PuigFlows

OAuth 2.1 en Producción: PKCE Obligatorio, Rotación de Refresh Tokens, DPoP y Token Exchange para Cadenas de Servicios

OAuth 2.1 in Production: Mandatory PKCE, Refresh Token Rotation, DPoP, and Token Exchange Across Service Chains

Guía / Guide · Premium

Actualizado el 23 de septiembre de 2026 / Updated September 23, 2026

Contenido: versión en español seguida de la versión en inglés. / Contents: Spanish version followed by the English version.

Versión en español

OAuth 2.1 en Producción: PKCE Obligatorio, Rotación de Refresh Tokens, DPoP y Token Exchange para Cadenas de Servicios

OAuth 2.0 nunca fue un protocolo: era un menú

OAuth 2.0 se publicó en 2012 como un marco de trabajo, no como un protocolo cerrado. Traía cuatro tipos de concesión —dos de ellos inseguros para la web moderna— y una docena larga de decisiones que el implementador tenía que tomar sin que la especificación le dijera cuál era la correcta. En los años siguientes el IETF publicó más de veinte RFCs entre extensiones y correcciones de seguridad. El resultado práctico es que nadie implementaba "OAuth 2.0": cada uno implementaba OAuth 2.0 más las cinco RFCs que había llegado a leer.

OAuth 2.1 no inventa nada nuevo. Consolida el Security Best Current Practice (RFC 9700) y las extensiones que ya eran obligatorias de facto en un único documento, y borra del mapa lo que llevaba años desaconsejado. Sigue siendo un Internet-Draft —la revisión 15 es de marzo de 2026—, pero en la práctica ya es el perfil que aplican los proveedores serios y el que exige la especificación de autorización de MCP.

Lo que cambia, en seis puntos:

- PKCE es obligatorio en todos los clientes que usan el flujo de código de autorización, no sólo en aplicaciones móviles y SPAs. Un backend con secreto de cliente también lo necesita.
- El grant implícito desaparece. response_type=token devolvía el access token en el fragmento de la URL, donde acaba en el historial del navegador, en los logs del referer y al alcance de cualquier script de terceros de la página.
- Resource Owner Password Credentials desaparece. Pedir la contraseña del usuario dentro de tu aplicación mata todo lo que OAuth venía a resolver.
- Las redirect URIs se comparan con igualdad exacta de cadenas. Nada de prefijos, nada de comodines en el path, nada de normalizar antes de comparar.
- Los bearer tokens no viajan en la query string. Ni ?access_token= ni nada parecido: van en la cabecera Authorization.
- Los refresh tokens de clientes públicos deben ser de un solo uso o estar ligados al remitente. Este punto es el que más código obliga a escribir, y lo vemos en detalle más abajo.

Si tu integración es de hace tres años y no has tocado nada, es muy probable que incumplas al menos tres de los seis.

PKCE: el ataque que previene, y el que no

PKCE (Proof Key for Code Exchange, RFC 7636) se explica casi siempre como "lo que hacen las apps móviles porque no pueden guardar un secreto". Esa descripción es correcta pero incompleta, y es la razón por la que tanta gente lo deja fuera del backend.

El ataque real se llama inyección del código de autorización. Imagina que un atacante consigue un código de autorización que el servidor emitió para la víctima: lo roba del historial del navegador, de un

log de un proxy mal configurado, de un redirect abierto en el sitio del cliente o de una app maliciosa que registró el mismo esquema de URL personalizado. Sin PKCE, ese código es un vale al portador: el atacante lo pega en su propia sesión del cliente, el cliente lo canjea contra el servidor de autorización y recibe un token válido de la víctima. El atacante nunca vio la contraseña y aun así entró.

PKCE cierra esa puerta añadiendo un secreto efímero por transacción. El cliente genera un code_verifier aleatorio, envía su hash en la petición de autorización y presenta el original al canjear el código. El servidor sólo entrega tokens si el hash cuadra. Un código robado sin su verifier no vale nada.

La generación correcta del par, en Python. Los detalles que importan están en los comentarios:

```
import base64
import hashlib
import secrets

def generate_pkce_pair() -> tuple[str, str]:
    """Devuelve (code_verifier, code_challenge) para el método S256.

    El verifier debe tener entre 43 y 128 caracteres del conjunto
    [A-Za-z0-9-._~]. token_urlsafe(64) produce 86 caracteres, que
    aporta 512 bits de entropía: muy por encima del mínimo.
    """
    verifier = secrets.token_urlsafe(64)
    digest = hashlib.sha256(verifier.encode("ascii")).digest()
    # base64url SIN relleno: el '=' final rompe la comparación en
    # bastantes servidores de autorización.
    challenge = base64.urlsafe_b64encode(digest).decode("ascii").rstrip("=")
    return verifier, challenge
```

Y la petición de autorización que lo usa, en un backend FastAPI. El code_verifier se guarda en la sesión del servidor, nunca en localStorage:

```
from urllib.parse import urlencode

from fastapi import APIRouter, Request
from fastapi.responses import RedirectResponse

router = APIRouter()

AUTH_ENDPOINT = "https://auth.example.com/oauth2/authorize"
CLIENT_ID = "app-web"
REDIRECT_URI = "https://app.example.com/callback"

@router.get("/login")
async def login(request: Request) -> RedirectResponse:
    verifier, challenge = generate_pkce_pair()
    state = secrets.token_urlsafe(32)

    # Sesión del servidor respaldada por cookie HttpOnly + Secure + SameSite=Lax.
    # En una SPA sin backend, sessionStorage; jamás localStorage, que sobrevive
    # al cierre de pestaña y es legible por cualquier script de la página.
    request.session["pkce_verifier"] = verifier
    request.session["oauth_state"] = state

    params = {
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": REDIRECT_URI,
        "scope": "openid profile invoices:read",
        "state": state,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
        # RFC 8707: restringe la audiencia del token que se va a emitir.
        "resource": "https://api.example.com/",
    }
```

```
    return RedirectResponse(f"{AUTH_ENDPOINT}?{urlencode(params)}", status_code=302)
```

El downgrade que deja PKCE en nada

Aquí está la parte que casi ningún tutorial menciona. Que el cliente implemente PKCE perfectamente no garantiza que esté protegido, porque quien decide si se aplica es el servidor de autorización.

El ataque de downgrade funciona así: el atacante intercepta o reconstruye la petición de autorización y elimina los parámetros code_challenge y code_challenge_method. Si el servidor acepta esa petición y después acepta canjear el código sin code_verifier, acabamos de volver a 2012. Hay dos defensas, y hacen falta las dos:

- En el servidor de autorización: si un código se emitió con un challenge, el canje debe fallar sin el verifier correcto. Y en OAuth 2.1, una petición de autorización sin code_challenge se rechaza directamente.
- En el cliente: comprueba en los metadatos del servidor que S256 está soportado, y niégate a arrancar si no lo está. Un servidor que no anuncia code_challenge_methods_supported es, con alta probabilidad, un servidor que ignora el challenge que le mandas.

```
import httpx

async def assert_pkce_supported(issuer: str) -> None:
    """Falla al arrancar, no en producción a las tres de la mañana."""
    url = f"{issuer.rstrip('/')}/.well-known/oauth-authorization-server"
    async with httpx.AsyncClient(timeout=5.0) as client:
        meta = (await client.get(url)).raise_for_status().json()

    methods = meta.get("code_challenge_methods_supported", [])
    if "S256" not in methods:
        raise RuntimeError(
            f"{issuer} no anuncia S256 en code_challenge_methods_supported "
            f"(anuncia {methods!r}). El flujo no está protegido contra "
            f"inyección de código."
        )
```

Un apunte sobre state: en OAuth 2.0 era la defensa contra CSRF. Con PKCE obligatorio esa función queda cubierta, porque un código inyectado no se puede canjear. OAuth 2.1 sigue permitiendo state, y sigue siendo útil para llevar estado de la aplicación —a qué página volver después del login, por ejemplo—. Si lo envías, valídalo. Lo que ya no es aceptable es usar state en lugar de PKCE.

Redirect URI: por qué un prefijo es una cuenta comprometida

OAuth 2.1 exige comparación de cadenas exacta entre la redirect_uri de la petición y las registradas para el cliente. Suena a tecnicismo pedante hasta que ves cómo se explota.

Supongamos que registras https://app.example.com/callback y tu servidor compara por prefijo, como hacen varias implementaciones caseras. El atacante prueba https://app.example.com/callback/../../redirect?to=https://evil.test. Pasa el prefijo. Si tu aplicación tiene en cualquier parte un redirect abierto —una página de cierre de sesión que acepta ?next=, un tracker de campañas, un acortador interno—, el código de autorización de la víctima acaba en el servidor del atacante. Y no hace falta ni eso: con comparación por prefijo, https://app.example.com.evil.test/callback pasa un startswith mal escrito sobre el host.

La comparación correcta cabe en cuatro líneas y no admite creatividad:

```
REGISTERED_REDIRECT_URIS = frozenset({
    "https://app.example.com/callback",
    "https://app.example.com/callback/silent",
})

def validate_redirect_uri(candidate: str) -> str:
    # Sin urlparse, sin normalizar, sin bajar a minúsculas, sin quitar
    # barras finales. Igualdad de octetos contra un conjunto cerrado.
    # Cualquier transformación previa es una superficie de bypass.
    if candidate not in REGISTERED_REDIRECT_URIS:
        # No redirijas al informar del error: si lo haces, acabas de
        # convertir tu mensaje de error en el redirect abierto que
        # intentabas evitar. Renderiza el error en tu propio dominio.
        raise ValueError("redirect_uri no registrada")
    return candidate
```

El ataque mix-up y el parámetro iss

Si tu cliente soporta varios proveedores de identidad —"entra con Google, con GitHub o con tu cuenta corporativa"— tienes un problema adicional. En el ataque mix-up, el atacante consigue que empieces el flujo contra un servidor de autorización que él controla y que la respuesta te llegue como si viniera del servidor legítimo. Tu cliente coge el código y lo canjea contra el servidor legítimo, enviando en el proceso el secreto de cliente de ese servidor al sitio equivocado, o aceptando un código emitido por un servidor que no es el que creías.

La RFC 9207 resuelve esto con un parámetro: el servidor de autorización incluye iss en la respuesta de autorización, y el cliente comprueba que coincide con el emisor con el que arrancó el flujo. Son cinco líneas en el callback:

```
from fastapi import HTTPException, Request

EXPECTED_ISSUER = "https://auth.example.com"

@router.get("/callback")
async def callback(request: Request, code: str, state: str, iss: str | None = None):
    # 1. iss (RFC 9207). Si el servidor lo anuncia con
    #    authorization_response_iss_parameter_supported, su ausencia
    #    también es un fallo: significa que alguien lo ha quitado.
    if iss != EXPECTED_ISSUER:
        raise HTTPException(400, "emisor inesperado en la respuesta de autorización")

    # 2. state, en tiempo constante.
    expected_state = request.session.pop("oauth_state", None)
    if not expected_state or not secrets.compare_digest(state, expected_state):
        raise HTTPException(400, "state inválido")

    # 3. El verifier se consume: un código sólo se canjea una vez.
    verifier = request.session.pop("pkce_verifier", None)
    if not verifier:
        raise HTTPException(400, "no hay flujo PKCE en curso")

    async with httpx.AsyncClient(timeout=10.0) as client:
        resp = await client.post(
            "https://auth.example.com/oauth2/token",
            data={
                "grant_type": "authorization_code",
                "code": code,
                "redirect_uri": REDIRECT_URI,
                "client_id": CLIENT_ID,
                "code_verifier": verifier,
                "resource": "https://api.example.com/",
            },
            auth=(CLIENT_ID, CLIENT_SECRET),  # clientes confidenciales
        )
```

```
    resp.raise_for_status()
    return resp.json()
```

Refresh tokens: rotación, familias y la carrera que desconecta usuarios

Un refresh token es, por diseño, la credencial de larga duración de tu sistema. Si se filtra uno de un cliente público —una SPA, una app móvil, un agente que corre en la máquina de alguien— el atacante tiene acceso indefinido y tú no tienes forma de distinguirlo del usuario legítimo.

OAuth 2.1 exige que los refresh tokens de clientes públicos sean de un solo uso (rotación) o estén ligados al remitente (DPoP o mTLS). La rotación es lo más barato de implementar y lo que la mayoría elige, pero sólo funciona si haces la parte que casi nadie hace: detección de reuso.

La idea es esta. Cada vez que se canjea un refresh token, lo invalidas y emites uno nuevo, encadenado al anterior. Todos los tokens descendientes de un mismo login forman una familia. Si alguna vez llega un token ya consumido, sólo hay dos explicaciones: o el cliente reintentó, o alguien robó el token y lo está usando en paralelo con el legítimo. Como no puedes distinguirlas, revocas la familia entera y obligas a un login nuevo. Es agresivo a propósito: convierte un robo silencioso en un evento visible.

```
CREATE TABLE refresh_tokens (
    id            uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    family_id     uuid NOT NULL,              -- constante durante toda la cadena
    token_hash    bytea NOT NULL UNIQUE,      -- SHA-256; nunca el token en claro
    user_id       uuid NOT NULL,
    client_id     text NOT NULL,
    scope         text NOT NULL,
    jkt           text,                       -- huella DPoP, si aplica
    issued_at     timestamptz NOT NULL DEFAULT now(),
    expires_at    timestamptz NOT NULL,
    consumed_at   timestamptz,                -- NULL = todavía vivo
    replaced_by   uuid REFERENCES refresh_tokens(id)
);

-- Barrido rápido al revocar una familia entera.
CREATE INDEX idx_rt_family_live ON refresh_tokens (family_id)
    WHERE consumed_at IS NULL;

-- La familia se revoca de golpe cuando se detecta reuso.
CREATE TABLE revoked_families (
    family_id  uuid PRIMARY KEY,
    revoked_at timestamptz NOT NULL DEFAULT now(),
    reason     text NOT NULL
);
```

Y ahora la parte que rompe en producción. La rotación pura tiene una condición de carrera desagradable: el cliente canjea el refresh token, el servidor emite el nuevo, y la respuesta se pierde —red inestable, túnel que se cae, el móvil que cambia de wifi a datos—. El cliente reintenta con el token antiguo, que ya está consumido, y tu detección de reuso revoca la familia. El usuario, que no ha hecho nada malo, se encuentra en la pantalla de login.

