string, method name, and message id that are not security-relevant).

The procedure is as follows:

1. Both parties compute:

   ```
   transcript_hash = SHA-256(
     JCS(client_initialize_params) ||
     JCS(server_initialize_result)
   )
   ```

   where "||" denotes byte concatenation, and JCS is the RFC 8785 canonicalization. "client_initialize_params" is the value of the "params" field from the client's initialize request. "server_initialize_result" is the value of the "result" field from the server's initialize response.

2. Each party signs the transcript_hash with their private key using ECDSA P-256, producing an IEEE P1363 signature with low-S normalization (Section 3.7).

3. The signed transcript is exchanged as the first signed message after the "initialize" handshake, using the method "mcps/transcript_verify":

   ```
   {
     "mcps": {
       "version": "1.0",
       "passport_id": "ap_<uuid>",
       "timestamp": "<ISO 8601 UTC>",
       "nonce": "<hex, 16 random bytes>",
       "signature": "<base64, IEEE P1363>"
     },
     "jsonrpc": "2.0",
     "method": "mcps/transcript_verify",
     "params": {
       "transcript_hash": "<hex SHA-256>",
       "transcript_signature": "<base64, IEEE P1363>"
     },
     "id": 2
   }
   ```

4. Each party verifies the other's transcript binding by:
   a. Recomputing the transcript_hash from their own records.
   b. Verifying the received transcript_hash matches.
   c. Verifying the transcript_signature against the other party's passport public key.

5. If the transcript bindings do not match, the connection MUST be terminated with error code -33012 (MCPS_TRANSCRIPT_MISMATCH).

This mechanism is analogous to the TLS Finished message, which prevents downgrade attacks by binding both parties to the same view of the negotiated parameters.

## 10. Error Codes

MCPS defines the following JSON-RPC error codes in the -33xxx range, avoiding collision with JSON-RPC's reserved implementation-defined range (-32000 to -32099). Each error has both a numeric code and a string code for programmatic handling:

| Code | String | Name |
|---|---|---|
| -33001 | MCPS-001 | MCPS_INVALID_PASSPORT<br>Passport format invalid. |
| -33002 | MCPS-002 | MCPS_PASSPORT_EXPIRED<br>Passport has expired. |
| -33003 | MCPS-003 | MCPS_PASSPORT_REVOKED<br>Passport has been revoked. |
| -33004 | MCPS-004 | MCPS_INVALID_SIGNATURE<br>Message signature<br>verification failed. |
| -33005 | MCPS-005 | MCPS_REPLAY_DETECTED<br>Duplicate nonce detected. |
| -33006 | MCPS-006 | MCPS_TIMESTAMP_EXPIRED<br>Message timestamp outside<br>acceptable window. |
| -33007 | MCPS-007 | MCPS_AUTHORITY_UNREACHABLE<br>Trust Authority unreachable<br>(fail-closed). |
| -33008 | MCPS-008 | MCPS_TOOL_INTEGRITY_FAILED<br>Tool definition signature<br>invalid or hash changed. |
| -33009 | MCPS-009 | MCPS_TRUST_LEVEL_INSUFFICIENT<br>Client trust level below<br>server minimum. |
| -33010 | MCPS-010 | MCPS_RATE_LIMITED<br>Rate limit exceeded. |
| -33011 | MCPS-011 | MCPS_ORIGIN_MISMATCH<br>Passport origin does not<br>match server URI. |
| -33012 | MCPS-012 | MCPS_TRANSCRIPT_MISMATCH<br>Transcript binding<br>verification failed<br>(downgrade detected). |
| -33013 | MCPS-013 | MCPS_PASSPORT_TOO_LARGE<br>Passport exceeds maximum<br>size (8192 bytes). |
| -33014 | MCPS-014 | MCPS_CHAIN_TOO_DEEP<br>Issuer chain exceeds<br>maximum depth (5). |
| -33015 | MCPS-015 | MCPS_VERSION_MISMATCH<br>No mutually supported MCPS<br>version. |

Error responses MUST include the standard JSON-RPC error format with a "data" object providing additional context:

```
{
  "jsonrpc": "2.0",
```

```
  "error": {
    "code": -33004,
    "message": "MCPS_INVALID_SIGNATURE",
    "data": {
      "string_code": "MCPS-004",
      "passport_id": "ap_<uuid>",
      "reason": "Signature does not match message
                 content"
    }
  },
  "id": 1
}
```

The "data" object SHOULD include:

string_code: The MCPS string code (e.g., "MCPS-004") for programmatic error handling across implementations that may use different numeric error code conventions.

passport_id: The passport identifier associated with the error, if applicable.

reason: A human-readable description of the failure cause.

