Sender-Constrained Tokens for Machine Identity: mTLS (RFC 8705) and DPoP (RFC 9449)

Machine Identity / Last updated 2026-06-11 / https://www.scrambleid.com/learn/machine-identity-pop-dpop-mtls

In one sentence: Sender-constrained tokens (via mTLS or DPoP) bind OAuth tokens to the caller's private key, making stolen tokens useless without the key, eliminating bearer token replay attacks.

Problem: OAuth bearer tokens are replayable. If a token is stolen (logs, proxies, memory dump), an attacker can reuse it until it expires.

Solution: Bind the token to the caller using proof-of-possession (PoP). A stolen token without the private key becomes useless.

TL;DR (canonical)

- Use **mTLS** when you can control network paths end-to-end (service mesh, internal gateways).
- Use **DPoP** when you need PoP over diverse clients or where mTLS is impractical.
- Treat PoP failures as high-signal security events.
- Keep TTLs short even with PoP.

What "sender-constrained" means

A token is sender-constrained when the resource server enforces: "only the client holding key X can use this token".

This is commonly expressed via the OAuth `cnf` (confirmation) claim, e.g. a JWK thumbprint (`jkt`).

---

mTLS sender-constrained tokens (RFC 8705)

When to use mTLS

- internal service-to-service traffic
- service mesh where client certs already exist
- API gateways that preserve client certificates

What can go wrong (mTLS)

- TLS termination strips the client certificate before the resource server
- inconsistent cert rotation across clusters
- relying on source IP instead of client cert

Operator checklist (mTLS)

- [ ] Ensure the resource server sees the client certificate (or the validated thumbprint) after any proxy.
- [ ] Define rotation overlap (old + new cert valid) and the emergency revoke path.
- [ ] Log certificate thumbprint for every token use.

DPoP (RFC 9449)

> **Key lifecycle baseline:** rotation overlap windows, in-flight request handling during rotation, and revocation propagation SLAs apply to mTLS client certificates and DPoP keys the same way they apply to user device keys. The canonical specification is in Device key lifecycle.

In practice, mid-flow rotation means: finish the request sequence you started under the old key, start new flows under the new key, and rely on the verifier accepting both keys during the overlap window. A DPoP proof signed by the old key against a token bound to the new key's thumbprint fails the cnf.jkt check by design; that is the signal to re-request the token, not to retry the proof.

When to use DPoP

- public APIs with diverse clients
- edge environments where mTLS is hard
- mobile/desktop clients that can hold keys but not cert chains

How DPoP works

- client has a DPoP key pair
- client signs a proof JWT per request
- token is bound to the DPoP key

Common proof fields:

- `htu` (HTTP URL)
- `htm` (method)
- `iat` (issued at)
- `jti` (unique id)

Minimal DPoP example (copy/paste)

HTTP request (illustrative):

```
GET /v1/transfers/tx_123 HTTP/1.1
Host: api.example.com
Authorization: DPoP eyJ...
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2Iiwi...<snip>
```

DPoP proof header (illustrative):

```
{
  "typ": "dpop+jwt",
  "alg": "ES256",
  "jwk": {"kty":"EC","crv":"P-256","x":"...","y":"..."}
}
```

DPoP proof payload (illustrative):

```
{
  "htu": "https://api.example.com/v1/transfers/tx_123",
  "htm": "GET",
  "iat": 1737158400,
  "jti": "9f9e5c02-1b70-4d6a-9f5b-2a0a9f...",
  "ath": "u2F1..."
}
```

Notes:

- `ath` binds the proof to the specific access token being used. The token's `cnf.jkt` thumbprint already blocks a stolen token from being used with an attacker's own DPoP key; what `ath` adds is binding each proof to one specific access token, so a captured proof cannot be replayed with a different token under the same key.
- `jti` must be unique per proof to enable replay detection.

What can go wrong (DPoP)

- URL rewriting by proxies (htu mismatch)
- Method mismatch on retries: a client may issue an idempotent retry with the same DPoP proof, but if the retry library rewrites POST to GET (or vice versa) the `htm` claim no longer matches the

request method and the proof is rejected. Configure retry libraries to either mint a fresh DPoP proof per attempt or to preserve the original method exactly.

- Clock skew (iat validation)
- Key reuse across multiple services (over-broad blast radius). A shared DPoP key means a single compromise affects every service that workload talks to. If the workload's signing key leaks via memory dump, log file, or accidental serialization, every downstream API is at risk simultaneously. Mint a separate DPoP key per workload identity, not per organization.

Operator checklist (DPoP)

- [ ] Normalize URLs consistently between client and server (especially behind gateways).
- [ ] Enforce `jti` reuse detection.
- [ ] Keep `iat` window tight.
- [ ] Rotate DPoP keys per workload (not a shared org-wide key).

Resource server validation (server-side)

The most common DPoP failures are **normalization mistakes** and **clock skew**. Keep validation simple and explicit.

Illustrative pseudocode:

```
function validateDPoP({ method, url, accessToken, dpopJwt, now }) {
  const proof = verifyJwtSignatureAndGetClaims(dpopJwt) // uses public jwk in header

  // 1) Request binding
  assertEqual(proof.htm, method)
  assertEqual(normalizeHtu(proof.htu), normalizeHtu(url))

  // 2) Time window (allow small leeway)
  assertWithinWindow(proof.iat, now, { leewaySeconds: 5, maxAgeSeconds: 300 })

  // 3) Replay detection
  assertNotSeenBefore({ keyThumbprint: jwkThumbprint(proof.jwk), jti: proof.jti, ttlSeconds: 300 })

  // 4) Token binding
  assertEqual(proof.ath, sha256Base64url(accessToken))

  return true
}
```

Recommended defaults (starter):

- `iat` leeway: **5 seconds** (tune to your infra)
- proof max age: **5 minutes** (or align to token TTL)
- replay cache TTL: **max(proofMaxAge, tokenTTL)**

Decision guide

Environment	Prefer	Why
Internal services in Kubernetes/service mesh	mTLS	network already supports cert binding
Public APIs / multi-tenant edge	DPoP	app-layer PoP works without cert plumbing
Zero trust w/ strict gateways	mTLS	simplest enforcement at the boundary
Rapidly scaling short-lived clients	DPoP	per-instance keys are easy

---

Monitoring and incident response

Track at minimum:

- PoP mismatch (cnf/jkt mismatch or DPoP failure)
- replay detection signals (duplicate `jti`)
- sudden token issuance spikes
- unexpected audience/scope changes

Key Takeaway

Sender-constrained tokens eliminate bearer token replay attacks by binding tokens to the caller's private key. Use mTLS (RFC 8705) when you control network paths end-to-end; use DPoP (RFC 9449) when mTLS is impractical. Treat PoP validation failures as high-signal security events, they indicate potential token theft or misconfiguration.

FAQ

Is PoP required if tokens are short-lived?

Short TTL reduces exposure, but PoP prevents replay during that TTL. Use both for high-risk APIs.

Which is "more secure", mTLS or DPoP?

Both can be secure. The biggest risk is operational: proxies, termination, and inconsistent enforcement.

What should we log?

Log client identity, `kid`, proof result, cnf/jkt, and the gateway/resource server that made the decision.

References (public)

- RFC 8705 (mTLS): https://datatracker.ietf.org/doc/html/rfc8705
- 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

---

Related reading

- M2M Without Shared Secrets
- AI Agent Authentication
- Evaluation Checklist + RFP