La solución estándar es una ventana de gracia: durante unos segundos después de consumirlo, el token antiguo devuelve exactamente el mismo par que se emitió en su canje original, sin rotar de nuevo y sin revocar nada. Pasada la ventana, el reuso es reuso. Es un compromiso consciente: abres unos segundos en los que un token robado sirve, a cambio de no expulsar a usuarios legítimos por una red mala.

```
import hashlib
from dataclasses import dataclass
from datetime import datetime, timedelta, timezone

GRACE_PERIOD = timedelta(seconds=30)

class TokenReuseDetected(Exception):
    """La familia entera queda revocada."""

@dataclass(frozen=True)
class TokenPair:
    access_token: str
    refresh_token: str

def _hash(token: str) -> bytes:
    return hashlib.sha256(token.encode("utf-8")).digest()

async def rotate_refresh_token(conn, presented: str) -> TokenPair:
    now = datetime.now(timezone.utc)

    # SELECT ... FOR UPDATE serializa los canjes concurrentes del MISMO
    # token. Sin esto, dos peticiones simultáneas emiten dos cadenas
    # distintas y la siguiente rotación revoca la familia sin motivo.
    row = await conn.fetchrow(
        """
        SELECT id, family_id, user_id, client_id, scope, jkt,
               expires_at, consumed_at, replaced_by
          FROM refresh_tokens
         WHERE token_hash = $1
           FOR UPDATE
        """,
        _hash(presented),
    )
    if row is None:
        raise TokenReuseDetected("token desconocido o ya purgado")

    if await conn.fetchval(
        "SELECT 1 FROM revoked_families WHERE family_id = $1", row["family_id"]
    ):
        raise TokenReuseDetected("familia previamente revocada")

    if row["expires_at"] <= now:
        raise TokenReuseDetected("refresh token caducado")

    if row["consumed_at"] is not None:
        within_grace = now - row["consumed_at"] <= GRACE_PERIOD
        if within_grace and row["replaced_by"] is not None:
            # Reintento legítimo: devolvemos el mismo sucesor, no rotamos.
            return await reissue_from(conn, row["replaced_by"])

        # Fuera de la ventana: alguien tiene una copia. Cortamos todo.
        await conn.execute(
            """
            INSERT INTO revoked_families (family_id, reason)
                 VALUES ($1, 'refresh_token_reuse')
            ON CONFLICT (family_id) DO NOTHING
            """,
            row["family_id"],
        )
        raise TokenReuseDetected(f"reuso en la familia {row['family_id']}")

    # Camino feliz: emitir el sucesor y marcar el actual como consumido,
    # todo dentro de la misma transacción.
    new_refresh = secrets.token_urlsafe(48)
    new_id = await conn.fetchval(
        """
        INSERT INTO refresh_tokens
               (family_id, token_hash, user_id, client_id, scope, jkt, expires_at)
        VALUES ($1, $2, $3, $4, $5, $6, $7)
```

```
         RETURNING id
            """,
            row["family_id"], _hash(new_refresh), row["user_id"],
            row["client_id"], row["scope"], row["jkt"], now + timedelta(days=30),
        )
        await conn.execute(
            "UPDATE refresh_tokens SET consumed_at = $1, replaced_by = $2 WHERE id = $3",
            now, new_id, row["id"],
        )

        access = mint_access_token(
            subject=row["user_id"], scope=row["scope"], jkt=row["jkt"],
            ttl=timedelta(minutes=10),
        )
        return TokenPair(access_token=access, refresh_token=new_refresh)
```

Tres decisiones de ese código que conviene no revertir:

- Guardamos el hash, no el token. Una fuga de la base de datos no debería ser una fuga de sesiones activas. Es el mismo razonamiento que con las contraseñas, aunque aquí basta SHA-256 porque el token tiene 384 bits de entropía y no es adivinable por fuerza bruta.
- El FOR UPDATE no es opcional. Dos pestañas del mismo navegador refrescando a la vez es un escenario cotidiano, no un caso exótico.
- Un token desconocido también dispara la alarma. Si ya lo purgaste por antigüedad no puedes saber si era legítimo; y si nunca existió, alguien está probando.

Sobre la vida del access token: la rotación sólo tiene sentido con access tokens cortos. Si tu access token dura ocho horas, revocar la familia no impide nada durante las próximas ocho horas. Entre cinco y quince minutos es el rango razonable para la mayoría de APIs; por debajo de eso el coste de refrescar empieza a notarse en la latencia p99.

DPoP: cuando robar el token deja de servir de nada

Todo lo anterior protege la obtención del token. Ninguna de esas defensas hace nada si el atacante roba el access token ya emitido: un bearer token es literalmente un vale al portador. Quien lo tenga, entra.

DPoP (Demonstrating Proof-of-Possession, RFC 9449, septiembre de 2023) ataca ese problema. El cliente genera un par de claves asimétricas, y cada petición —tanto al servidor de autorización como a la API— lleva una prueba firmada con la clave privada. El token queda ligado a la clave pública. Un token robado sin la clave privada correspondiente es inerte.

El mecanismo tiene tres piezas:

- Un DPoP proof: un JWT efímero con typ: "dpop+jwt", la clave pública en la cabecera jwk y, en el cuerpo, el método HTTP (htm), la URI (htu), un identificador único (jti) y la marca de tiempo (iat). Viaja en la cabecera DPoP.
- La confirmación cnf.jkt dentro del access token: la huella SHA-256 de la clave pública (JWK thumbprint, RFC 7638). Es lo que ata el token a la clave.
- El claim ath en el proof cuando se presenta junto a un access token: el hash del propio token. Impide que alguien reutilice un proof capturado con otro token.

La petición completa se ve así. Fíjate en que el esquema de Authorization ya no es Bearer:

```
GET /v1/invoices HTTP/1.1
Host: api.example.com
```

```
Authorization: DPoP eyJhbGciOiJFUzI1NiIsInR5cCI6ImF0K2p3dCIsImtpZCI6...
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6...
```

La generación del proof en el navegador, con WebCrypto y la clave marcada como no extraíble. Ese detalle es la mitad del valor de DPoP: una clave no extraíble guardada en IndexedDB no puede ser leída por JavaScript, ni siquiera por el script del atacante que se coló por un XSS. Puede usarla mientras la página esté comprometida, pero no puede exfiltrarla y seguir usándola mañana desde otra máquina.

```
const ALG: EcKeyGenParams = { name: "ECDSA", namedCurve: "P-256" };

function b64url(bytes: ArrayBuffer | Uint8Array): string {
  const arr = bytes instanceof Uint8Array ? bytes : new Uint8Array(bytes);
  return btoa(String.fromCharCode(...arr))
    .replace(/\+/g, "-")
    .replace(/\//g, "_")
    .replace(/=+$/, "");
}

export async function createDPoPKeyPair(): Promise<CryptoKeyPair> {
  // extractable = false. Ni tu propio código puede exportar la privada.
  // Persiste el CryptoKeyPair en IndexedDB: structured clone lo admite.
  return crypto.subtle.generateKey(ALG, false, ["sign", "verify"]);
}

async function publicJwk(key: CryptoKey): Promise<JsonWebKey> {
  const jwk = await crypto.subtle.exportKey("jwk", key);
  // La cabecera jwk sólo puede llevar la parte pública, y en forma
  // canónica: kty, crv, x, y para EC. Cualquier campo extra cambia
  // la huella y el servidor rechazará el proof.
  return { kty: jwk.kty, crv: jwk.crv, x: jwk.x, y: jwk.y };
}

export async function createProof(
  keys: CryptoKeyPair,
  method: string,
  url: string,
  opts: { accessToken?: string; nonce?: string } = {},
): Promise<string> {
  // htu = URI sin query ni fragmento. Incluirlos es el error nº1
  // y produce un 401 que cuesta horas de depurar.
  const u = new URL(url);
  const htu = u.origin + u.pathname;

  const header = {
    typ: "dpop+jwt",
    alg: "ES256",
    jwk: await publicJwk(keys.publicKey),
  };

  const payload: Record<string, unknown> = {
    jti: crypto.randomUUID(),
    htm: method.toUpperCase(),
    htu,
    iat: Math.floor(Date.now() / 1000),
  };

  if (opts.accessToken) {
    // ath: SHA-256 del access token, base64url. Obligatorio cuando el
    // proof acompaña a un token; ata este proof a ESE token concreto.
    const digest = await crypto.subtle.digest(
      "SHA-256",
      new TextEncoder().encode(opts.accessToken),
    );
    payload.ath = b64url(digest);
  }
```

```
  if (opts.nonce) payload.nonce = opts.nonce;

  const signingInput =
    b64url(new TextEncoder().encode(JSON.stringify(header))) +
    "." +
    b64url(new TextEncoder().encode(JSON.stringify(payload)));

  const sig = await crypto.subtle.sign(
    { name: "ECDSA", hash: "SHA-256" },
    keys.privateKey,
    new TextEncoder().encode(signingInput),
  );
  return signingInput + "." + b64url(sig);
}
```

La validación en el servidor de recursos es donde se gana o se pierde la protección. Ocho comprobaciones, y saltarse cualquiera deja un agujero concreto:

```
import base64
import hashlib
import json
from datetime import datetime, timezone

from jwcrypto import jwk, jws
import redis.asyncio as redis

ALLOWED_ALGS = {"ES256", "ES384", "PS256", "RS256"}
IAT_SKEW_SECONDS = 60          # ventana de aceptación del proof
JTI_TTL_SECONDS = 300          # caché de replay, holgada respecto al skew

def _b64u(data: bytes) -> str:
    return base64.urlsafe_b64encode(data).decode("ascii").rstrip("=")

def _b64u_decode(segment: str) -> bytes:
    # base64url sin relleno: hay que reponerlo antes de decodificar.
    return base64.urlsafe_b64decode(segment + "=" * (-len(segment) % 4))

class DPoPError(Exception):
    """Se traduce a 401 con WWW-Authenticate: DPoP error='invalid_dpop_proof'."""

async def verify_dpop_proof(
    rds: redis.Redis,
    proof: str,
    method: str,
    url: str,
    access_token: str,
    token_cnf_jkt: str,
    expected_nonce: str | None = None,
) -> None:
    header = json.loads(_b64u_decode(proof.split(".")[0]))

    # 1. typ. Sin esto, un JWT de otro contexto (un id_token, por ejemplo)
    #    podría colar como proof: es confusión de tipos de token.
    if header.get("typ") != "dpop+jwt":
        raise DPoPError("typ incorrecto")

    # 2. alg asimétrico y de la lista blanca. 'none' y los HS* deben morir
    #    aquí: con HS256 la 'clave pública' del header sería la clave de
    #    firma y cualquiera podría fabricar proofs.
    if header.get("alg") not in ALLOWED_ALGS:
        raise DPoPError("alg no permitido")

    # 3. La clave pública viene embebida y NO puede traer material privado.
    raw_jwk = header.get("jwk") or {}
    if raw_jwk.get("d") or raw_jwk.get("k"):
        raise DPoPError("jwk con material privado o simétrico")
    key = jwk.JWK(**raw_jwk)
```

```
    # 4. Firma válida con esa misma clave.
    verifier = jws.JWS()
    verifier.deserialize(proof)
    verifier.verify(key)                      # lanza si no cuadra
    claims = json.loads(verifier.payload)

    # 5. La petición que se está firmando es ESTA petición.
    u = url.split("?", 1)[0].split("#", 1)[0]
    if claims.get("htm") != method.upper() or claims.get("htu") != u:
        raise DPoPError("htm/htu no coinciden con la petición")

    # 6. Frescura. iat en el futuro también se rechaza.
    now = int(datetime.now(timezone.utc).timestamp())
    iat = claims.get("iat")
    if not isinstance(iat, int) or abs(now - iat) > IAT_SKEW_SECONDS:
        raise DPoPError("proof fuera de la ventana temporal")

    if expected_nonce is not None and claims.get("nonce") != expected_nonce:
        raise DPoPError("nonce ausente o caducado")

    # 7. Anti-replay. SET NX es atómico: el primero que llega gana.
    #    Sin esta caché, un proof interceptado se puede reenviar durante
    #    los 60 segundos de la ventana de iat.
    jti = claims.get("jti")
    if not jti or not await rds.set(f"dpop:jti:{jti}", "1", ex=JTI_TTL_SECONDS, nx=True):
        raise DPoPError("jti repetido")

    # 8. Las dos ataduras finales: el proof corresponde a ESTE access token,
    #    y la clave del proof es la que el token declara en cnf.jkt.
    expected_ath = _b64u(hashlib.sha256(access_token.encode()).digest())
    if claims.get("ath") != expected_ath:
        raise DPoPError("ath no corresponde al access token presentado")

    if key.thumbprint() != token_cnf_jkt:
        raise DPoPError("la clave del proof no es la ligada al token")
```

El paso 7 merece un comentario operativo. La caché de jti tiene que ser compartida entre todas las réplicas de tu API: una caché en memoria por proceso significa que el atacante sólo tiene que reintentar hasta que el balanceador lo mande a otro pod. Redis con SET NX EX resuelve el caso normal. Si Redis se cae, tienes que decidir de antemano si fallas abierto —aceptas proofs sin comprobar replay, y la petición sigue exigiendo firma válida, htu correcto e iat fresco— o fallas cerrado. Para la mayoría de APIs, fallar abierto durante una caída de Redis es el compromiso correcto: el atacante necesita además haber interceptado un proof válido en los últimos 60 segundos.

Nonces: cuando 60 segundos son demasiados

La ventana de iat es un compromiso entre desfase de relojes y superficie de replay. Si tu perfil de riesgo no la tolera, el servidor puede exigir un nonce que él mismo emite. El baile es un rechazo inicial que el cliente debe saber manejar:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP algs="ES256", error="use_dpop_nonce",
    error_description="Se requiere un nonce en el proof"
DPoP-Nonce: eyJ7S_zG.vYA.hM
```

El cliente guarda el nonce, rehace el proof incluyéndolo y reintenta. En el servidor de autorización el mismo caso devuelve un 400 con "error": "use_dpop_nonce" en el cuerpo JSON. Implementa el reintento automático una sola vez en tu capa HTTP; un bucle sin tope ante un servidor que rota nonces agresivamente es una tormenta de peticiones esperando a ocurrir.

DPoP o mTLS

Los certificados de cliente mutuos (RFC 8705) resuelven el mismo problema y llevan más tiempo en producción. La elección real es operativa, no criptográfica: mTLS exige que el TLS termine donde puedas ver el certificado del cliente —lo que choca con casi cualquier CDN o balanceador gestionado— y exige una PKI con su emisión, distribución y rotación. DPoP funciona a nivel de aplicación, atraviesa proxies sin ceremonia y la clave la genera el propio cliente. A cambio, pagas una firma por petición: entre 0,3 y 1 ms con ES256 en hardware moderno, que en el cliente casi nunca importa y en un servidor de recursos con mucho tráfico conviene medir. FAPI 2.0, el perfil de banca abierta, acepta las dos.

Token Exchange: delegación sin suplantación

Hasta aquí hemos hablado de un cliente y una API. En cuanto aparece una cadena de servicios —un gateway que llama a un servicio de facturación que llama a un servicio de pagos— surge una pregunta incómoda: ¿qué token presenta cada salto?

Las dos respuestas habituales son malas. Reenviar el token original del usuario a todos los servicios significa que el servicio de pagos recibe un token con todos los permisos del usuario, incluidos los que nada tienen que ver con pagos, y que cualquier servicio comprometido de la cadena se convierte en el usuario. Usar credenciales de servicio propias pierde por completo la identidad de quien inició la petición: tus logs de auditoría dicen "billing-service borró la factura" y no sirven para nada.

La RFC 8693 (Token Exchange, enero de 2020) define un grant específico para esto. Un servicio presenta el token que recibió y pide a cambio un token nuevo, restringido a la audiencia del siguiente salto y con menos permisos.

```
TOKEN_EXCHANGE = "urn:ietf:params:oauth:grant-type:token-exchange"
ACCESS_TOKEN_TYPE = "urn:ietf:params:oauth:token-type:access_token"

async def exchange_for_downstream(
    client: httpx.AsyncClient,
    incoming_token: str,
    audience: str,
    scope: str,
) -> str:
    """Canjea el token del usuario por uno acotado al servicio de destino."""
    resp = await client.post(
        "https://auth.example.com/oauth2/token",
        data={
            "grant_type": TOKEN_EXCHANGE,
            "subject_token": incoming_token,
            "subject_token_type": ACCESS_TOKEN_TYPE,
            # actor_token identifica a QUIÉN actúa en nombre del sujeto.
            # Su presencia es lo que convierte esto en delegación en vez
            # de suplantación: el token resultante llevará un claim 'act'.
            "actor_token": await service_identity_token(client),
            "actor_token_type": ACCESS_TOKEN_TYPE,
            # Restringir audiencia y scope es el punto de todo el ejercicio.
            # Un canje que devuelve los mismos permisos no aporta nada.
            "audience": audience,
            "scope": scope,
            "requested_token_type": ACCESS_TOKEN_TYPE,
        },
        auth=(SERVICE_CLIENT_ID, SERVICE_CLIENT_SECRET),
    )
    resp.raise_for_status()
    body = resp.json()

    # La RFC obliga al servidor a devolver issued_token_type. Compruébalo:
    # un servidor puede legítimamente devolver un tipo distinto del pedido.
```

```
    if body["issued_token_type"] != ACCESS_TOKEN_TYPE:
        raise RuntimeError(f"tipo inesperado: {body['issued_token_type']}")
    return body["access_token"]

# Uso desde el servicio de facturación, llamando a pagos:
downstream = await exchange_for_downstream(
    client,
    incoming_token=user_token,
    audience="https://payments.internal/",
    scope="payments:charge",     # y NADA más
)
```

El token resultante lleva el claim act, que es la diferencia entre delegación y suplantación. Con delegación, el token dice "soy la usuaria alice, y quien actúa por mí es billing-service":

```
{
  "iss": "https://auth.example.com",
  "sub": "alice@example.com",
  "aud": "https://payments.internal/",
  "scope": "payments:charge",
  "exp": 1790000600,
  "act": {
    "sub": "billing-service",
    "act": { "sub": "api-gateway" }
  }
}
```

Los act se anidan: el actor más reciente queda fuera y la cadena completa de delegación es legible. En una investigación post-incidente, eso es la diferencia entre reconstruir qué pasó y adivinarlo. La suplantación —el mismo canje sin actor_token— produce un token sin act, indistinguible de uno emitido directamente al usuario. Es útil para el caso de "soporte entra como el cliente para reproducir un bug", pero entonces la trazabilidad tiene que vivir en otro sitio, y conviene que sea una decisión explícita y no el efecto de haber copiado un ejemplo.

Del lado receptor, la regla no negociable: valida siempre aud. Un servicio que acepta cualquier token bien firmado por tu emisor es un deputy confuso. El token que el servicio de informes emitió para sí mismo sirve entonces para llamar al servicio de pagos.

