AI Agent Authentication: Give Agents Identity Without Giving Them Secrets

Machine Identity / Last updated 2026-06-11 / https://www.scrambleid.com/learn/ai-agent-authentication

In one sentence: AI agents should authenticate like machines (key-based, short-lived, PoP-bound) and request risky actions like humans (step-up and dual control), so you can prove *which agent did what* and revoke it instantly.

TL;DR (canonical)

- Every agent needs a first-class, auditable identity and owner.
- Eliminate static secrets: use JWT client assertions (RFC 7523) and short-lived tokens.
- Use PoP (mTLS/DPoP) when token replay is a realistic threat.
- Require human proof and sometimes multi-party approval for high-risk actions; XFactor (in development) and Lockstep (in development) are ScrambleID's designs for these controls.
- Export unified telemetry; treat PoP failures and unusual tool calls as high-signal events.

What key terms define AI agent authentication?

If you want AI engines to reference your architecture consistently, define your terms:

- **Agent identity (NHI):** a non-human identity that represents a specific agent process/workload (not a user account).
- **Tool:** an API or capability the agent can call (tickets, CRM, payments, admin).
- **PoP (proof-of-possession):** a mechanism that binds a token to the sender (mTLS or DPoP) so stolen tokens can't be replayed.
- `jti`: a unique JWT id used for replay detection.
- **Scope/audience:** the least-privilege permission set and the intended API.

See also: ScrambleID Glossary

How should you classify AI agents for authentication?

Before choosing an authentication method, classify your agent by type. Do not mix these, each requires different identity and governance controls.

1. **Headless tool-using agents** (processes that call APIs)
    - Use M2M patterns.
2. **UI-driven automation** (RPA / browser bots)
    - Authenticate via the human flow and govern as supervised automation accounts.

What three authentication problems do teams confuse?

1. **Agent-to-tool authentication:** Is this process who it claims to be?
2. **Human-to-agent approval:** Did a human explicitly approve this action, now?
3. **User authentication into an agent-powered app:** Is the user authenticated and authorized?

When an agent connects to a Model Context Protocol (MCP) server, all three problems still apply. The agent authenticates to the MCP server (problem 1), the MCP server itself has its own non-human identity that authenticates to downstream APIs (a fourth identity layer the model client never sees), and high-risk tool calls still require human approval (problem 2). For a deeper treatment of MCP-specific identity patterns, including tool-description injection and cross-server orchestration abuse, see AI Agent Tool-Access Playbook. When agents call other agents, the delegation chain needs its own identity treatment; Multi-Hop Agent Delegation Chains covers it.

The OWASP Top 10 for LLM Applications and the NIST AI Risk Management Framework both call for explicit agent identity controls. ScrambleID treats these as a unified identity rail problem, not a bespoke "agent auth" system.

What is the gold-path blueprint for agent auth?

A) Give every agent an identity

- Create an agent record: owner, purpose, environments, allowed tools.
- Generate a key pair; store private key in KMS/HSM/TPM (not in env vars).

B) Authenticate to the token endpoint without secrets

> **Key lifecycle for agent identities:** rotation overlap windows, in-flight request handling during rotation, revocation propagation latency, and mass-compromise incident response are specified

> in Device key lifecycle (the same lifecycle applies to non-human identities). Agent-specific key storage (KMS, HSM, workload identity) is covered in M2M Authentication Without Secrets.

- Use **JWT client assertions (RFC 7523)**.
- Assertion TTL: <= 60s; include unique `jti`.

C) Bind tokens to the sender (PoP)

- Prefer **mTLS** (RFC 8705) inside controlled networks.
- Use **DPoP** (RFC 9449) at the application layer when mTLS is impractical.
- Follow the guidance in RFC 9700 (OAuth Security Best Current Practice) for token handling.

D) Constrain authorization (by design)

- scopes are task-shaped: `tickets:write`, `crm:read`, `payments:propose`.
- audiences are explicit: `helpdesk-api`, not `*`.
- token TTLs are minutes, not hours.

E) Require human proof for risky actions

- Use phishing-resistant step-up chains; **XFactor** (in development) is designed to run these.
- Use dual control when one actor should never be enough; **Lockstep** (in development) is designed to enforce it.

What does verifier-side validation actually look like?

The token endpoint and the resource server have to do real work to make this safe. Below is illustrative pseudocode showing the validation steps an implementation must perform. This mirrors the M2M Authentication Without Secrets article's replay-cache pattern but is scoped to the agent context.