Design note: The -33xxx range is used because JSON-RPC 2.0 reserves error codes -32000 to -32099 for "implementation-defined server errors." The previous draft (01) incorrectly placed MCPS error codes in the -32001 to -32009 range, which collides with this reserved range. The -33xxx range is outside all JSON-RPC reserved ranges, providing a clean namespace for MCPS errors.

## 11. Security Considerations

### 11.1. Threat Model

MCPS defends against the following threats:

| Threat | Mitigation |
|---|---|
| Server impersonation | Origin-bound passports (Sec 4.4) |
| Message tampering | Per-message signatures (Sec 5) |
| Tool description poisoning | Full-tool-object hashing including description (Sec 6) |
| Replay attacks | Nonce + timestamp (Sec 7) |
| Signature malleability | Low-S normalization (Sec 3.7) |
| Capability downgrade | Transcript binding (Sec 9.5) |
| Revocation bypass | Signed revocation responses, fail-closed default (Sec 8.7-8.9) |
| Trust level inflation | Issuer chain validation, self-signed capped at L0 (Sec 3.4) |
| Oversized passport DoS | Size limits (Sec 4.3) |
| Tool relaying | Author origin binding (Sec 6.2) |
| Agent key compromise | Key rotation with previous_key_hash (Sec 4.5) |

| TLS relay (MITM) | Optional channel binding (Sec 5.3) |
|---|---|

MCPS does NOT defend against:

| Threat | Why Not |
|---|---|
| Malicious tool authored by legitimate server | Provenance proves WHO, not safety. See Section 11.2. |
| Compromised Trust Authority | Same as X.509: TA compromise is catastrophic. Mitigated by key rotation, multiple TAs (Sec 8.6). |
| Compromised agent key | Agent must protect its key. HSM recommended for L4. Key rotation provides recovery (Sec 4.5). |

## 11.2. Provenance vs Safety

MCPS provides provenance (WHO authored a tool or sent a message), not safety analysis. A malicious server operator can sign their own poisoned tool definitions with a valid MCPS signature.

Tool safety analysis -- detecting malicious descriptions, sandboxing execution, and input validation -- is a separate concern that complements MCPS. Implementations SHOULD NOT treat a valid MCPS signature as evidence that a tool is safe to execute.

## 11.3. Trust Authority Compromise

If a Trust Authority's signing key is compromised, an attacker can issue fraudulent Agent Passports. To mitigate this risk:

- Trust Authority private keys SHOULD be stored in hardware security modules (HSMs) or equivalent tamper-resistant hardware.
- Trust Authorities SHOULD implement key rotation procedures with documented key ceremony practices.
- Trust Authorities MUST publish revocation lists promptly upon detecting key compromise.
- Clients SHOULD support pinning Trust Authority public keys to detect unauthorized key changes.
- Deployments SHOULD configure multiple TAs (Section 8.6) to reduce single-point-of-failure risk.

This is the same risk model as X.509 Certificate Authority compromise. Mitigations are well-understood from the TLS/PKI ecosystem.

## 11.4. Nonce Store Exhaustion

An attacker could flood a server with messages containing unique nonces to exhaust the nonce store's memory. Implementations MUST mitigate this by:

- Garbage-collecting nonces whose associated timestamps have fallen outside the acceptance window.

- Implementing rate limiting on incoming messages.
- Setting a maximum nonce store size with a least-recently-used eviction policy as a last resort.