```
import jwt   # PyJWT

def verify_access_token(token: str, jwks_client: jwt.PyJWKClient) -> dict:
    signing_key = jwks_client.get_signing_key_from_jwt(token).key
    claims = jwt.decode(
        token,
        signing_key,
        algorithms=["ES256"],          # lista blanca explícita, nunca la del header
        issuer="https://auth.example.com",
        audience="https://payments.internal/",   # ESTE servicio, no un comodín
        options={"require": ["exp", "iat", "iss", "aud", "sub"]},
        leeway=30,                     # desfase de reloj, no más
    )
    if "payments:charge" not in claims.get("scope", "").split():
        raise PermissionError("scope insuficiente")
    return claims
```

Descubrimiento: dejar de cablear URLs a mano

Dos RFCs de metadatos evitan que la configuración de tu cliente sea una lista de URLs copiadas de un correo.

RFC 8414 (Authorization Server Metadata) publica en /.well-known/oauth-authorization-server los endpoints, los algoritmos soportados y —importante para lo que vimos antes— code_challenge_methods_supported y dpop_signing_alg_values_supported.

RFC 9728 (Protected Resource Metadata) es más reciente y resuelve el problema inverso: dado el recurso, ¿quién lo autoriza? El servidor de recursos publica sus metadatos y, cuando rechaza una petición sin token, apunta a ellos en la cabecera:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api",
    resource_metadata="https://api.example.com/.well-known/oauth-protected-resource"
```

```
{
  "resource": "https://api.example.com/",
  "authorization_servers": ["https://auth.example.com"],
  "scopes_supported": ["invoices:read", "invoices:write", "payments:charge"],
  "bearer_methods_supported": ["header"],
  "dpop_bound_access_tokens_required": true
}
```

Este par es exactamente lo que usa la especificación de autorización de MCP: un servidor MCP se declara servidor de recursos protegido, el cliente descubre su servidor de autorización, se registra dinámicamente si hace falta (RFC 7591) y ejecuta un flujo OAuth 2.1 con PKCE. Si estás construyendo integraciones con agentes, este es el camino trazado, y merece la pena seguirlo en vez de inventar un esquema de API keys propio.

Y una pieza más que se pasa por alto: los indicadores de recurso de la RFC 8707, el parámetro resource que aparece en los ejemplos de arriba. Sin él, el servidor de autorización emite un token de audiencia genérica que vale para todas tus APIs. Con él, pides explícitamente un token para https://api.example.com/ y ese token no sirve en ningún otro sitio. Es una línea de código y reduce el radio de explosión de cada token filtrado.

Validar el token en el resource server: las cinco comprobaciones y las dos que casi nadie hace

Toda la guía hasta aquí habla del lado del cliente y del servidor de autorización. Pero el eslabón donde de verdad se rompen las cosas es el tercero: la API que recibe el token y decide si lo cree. Un flujo impecable con PKCE, rotación y DPoP no sirve de nada si el servicio de destino acepta cualquier JWT que traiga una firma con buena pinta.

La validación completa son cinco comprobaciones y ninguna es opcional:

1. Firma, con una clave que hayas obtenido del emisor por un canal de confianza, y con el algoritmo fijado por ti.
2. Emisor (iss): comparación exacta contra el emisor que esperas. No "contiene", no "empieza por".
3. Audiencia (aud): este token era para ti, no para el servicio de al lado.
4. Vigencia (exp, y nbf si está): con una tolerancia de reloj pequeña y explícita.
5. Autorización (scope, y lo que uses para permisos finos): que el token permita esta operación sobre este recurso.

Las dos que se saltan con más frecuencia son la tercera y la quinta, y las dos tienen la misma consecuencia: un token perfectamente válido emitido para otra cosa entra por la puerta.

Confusión de algoritmos: nunca preguntes al token qué algoritmo usar

La cabecera de un JWT dice de qué algoritmo se trata. Es un dato que viene del atacante. Si tu librería lee alg de la cabecera y actúa en consecuencia, hay dos ataques clásicos esperando.

El primero es alg: none: el token afirma no estar firmado y una implementación ingenua lo acepta sin más. El segundo es más sutil: el token llega con alg: HS256 cuando tú esperabas RS256. Si tu código llama a una función genérica de verificación pasándole "la clave", y esa clave es la pública RSA, la librería intentará usarla como secreto HMAC. La clave pública es pública: el atacante la tiene, firma con ella, y tu verificación pasa.

La defensa es una línea: pasa siempre una lista explícita de algoritmos permitidos y no la derives del token.

```
import time, threading, requests
from jose import jwt, jwk
from jose.utils import base64url_decode

ISSUER   = "https://auth.ejemplo.com"
AUDIENCE = "https://api.ejemplo.com/facturacion"   # ESTE servicio, no "la API"
ALGOS    = ["RS256"]                               # fijado aquí, nunca leído del token


class JwksCache:
    """Caché de claves con dos límites: no refresca más de una vez por minuto
    (evita que un kid inventado convierta cada petición en un DoS contra el IdP)
    y refresca siempre que pasen 10 minutos (evita quedarse clavado tras una rotación)."""

    def __init__(self, url, min_interval=60, max_age=600):
        self._url = url
        self._keys = {}
        self._fetched_at = 0.0
        self._min_interval = min_interval
        self._max_age = max_age
        self._lock = threading.Lock()

    def _refresh(self):
        doc = requests.get(self._url, timeout=3).json()
        self._keys = {k["kid"]: k for k in doc["keys"]}
        self._fetched_at = time.monotonic()

    def get(self, kid):
        now = time.monotonic()
        if kid in self._keys and now - self._fetched_at < self._max_age:
            return self._keys[kid]
        with self._lock:
            if kid in self._keys and now - self._fetched_at < self._max_age:
                return self._keys[kid]
            if now - self._fetched_at < self._min_interval:
                # Demasiado pronto para volver a preguntar. Si la tenemos vieja,
                # la usamos; si no, rechazamos. No castigamos al IdP.
                if kid in self._keys:
                    return self._keys[kid]
                raise KeyUnavailable(kid)
            self._refresh()
        if kid not in self._keys:
            raise KeyUnavailable(kid)
        return self._keys[kid]


jwks = JwksCache(f"{ISSUER}/.well-known/jwks.json")


def validar_access_token(token: str, scope_requerido: str) -> dict:
    cabecera = jwt.get_unverified_header(token)
```

```
    # 1. El algoritmo lo decidimos nosotros.
    if cabecera.get("alg") not in ALGOS:
        raise TokenInvalido("algoritmo no permitido")

    # RFC 9068: un access token JWT se marca con typ "at+jwt". Si el emisor lo
    # pone, compruébalo: es lo que impide aceptar un ID token como si fuera de acceso.
    if cabecera.get("typ", "at+jwt").lower() not in ("at+jwt", "application/at+jwt"):
        raise TokenInvalido("este JWT no es un access token")

    clave = jwks.get(cabecera["kid"])

    # 2-4. Firma, emisor, audiencia y vigencia, en una sola llamada y sin excepciones.
    reclamaciones = jwt.decode(
        token,
        clave,
        algorithms=ALGOS,
        issuer=ISSUER,
        audience=AUDIENCE,
        options={
            "verify_signature": True,
            "verify_aud": True,
            "verify_iss": True,
            "verify_exp": True,
            "require_exp": True,
            "leeway": 30,          # tolerancia de reloj explícita y pequeña
        },
    )

    # 5. Autorización. Un token válido no es un token autorizado.
    concedidos = set(reclamaciones.get("scope", "").split())
    if scope_requerido not in concedidos:
        raise Prohibido(f"falta el scope {scope_requerido}")

    return reclamaciones
```

El ID token no es un access token

Merece su propio párrafo porque es, de largo, el error más repetido en integraciones con OpenID Connect. El ID token existe para que tu cliente sepa quién ha iniciado sesión. Su audiencia es el client_id, no la API. No lleva scopes. No está pensado para viajar a un servicio de recursos y, cuando lo hace, normalmente es porque alguien vio un JWT bonito con el nombre del usuario dentro y lo mandó.

Una API que acepta ID tokens acepta, en la práctica, "cualquier persona que haya iniciado sesión en cualquier aplicación de este tenant", porque la comprobación de audiencia —si es que existe— pasa con el client_id de otra aplicación. La comprobación de aud contra la URL concreta de tu servicio, más el typ: at+jwt, cierran los dos caminos a la vez.

JWT local o introspección: es una decisión sobre revocación, no sobre rendimiento

Validar el JWT en local no cuesta nada: una verificación de firma son microsegundos y no hay red de por medio. La introspección (RFC 7662) implica una llamada al servidor de autorización por cada petición, con su latencia y su punto único de fallo.

Puesto así parece que no hay debate, y por eso mucha gente elige mal. Lo que compra la introspección es lo único que el JWT local no puede darte: saber si el token sigue vivo ahora mismo. Un JWT es válido hasta su exp pase lo que pase — lo veremos en el capítulo siguiente. La regla práctica: valida en local y acorta la vida del access token a lo que estés dispuesto a esperar tras una revocación (cinco o diez minutos para la mayoría de las APIs); reserva la introspección para las operaciones donde ese

retraso no sea aceptable, que son muchas menos de las que parece.

Cerrar sesión de verdad: revocación, logout back-channel y el JWT que no se puede matar

"Cerrar sesión" es una de esas funciones que todo el mundo da por implementada porque hay un botón. Luego llega el incidente —un portátil robado, un empleado que se va, una cuenta comprometida— y aparece la pregunta incómoda: ¿cuánto tiempo sigue funcionando el token que ya no debería funcionar?

La respuesta honesta: hasta su exp

Un JWT autocontenido no se consulta con nadie. Esa es toda su gracia y también su límite: el resource server valida la firma y las reclamaciones sin preguntar al emisor, de modo que el emisor no tiene forma de decirle que ese token ya no vale. Puedes borrar la sesión del usuario, revocar el refresh token y desactivar la cuenta, y el access token que ya está en manos del atacante sigue siendo criptográficamente válido hasta el segundo exacto de su exp.

De ahí sale la conclusión que conviene escribir en la documentación del servicio: la vida del access token es tu ventana de revocación. No es un parámetro de rendimiento que se sube para ahorrar llamadas al endpoint de token; es el tiempo máximo que un atacante conserva el acceso después de que lo hayas echado. Quince minutos es un compromiso habitual. Una hora es una decisión que hay que poder defender. Veinticuatro horas, que se ve más de lo que debería, significa que el botón de cerrar sesión es decorativo durante un día entero.

Qué revoca de verdad la revocación (RFC 7009)

El endpoint de revocación existe y hay que llamarlo, pero conviene saber qué hace. Le mandas un token y el servidor invalida la concesión asociada. Dos detalles que sorprenden:

- Revocar el refresh token normalmente revoca también los access tokens emitidos a partir de él en el registro del servidor de autorización — pero eso sólo afecta a la introspección. Si tus APIs validan en local, no se enteran. Revocar cambia el futuro (no se emitirán tokens nuevos), no el presente.
- El servidor responde 200 aunque el token no exista o ya estuviera revocado. Es deliberado: una respuesta distinta convertiría el endpoint en un oráculo para averiguar si un token es válido. Así que un 200 no significa "he revocado algo", significa "deja de usar ese token". No lo uses como confirmación.

```
import httpx

async def cerrar_sesion(cliente_http, refresh_token, access_token):
    """Revoca en el orden correcto: primero el refresh (corta el futuro),
    después el access (por si el emisor lo registra)."""
    datos_base = {"client_id": CLIENT_ID, "client_secret": CLIENT_SECRET}

    for token, hint in ((refresh_token, "refresh_token"), (access_token, "access_token")):
        if not token:
            continue
        try:
            await cliente_http.post(
                f"{ISSUER}/oauth2/revoke",
                data={**datos_base, "token": token, "token_type_hint": hint},
                timeout=3,
            )
        except httpx.HTTPError:
            # Un fallo aquí NO debe impedir el cierre de sesión local. Se registra
```

```
            # y se reintenta en segundo plano; el usuario ya ha dicho que se va.
            log.warning("revocacion_fallida", extra={"hint": hint})
            encolar_reintento_revocacion(token, hint)

    borrar_sesion_local()
```

Back-channel logout: el único que sobrevive al bloqueo de cookies de terceros

Cuando el usuario cierra sesión en el proveedor de identidad, las demás aplicaciones tienen que enterarse. Durante años esto se hacía con front-channel logout: el IdP cargaba un iframe oculto por cada aplicación y cada una limpiaba su cookie. Ese mecanismo depende de enviar cookies en peticiones de terceros, exactamente lo que Safari, Firefox y ahora Chrome bloquean por defecto. El iframe carga, la aplicación no ve la cookie de sesión, y el logout falla en silencio: nadie devuelve un error porque nadie está mirando.

El sustituto es back-channel logout: el IdP hace una llamada de servidor a servidor a un endpoint tuyo con un logout token, un JWT firmado que identifica la sesión que se cierra. No hay navegador de por medio, así que no hay cookies de terceros que bloquear.

Ese endpoint es público y recibe JWTs de fuera, con lo que se valida con el mismo rigor que un access token, más tres comprobaciones propias:

```
from fastapi import APIRouter, Form, HTTPException

router = APIRouter()

@router.post("/oidc/backchannel-logout")
async def backchannel_logout(logout_token: str = Form(...)):
    cabecera = jwt.get_unverified_header(logout_token)
    if cabecera.get("alg") not in ALGOS:
        raise HTTPException(400, "algoritmo no permitido")

    r = jwt.decode(
        logout_token, jwks.get(cabecera["kid"]),
        algorithms=ALGOS, issuer=ISSUER, audience=CLIENT_ID,
        options={"verify_exp": True, "require_exp": True, "leeway": 30},
    )

    # 1. Tiene que declararse como logout token. Sin esto, un ID token normal
    #    serviría para cerrar la sesión de cualquiera.
    evento = r.get("events", {})
    if "http://schemas.openid.net/event/backchannel-logout" not in evento:
        raise HTTPException(400, "no es un logout token")

    # 2. Un logout token NUNCA lleva nonce. Si lo lleva, es un ID token reciclado.
    if "nonce" in r:
        raise HTTPException(400, "logout token con nonce")

    # 3. Anti-replay: el mismo jti no se procesa dos veces.
    if not marcar_jti_una_vez(r["jti"], ttl=600):
        return {"ok": True}

    sid, sub = r.get("sid"), r.get("sub")
    if sid:
        await terminar_sesiones_por_sid(sid)      # preferible: sólo esa sesión
    elif sub:
        await terminar_sesiones_por_sujeto(sub)   # todas las del usuario
    else:
        raise HTTPException(400, "el logout token no identifica ninguna sesión")

    return {"ok": True}
```

El sid es lo que distingue un logout correcto de uno grosero. Sin él sólo puedes cerrar todas las

sesiones del usuario, y el que estaba trabajando en el portátil se queda fuera porque alguien cerró sesión en el móvil. Si tu IdP lo emite, guárdalo en tu sesión cuando la creas: es la única forma de correlacionar después.

