M2M Authentication Without Secrets: JWT Client Assertions Instead of Client Secrets

Machine Identity / Last updated 2026-06-11 / https://www.scrambleid.com/learn/m2m-authentication-without-secrets

In one sentence: Replace OAuth client secrets with **asymmetric keys** and **JWT client assertions**, minting short-lived tokens that can be made replay-resistant with proof-of-possession.

TL;DR (canonical)

- Client secrets leak; key-based client authentication dramatically reduces shared-secret blast radius.
- Use **JWT client assertions (RFC 7523)** for the token endpoint.
- Use very short assertion TTLs (tens of seconds) and unique `jti` values for replay prevention.
- Add sender constraints (mTLS/DPoP) when token replay risk is material.

Everything in this article is standards-based, and it's also how ScrambleID's own machine-to-machine layer works: the standardized M2M API that shipped in March 2026 authenticates non-human clients with key-based assertions and short-lived credentials instead of stored secrets.

The problem with client secrets

Client secrets tend to end up:

- in repos and build logs
- duplicated across environments
- copied into too many vault paths
- shared between teams

A leaked secret often enables silent, long-lived impersonation.

JWT client assertions (RFC 7523)

A standard pattern:

1. Workload has a key pair; public key is registered.

2. Workload signs a JWT and posts it to the OAuth token endpoint.
3. Server validates signature, audience, TTL, and replay.
4. Server issues a short-lived access token.

Example assertion claims

```
{
  "iss": "orders-service",
  "sub": "orders-service",
  "aud": "https://id.example.com/oauth2/token",
  "exp": 1895010000,
  "jti": "b8a6c0b2-1f6b-4b2f-9c31-..."
}
```

Recommended practices:

- `exp` should be short (<= 60 seconds) to reduce replay window.
- `jti` should be unique and checked for reuse.
- `aud` must be exact (token endpoint).

Clock skew + replay caches (avoid false positives)

Replay prevention only works if the verifier is tolerant of real-world time drift.

Prerequisite: time synchronization

Before discussing leeway, the foundational requirement is that every workload that signs assertions and every server that validates them runs **NTP or an equivalent time-sync service**. Without it, even modest drift breaks tight TTLs and produces false-positive replay detections during normal operations.

- Workloads: `chronyd`, `systemd-timesyncd`, AWS Time Sync Service, GCP metadata server time, Azure-provided time.
- Containers: ensure the host's NTP propagates to the container; do not assume container clocks are synced independently.
- Verify drift is sub-second in production. Anything beyond a couple of seconds is a configuration smell.

Recommended defaults (starter)

- Allow **small clock leeway** (e.g., 5 seconds) when validating `iat` / `exp`.
- Store assertion `jti` values in a replay cache with TTL ≈ `exp - now + leeway`.

- Scope replay detection **per client identity** (iss/sub/kid) to prevent cross-tenant collisions.

Redis-style replay cache pattern (pseudocode)

```
// key: "jti:{clientId}:{jti}"
// SETNX ensures only first use succeeds
const ttlSeconds = Math.max(1, exp - now + 5)
const ok = await redis.set(key, "1", { NX: true, EX: ttlSeconds })
if (!ok) throw new Error("replay_detected")
```

Why this matters: without leeway + correct TTLs, you will flag legitimate traffic as replay during deploys, node restarts, or time-sync hiccups.

Key storage and signing patterns (what actually works)

- Cloud KMS/HSM-backed signing (best for high-value workloads)
- Platform secure hardware (TPM / Secure Enclave)
- Kubernetes: use a sidecar or node-level signing agent instead of env-var keys

Avoid:

- private keys in plain environment variables
- long-lived keys shared across multiple workloads

Cloud workload identity vs. self-managed key pairs

If you run on AWS, GCP, or Azure, the major cloud providers offer workload identity systems that solve a related problem: removing static credentials from workloads entirely. Each issues short-lived, per-workload identity proofs that downstream services can validate, similar in spirit to JWT client assertions but managed by the platform.

Approach	Best fit	Trade-offs
AWS IAM roles for service accounts (IRSA), EC2 instance profiles, ECS task roles	AWS-native workloads accessing AWS APIs and AWS-aware downstream services	Strong inside AWS. Calling external systems still requires JWT assertions or federation back into the external IdP.
GCP Workload Identity Federation	GKE workloads, federated identity with external IdPs (including JWT-based)	GKE-native is excellent. Federation to non-Google IdPs requires careful audience configuration.
Azure Managed Identities	Azure-hosted workloads accessing Azure resources or Entra ID-protected APIs	Tight Azure integration. External (non-Azure) consumers need explicit OIDC trust.

Approach	Best fit	Trade-offs
Self-managed JWT client assertions (RFC 7523)	Multi-cloud, on-prem, hybrid; calling third-party APIs that don't accept cloud-native identity	More operational responsibility (key rotation, key storage, JWKS hosting). Maximum portability.