### 11.5. Clock Skew Attacks

An attacker with the ability to manipulate system clocks on either the sender or receiver could bypass timestamp validation. Mitigations include:

- Implementations at Trust Level L3 and above SHOULD require NTP synchronization or equivalent time source.
- The configurable clock skew tolerance limits the exploitable window.
- The nonce mechanism provides a secondary protection layer independent of timestamps.

### 11.6. Algorithm Agility

This specification mandates ECDSA P-256 as the sole algorithm. Future versions of MCPS MAY introduce algorithm negotiation to support additional curves or signature schemes via the version negotiation mechanism defined in Section 9.1.

Implementations SHOULD be designed to accommodate future algorithm changes by parameterizing the signature algorithm rather than hard-coding P-256.

If algorithm negotiation is introduced in future versions, the "mcps" header MUST include an "alg" field indicating the algorithm used, and both parties MUST agree on a common algorithm during capability negotiation.

### 11.7. Privacy Considerations

Agent Passports contain an agent name, origin, and public key. The origin reveals the server the agent is authorized to communicate with.

Passport identifiers are pseudonymous (UUID-based). They do not reveal the agent operator’s identity without Trust Authority cooperation.

Signed messages enable non-repudiation: it is possible to prove cryptographically that a specific agent sent a specific message. This property may conflict with privacy requirements in certain jurisdictions. Operators SHOULD consider data retention policies for signed message logs.

Implementations SHOULD NOT include sensitive or personally identifiable information in the "agent_name" or "capabilities" fields of Agent Passports.

### 11.8. Non-Repudiation Implications

Because MCPS signatures are produced using long-lived agent keys, signed messages provide strong non-repudiation properties. This means that a signed message can be presented as evidence that a particular agent sent a particular message at a particular time.

Operators deploying MCPS SHOULD be aware that signed message logs may have legal implications, particularly in regulated industries.

Message retention and disposal policies SHOULD be established as part of MCPS deployment planning.

## 12. IANA Considerations

### 12.1. MCPS Error Code Registry

This document requests IANA to create a new registry titled "MCPS Error Codes" with the following initial entries:

| Code | String | Name | Reference |
|---|---|---|---|
| -33001 | MCPS-001 | MCPS_INVALID_PASSPORT | [this doc] |
| -33002 | MCPS-002 | MCPS_PASSPORT_EXPIRED | [this doc] |
| -33003 | MCPS-003 | MCPS_PASSPORT_REVOKED | [this doc] |
| -33004 | MCPS-004 | MCPS_INVALID_SIGNATURE | [this doc] |
| -33005 | MCPS-005 | MCPS_REPLAY_DETECTED | [this doc] |
| -33006 | MCPS-006 | MCPS_TIMESTAMP_EXPIRED | [this doc] |
| -33007 | MCPS-007 | MCPS_AUTHORITY_UNREACHABLE | [this doc] |
| -33008 | MCPS-008 | MCPS_TOOL_INTEGRITY_FAILED | [this doc] |
| -33009 | MCPS-009 | MCPS_TRUST_LEVEL_INSUFFICIENT | [this doc] |
| -33010 | MCPS-010 | MCPS_RATE_LIMITED | [this doc] |
| -33011 | MCPS-011 | MCPS_ORIGIN_MISMATCH | [this doc] |
| -33012 | MCPS-012 | MCPS_TRANSCRIPT_MISMATCH | [this doc] |
| -33013 | MCPS-013 | MCPS_PASSPORT_TOO_LARGE | [this doc] |
| -33014 | MCPS-014 | MCPS_CHAIN_TOO_DEEP | [this doc] |
| -33015 | MCPS-015 | MCPS_VERSION_MISMATCH | [this doc] |

New entries in this registry require Standards Action [RFC8126].

### 12.2. MCPS Trust Level Registry

This document requests IANA to create a new registry titled "MCPS Trust Levels" with the following initial entries:

| Level | Name | Ref | Description |
|---|---|---|---|
| 0 | None | [td] | No MCPS verification |
| 1 | Signed | [td] | Signed, any TA |
| 2 | Verified | [td] | Signed, recognized TA required |
| 3 | Strict | [td] | L2 + tool sigs + origin verified |
| 4 | Full | [td] | L3 + mutual auth + revocation |

[td] = [this document]

New entries in this registry require Specification Required [RFC8126].