La lista de denegación acotada: cómo acortar la ventana sin volver a la introspección

Entre "validar en local y aguantar la ventana" y "introspección en cada petición" hay una opción intermedia que funciona bien: una lista de denegación en Redis, consultada en local, con una propiedad que la hace barata — cada entrada caduca cuando caduca el token. La lista nunca crece más allá de los tokens que siguen vivos.

```
import time

async def revocar_sesion(redis, sid: str, exp_maximo: int):
    """Marca una sesión como terminada sólo hasta que sus tokens caduquen solos."""
    ttl = max(1, exp_maximo - int(time.time()) + 30)
    await redis.setex(f"revoked:sid:{sid}", ttl, "1")


async def esta_revocado(redis, reclamaciones: dict) -> bool:
    sid = reclamaciones.get("sid")
    if sid and await redis.exists(f"revoked:sid:{sid}"):
        return True
    return bool(await redis.exists(f"revoked:jti:{reclamaciones['jti']}"))
```

Cuesta una consulta a Redis por petición —decenas de microsegundos, no un viaje al IdP—, funciona sin coordinación entre servicios y degrada de forma razonable: si Redis no responde, decides si fallas abierto (sigues aceptando tokens válidos, que es lo que hacías antes de tener lista) o cerrado. Para la mayoría de las APIs, fallar abierto con una alerta es la respuesta correcta, porque la alternativa es que una caída de Redis tumbe el servicio entero por un mecanismo que es una mejora opcional sobre la caducidad.

Aplicaciones en el navegador: por qué el token no debería vivir ahí y el patrón BFF

La discusión sobre dónde guarda una SPA sus tokens lleva años dando vueltas en los mismos términos — localStorage frente a sessionStorage frente a una variable en memoria — y el marco entero está mal planteado. El IETF publicó la guía definitiva como RFC 10017, OAuth 2.0 for Browser-Based Apps, y OAuth 2.1 la incorpora junto con el Security BCP (RFC 9700). Conviene leerla entera, pero su conclusión práctica se resume en una frase: el navegador es un entorno donde no se puede guardar nada a salvo de un XSS, así que la pregunta útil no es dónde guardar el token sino si el token tiene que llegar al navegador.

Por qué la comparación localStorage/memoria es menos importante de lo que parece

Si hay ejecución de JavaScript arbitrario en tu origen, el atacante ya ha ganado en todos los escenarios. Lo que cambia es el radio de explosión, y esa diferencia sí importa:

- localStorage: el token se lee con una línea y se envía a un servidor del atacante. El acceso sobrevive al cierre de la pestaña y viaja fuera de la máquina. Es el peor caso porque el robo es exfiltrable.

- Variable en memoria: no hay API para leerla desde otro contexto, así que el atacante no puede extraerla tan fácilmente; en cambio sí puede usar la sesión desde dentro de la página mientras esté abierta. El daño es real pero está acotado a la vida de la pestaña.
- Cookie HttpOnly: el JavaScript no la lee, punto. El atacante sigue pudiendo hacer peticiones desde el origen víctima y el navegador adjuntará la cookie, pero no puede llevarse la credencial. El robo pasa de ser un token exfiltrado a ser una sesión montada, que termina cuando la pestaña se cierra.

Ninguna de las tres impide el XSS. La tercera convierte una fuga permanente en un abuso temporal, y ese es el único eje en el que se puede mejorar sin salir del navegador.

El BFF: el token no cruza la frontera

El patrón backend-for-frontend lleva la idea hasta el final. La SPA deja de ser un cliente OAuth. El cliente pasa a ser un backend pequeño, del mismo origen que la aplicación, que hace el flujo completo, guarda los tokens en su lado y expone al navegador únicamente una cookie de sesión opaca.

El navegador nunca ve un access token. No hay nada que exfiltrar. Como efecto secundario, el cliente pasa a ser confidencial: puede autenticarse con un secreto, lo que habilita rotación de refresh tokens con detección de reutilización y comprobaciones que un cliente público no puede hacer.

```
import secrets, hashlib, base64
from fastapi import FastAPI, Request, Response, HTTPException
from fastapi.responses import RedirectResponse
import httpx

app = FastAPI()
SESIONES = redis_client   # en producción: Redis con TTL, no un diccionario

def _pkce():
    verifier = base64.urlsafe_b64encode(secrets.token_bytes(64)).decode().rstrip("=")
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()).decode().rstrip("=")
    return verifier, challenge


@app.get("/auth/login")
async def login(response: Response):
    verifier, challenge = _pkce()
    state = secrets.token_urlsafe(32)
    # El estado del flujo vive en el servidor, atado a una cookie temporal.
    await SESIONES.setex(f"flow:{state}", 300, verifier)

    response = RedirectResponse(
        f"{ISSUER}/oauth2/authorize?response_type=code"
        f"&client_id={CLIENT_ID}&redirect_uri={REDIRECT_URI}"
        f"&scope=openid%20facturacion.read&state={state}"
        f"&code_challenge={challenge}&code_challenge_method=S256"
    )
    response.set_cookie("flow_state", state, httponly=True, secure=True,
                        samesite="lax", max_age=300, path="/auth")
    return response


@app.get("/auth/callback")
async def callback(request: Request, code: str, state: str):
    # El state de la URL tiene que coincidir con el de la cookie: esto es lo que
    # impide que alguien te inyecte SU código de autorización (login CSRF).
    if not secrets.compare_digest(state, request.cookies.get("flow_state", "")):
        raise HTTPException(400, "state no coincide")

    verifier = await SESIONES.getdel(f"flow:{state}")
    if not verifier:
        raise HTTPException(400, "flujo caducado o reutilizado")
```

```
    async with httpx.AsyncClient() as c:
        r = await c.post(f"{ISSUER}/oauth2/token", data={
            "grant_type": "authorization_code", "code": code,
            "redirect_uri": REDIRECT_URI, "client_id": CLIENT_ID,
            "client_secret": CLIENT_SECRET,      # cliente confidencial de verdad
            "code_verifier": verifier,
        }, timeout=5)
    r.raise_for_status()
    tokens = r.json()

    # Los tokens se quedan AQUÍ. Al navegador sólo le llega un identificador opaco.
    sid = secrets.token_urlsafe(32)
    await SESIONES.setex(f"sess:{sid}", 28800, json.dumps(tokens))

    resp = RedirectResponse("/")
    resp.set_cookie("sid", sid, httponly=True, secure=True,
                    samesite="lax", path="/", max_age=28800)
    resp.delete_cookie("flow_state", path="/auth")
    return resp


@app.api_route("/api/{ruta:path}", methods=["GET", "POST", "PUT", "DELETE"])
async def proxy(ruta: str, request: Request):
    sid = request.cookies.get("sid")
    bruto = await SESIONES.get(f"sess:{sid}") if sid else None
    if not bruto:
        raise HTTPException(401)
    tokens = json.loads(bruto)

    # Con SameSite=Lax una petición POST de otro sitio no lleva la cookie, pero
    # Lax no cubre todos los casos: exige la cabecera de origen en métodos que mutan.
    if request.method != "GET" and request.headers.get("origin") not in ORIGENES_PERMITIDOS:
        raise HTTPException(403, "origen no permitido")

    async with httpx.AsyncClient() as c:
        upstream = await c.request(
            request.method, f"{API_BASE}/{ruta}",
            headers={"authorization": f"Bearer {tokens['access_token']}"},
            content=await request.body(), timeout=10)
    return Response(upstream.content, upstream.status_code,
                    media_type=upstream.headers.get("content-type"))
```

Lo que el BFF te cobra

Ningún patrón es gratis y este tiene una factura concreta que conviene mirar antes de adoptarlo:

- Vuelve el estado de servidor. Acabas de reintroducir sesiones, con su almacén, su caducidad y su necesidad de compartirse entre instancias. Si tu frontend era estático en una CDN, ahora hay un servicio que escalar y vigilar.
- Vuelve el CSRF. Al autenticar con cookie en lugar de con una cabecera Authorization, el navegador las adjunta solo. SameSite=Lax cubre la mayor parte, pero no es una defensa completa: hace falta comprobar el origen en los métodos que mutan, o un token anti-CSRF.
- El BFF es ahora la superficie de ataque. Un proxy que reenvía rutas arbitrarias a una API interna con un token privilegiado adjunto es un SSRF esperando a nacer si no validas qué rutas se pueden reenviar.

Dicho eso, para una aplicación web que ya tiene servidor la cuenta sale casi siempre a favor. La comparación honesta no es "SPA pura frente a BFF", sino "una SPA pura con tokens en el navegador y una ventana de exfiltración permanente, frente a un servicio más que ya sabes operar". La segunda es la que puedes explicar en una revisión de seguridad sin ponerte creativo.

Autenticar al cliente sin secretos compartidos: private_key_jwt y la audiencia que ya no admite ambigüedad

Todo lo anterior protege el token una vez emitido. Pero hay una pregunta previa que casi nadie se hace: ¿cómo demuestra tu backend al servidor de autorización que es quien dice ser? En la mayoría de las integraciones la respuesta es un client_secret: una cadena larga que vive en una variable de entorno, viaja en cada petición al token endpoint y que el servidor de autorización también guarda. Es, a todos los efectos, una contraseña compartida, con todos los problemas de una contraseña compartida: se copia a CI, aparece en un volcado de configuración, acaba pegada en un ticket de soporte y, cuando se filtra, no hay forma de saber quién la está usando.

La alternativa madura es private_key_jwt (RFC 7523 y sección 9 de OpenID Connect Core): el cliente guarda una clave privada que nunca sale de su proceso (o de su KMS), publica la parte pública en un JWKS y, en cada petición al token endpoint, firma una aserción corta que demuestra que posee esa clave. El servidor de autorización no guarda ningún secreto del cliente, sólo su clave pública. Si alguien roba la base de datos del servidor de autorización, no se lleva nada con lo que suplantar a tus clientes. Es uno de los dos métodos que exige FAPI 2.0 (el otro es mTLS) y el que recomendaría por defecto para cualquier cliente confidencial nuevo.

La aserción, campo a campo

- iss y sub: ambos, el client_id. No es redundancia: el mismo formato sirve para concesiones por aserción en las que sub es un usuario.
- aud: el issuer del servidor de autorización. Un poco más abajo explico por qué esto ha dejado de ser una elección.
- jti: un identificador único; el servidor debe rechazar un jti que ya ha visto mientras la aserción siga vigente.
- exp e iat: vida de 60 segundos o menos. Una aserción de cliente no se reutiliza; se genera para cada petición.
- Cabecera typ: client-authentication+jwt, el tipo explícito que introducen las actualizaciones de RFC 7523.

```
import time, uuid
import httpx
import jwt  # PyJWT >= 2.8 con cryptography

ISSUER = "https://auth.example.com"          # el issuer validado por discovery
TOKEN_ENDPOINT = "https://auth.example.com/oauth2/token"
CLIENT_ID = "billing-service"

def client_assertion(private_key_pem: bytes, kid: str) -> str:
    now = int(time.time())
    claims = {
        "iss": CLIENT_ID,
        "sub": CLIENT_ID,
        "aud": ISSUER,               # el issuer, nunca la URL del endpoint
        "jti": str(uuid.uuid4()),
        "iat": now,
        "exp": now + 60,
    }
    headers = {"kid": kid, "typ": "client-authentication+jwt"}
    return jwt.encode(claims, private_key_pem, algorithm="ES256", headers=headers)

def client_credentials_token(private_key_pem: bytes, kid: str, scope: str) -> dict:
    resp = httpx.post(
```

```
        TOKEN_ENDPOINT,
        data={
            "grant_type": "client_credentials",
            "scope": scope,
            "client_assertion_type": "urn:ietf:params:oauth:client-assertion-type:jwt-bearer",
            "client_assertion": client_assertion(private_key_pem, kid),
        },
        timeout=5.0,
    )
    resp.raise_for_status()
    return resp.json()
```

Qué hace y por qué importa: cada llamada genera una aserción nueva con un jti distinto y una vida de un minuto, de modo que una aserción capturada en un log o en un proxy caduca antes de ser útil y, aunque no caducara, el servidor la rechazaría por repetida. Fíjate en que el cliente no envía ningún secreto: la identidad va dentro de la firma.

El ataque de inyección de audiencia y por qué aud ya no es negociable

RFC 7523 permitía que el aud fuera el issuer o la URL del token endpoint, y muchas librerías ponían la del endpoint porque "es a donde se envía". En enero de 2025, Pedram Hosseyni, Ralf Küsters y Tim Würtele (Universidad de Stuttgart) comunicaron al grupo de trabajo OAuth una vulnerabilidad que nace precisamente de esa ambigüedad. El escenario: un cliente que habla con varios servidores de autorización, uno de ellos malicioso o comprometido. Ese servidor publica metadatos en los que alguno de sus endpoints (por ejemplo, el de PAR) apunta a la URL del token endpoint del servidor honesto. El cliente construye una aserción con aud igual a "la URL a la que voy a enviar esto", se la entrega al atacante, y el atacante la reenvía al servidor honesto, que la acepta porque el aud coincide con su propio endpoint. Resultado: el atacante se autentica como tu cliente.

El borrador draft-ietf-oauth-rfc7523bis (versión 06, marzo de 2026) cierra la puerta en las dos direcciones:

- El cliente debe usar como aud el issuer del servidor de autorización, y sólo ese valor. La URL del token endpoint queda expresamente prohibida.
- El servidor de autorización debe rechazar cualquier aserción cuyo único aud no sea su propio issuer, comparado como cadena exacta.
- El typ explícito client-authentication+jwt se recomienda a los clientes, pero se desaconseja que el servidor rechace aserciones sin él, para no romper a los clientes que ya cumplen con la audiencia.

La lección práctica es sencilla: el issuer que pones en aud es el que validaste al hacer discovery (RFC 8414 exige comprobar que el issuer de los metadatos coincide con el que pediste), no cualquier URL que aparezca en ellos. Y si operas el servidor de autorización, revisa tu producto: varios proveedores publicaron parches en 2025 para dejar de aceptar el token endpoint como audiencia.

```
# Lado servidor de autorización: validación estricta de la aserción de cliente
import jwt
from jwt import PyJWKClient

ISSUER = "https://auth.example.com"

def authenticate_client(assertion: str, client, replay_cache) -> str:
    jwks = PyJWKClient(client.jwks_uri, cache_keys=True)
    key = jwks.get_signing_key_from_jwt(assertion).key
    claims = jwt.decode(
        assertion,
        key,
        algorithms=["ES256", "PS256"],      # lista cerrada, nunca la cabecera
        audience=ISSUER,
```

```
        options={"require": ["iss", "sub", "aud", "exp", "jti"]},
        leeway=5,
    )
    aud = claims["aud"]
    if isinstance(aud, list) and aud != [ISSUER]:
        raise PermissionError("aud debe contener sólo el issuer")
    if claims["iss"] != client.client_id or claims["sub"] != client.client_id:
        raise PermissionError("iss/sub no coinciden con el cliente")
    if claims["exp"] - claims.get("iat", claims["exp"]) > 300:
        raise PermissionError("aserción con vida excesiva")
    # SET NX con caducidad igual a la vida restante de la aserción
    if not replay_cache.add(f"jti:{client.client_id}:{claims['jti']}", expires_at=claims["exp"]):
        raise PermissionError("aserción repetida")
    return client.client_id
```

PyJWT ya comprueba que ISSUER esté en aud, pero acepta listas con varios valores; la comprobación explícita de la lista cierra el caso aud: [issuer, atacante]. La caché de jti con SET NX es la misma pieza que usamos para DPoP: si ya la tienes, reutilízala.

Rotación de claves del cliente sin cortes

La ventaja operativa de private_key_jwt frente a un secreto es que rotar no exige coordinación. El patrón: generas la clave nueva, la añades al JWKS publicado junto a la antigua, esperas a que caduque la caché del servidor de autorización (normalmente minutos; pregunta a tu proveedor o mira la cabecera Cache-Control del JWKS), empiezas a firmar con la nueva usando su kid y, pasado un margen, retiras la antigua. Si guardas la clave en un KMS o HSM, la clave privada no existe fuera del módulo y la firma es una llamada a la API; el coste es una latencia de uno o dos dígitos de milisegundos por petición de token, irrelevante con tokens de 5 a 15 minutos que ya cacheas.