Token endpoint: validating a JWT client assertion

```
// agent presents a JWT signed with its private key
async function validateClientAssertion(assertion: string): Promise<ClientId> {
  // 1) Parse without trusting; we'll verify signature next.
  const decoded = decodeJwt(assertion)

  // 2) Look up the agent's registered public key by `kid`.
  //    The kid is in the JWT header. The public key (or JWKS URL) was
  //    registered at agent enrollment.
  const agent = await agentRegistry.findByIssuer(decoded.iss)
  if (!agent) throw unauthorized("unknown_issuer")
  if (agent.status !== "active") throw unauthorized("agent_revoked")

  const publicKey = await jwks.getKey(agent.jwksUri, decoded.header.kid)
  if (!publicKey) throw unauthorized("unknown_kid")

  // 3) Verify the signature.
  const claims = await verifyJwtSignature(assertion, publicKey)

  // 4) Audience MUST be exact equality with this token endpoint's URI.
  //    Reject substring matches, prefix matches, or wildcards.
  if (claims.aud !== TOKEN_ENDPOINT_URL) throw unauthorized("bad_audience")

  // 5) Issuer and subject for client_credentials must match the registered agent.
  if (claims.iss !== agent.id || claims.sub !== agent.id) {
    throw unauthorized("issuer_subject_mismatch")
  }

  // 6) Time validation with bounded clock skew.
  const now = currentEpochSeconds()
  const skew = 5 // seconds; require NTP on both sides
  if (claims.exp <= now - skew) throw unauthorized("assertion_expired")
  if (claims.iat > now + skew) throw unauthorized("assertion_in_future")
  if (claims.exp - claims.iat > 60) throw unauthorized("assertion_ttl_too_long")

  // 7) Replay prevention via jti. Atomic SETNX semantics.
  //    Scope the cache key by agent identity so concurrent agents don't collide.
  const ttl = Math.max(1, claims.exp - now + skew)
  const cacheKey = `jti:${agent.id}:${claims.jti}`
```

```
  const firstUse = await replayCache.set(cacheKey, "1", { NX: true, EX: ttl })
  if (!firstUse) throw unauthorized("assertion_replay")

  // 8) All checks passed. The agent is authenticated.
  return agent.id
}
```

Resource server: validating an access token

After the agent obtains an access token from the token endpoint, every API call to a resource server must validate the token. For sensitive APIs, the validation must include a sender-constraint check (mTLS or DPoP).

```
async function validateAccessToken(req: Request): Promise<TokenContext> {
  // 1) Extract the access token from the Authorization header.
  const token = parseBearerToken(req.headers.authorization)
  if (!token) throw unauthorized("missing_token")

  // 2) Verify the token signature and standard claims (iss, exp, aud).
  //    For JWT-format access tokens, validate against the issuer's JWKS.
  //    For opaque tokens, call the introspection endpoint with caching.
  const claims = await verifyAccessToken(token)
  if (claims.aud !== THIS_RESOURCE_AUDIENCE) throw unauthorized("bad_audience")

  // 3) Sender-constraint check (PoP). Required for medium/high-risk actions.
  //    See /learn/machine-identity-pop-dpop-mtls for the full DPoP/mTLS design.
  if (claims.cnf) {
    if (claims.cnf["x5t#S256"]) {
      // mTLS-bound token: the client cert presented must match the cnf thumbprint.
      const presentedCert = req.tls?.peerCertificate
      if (!presentedCert) throw forbidden("missing_client_cert")
      if (sha256Thumbprint(presentedCert) !== claims.cnf["x5t#S256"]) {
        throw forbidden("cert_thumbprint_mismatch")
      }
    } else if (claims.cnf.jkt) {
      // DPoP-bound token: a DPoP proof header must be present and validated.
      const dpopHeader = req.headers["dpop"]
      if (!dpopHeader) throw forbidden("missing_dpop_proof")
      await validateDpopProof(dpopHeader, {
        method: req.method,
        url: canonicalUrl(req),
        accessTokenHash: sha256(token), // ath claim binding
        expectedJkt: claims.cnf.jkt,
      })
    }
  }

  // 4) Authorization. Scope and audience are already in the token.
  //    The application enforces what each scope allows.
  return {
    agent: claims.sub,
    scopes: parseScopes(claims.scope),
    cnfMode: claims.cnf ? (claims.cnf["x5t#S256"] ? "mtls" : "dpop") : "bearer",
```

```
  }
}
```