### 12.3. MCPS Capability Identifier

This document registers the capability identifier "mcps" for use within the MCP capabilities negotiation framework. Implementations using this capability identifier MUST conform to the negotiation procedures defined in Section 9 of this document.

## 13. References

### 13.1. Normative References

[FIPS186-5]
National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-5,

DOI 10.6028/NIST.FIPS.186-5, February 2023.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[RFC9266] Shoemaker, R., "Channel Bindings for TLS 1.3", RFC 9266, DOI 10.17487/RFC9266, July 2022, <https://www.rfc-editor.org/info/rfc9266>.

### 13.2. Informative References

[RFC9421] Backman, A., Richer, J., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[JSONRPC] JSON-RPC Working Group, "JSON-RPC 2.0 Specification", January 2013, <https://www.jsonrpc.org/specification>.

[SEMVER] Preston-Werner, T., "Semantic Versioning 2.0.0",

<https://semver.org/>.

## Appendix A. Example Protocol Exchange

This appendix provides a complete example of an MCPS-secured MCP session, illustrating capability negotiation, transcript binding verification, and signed tool invocation.

### A.1. Capability Negotiation

Client sends initialize request with MCPS capability:

```
{
  "jsonrpc": "2.0",
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "mcps": {
        "version": "1.0",
        "trust_level": 2,
        "passport": {
          "mcps_version": "1.0",
          "passport": {
            "id":
              "ap_550e8400-e29b-41d4-a716-446655440000",
            "agent_name": "research-agent",
            "agent_version": "1.2.0",
            "issuer": "ta.example.com",
            "origin": "https://api.example.com",
            "issued_at": "2026-03-01T00:00:00Z",
            "expires_at": "2026-09-01T00:00:00Z",
            "public_key": {
              "kty": "EC",
              "crv": "P-256",
              "x": "f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8T
                    us9uPHvRVEU",
              "y": "x_FEzRu9m36HLN_tue659LNpXW6pCySt
                    ikYjKIWI5a0"
            },
            "capabilities": [
              "tools/call",
              "tools/list"
            ],
            "trust_level": 2,
            "issuer_chain": []
          },
          "signature": "MEUCIQC...base64..."
        }
      }
    },
    "clientInfo": {
      "name": "research-agent",
      "version": "1.2.0"
    }
  },
  "id": 1
}
```

Server responds with its MCPS capabilities and passport:

```
{
  "jsonrpc": "2.0",
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
```

```
        "mcps": {
          "version": "1.0",
          "min_trust_level": 2,
          "passport": {
            "mcps_version": "1.0",
            "passport": {
              "id":
                "ap_660f9511-f30c-52e5-b827-557766551111",
              "agent_name": "file-server",
              "agent_version": "2.0.0",
              "issuer": "ta.example.com",
              "origin": "https://api.example.com",
              "issued_at": "2026-02-15T00:00:00Z",
              "expires_at": "2026-08-15T00:00:00Z",
              "public_key": {
                "kty": "EC",
                "crv": "P-256",
                "x": "2ygLm3slRf6MtoN_-D2bLMMq_10eCMei
                      -yLAbFYmPFE",
                "y": "4PUOD5BNzGMIAnvMjMVHRwZ3XwGBJjB0
                      NJEwPHLJDjg"
              },
              "capabilities": ["tools/call"],
              "trust_level": 2,
              "issuer_chain": []
            },
            "signature": "MEQCIAo...base64..."
          }
        }
      },
      "serverInfo": {
        "name": "file-server",
        "version": "2.0.0"
      }
    },
    "id": 1
  }
```