PAR: sacar la petición de autorización del navegador

En el flujo clásico, todos los parámetros de la autorización viajan en la URL que sigue el navegador del usuario: client_id, redirect_uri, scope, state, code_challenge y, si usas autorizaciones ricas, un JSON entero codificado. Esa URL pasa por el historial, por extensiones, por proxies corporativos y por los logs de cualquier cosa que la toque, y además el usuario (o un atacante) puede modificarla antes de que llegue al servidor. Pushed Authorization Requests (RFC 9126) invierte el orden: el backend del cliente envía primero los parámetros por un canal autenticado al endpoint de PAR, recibe un request_uri opaco de un solo uso y vida corta (típicamente 60 segundos), y el navegador sólo transporta ese identificador.

Lo que compras con esto es más de lo que parece:

- Integridad: nadie puede cambiar scope ni redirect_uri en tránsito, porque ya no están en la URL.
- Autenticación temprana del cliente: el servidor de autorización sabe quién pide la autorización antes de mostrar nada al usuario, así que puede rechazar peticiones malformadas sin exponer una pantalla de consentimiento.
- Confidencialidad: los parámetros no aparecen en logs de navegador ni de proxies.
- URLs cortas: el límite práctico de longitud de URL deja de ser un problema con authorization_details grandes.

```
import base64, hashlib, secrets
from urllib.parse import urlencode
import httpx

PAR_ENDPOINT = "https://auth.example.com/oauth2/par"
AUTHZ_ENDPOINT = "https://auth.example.com/oauth2/authorize"
```

```
def start_login(session, private_key_pem, kid) -> str:
    verifier = secrets.token_urlsafe(64)
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()
    ).rstrip(b"=").decode()
    state = secrets.token_urlsafe(32)
    session["pkce_verifier"], session["oauth_state"] = verifier, state

    resp = httpx.post(PAR_ENDPOINT, data={
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": "https://app.example.com/callback",
        "scope": "openid profile invoices:read",
        "state": state,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
        "client_assertion_type": "urn:ietf:params:oauth:client-assertion-type:jwt-bearer",
        "client_assertion": client_assertion(private_key_pem, kid),  # aud = issuer
    }, timeout=5.0)
    resp.raise_for_status()
    request_uri = resp.json()["request_uri"]   # urn:ietf:params:oauth:request_uri:...
    # El navegador sólo lleva client_id y request_uri
    return AUTHZ_ENDPOINT + "?" + urlencode({"client_id": CLIENT_ID, "request_uri": request_uri})
```

Fíjate en el comentario del aud: la aserción que envías al endpoint de PAR lleva como audiencia el issuer, no la URL de PAR. Es exactamente la situación que explotaba el ataque de inyección de audiencia, y la razón por la que rfc7523bis actualiza también RFC 9126.

Si operas el servidor de autorización, publica require_pushed_authorization_requests: true en los metadatos (del servidor o por cliente) cuando todos tus clientes confidenciales lo soporten; mientras sea opcional, un atacante puede seguir construyendo URLs de autorización manipuladas con tu client_id. PAR es obligatorio en el perfil de seguridad de FAPI 2.0, que lo combina con PKCE y con tokens restringidos por remitente mediante DPoP o mTLS.

PAR y JAR no son lo mismo

JAR (RFC 9101) consiste en firmar la petición de autorización como un JWT, el llamado request object. PAR resuelve la integridad por el canal; JAR la resuelve con una firma. Si ya autenticas al cliente en PAR, JAR añade sobre todo no repudio: una prueba firmada de qué pidió exactamente el cliente, útil en entornos regulados. Para la mayoría de aplicaciones, PAR más private_key_jwt es la combinación con mejor relación entre complejidad y seguridad.

Step-up: cuando el token es válido pero la autenticación no basta

Un access token dice qué puede hacer el cliente en nombre del usuario, pero no dice cómo se autenticó ese usuario ni cuándo. Para ver facturas, un login con contraseña de hace seis horas es aceptable; para cambiar la cuenta bancaria de cobro, quieres un segundo factor de los últimos cinco minutos. Hasta RFC 9470 (septiembre de 2023), cada API resolvía esto a su manera, con errores 403 inventados que el cliente no sabía interpretar. RFC 9470 lo estandariza en tres piezas:

- El servidor de autorización incluye en el token (o en la respuesta de introspección) los claims acr, el nivel de autenticación alcanzado, y auth_time, cuándo ocurrió.
- El resource server, si el nivel o la antigüedad no bastan para esa operación, responde 401 con WWW-Authenticate: Bearer error="insufficient_user_authentication" y los parámetros acr_values y/o max_age que necesita.
- El cliente repite el flujo de autorización con esos parámetros, obtiene un token nuevo y reintenta la

operación.

```
import time
from fastapi import Depends, HTTPException

STRONG_ACR = "urn:example:acr:mfa"   # acuerda los valores con tu servidor de autorización

def require_recent_mfa(max_age: int = 300):
    def dependency(claims: dict = Depends(verified_access_token)):
        acr_ok = claims.get("acr") == STRONG_ACR
        auth_time = claims.get("auth_time")
        fresh = auth_time is not None and time.time() - auth_time <= max_age
        if not (acr_ok and fresh):
            raise HTTPException(
                status_code=401,
                headers={"WWW-Authenticate": (
                    'Bearer error="insufficient_user_authentication", '
                    'error_description="Se requiere MFA reciente", '
                    f'acr_values="{STRONG_ACR}", max_age="{max_age}"'
                )},
            )
        return claims
    return dependency

@app.put("/billing/payout-account")
def change_payout_account(body: PayoutAccount, claims=Depends(require_recent_mfa(300))):
    ...
```

Dos detalles marcan la diferencia en producción. El primero: acr_values es orientativo, mientras que max_age es exigible; OpenID Connect obliga al servidor de autorización a reautenticar si se supera max_age, pero sólo le pide que intente satisfacer acr_values. Por eso el resource server nunca debe dar por hecho que el token nuevo cumple: vuelve a comprobar acr y auth_time en cada petición, como hace la dependencia de arriba. El segundo: si validas tokens con introspección, asegúrate de que tu servidor devuelve acr y auth_time en la respuesta; algunos productos sólo los incluyen en el ID token, que, como vimos, no es un access token.

En el cliente, el tratamiento es un bucle acotado: si una llamada devuelve este error, se lanza una autorización con los parámetros recibidos y se reintenta una sola vez. Un bucle sin límite convierte un desajuste de configuración entre el resource server y el servidor de autorización (un acr que el servidor nunca emite) en un usuario atrapado en pantallas de login infinitas.

Aplicaciones nativas, CLIs y dispositivos sin teclado

Loopback en lugar de esquemas personalizados

Una aplicación de escritorio o una CLI no puede guardar secretos: cualquiera puede extraerlos del binario. Es un cliente público, y RFC 8252 describe cómo debe hacer el flujo: abrir el navegador del sistema (nunca un WebView embebido, que permitiría a la aplicación leer la contraseña del usuario) y recibir el código en una redirección a una interfaz loopback. El servidor de autorización debe aceptar cualquier puerto en las redirect URIs de loopback, porque la aplicación elige un puerto libre en cada ejecución; es la única excepción legítima a la comparación exacta que defendimos antes, y está acotada a 127.0.0.1 y [::1]. Usa la IP literal y no localhost: el nombre puede resolverse a otra interfaz o manipularse con el fichero hosts, y OAuth 2.1 desaconseja usarlo.

```
import base64, hashlib, secrets, webbrowser
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlencode, urlparse, parse_qs

def cli_login(authz_endpoint: str, client_id: str, scope: str) -> dict:
```

```
    verifier = secrets.token_urlsafe(64)
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()).rstrip(b"=").decode()
    state = secrets.token_urlsafe(24)
    result: dict = {}

    class Handler(BaseHTTPRequestHandler):
        def do_GET(self):
            qs = parse_qs(urlparse(self.path).query)
            if qs.get("state", [None])[0] != state:
                self.send_response(400); self.end_headers(); return
            result["code"] = qs.get("code", [None])[0]
            self.send_response(200); self.end_headers()
            self.wfile.write("Puedes cerrar esta ventana.".encode())
        def log_message(self, *args):  # no escribir el código en la terminal
            pass

    server = HTTPServer(("127.0.0.1", 0), Handler)   # puerto libre elegido por el SO
    redirect_uri = f"http://127.0.0.1:{server.server_port}/callback"
    webbrowser.open(authz_endpoint + "?" + urlencode({
        "response_type": "code", "client_id": client_id, "scope": scope,
        "redirect_uri": redirect_uri, "state": state,
        "code_challenge": challenge, "code_challenge_method": "S256",
    }))
    server.timeout = 120
    server.handle_request()          # atiende una sola petición y termina
    server.server_close()
    if not result.get("code"):
        raise RuntimeError("login cancelado o state inválido")
    return {"code": result["code"], "verifier": verifier, "redirect_uri": redirect_uri}
```

El servidor escucha sólo en 127.0.0.1, atiende una única petición y desaparece; el state protege contra otra aplicación local que intente inyectar un código, y PKCE hace inútil un código interceptado. Después, el intercambio en el token endpoint usa el mismo redirect_uri y el verifier, sin secreto de cliente. Si guardas el refresh token, hazlo en el llavero del sistema operativo (Keychain, Credential Manager o Secret Service), nunca en un fichero de texto en el directorio del usuario.

Device flow: útil, y el vector de phishing de moda

El Device Authorization Grant (RFC 8628) resuelve el login en televisores, consolas o terminales sin navegador: el dispositivo muestra un código corto, el usuario lo introduce en otra pantalla y el dispositivo sondea el token endpoint hasta recibir el token. El problema es que la pantalla en la que el usuario introduce el código no sabe qué dispositivo lo pidió. Un atacante puede iniciar el flujo él mismo, enviar el código a la víctima con un pretexto creíble ("introduce este código para acceder al documento compartido") y recibir los tokens cuando la víctima se autentica en la página legítima del proveedor, MFA incluido. Campañas como la de Storm-2372 contra tenants de Microsoft 365 desde 2024, y la venta del kit completo como servicio en 2026, lo han convertido en una técnica habitual; Salesforce llegó a retirar el device flow por completo en septiembre de 2025.

Si lo necesitas, trátalo como un permiso excepcional:

- Habilítalo sólo para los client_id que lo necesitan de verdad, nunca como grant disponible por defecto.
- En la pantalla donde se introduce el código, muestra de forma prominente qué aplicación lo pide y desde dónde (IP aproximada y ubicación), y avisa de que nunca debe introducirse un código recibido por correo o mensaje.
- Vidas cortas para el device_code (cinco minutos o menos) y límite de intentos del user_code, para que el atacante tenga una ventana pequeña.

- No emitas refresh tokens de larga duración a clientes de device flow sin restricción por remitente, y alerta cuando la IP que sondea el token endpoint esté en un país distinto del de quien aprueba.

Tests que prueban que tu integración resiste ataques

Los tests de una integración OAuth suelen comprobar que el login funciona. Los que importan comprueban que lo que no debe funcionar, no funciona. Son baratos de escribir contra un servidor de autorización de pruebas (Keycloak en un contenedor o el entorno de desarrollo de tu proveedor) y atrapan regresiones que ninguna revisión de código ve, como una actualización de librería que vuelve a aceptar alg=none o una configuración nueva que relaja la validación de redirect_uri.

```
import pytest

def test_redirect_uri_con_sufijo_rechazado(authz):
    r = authz.authorize(redirect_uri="https://app.example.com/callback/../evil")
    assert r.status_code == 400 and "redirect_uri" in r.text

def test_codigo_sin_verifier_rechazado(authz, code_with_pkce):
    r = authz.token(code=code_with_pkce, code_verifier=None)
    assert r.status_code == 400 and r.json()["error"] == "invalid_grant"

def test_codigo_de_un_solo_uso(authz, code_with_pkce, verifier):
    assert authz.token(code=code_with_pkce, code_verifier=verifier).status_code == 200
    assert authz.token(code=code_with_pkce, code_verifier=verifier).status_code == 400

def test_reuso_de_refresh_revoca_la_familia(authz, tokens):
    rt1 = tokens["refresh_token"]
    rt2 = authz.refresh(rt1).json()["refresh_token"]
    authz.refresh(rt1)                              # reuso: señal de robo
    assert authz.refresh(rt2).status_code == 400    # cae la familia entera

def test_asercion_con_aud_de_endpoint_rechazada(authz, make_assertion):
    bad = make_assertion(aud=authz.token_endpoint)
    assert authz.client_credentials(client_assertion=bad).status_code == 401

@pytest.mark.parametrize("alg", ["none", "HS256"])
def test_resource_server_rechaza_algoritmos_no_permitidos(api, forge_token, alg):
    assert api.get("/invoices", token=forge_token(alg=alg)).status_code == 401
```

Cada test corresponde a una sección de esta guía: redirect_uri exacto, PKCE obligatorio, códigos de un solo uso, detección de reuso por familias, audiencia estricta en la autenticación del cliente y lista cerrada de algoritmos en el resource server. Los fixtures (authz, api, forge_token) son envoltorios finos sobre httpx apuntando a tu entorno de pruebas. Ejecútalos en CI contra la misma versión y configuración del servidor de autorización que usas en producción; un test que pasa contra una configuración distinta sólo te da falsa tranquilidad.

Observabilidad: las señales que avisan antes que el incidente

Un sistema OAuth sano produce pocos errores, y casi todos aburridos. Los interesantes son los que indican que alguien está probando tus defensas o que una defensa se ha roto. Estas son las métricas que merece la pena exportar desde el servidor de autorización, el BFF o los resource servers:

- oauth_refresh_reuse_detected_total: cada reuso de un refresh ya rotado. Un pico aislado puede ser la carrera de la que hablamos; una tendencia es robo de tokens.
- oauth_token_errors_total{error} por tipo: invalid_grant, invalid_client, use_dpop_nonce. Un salto de invalid_client tras un despliegue suele ser una rotación de claves mal hecha o un aud incorrecto.
- oauth_dpop_replay_rejected_total: proofs con jti repetido. En condiciones normales debería ser

prácticamente cero.

- oauth_step_up_challenges_total y su tasa de completado: si los usuarios abandonan el step-up, la operación protegida es segura pero inservible.
- oauth_device_code_approvals_total{country_mismatch}: aprobaciones de device flow en las que el país del sondeo no coincide con el del usuario.
- oauth_jwks_refresh_failures_total: si el resource server no puede refrescar las claves, fallará en cuanto el servidor rote.

```
groups:
  - name: oauth-security
    rules:
      - alert: OAuthRefreshReuseSpike
        expr: sum(rate(oauth_refresh_reuse_detected_total[10m])) > 0.05
        for: 10m
        labels: { severity: page }
        annotations:
          summary: "Reuso de refresh tokens por encima de lo normal: posible robo"
      - alert: OAuthInvalidClientAfterDeploy
        expr: sum(rate(oauth_token_errors_total{error="invalid_client"}[5m])) > 1
        for: 5m
        labels: { severity: ticket }
        annotations:
          summary: "Clientes fallando al autenticarse: revisa rotación de claves o aud"
      - alert: OAuthJWKSRefreshFailing
        expr: increase(oauth_jwks_refresh_failures_total[15m]) > 3
        labels: { severity: page }
        annotations:
          summary: "El resource server no puede refrescar el JWKS"
```

Registra también, en los logs de auditoría, el client_id, el sub, el jkt de DPoP cuando exista y el identificador de familia del refresh token en cada emisión. Cuando llegue la pregunta "¿qué hizo este token robado?", esos cuatro campos son la diferencia entre responder en diez minutos o en tres días. Y nunca registres el token en sí: registra su jti o un hash truncado.

Ocho errores que verás en producción

1. Guardar el code_verifier en localStorage. Sobrevive al cierre de la pestaña y es legible por cualquier script de la página, incluidos los de terceros que metió marketing. Usa sessionStorage o, mejor, la sesión del servidor.
2. Comparar redirect_uri por prefijo o tras normalizar. Ya vimos cómo acaba. Igualdad de octetos contra un conjunto cerrado.
3. No validar aud en el servidor de recursos. El error más extendido y el más caro: convierte cualquier token de tu ecosistema en una llave maestra.
4. Aceptar el alg que viene en el header del JWT. Clásico eterno. Lista blanca explícita en la llamada a decode, y nunca none ni algoritmos simétricos donde esperas asimétricos.
5. Rotación de refresh tokens sin ventana de gracia. Funciona de maravilla en el entorno de pruebas con fibra y falla con usuarios reales en el metro. Se manifiesta como "la app me echa sola" en las reseñas.
6. Access tokens de horas "porque revocar es difícil". Es circular: son difíciles de revocar porque son largos. Tokens de diez minutos más rotación de refresh te dan una ventana de revocación real sin introspección en cada petición.
7. Incluir la query string en htu del proof DPoP. Produce un 401 sin explicación. La especificación es clara: sólo origen y path.