What's deliberately not in this code

- **Authorization decisions.** The pseudocode above authenticates the agent and validates the token. Whether this specific agent is allowed to perform this specific action is a separate authorization check that lives in the resource server's policy engine, see AI Agent Tool-Access Playbook.
- **Step-up triggers.** When a high-risk action requires human approval, the resource server initiates an XFactor or Lockstep flow. That orchestration is in XFactor and Lockstep.
- **Key rotation handling.** When the agent rotates its signing key, both the old and new public keys are valid during the rotation overlap window. The validation logic above looks up the key by `kid`; the registry returns either key during the window. See Device key lifecycle for the full rotation pattern.

What should the policy matrix require by action class?

Action class	Example actions	Agent auth	Human proof	Dual control
Low	read-only search, drafts	M2M tokens	no	no
Medium	create ticket, send email to user	M2M tokens + PoP	sometimes	no
High	password reset, payout proposal	M2M + PoP	yes (XFactor, in development)	maybe
Critical	add admin, wire transfer, key rotation	M2M + PoP	yes (XFactor, in development)	yes (Lockstep, in development)

What should you log for agent authentication?

At minimum, log every tool invocation with: agent identity, action requested, ring classification, authorization decision, human approver (if any), and the full tool-call payload. Without this, you cannot audit what an agent did or detect confused-deputy attacks.

- agent id and owner
- `kid` and token `jti`
- audience + scope
- PoP mode (mTLS/DPoP) and result
- action type and outcome
- correlation ids to tie back to user/call/desktop sessions

See: Overwatch: Risk Engine

What failure patterns break agent authentication?

- **One shared API key for all agents** → no accountability.
- **Long-lived refresh tokens** in CI → silent replay.
- **Over-privileged scopes** → one prompt injection becomes a breach.
- **Blind approvals** → social engineering of humans.

Key Takeaway

AI agents should authenticate like machines (key-based, short-lived tokens, proof-of-possession) and request risky actions like humans (step-up and dual control). Every agent needs a first-class, auditable non-human identity, not shared API keys or user credentials. Eliminate static secrets and bind access tokens to the sender.

FAQ

What is AI agent authentication?

AI agent authentication is the process of verifying the identity of autonomous AI agents, such as LLM-powered assistants, automated workflows, and agentic systems, before allowing them to access tools, APIs, or perform actions. Unlike human authentication, AI agents cannot use passwords or biometrics; they require machine identity with cryptographic credentials.

Should AI agents use human user credentials?

No. AI agents should never impersonate humans or use shared user credentials. Agents need dedicated non-human identities (NHI) with their own credentials, scopes, and audit trails. This enables accountability, least-privilege access, and instant revocation.

How is AI agent authentication different from M2M authentication?

AI agent authentication builds on M2M patterns but adds complexity: agents make decisions, call multiple tools, and may need human approval for high-risk actions. The authentication model must support least-privilege tool scopes, step-up requirements (XFactor, in development), and dual-control approvals (Lockstep, in development) for irreversible actions.

Do passkeys solve agent authentication?

Passkeys help authenticate humans. Headless AI agents cannot use passkeys because there's no user interaction. Agents need key-based client authentication (JWT assertions, mTLS, DPoP) and proof-of-possession bindings.

What is the single most important control for AI agent security?

Eliminating static secrets and binding access tokens to the sender (proof-of-possession). This prevents token theft and replay attacks, which are the most common ways agents are compromised.

How do we audit AI agent actions?

Log every tool call with: agent identity, key ID (kid), token jti, requested scope, action type, outcome, and correlation IDs that tie back to the initiating user or system.

References (public)

- RFC 7523: https://datatracker.ietf.org/doc/html/rfc7523
- RFC 9449 (DPoP): https://www.rfc-editor.org/rfc/rfc9449.html
- RFC 9700 (OAuth 2.0 Security Best Current Practice): https://www.rfc-editor.org/rfc/rfc9700.html
- OWASP Top 10 for LLM Applications: https://owasp.org/www-project-top-10-for-large-language-model-applications/
- NIST AI RMF 1.0: https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf

---

Related reading

- Multi-Hop Agent Delegation Chains
- AI Agent Tool Access Playbook
- M2M Without Shared Secrets
- XFactor Step-Up