### A.2. Transcript Binding Verification

Both parties exchange transcript bindings after initialize. The transcript hash is computed over the "params" field of the client's initialize request and the "result" field of the server's initialize response (NOT the full JSON-RPC envelopes):

```
  {
    "mcps": {
      "version": "1.0",
      "passport_id":
        "ap_550e8400-e29b-41d4-a716-446655440000",
      "timestamp": "2026-03-13T14:29:55Z",
      "nonce":
        "b2c3d4e5f6a748b90c1d2e3f4a5b6c7d",
      "signature": "MEQCIBx...base64..."
    },
    "jsonrpc": "2.0",
    "method": "mcps/transcript_verify",
    "params": {
      "transcript_hash":
        "a1b2c3...64-char-hex-sha256...",
      "transcript_signature":
        "MEUCIQDk...base64..."
    },
    "id": 2
  }
```

### A.3. Signed Tool Call

Client invokes a tool with MCPS signature:

```
{
  "mcps": {
    "version": "1.0",
    "passport_id":
      "ap_550e8400-e29b-41d4-a716-446655440000",
    "timestamp": "2026-03-13T14:30:00Z",
    "nonce":
      "a1b2c3d4e5f647a89b0c1d2e3f4a5b6c",
    "signature": "MEUCIQDr...base64..."
  },
  "jsonrpc": "2.0",
  "method": "tools/call",
  "params": {
    "name": "read_file",
    "arguments": {
      "path": "/etc/hostname"
    }
  },
  "id": 3
}
```

## Appendix B. OWASP Risk Mapping

The following table maps MCPS features to risks identified by the OWASP Top 10 for Agentic Applications (2026) and the OWASP MCP Top 10:

| Risk | MCPS Mitigation |
|---|---|
| Inadequate Agent Identity (ASI02) | Origin-bound Agent Passports with issuer chain validation |
| Inadequate Tool Validation (ASI03) | Full tool definition signing with description coverage |
| Delegated Trust Boundaries (ASI04) | Trust levels enforce minimum security per connection |
| Tool Poisoning (MCP03) | Signed tool definitions with pinned tool_hash detection |
| Server Impersonation | Origin-bound passport-based mutual authentication |
| Message Tampering | Per-message ECDSA signatures with JCS canonicalization |
| Replay Attacks | Nonce + timestamp window |
| Capability Downgrade | Transcript binding (Section 9.5) |
| Insufficient Authorization (MCP07) | Trust level gating with capability lists |

## Appendix C. Design Rationale

### C.1. Why a Separate Envelope (Not Modifying JSON-RPC)

MCPS wraps existing JSON-RPC messages rather than modifying the

protocol schema. This ensures backward compatibility, transport independence, and composability with existing OAuth authorization and DPoP.

### C.2. Why ECDSA P-256 (Not Ed25519 or RSA)

P-256 is mandated by FIPS 186-5, required for government and enterprise compliance, and supported by Web Crypto API. Ed25519 has superior performance but lacks FIPS certification. RSA key sizes create larger signatures.

### C.3. Why IEEE P1363 (Not DER)

DER encoding produces variable-length signatures (typically 70-72 bytes for P-256). IEEE P1363 produces fixed-length signatures (exactly 64 bytes for P-256). Fixed-length format eliminates parsing ambiguity, is required by RFC 7518 Section 3.4, and simplifies cross-platform interoperability.

### C.4. Why message_hash (Not Nested JCS)

Earlier drafts used "jsonrpc_message: JCS(message)" in the signing payload, nesting one JCS string inside another JCS object. This creates fragility: the inner JCS string must be JSON-escaped when embedded in the outer object, and different JSON libraries handle this escaping differently. Using "message_hash: SHA-256(JCS(message))" eliminates this ambiguity, is more robust, and is more efficient.

### C.5. Why -33xxx Error Codes (Not -32xxx)

JSON-RPC 2.0 reserves error codes -32000 to -32099 for "implementation-defined server errors." Using this range for MCPS-specific codes would risk collisions with other JSON-RPC extensions. The -33xxx range is outside all reserved ranges, providing a clean namespace for MCPS errors.

### C.6. Why a Lightweight Passport Format (Not Raw X.509)

Agent Passports serve a similar role to X.509 certificates but use a JSON-native format because: (1) MCP is JSON-RPC, so JSON credentials avoid ASN.1/DER complexity; (2) the chain model is simpler than X.509 path validation; (3) JSON is familiar to the MCP developer ecosystem.