8. Token exchange sin reducir permisos. Si el token de salida tiene el mismo scope y la misma audiencia que el de entrada, has añadido una llamada de red y cero seguridad.

Checklist de producción

- PKCE con S256 en todos los clientes, y verificación al arrancar de que el servidor anuncia S256.
- redirect_uri validadas por igualdad exacta contra una lista cerrada; ningún redirect en las respuestas de error.
- Validación de iss (RFC 9207) si el cliente habla con más de un emisor.
- Access tokens de 5 a 15 minutos, con aud acotada por resource (RFC 8707).
- Refresh tokens rotados, almacenados como hash, con familias, detección de reuso y ventana de gracia de 15–60 segundos.
- DPoP en clientes públicos: clave no extraíble, ath siempre, caché de jti compartida entre réplicas y comprobación de cnf.jkt.
- Token exchange con actor_token, audiencia del siguiente salto y scope reducido en cada canje.
- Endpoint de revocación (RFC 7009) implementado y llamado en el logout. Cerrar sesión borrando una cookie no revoca nada.
- Métricas: tasa de invalid_grant, revocaciones por reuso detectado, rechazos de DPoP por motivo y edad p99 de los access tokens en uso.
- Alertas sobre picos de revocación por familia: un aumento súbito suele ser un despliegue que rompió el reintento, pero podría ser un robo de tokens en curso.

Preguntas frecuentes

¿Necesito PKCE si mi cliente es un backend con secreto? Sí. El secreto de cliente autentica al cliente ante el servidor de autorización; PKCE ata el código a la transacción concreta. Son propiedades distintas y OAuth 2.1 exige las dos. El coste son diez líneas.

¿Puedo guardar tokens en localStorage si mi SPA usa DPoP? Mejora mucho la situación —el token sin la clave privada no vale— pero no la arregla. Un XSS activo puede pedir proofs a la clave no extraíble mientras la página está abierta. Lo correcto sigue siendo el patrón BFF: tokens en el backend, cookie de sesión HttpOnly hacia el navegador.

¿Rotación o DPoP para refresh tokens? OAuth 2.1 te pide una de las dos en clientes públicos. La rotación es más fácil de desplegar y detecta el robo después de que ocurra; DPoP lo previene pero exige soporte en el servidor de autorización. Si puedes, las dos: son ortogonales.

¿Tengo que migrar ya si uso el grant implícito? Sí, y no por cumplimiento. El token en el fragmento de la URL aparece en el historial, puede filtrarse por Referer y es accesible a cualquier script inyectado. La migración a código de autorización con PKCE es mecánica y está muy documentada.

¿Qué hago con los relojes desfasados y el iat del proof? Sincroniza con NTP y usa una ventana de 60 segundos. Si ves rechazos por desfase, mide antes de ampliarla: ampliar la ventana amplía la superficie de replay, y el problema casi siempre es un contenedor sin NTP, no la especificación.

Glosario

- PKCE — Proof Key for Code Exchange (RFC 7636). Secreto efímero por transacción que ata el

código de autorización al cliente que lo pidió.

- Authorization code injection — Ataque en el que un código robado se canjea desde la sesión del atacante.
- DPoP — Demonstrating Proof-of-Possession (RFC 9449). Liga los tokens a una clave que el cliente posee.
- cnf.jkt — Claim de confirmación con la huella SHA-256 (RFC 7638) de la clave pública ligada al token.
- ath — Hash del access token dentro del proof DPoP; impide reutilizar un proof con otro token.
- Familia de refresh tokens — Cadena de tokens descendientes de un mismo login; se revoca entera al detectar reuso.
- Token exchange — RFC 8693. Canje de un token por otro acotado a un destino y unos permisos menores.
- Claim act — Identifica al actor que opera en nombre del sujeto; anidable para representar la cadena de delegación.
- Confused deputy — Servicio con permisos que acepta órdenes de quien no debería, típicamente por no validar aud.
- Sender-constrained token — Token que sólo sirve a quien posee una clave o certificado concretos (DPoP o mTLS).

Por dónde empezar mañana

Si sólo puedes hacer una cosa esta semana, valida aud en todos tus servidores de recursos y comprueba que el algorithms de tu verificador es una lista explícita. Son los dos fallos más comunes y los dos que convierten un token cualquiera en acceso total.

Si puedes hacer dos, añade PKCE con S256 donde falte y baja la vida de los access tokens a diez minutos con rotación de refresh. A partir de ahí, DPoP y token exchange son mejoras incrementales que ya tienen dónde apoyarse.

English version

OAuth 2.1 in Production: Mandatory PKCE, Refresh Token Rotation, DPoP, and Token Exchange Across Service Chains

OAuth 2.0 was never a protocol: it was a menu

OAuth 2.0 shipped in 2012 as a framework, not a closed protocol. It came with four grant types —two of them unsafe for the modern web— and a long list of decisions the implementer had to make without the spec telling them which one was right. Over the following years the IETF published more than twenty RFCs of extensions and security corrections. The practical result is that nobody implemented "OAuth 2.0": everyone implemented OAuth 2.0 plus whichever five RFCs they happened to read.

OAuth 2.1 invents nothing new. It consolidates the Security Best Current Practice (RFC 9700) and the extensions that were already mandatory in practice into a single document, and it removes what has been discouraged for years. It is still an Internet-Draft —revision 15 dates from March 2026— but in practice it is already the profile serious providers apply, and the one the MCP authorization specification requires.

What changes, in six points:

- PKCE is mandatory for every client using the authorization code flow, not just mobile apps and SPAs. A backend holding a client secret needs it too.
- The implicit grant is gone. response_type=token returned the access token in the URL fragment, where it lands in browser history, in referer logs, and within reach of any third-party script on the page.
- Resource Owner Password Credentials is gone. Asking for the user's password inside your application defeats everything OAuth set out to solve.
- Redirect URIs are compared with exact string matching. No prefixes, no path wildcards, no normalizing before comparing.
- Bearer tokens never travel in the query string. No ?access_token=, no variations: they go in the Authorization header.
- Refresh tokens for public clients must be single-use or sender-constrained. This is the point that forces the most code, and we cover it in detail below.

If your integration is three years old and untouched, odds are you violate at least three of the six.

PKCE: the attack it prevents, and the one it doesn't

PKCE (Proof Key for Code Exchange, RFC 7636) is almost always explained as "what mobile apps do because they can't keep a secret". That description is accurate but incomplete, and it is the reason so many people leave it out of the backend.

The real attack is called authorization code injection. Suppose an attacker gets hold of an authorization code the server issued for a victim: stolen from browser history, from a misconfigured proxy log, from an open redirect on the client's site, or from a malicious app that registered the same custom URL scheme. Without PKCE that code is a bearer voucher: the attacker pastes it into their own client session, the

client redeems it against the authorization server, and out comes a valid token for the victim. The attacker never saw a password and still got in.

PKCE closes that door by adding a per-transaction ephemeral secret. The client generates a random code_verifier, sends its hash in the authorization request, and presents the original when redeeming the code. The server only issues tokens if the hash matches. A stolen code without its verifier is worthless.

Generating the pair correctly, in Python. The details that matter are in the comments:

```
import base64
import hashlib
import secrets

def generate_pkce_pair() -> tuple[str, str]:
    """Return (code_verifier, code_challenge) for the S256 method.

    The verifier must be 43 to 128 characters from the set
    [A-Za-z0-9-._~]. token_urlsafe(64) produces 86 characters,
    giving 512 bits of entropy: far above the minimum.
    """
    verifier = secrets.token_urlsafe(64)
    digest = hashlib.sha256(verifier.encode("ascii")).digest()
    # base64url WITHOUT padding: the trailing '=' breaks comparison
    # on a good number of authorization servers.
    challenge = base64.urlsafe_b64encode(digest).decode("ascii").rstrip("=")
    return verifier, challenge
```

And the authorization request that uses it, in a FastAPI backend. The code_verifier lives in the server-side session, never in localStorage:

```
from urllib.parse import urlencode

from fastapi import APIRouter, Request
from fastapi.responses import RedirectResponse

router = APIRouter()

AUTH_ENDPOINT = "https://auth.example.com/oauth2/authorize"
CLIENT_ID = "app-web"
REDIRECT_URI = "https://app.example.com/callback"

@router.get("/login")
async def login(request: Request) -> RedirectResponse:
    verifier, challenge = generate_pkce_pair()
    state = secrets.token_urlsafe(32)

    # Server-side session backed by an HttpOnly + Secure + SameSite=Lax
    # cookie. In a backendless SPA, sessionStorage; never localStorage,
    # which outlives the tab and is readable by any script on the page.
    request.session["pkce_verifier"] = verifier
    request.session["oauth_state"] = state

    params = {
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": REDIRECT_URI,
        "scope": "openid profile invoices:read",
        "state": state,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
        # RFC 8707: constrain the audience of the token about to be issued.
        "resource": "https://api.example.com/",
    }
    return RedirectResponse(f"{AUTH_ENDPOINT}?{urlencode(params)}", status_code=302)
```

The downgrade that reduces PKCE to nothing

Here is the part almost no tutorial mentions. A client implementing PKCE perfectly is not guaranteed to be protected, because the authorization server is what decides whether it gets enforced.

The downgrade attack works like this: the attacker intercepts or reconstructs the authorization request and strips the code_challenge and code_challenge_method parameters. If the server accepts that request and then accepts redeeming the code without a code_verifier, we are back in 2012. Two defenses are needed, and you need both:

- On the authorization server: if a code was issued with a challenge, redemption must fail without the correct verifier. And under OAuth 2.1, an authorization request with no code_challenge is rejected outright.
- On the client: check the server's metadata to confirm S256 is supported, and refuse to start if it isn't. A server that does not advertise code_challenge_methods_supported is, with high probability, a server that ignores the challenge you send it.

```
import httpx

async def assert_pkce_supported(issuer: str) -> None:
    """Fail at boot, not in production at three in the morning."""
    url = f"{issuer.rstrip('/')}/.well-known/oauth-authorization-server"
    async with httpx.AsyncClient(timeout=5.0) as client:
        meta = (await client.get(url)).raise_for_status().json()

    methods = meta.get("code_challenge_methods_supported", [])
    if "S256" not in methods:
        raise RuntimeError(
            f"{issuer} does not advertise S256 in code_challenge_methods_supported "
            f"(it advertises {methods!r}). The flow is not protected against "
            f"code injection."
        )
```

A note on state: in OAuth 2.0 it was the CSRF defense. With PKCE mandatory that job is covered, because an injected code cannot be redeemed. OAuth 2.1 still allows state, and it remains useful for carrying application state —which page to return to after login, for instance. If you send it, validate it. What is no longer acceptable is using state instead of PKCE.

Redirect URI: why a prefix is a compromised account

OAuth 2.1 requires exact string comparison between the request's redirect_uri and the ones registered for the client. It sounds like pedantic nitpicking until you see how it gets exploited.

Say you register https://app.example.com/callback and your server compares by prefix, as several homegrown implementations do. The attacker tries https://app.example.com/callback/../../redirect?to=https://evil.test. It passes the prefix check. If your application has an open redirect anywhere —a logout page that accepts ?next=, a campaign tracker, an internal shortener— the victim's authorization code ends up on the attacker's server. And you don't even need that: with prefix comparison, https://app.example.com.evil.test/callback passes a sloppy startswith on the host.

The correct comparison fits in four lines and leaves no room for creativity:

```
REGISTERED_REDIRECT_URIS = frozenset({
    "https://app.example.com/callback",
    "https://app.example.com/callback/silent",
```

```
})

def validate_redirect_uri(candidate: str) -> str:
    # No urlparse, no normalizing, no lowercasing, no stripping trailing
    # slashes. Byte equality against a closed set. Any transformation
    # beforehand is a bypass surface.
    if candidate not in REGISTERED_REDIRECT_URIS:
        # Do not redirect when reporting the error: if you do, you just
        # turned your error message into the open redirect you were
        # trying to avoid. Render the error on your own domain.
        raise ValueError("redirect_uri is not registered")
    return candidate
```

The mix-up attack and the iss parameter

If your client supports several identity providers —"sign in with Google, with GitHub, or with your corporate account"— you have an extra problem. In the mix-up attack, the attacker gets you to start the flow against an authorization server they control and arranges for the response to reach you as if it came from the legitimate server. Your client takes the code and redeems it against the legitimate server, in the process sending that server's client secret to the wrong place, or accepting a code issued by a server other than the one you thought.

RFC 9207 solves this with one parameter: the authorization server includes iss in the authorization response, and the client checks it matches the issuer it started the flow with. Five lines in the callback:

```
from fastapi import HTTPException, Request

EXPECTED_ISSUER = "https://auth.example.com"

@router.get("/callback")
async def callback(request: Request, code: str, state: str, iss: str | None = None):
    # 1. iss (RFC 9207). If the server advertises
    #    authorization_response_iss_parameter_supported, its absence is
    #    also a failure: it means somebody stripped it.
    if iss != EXPECTED_ISSUER:
        raise HTTPException(400, "unexpected issuer in the authorization response")

    # 2. state, in constant time.
    expected_state = request.session.pop("oauth_state", None)
    if not expected_state or not secrets.compare_digest(state, expected_state):
        raise HTTPException(400, "invalid state")

    # 3. The verifier is consumed: a code is redeemed exactly once.
    verifier = request.session.pop("pkce_verifier", None)
    if not verifier:
        raise HTTPException(400, "no PKCE flow in progress")

    async with httpx.AsyncClient(timeout=10.0) as client:
        resp = await client.post(
            "https://auth.example.com/oauth2/token",
            data={
                "grant_type": "authorization_code",
                "code": code,
                "redirect_uri": REDIRECT_URI,
                "client_id": CLIENT_ID,
                "code_verifier": verifier,
                "resource": "https://api.example.com/",
            },
            auth=(CLIENT_ID, CLIENT_SECRET),  # confidential clients
        )
    resp.raise_for_status()
    return resp.json()
```

Refresh tokens: rotation, families, and the race that logs users out

A refresh token is, by design, the long-lived credential in your system. If one leaks from a public client —a SPA, a mobile app, an agent running on somebody's machine— the attacker has open-ended access and you have no way to tell them apart from the legitimate user.

OAuth 2.1 requires that refresh tokens for public clients be single-use (rotation) or sender-constrained (DPoP or mTLS). Rotation is the cheaper one to implement and what most people choose, but it only works if you do the part almost nobody does: reuse detection.

The idea is this. Every time a refresh token is redeemed, you invalidate it and issue a new one chained to the previous. All tokens descending from a single login form a family. If an already-consumed token ever shows up, there are only two explanations: either the client retried, or somebody stole the token and is using it in parallel with the legitimate one. Since you cannot tell them apart, you revoke the whole family and force a fresh login. It is aggressive on purpose: it turns a silent theft into a visible event.

```
CREATE TABLE refresh_tokens (
    id            uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    family_id     uuid NOT NULL,              -- constant across the chain
    token_hash    bytea NOT NULL UNIQUE,      -- SHA-256; never the raw token
    user_id       uuid NOT NULL,
    client_id     text NOT NULL,
    scope         text NOT NULL,
    jkt           text,                       -- DPoP thumbprint, if any
    issued_at     timestamptz NOT NULL DEFAULT now(),
    expires_at    timestamptz NOT NULL,
    consumed_at   timestamptz,                -- NULL = still live
    replaced_by   uuid REFERENCES refresh_tokens(id)
);

-- Fast sweep when revoking an entire family.
CREATE INDEX idx_rt_family_live ON refresh_tokens (family_id)
    WHERE consumed_at IS NULL;

-- The family is revoked in one shot when reuse is detected.
CREATE TABLE revoked_families (
    family_id  uuid PRIMARY KEY,
    revoked_at timestamptz NOT NULL DEFAULT now(),
    reason     text NOT NULL
);
```

And now the part that breaks in production. Pure rotation has an unpleasant race: the client redeems the refresh token, the server issues the new one, and the response is lost —flaky network, a tunnel that drops, a phone switching from wifi to cellular. The client retries with the old token, which is already consumed, and your reuse detection revokes the family. The user, who did nothing wrong, lands on the login screen.