Practical guidance:

- **Prefer cloud-native workload identity for cloud-internal traffic.** It eliminates an entire class of credential-management work.
- **Use JWT client assertions when the downstream service is outside your cloud** (e.g., a third-party API or your own ScrambleID deployment). Combine with sender-constrained tokens for replay defense.
- **You can chain them.** A workload may use AWS IRSA to obtain a short-lived AWS identity, then exchange it for an OIDC token via STS that a third-party API accepts. This pattern is well-supported and avoids static keys end-to-end.

Hardening: sender-constrained access tokens

If an access token can be stolen (logs, proxies, memory dump), it can be replayed unless sender-constrained.

Two common sender constraints:

- **mTLS** (RFC 8705): certificate-bound tokens
- **DPoP** (RFC 9449): application-layer proof-of-possession

See: PoP, DPoP, and mTLS

Rotation and revocation (a real runbook)

> **Lifecycle baseline:** the architecture-level specification of key lifecycle (enrollment, rotation overlap windows, revocation propagation SLAs, in-flight request handling) is in Device key lifecycle. This section covers the M2M-specific runbook on top of that baseline.

Rotation

- Register new public key (`kid` B) while keeping old key (`kid` A) valid.
- Roll workloads to start signing with `kid` B.
- After the overlap window, revoke `kid` A.

Emergency revoke

- Revoke the compromised key id.
- Block token minting for that client alias.
- Force re-provisioning of keys.

Mass key compromise (incident response)

A single compromised key is contained by the steps above. A broader incident, where a CI/CD pipeline, signing agent, or KMS partition is suspected to have leaked many workloads' keys at once, requires a different posture:

Forensics scope the blast radius before re-issuance finishes: pull the key-usage logs for every suspect key id (cloud KMS audit trails such as AWS CloudTrail KMS events, GCP Cloud Audit Logs, or your HSM partition logs), diff observed issuance and signing activity against the workload expected baseline, and treat any use from an unexpected principal, region, or time window as confirmed compromise for that key. The log review bounds the incident window and tells you which downstream tokens to revoke rather than guessing.

- **Quarantine first, audit second.** Block token minting for every affected client alias rather than trying to identify exactly which keys are compromised. Bias toward false-positive lockouts; you can re-enable rapidly after re-keying.
- **Stage a bulk re-issuance.** Generate fresh keys via the KMS or signing agent and roll workloads in waves. Maintain a brief overlap window for graceful drains; do not run the rotation as a hard cutover unless your SLAs require it.
- **Force outstanding token revocation.** Short-lived tokens age out naturally, but for incidents you should also invalidate active sessions/tokens issued under the compromised keys. PoP-bound tokens (mTLS/DPoP) shrink the blast radius significantly here.
- **Capture audit detail.** Log each `kid` revoked, each new `kid` issued, the workload it was bound to, and timestamps. Compliance frameworks (SOC 2, ISO 27001) want this artifact for the incident record.
- **Communicate downstream.** If your customers consume your APIs with their own clients, they may need to re-import your JWKS. Plan that comms loop ahead of the incident.

What to measure

- percent of machine identities without secrets
- token mint success rate and p95 latency
- replay attempts (`jti` reuse)
- PoP failures (cnf mismatch / DPoP failure)
- rotation success rate

Key Takeaway

Replace OAuth client secrets with asymmetric keys and JWT client assertions (RFC 7523). Client secrets leak; key-based authentication dramatically reduces blast radius. Use very short assertion TTLs, unique jti values for replay prevention, and add sender constraints (mTLS/DPoP) when token replay risk is material.

FAQ

What is machine-to-machine (M2M) authentication?

Machine-to-machine authentication verifies the identity of non-human entities, services, APIs, microservices, batch jobs, and automated processes, before allowing them to access resources. Unlike human authentication, M2M authentication cannot use interactive methods like passwords or biometrics; it relies on credentials exchanged programmatically.

What is the problem with shared secrets in M2M authentication?

Shared secrets (client secrets, API keys, passwords) are problematic because they can be leaked through logs, config files, environment variables, or memory dumps. Once leaked, they remain valid until manually rotated. They also cannot distinguish between legitimate use and stolen credential use.

What should replace shared secrets for M2M authentication?

Replace shared secrets with asymmetric key pairs and JWT client assertions (RFC 7523). The service holds a private key and presents a signed assertion to the authorization server. Add proof-of-possession (mTLS or DPoP) to bind tokens to the caller, making stolen tokens useless.

Why are short-lived tokens important?

Short-lived tokens (minutes, not hours) limit the window of exposure if a token is stolen. Combined with proof-of-possession, even a stolen token becomes useless because the attacker lacks the private key needed to use it.

When should we use mTLS vs DPoP?

Use mTLS when you control the network path end-to-end (service mesh, internal gateways). Use DPoP when clients are diverse or when mTLS is impractical. Both achieve sender-constrained tokens.

What about service accounts in the cloud?

Cloud workload identity (AWS IAM roles, GCP service accounts, Azure managed identities) provides short-lived credentials tied to the workload. Combine with sender constraints where possible.

References (public)

- RFC 7523 (JWT bearer client authentication): https://datatracker.ietf.org/doc/html/rfc7523
- RFC 8705 (mTLS for OAuth): 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

- PoP, DPoP, and mTLS
- AI Agent Authentication
- Overwatch: Risk Engine