However, the passport model deliberately borrows proven concepts from X.509: issuer chains, validity periods, key usage constraints, and revocation checking. Implementations MAY bridge to X.509 by embedding a certificate reference in the passport.

### C.7. Why Not JWS (RFC 7515) for Message Signing

The per-message envelope is comparable in size to JWS compact serialization (~280 bytes vs ~200 bytes for JWS). The primary motivation for a purpose-built envelope is explicitness: the "mcps" field contains exactly the fields needed for MCP verification (passport_id, nonce, timestamp) without requiring implementers to parse JWS headers and map them to MCP semantics.

An alternative formulation using JWS as the signing container is viable and could be explored if the community prefers standards composition over a purpose-built format.

### C.8. Why Self-Hostable Trust Authority

A centralized TA would create a single point of failure. The

self-hostable design ensures no vendor lock-in, air-gapped deployment support, regulatory compliance, and resilience.

## Appendix D. Changes from draft-sharif-mcps-secure-mcp-01

This section summarizes the changes from version 01 to version 02:

CRITICAL changes:

- Specified IEEE P1363 signature format (r||s concatenation, exactly 64 bytes for P-256) per RFC 7518 Section 3.4. Explicitly prohibits DER encoding. Added low-S normalization requirement (s <= n/2) with the P-256 curve order constant. (Section 3.6, Section 3.7)

- Added ECDSA malleability defense: low-S normalization prevents (r, n-s) from being accepted as valid alternative. Nonce stores MUST key on nonce string, not full message bytes. (Section 3.7, Section 7.3)

- Defined issuer chain entry format: each entry is a base64-encoded JSON intermediate TA passport with its own signature, public_key, and issuer fields. Added step-by-step chain verification algorithm. Max depth: 5. (Section 8.4, Section 8.5)

- Changed error code range from -32001..-32009 (which collides with JSON-RPC reserved range -32000..-32099) to -33001..-33015. Added string codes MCPS-001 through MCPS-015. (Section 10)

HIGH changes:

- Renamed "Transcript MAC" to "Transcript Binding" throughout. MCPS uses ECDSA signatures (asymmetric), not MAC (symmetric). (Section 2, Section 9.5)

- Added version negotiation mechanism during initialize. Client may announce single version or array of versions. Server responds with highest mutual version or -33015 error. (Section 9.1)

- Added passport size limits: MAX_PASSPORT_BYTES = 8192, MAX_ISSUER_CHAIN_DEPTH = 5, MAX_CAPABILITIES = 64. Added corresponding error codes -33013 and -33014. (Section 4.3)

- Changed signing payload from "jsonrpc_message: JCS(msg)" to "message_hash: SHA-256(JCS(msg))" to avoid double-canonicalization fragility. (Section 5.2)

- Added optional TLS channel binding field (RFC 9266 tls-exporter) to signing payload. (Section 5.3)

- Added multi-TA deployment support with documented cross-TA trust model. (Section 8.6)

- Added key_rotation field with previous_key_hash and rotated_at for agent key compromise recovery. (Section 4.5)

- Clarified transcript hash boundary: covers "params" (client) and "result" (server), NOT full JSON-RPC envelope.

(Section 9.5)

- Added author_origin to tool signing payload for tool-server origin binding.
  (Section 6.1, Section 6.2)

- Updated all references from FIPS 186-4 to FIPS 186-5.
  Added note on RFC 6979 vs HSM compatibility with FIPS 186-5 nonce generation.
  (Section 3.5)

Additional changes:

- Added RFC 7518 and RFC 9266 to normative references.

- Updated reference implementation test counts: Node.js 75 tests, Python 53 tests.

- Added six new error codes: MCPS_AUTHORITY_UNREACHABLE (-33007), MCPS_RATE_LIMITED (-33010), MCPS_ORIGIN_MISMATCH (-33011), MCPS_PASSPORT_TOO_LARGE (-33013), MCPS_CHAIN_TOO_DEEP (-33014), MCPS_VERSION_MISMATCH (-33015).

## Author's Address

Raza Sharif
CyberSecAI Ltd

Email: contact@agentsign.dev