The standard fix is a grace window: for a few seconds after it is consumed, the old token returns exactly the same pair issued during its original redemption, without rotating again and without revoking anything. Past the window, reuse is reuse. It is a conscious trade-off: you open a few seconds in which a stolen token works, in exchange for not throwing legitimate users out over a bad network.

```
import hashlib
from dataclasses import dataclass
from datetime import datetime, timedelta, timezone

GRACE_PERIOD = timedelta(seconds=30)

class TokenReuseDetected(Exception):
```

```
    """The entire family gets revoked."""

@dataclass(frozen=True)
class TokenPair:
    access_token: str
    refresh_token: str

def _hash(token: str) -> bytes:
    return hashlib.sha256(token.encode("utf-8")).digest()

async def rotate_refresh_token(conn, presented: str) -> TokenPair:
    now = datetime.now(timezone.utc)

    # SELECT ... FOR UPDATE serializes concurrent redemptions of the SAME
    # token. Without it, two simultaneous requests issue two different
    # chains and the next rotation revokes the family for no reason.
    row = await conn.fetchrow(
        """
        SELECT id, family_id, user_id, client_id, scope, jkt,
               expires_at, consumed_at, replaced_by
          FROM refresh_tokens
         WHERE token_hash = $1
           FOR UPDATE
        """,
        _hash(presented),
    )
    if row is None:
        raise TokenReuseDetected("unknown or already-purged token")

    if await conn.fetchval(
        "SELECT 1 FROM revoked_families WHERE family_id = $1", row["family_id"]
    ):
        raise TokenReuseDetected("family previously revoked")

    if row["expires_at"] <= now:
        raise TokenReuseDetected("refresh token expired")

    if row["consumed_at"] is not None:
        within_grace = now - row["consumed_at"] <= GRACE_PERIOD
        if within_grace and row["replaced_by"] is not None:
            # Legitimate retry: hand back the same successor, do not rotate.
            return await reissue_from(conn, row["replaced_by"])

        # Outside the window: somebody holds a copy. Cut everything.
        await conn.execute(
            """
            INSERT INTO revoked_families (family_id, reason)
                 VALUES ($1, 'refresh_token_reuse')
            ON CONFLICT (family_id) DO NOTHING
            """,
            row["family_id"],
        )
        raise TokenReuseDetected(f"reuse in family {row['family_id']}")

    # Happy path: issue the successor and mark the current one consumed,
    # all inside the same transaction.
    new_refresh = secrets.token_urlsafe(48)
    new_id = await conn.fetchval(
        """
        INSERT INTO refresh_tokens
               (family_id, token_hash, user_id, client_id, scope, jkt, expires_at)
        VALUES ($1, $2, $3, $4, $5, $6, $7)
     RETURNING id
        """,
        row["family_id"], _hash(new_refresh), row["user_id"],
        row["client_id"], row["scope"], row["jkt"], now + timedelta(days=30),
    )
    await conn.execute(
        "UPDATE refresh_tokens SET consumed_at = $1, replaced_by = $2 WHERE id = $3",
```

```
        now, new_id, row["id"],
    )

    access = mint_access_token(
        subject=row["user_id"], scope=row["scope"], jkt=row["jkt"],
        ttl=timedelta(minutes=10),
    )
    return TokenPair(access_token=access, refresh_token=new_refresh)
```

Three decisions in that code worth not reverting:

- We store the hash, not the token. A database leak should not be a leak of live sessions. Same reasoning as with passwords, though plain SHA-256 suffices here because the token carries 384 bits of entropy and is not brute-forceable.
- The FOR UPDATE is not optional. Two tabs of the same browser refreshing at once is an everyday scenario, not an exotic edge case.
- An unknown token also raises the alarm. If you already purged it by age you cannot tell whether it was legitimate; and if it never existed, somebody is probing.

On access token lifetime: rotation only makes sense with short access tokens. If your access token lives eight hours, revoking the family prevents nothing for the next eight hours. Five to fifteen minutes is the sensible range for most APIs; below that, the cost of refreshing starts showing up in p99 latency.

DPoP: when stealing the token stops being useful

Everything above protects how the token is obtained. None of those defenses does anything if the attacker steals the access token after issuance: a bearer token is literally a voucher to whoever holds it.

DPoP (Demonstrating Proof-of-Possession, RFC 9449, September 2023) attacks that problem. The client generates an asymmetric key pair, and every request —to the authorization server and to the API alike— carries a proof signed with the private key. The token is bound to the public key. A stolen token without the matching private key is inert.

The mechanism has three pieces:

- A DPoP proof: an ephemeral JWT with typ: "dpop+jwt", the public key in the jwk header, and, in the body, the HTTP method (htm), the URI (htu), a unique identifier (jti), and a timestamp (iat). It travels in the DPoP header.
- The confirmation cnf.jkt inside the access token: the SHA-256 thumbprint of the public key (JWK thumbprint, RFC 7638). This is what binds the token to the key.
- The ath claim in the proof when presented alongside an access token: the hash of the token itself. It stops anyone from replaying a captured proof with a different token.

The full request looks like this. Note that the Authorization scheme is no longer Bearer:

```
GET /v1/invoices HTTP/1.1
Host: api.example.com
Authorization: DPoP eyJhbGciOiJFUzI1NiIsInR5cCI6ImF0K2p3dCIsImtpZCI6...
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6...
```

Generating the proof in the browser, with WebCrypto and the key marked non-extractable. That detail is half the value of DPoP: a non-extractable key stored in IndexedDB cannot be read by JavaScript, not even by the attacker's script that slipped in through an XSS. It can use the key while the page is

compromised, but it cannot exfiltrate it and keep using it tomorrow from another machine.

```
const ALG: EcKeyGenParams = { name: "ECDSA", namedCurve: "P-256" };

function b64url(bytes: ArrayBuffer | Uint8Array): string {
  const arr = bytes instanceof Uint8Array ? bytes : new Uint8Array(bytes);
  return btoa(String.fromCharCode(...arr))
    .replace(/\+/g, "-")
    .replace(/\//g, "_")
    .replace(/=+$/, "");
}

export async function createDPoPKeyPair(): Promise<CryptoKeyPair> {
  // extractable = false. Not even your own code can export the private key.
  // Persist the CryptoKeyPair in IndexedDB: structured clone supports it.
  return crypto.subtle.generateKey(ALG, false, ["sign", "verify"]);
}

async function publicJwk(key: CryptoKey): Promise<JsonWebKey> {
  const jwk = await crypto.subtle.exportKey("jwk", key);
  // The jwk header may only carry the public part, in canonical form:
  // kty, crv, x, y for EC. Any extra field changes the thumbprint and
  // the server will reject the proof.
  return { kty: jwk.kty, crv: jwk.crv, x: jwk.x, y: jwk.y };
}

export async function createProof(
  keys: CryptoKeyPair,
  method: string,
  url: string,
  opts: { accessToken?: string; nonce?: string } = {},
): Promise<string> {
  // htu = URI without query or fragment. Including them is mistake #1
  // and produces a 401 that costs hours to debug.
  const u = new URL(url);
  const htu = u.origin + u.pathname;

  const header = {
    typ: "dpop+jwt",
    alg: "ES256",
    jwk: await publicJwk(keys.publicKey),
  };

  const payload: Record<string, unknown> = {
    jti: crypto.randomUUID(),
    htm: method.toUpperCase(),
    htu,
    iat: Math.floor(Date.now() / 1000),
  };

  if (opts.accessToken) {
    // ath: SHA-256 of the access token, base64url. Required whenever the
    // proof accompanies a token; it binds this proof to THAT token.
    const digest = await crypto.subtle.digest(
      "SHA-256",
      new TextEncoder().encode(opts.accessToken),
    );
    payload.ath = b64url(digest);
  }
  if (opts.nonce) payload.nonce = opts.nonce;

  const signingInput =
    b64url(new TextEncoder().encode(JSON.stringify(header))) +
    "." +
    b64url(new TextEncoder().encode(JSON.stringify(payload)));

  const sig = await crypto.subtle.sign(
    { name: "ECDSA", hash: "SHA-256" },
    keys.privateKey,
```

```
    new TextEncoder().encode(signingInput),
  );
  return signingInput + "." + b64url(sig);
}
```

Validation on the resource server is where the protection is won or lost. Eight checks, and skipping any one of them leaves a specific hole:

```
import base64
import hashlib
import json
from datetime import datetime, timezone

from jwcrypto import jwk, jws
import redis.asyncio as redis

ALLOWED_ALGS = {"ES256", "ES384", "PS256", "RS256"}
IAT_SKEW_SECONDS = 60          # proof acceptance window
JTI_TTL_SECONDS = 300          # replay cache, generous relative to the skew

def _b64u(data: bytes) -> str:
    return base64.urlsafe_b64encode(data).decode("ascii").rstrip("=")

def _b64u_decode(segment: str) -> bytes:
    # base64url without padding: put it back before decoding.
    return base64.urlsafe_b64decode(segment + "=" * (-len(segment) % 4))

class DPoPError(Exception):
    """Maps to 401 with WWW-Authenticate: DPoP error='invalid_dpop_proof'."""

async def verify_dpop_proof(
    rds: redis.Redis,
    proof: str,
    method: str,
    url: str,
    access_token: str,
    token_cnf_jkt: str,
    expected_nonce: str | None = None,
) -> None:
    header = json.loads(_b64u_decode(proof.split(".")[0]))

    # 1. typ. Without it, a JWT from another context (an id_token, say)
    #    could pass as a proof: that is token type confusion.
    if header.get("typ") != "dpop+jwt":
        raise DPoPError("wrong typ")

    # 2. Asymmetric alg from an allowlist. 'none' and the HS* family must
    #    die here: with HS256 the 'public key' in the header would be the
    #    signing key and anyone could forge proofs.
    if header.get("alg") not in ALLOWED_ALGS:
        raise DPoPError("alg not allowed")

    # 3. The public key is embedded and must NOT carry private material.
    raw_jwk = header.get("jwk") or {}
    if raw_jwk.get("d") or raw_jwk.get("k"):
        raise DPoPError("jwk carries private or symmetric material")
    key = jwk.JWK(**raw_jwk)

    # 4. Valid signature under that same key.
    verifier = jws.JWS()
    verifier.deserialize(proof)
    verifier.verify(key)                      # raises on mismatch
    claims = json.loads(verifier.payload)

    # 5. The request being signed is THIS request.
    u = url.split("?", 1)[0].split("#", 1)[0]
    if claims.get("htm") != method.upper() or claims.get("htu") != u:
```

```
        raise DPoPError("htm/htu do not match the request")

    # 6. Freshness. An iat in the future is rejected too.
    now = int(datetime.now(timezone.utc).timestamp())
    iat = claims.get("iat")
    if not isinstance(iat, int) or abs(now - iat) > IAT_SKEW_SECONDS:
        raise DPoPError("proof outside the time window")

    if expected_nonce is not None and claims.get("nonce") != expected_nonce:
        raise DPoPError("missing or stale nonce")

    # 7. Anti-replay. SET NX is atomic: first one in wins. Without this
    #    cache, an intercepted proof can be replayed for the whole 60
    #    seconds of the iat window.
    jti = claims.get("jti")
    if not jti or not await rds.set(f"dpop:jti:{jti}", "1", ex=JTI_TTL_SECONDS, nx=True):
        raise DPoPError("repeated jti")

    # 8. The two final bindings: the proof belongs to THIS access token,
    #    and the proof key is the one the token declares in cnf.jkt.
    expected_ath = _b64u(hashlib.sha256(access_token.encode()).digest())
    if claims.get("ath") != expected_ath:
        raise DPoPError("ath does not match the presented access token")

    if key.thumbprint() != token_cnf_jkt:
        raise DPoPError("proof key is not the one bound to the token")
```

Step 7 deserves an operational note. The jti cache has to be shared across every replica of your API: an in-process cache means the attacker only has to retry until the load balancer sends them to a different pod. Redis with SET NX EX covers the normal case. If Redis goes down, you need to have decided in advance whether you fail open —accepting proofs without a replay check, while the request still demands a valid signature, correct htu, and fresh iat— or fail closed. For most APIs, failing open during a Redis outage is the right trade-off: the attacker additionally needs to have intercepted a valid proof within the last 60 seconds.

Nonces: when 60 seconds is too long

The iat window is a compromise between clock skew and replay surface. If your risk profile cannot tolerate it, the server can demand a nonce it issues itself. The dance starts with a rejection the client must know how to handle:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP algs="ES256", error="use_dpop_nonce",
    error_description="A nonce is required in the proof"
DPoP-Nonce: eyJ7S_zG.vYA.hM
```

The client stores the nonce, rebuilds the proof including it, and retries. On the authorization server the same case returns a 400 with "error": "use_dpop_nonce" in the JSON body. Implement the automatic retry exactly once in your HTTP layer; an uncapped loop against a server that rotates nonces aggressively is a request storm waiting to happen.

DPoP or mTLS

Mutual client certificates (RFC 8705) solve the same problem and have been in production longer. The real choice is operational, not cryptographic: mTLS requires TLS to terminate somewhere you can see the client certificate —which collides with nearly any CDN or managed load balancer— and it requires a PKI with issuance, distribution, and rotation. DPoP works at the application layer, traverses proxies without ceremony, and the client generates its own key. In exchange you pay one signature per request:

roughly 0.3 to 1 ms with ES256 on modern hardware, which almost never matters on the client and is worth measuring on a high-traffic resource server. FAPI 2.0, the open-banking profile, accepts both.

Token Exchange: delegation without impersonation

So far we have talked about one client and one API. The moment a chain of services appears —a gateway calling a billing service calling a payments service— an awkward question shows up: which token does each hop present?

The two usual answers are both bad. Forwarding the user's original token to every service means the payments service receives a token with all of the user's permissions, including the ones that have nothing to do with payments, and any compromised service in the chain becomes the user. Using each service's own credentials loses the identity of whoever started the request entirely: your audit logs say "billing-service deleted the invoice" and are useless.

RFC 8693 (Token Exchange, January 2020) defines a grant for exactly this. A service presents the token it received and asks for a new one, restricted to the audience of the next hop and with fewer permissions.

```
TOKEN_EXCHANGE = "urn:ietf:params:oauth:grant-type:token-exchange"
ACCESS_TOKEN_TYPE = "urn:ietf:params:oauth:token-type:access_token"

async def exchange_for_downstream(
    client: httpx.AsyncClient,
    incoming_token: str,
    audience: str,
    scope: str,
) -> str:
    """Exchange the user's token for one scoped to the target service."""
    resp = await client.post(
        "https://auth.example.com/oauth2/token",
        data={
            "grant_type": TOKEN_EXCHANGE,
            "subject_token": incoming_token,
            "subject_token_type": ACCESS_TOKEN_TYPE,
            # actor_token identifies WHO is acting on the subject's behalf.
            # Its presence is what makes this delegation rather than
            # impersonation: the resulting token will carry an 'act' claim.
            "actor_token": await service_identity_token(client),
            "actor_token_type": ACCESS_TOKEN_TYPE,
            # Narrowing audience and scope is the entire point of the
            # exercise. An exchange that returns the same permissions
            # buys you nothing.
            "audience": audience,
            "scope": scope,
            "requested_token_type": ACCESS_TOKEN_TYPE,
        },
        auth=(SERVICE_CLIENT_ID, SERVICE_CLIENT_SECRET),
    )
    resp.raise_for_status()
    body = resp.json()

    # The RFC requires the server to return issued_token_type. Check it:
    # a server may legitimately return a type other than the one requested.
    if body["issued_token_type"] != ACCESS_TOKEN_TYPE:
        raise RuntimeError(f"unexpected type: {body['issued_token_type']}")
    return body["access_token"]

# Used from the billing service, calling payments:
downstream = await exchange_for_downstream(
    client,
    incoming_token=user_token,
    audience="https://payments.internal/",
```

```
    scope="payments:charge",     # and NOTHING else
)
```

The resulting token carries the act claim, which is the difference between delegation and impersonation. With delegation, the token says "I am the user alice, and the one acting for her is billing-service":

```
{
  "iss": "https://auth.example.com",
  "sub": "alice@example.com",
  "aud": "https://payments.internal/",
  "scope": "payments:charge",
  "exp": 1790000600,
  "act": {
    "sub": "billing-service",
    "act": { "sub": "api-gateway" }
  }
}
```

act claims nest: the most recent actor sits outermost and the full delegation chain is readable. In a post-incident investigation, that is the difference between reconstructing what happened and guessing at it. Impersonation —the same exchange without actor_token— produces a token with no act, indistinguishable from one issued directly to the user. It is useful for the "support signs in as the customer to reproduce a bug" case, but then traceability has to live somewhere else, and it should be an explicit decision rather than the side effect of copying an example.

On the receiving side, the non-negotiable rule: always validate aud. A service that accepts any token correctly signed by your issuer is a confused deputy. The token the reporting service minted for itself then works for calling the payments service.

```
import jwt   # PyJWT

def verify_access_token(token: str, jwks_client: jwt.PyJWKClient) -> dict:
    signing_key = jwks_client.get_signing_key_from_jwt(token).key
    claims = jwt.decode(
        token,
        signing_key,
        algorithms=["ES256"],          # explicit allowlist, never the header's
        issuer="https://auth.example.com",
        audience="https://payments.internal/",   # THIS service, not a wildcard
        options={"require": ["exp", "iat", "iss", "aud", "sub"]},
        leeway=30,                     # clock skew, no more
    )
    if "payments:charge" not in claims.get("scope", "").split():
        raise PermissionError("insufficient scope")
    return claims
```

Discovery: stop hardcoding URLs

Two metadata RFCs keep your client configuration from being a list of URLs copied out of an email.

RFC 8414 (Authorization Server Metadata) publishes endpoints, supported algorithms, and —relevant to what we saw earlier— code_challenge_methods_supported and dpop_signing_alg_values_supported at /.well-known/oauth-authorization-server.

RFC 9728 (Protected Resource Metadata) is more recent and solves the inverse problem: given the resource, who authorizes it? The resource server publishes its metadata and, when it rejects a request with no token, points at it in the header:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api",
    resource_metadata="https://api.example.com/.well-known/oauth-protected-resource"
```

```
{
  "resource": "https://api.example.com/",
  "authorization_servers": ["https://auth.example.com"],
  "scopes_supported": ["invoices:read", "invoices:write", "payments:charge"],
  "bearer_methods_supported": ["header"],
  "dpop_bound_access_tokens_required": true
}
```

This pair is exactly what the MCP authorization specification uses: an MCP server declares itself a protected resource, the client discovers its authorization server, registers dynamically if needed (RFC 7591), and runs an OAuth 2.1 flow with PKCE. If you are building agent integrations, this is the marked path, and it is worth following instead of inventing your own API key scheme.

One more piece that gets overlooked: resource indicators from RFC 8707, the resource parameter appearing in the examples above. Without it, the authorization server issues a generically-scoped token that works against all of your APIs. With it, you explicitly request a token for https://api.example.com/ and that token is useless anywhere else. It is one line of code and it shrinks the blast radius of every leaked token.

Validating the token at the resource server: the five checks, and the two almost nobody does

Everything in this guide so far is about the client and the authorization server. But the link where things actually break is the third one: the API that receives the token and decides whether to believe it. A flawless flow with PKCE, rotation and DPoP buys you nothing if the destination service accepts any JWT that arrives with a plausible-looking signature.

Full validation is five checks and none of them is optional:

1. Signature, with a key you obtained from the issuer over a trusted channel, using an algorithm you pinned.
2. Issuer (iss): exact string comparison against the issuer you expect. Not "contains", not "starts with".
3. Audience (aud): this token was for you, not for the service next door.
4. Validity window (exp, and nbf when present): with a small, explicit clock tolerance.
5. Authorization (scope, plus whatever you use for fine-grained permissions): the token permits this operation on this resource.

The two most often skipped are the third and the fifth, and both have the same consequence: a perfectly valid token issued for something else walks straight in.

Algorithm confusion: never ask the token which algorithm to use

A JWT's header states the algorithm. That is attacker-supplied data. If your library reads alg from the header and acts on it, two classic attacks are waiting.

The first is alg: none: the token claims to be unsigned and a naive implementation accepts it. The second is subtler: the token arrives with alg: HS256 when you expected RS256. If your code calls a generic verify function and hands it "the key", and that key is the RSA public key, the library will happily

use it as an HMAC secret. The public key is public: the attacker has it, signs with it, and your verification passes.

The defence is one line: always pass an explicit allow-list of algorithms and never derive it from the token.

```
import time, threading, requests
from jose import jwt

ISSUER   = "https://auth.example.com"
AUDIENCE = "https://api.example.com/billing"   # THIS service, not "the API"
ALGOS    = ["RS256"]                           # pinned here, never read from the token


class JwksCache:
    """Key cache with two limits: it refuses to refetch more than once a minute
    (so a made-up kid can't turn every request into a DoS against the IdP) and it
    always refetches after 10 minutes (so it can't get stuck after a rotation)."""

    def __init__(self, url, min_interval=60, max_age=600):
        self._url = url
        self._keys = {}
        self._fetched_at = 0.0
        self._min_interval = min_interval
        self._max_age = max_age
        self._lock = threading.Lock()

    def _refresh(self):
        doc = requests.get(self._url, timeout=3).json()
        self._keys = {k["kid"]: k for k in doc["keys"]}
        self._fetched_at = time.monotonic()

    def get(self, kid):
        now = time.monotonic()
        if kid in self._keys and now - self._fetched_at < self._max_age:
            return self._keys[kid]
        with self._lock:
            if kid in self._keys and now - self._fetched_at < self._max_age:
                return self._keys[kid]
            if now - self._fetched_at < self._min_interval:
                # Too soon to ask again. Use the stale key if we have it, otherwise
                # reject. We do not hammer the IdP.
                if kid in self._keys:
                    return self._keys[kid]
                raise KeyUnavailable(kid)
            self._refresh()
        if kid not in self._keys:
            raise KeyUnavailable(kid)
        return self._keys[kid]


jwks = JwksCache(f"{ISSUER}/.well-known/jwks.json")


def validate_access_token(token: str, required_scope: str) -> dict:
    header = jwt.get_unverified_header(token)

    # 1. We decide the algorithm.
    if header.get("alg") not in ALGOS:
        raise InvalidToken("algorithm not allowed")

    # RFC 9068: a JWT access token is typed "at+jwt". If your issuer sets it, check
    # it: this is what stops you accepting an ID token as an access token.
    if header.get("typ", "at+jwt").lower() not in ("at+jwt", "application/at+jwt"):
        raise InvalidToken("this JWT is not an access token")

    key = jwks.get(header["kid"])
```

```
    # 2-4. Signature, issuer, audience and validity, in one call and with no escapes.
    claims = jwt.decode(
        token,
        key,
        algorithms=ALGOS,
        issuer=ISSUER,
        audience=AUDIENCE,
        options={
            "verify_signature": True,
            "verify_aud": True,
            "verify_iss": True,
            "verify_exp": True,
            "require_exp": True,
            "leeway": 30,          # explicit, small clock tolerance
        },
    )

    # 5. Authorization. A valid token is not an authorized token.
    granted = set(claims.get("scope", "").split())
    if required_scope not in granted:
        raise Forbidden(f"missing scope {required_scope}")

    return claims
```

An ID token is not an access token

This deserves its own paragraph because it is comfortably the most repeated mistake in OpenID Connect integrations. The ID token exists so that your client knows who signed in. Its audience is the client_id, not the API. It carries no scopes. It was never meant to travel to a resource server, and when it does it is usually because somebody saw a nice JWT with the user's name inside and sent it.

An API that accepts ID tokens is, in practice, accepting "anyone who has signed in to any application in this tenant", because the audience check — if it exists at all — passes with some other application's client_id. Checking aud against your service's specific URL, plus typ: at+jwt, closes both routes at once.

Local JWT or introspection: that is a revocation decision, not a performance one

Validating the JWT locally costs nothing: a signature check is microseconds and there is no network involved. Introspection (RFC 7662) means a call to the authorization server on every request, with its latency and its single point of failure.

Put that way it sounds like there is no debate, which is exactly why many teams choose wrong. What introspection buys is the one thing a local JWT check cannot give you: knowing whether the token is still alive right now. A JWT is valid until its exp no matter what — the next chapter is entirely about that. The practical rule: validate locally and shorten the access token lifetime to whatever delay you are willing to accept after a revocation (five or ten minutes for most APIs); reserve introspection for operations where that delay is genuinely unacceptable, which are far fewer than they first appear.

Really logging out: revocation, back-channel logout, and the JWT you cannot kill

"Log out" is one of those features everybody assumes is implemented because there is a button. Then an incident arrives — a stolen laptop, a departing employee, a compromised account — and with it the uncomfortable question: how long does the token that should no longer work keep working?

The honest answer: until its exp

A self-contained JWT consults nobody. That is its whole appeal and also its limit: the resource server validates the signature and the claims without asking the issuer, so the issuer has no way to tell it that this token is finished. You can delete the user's session, revoke the refresh token and disable the account, and the access token already in the attacker's hands remains cryptographically valid until the exact second of its exp.

From which comes the sentence worth writing into your service documentation: the access token lifetime is your revocation window. It is not a performance knob you turn up to save calls to the token endpoint; it is the maximum time an attacker keeps access after you have thrown them out. Fifteen minutes is a common compromise. One hour is a decision you should be able to defend. Twenty-four hours, which shows up more often than it should, means the log-out button is decorative for a full day.

What revocation actually revokes (RFC 7009)

The revocation endpoint exists and you should call it, but it helps to know what it does. You send it a token and the server invalidates the grant behind it. Two details surprise people:

- Revoking the refresh token typically also revokes the access tokens issued from it in the authorization server's own records — but that only affects introspection. If your APIs validate locally, they never find out. Revocation changes the future (no new tokens will be issued), not the present.
- The server returns 200 even when the token does not exist or was already revoked. This is deliberate: a different response would turn the endpoint into an oracle for probing token validity. So a 200 does not mean "I revoked something", it means "stop using that token". Do not treat it as confirmation.

```
import httpx

async def log_out(http_client, refresh_token, access_token):
    """Revoke in the right order: refresh first (cuts off the future),
    then access (in case the issuer tracks it)."""
    base = {"client_id": CLIENT_ID, "client_secret": CLIENT_SECRET}

    for token, hint in ((refresh_token, "refresh_token"), (access_token, "access_token")):
        if not token:
            continue
        try:
            await http_client.post(
                f"{ISSUER}/oauth2/revoke",
                data={**base, "token": token, "token_type_hint": hint},
                timeout=3,
            )
        except httpx.HTTPError:
            # A failure here must NOT block the local logout. Log it and retry in
            # the background; the user has already said they are leaving.
            log.warning("revocation_failed", extra={"hint": hint})
            enqueue_revocation_retry(token, hint)

    clear_local_session()
```

Back-channel logout: the only one that survives third-party cookie blocking

When a user signs out at the identity provider, the other applications have to find out. For years this was done with front-channel logout: the IdP loaded a hidden iframe per application and each one cleared its cookie. That mechanism depends on sending cookies in third-party requests, which is exactly what Safari, Firefox and now Chrome block by default. The iframe loads, the application cannot see its session cookie, and the logout fails silently — nobody returns an error because nobody is watching.

The replacement is back-channel logout: the IdP makes a server-to-server call to an endpoint of yours

carrying a logout token, a signed JWT identifying the session being closed. No browser involved, so no third-party cookies to block.

That endpoint is public and receives JWTs from outside, so it gets validated as rigorously as an access token, plus three checks of its own:

```
from fastapi import APIRouter, Form, HTTPException

router = APIRouter()

@router.post("/oidc/backchannel-logout")
async def backchannel_logout(logout_token: str = Form(...)):
    header = jwt.get_unverified_header(logout_token)
    if header.get("alg") not in ALGOS:
        raise HTTPException(400, "algorithm not allowed")

    c = jwt.decode(
        logout_token, jwks.get(header["kid"]),
        algorithms=ALGOS, issuer=ISSUER, audience=CLIENT_ID,
        options={"verify_exp": True, "require_exp": True, "leeway": 30},
    )

    # 1. It has to declare itself a logout token. Without this, an ordinary ID
    #    token would be enough to sign anybody out.
    if "http://schemas.openid.net/event/backchannel-logout" not in c.get("events", {}):
        raise HTTPException(400, "not a logout token")

    # 2. A logout token NEVER carries a nonce. If it does, it is a recycled ID token.
    if "nonce" in c:
        raise HTTPException(400, "logout token carries a nonce")

    # 3. Anti-replay: the same jti is not processed twice.
    if not mark_jti_once(c["jti"], ttl=600):
        return {"ok": True}

    sid, sub = c.get("sid"), c.get("sub")
    if sid:
        await terminate_sessions_by_sid(sid)       # preferred: just that session
    elif sub:
        await terminate_sessions_by_subject(sub)   # all of the user's sessions
    else:
        raise HTTPException(400, "logout token identifies no session")

    return {"ok": True}
```

The sid is what separates a precise logout from a rude one. Without it you can only close every session the user has, and the person working on their laptop gets kicked out because somebody signed out on their phone. If your IdP issues it, store it in your session when you create it: it is the only way to correlate later.

The bounded denylist: shortening the window without going back to introspection

Between "validate locally and live with the window" and "introspect on every request" there is a middle option that works well: a denylist in Redis, checked locally, with one property that makes it cheap — every entry expires when the token expires. The list never grows beyond the tokens that are still alive.

```
import time

async def revoke_session(redis, sid: str, max_exp: int):
    """Mark a session as terminated only until its tokens expire on their own."""
    ttl = max(1, max_exp - int(time.time()) + 30)
    await redis.setex(f"revoked:sid:{sid}", ttl, "1")
```

```
async def is_revoked(redis, claims: dict) -> bool:
    sid = claims.get("sid")
    if sid and await redis.exists(f"revoked:sid:{sid}"):
        return True
    return bool(await redis.exists(f"revoked:jti:{claims['jti']}"))
```

It costs one Redis lookup per request — tens of microseconds, not a round trip to the IdP — works without coordination between services, and degrades sensibly: if Redis stops answering you decide whether to fail open (keep accepting valid tokens, which is what you did before you had a list at all) or closed. For most APIs, failing open with an alert is the right call, because the alternative is a Redis outage taking down the whole service over a mechanism that is an optional improvement on expiry.

Browser apps: why the token shouldn't live there, and the BFF pattern

The argument about where a SPA keeps its tokens has been going round the same track for years — localStorage versus sessionStorage versus a variable in memory — and the framing is wrong. The IETF published the definitive guidance as RFC 10017, OAuth 2.0 for Browser-Based Apps, and OAuth 2.1 folds it in alongside the Security BCP (RFC 9700). It is worth reading in full, but its practical conclusion fits in a sentence: the browser is an environment where nothing can be kept safe from an XSS, so the useful question is not where to store the token but whether the token needs to reach the browser at all.

Why the localStorage-versus-memory comparison matters less than it seems

If arbitrary JavaScript runs on your origin, the attacker has already won in every scenario. What changes is the blast radius, and that difference does matter:

- localStorage: the token is read in one line and posted to the attacker's server. Access survives closing the tab and leaves the machine entirely. This is the worst case, because the theft is exfiltratable.
- In-memory variable: there is no API to read it from another context, so the attacker cannot lift it as easily; they can, however, ride the session from inside the page while it is open. Real damage, but bounded by the life of the tab.
- HttpOnly cookie: JavaScript cannot read it, full stop. The attacker can still make requests from the victim origin and the browser will attach the cookie, but they cannot take the credential away. The theft goes from an exfiltrated token to a ridden session, which ends when the tab closes.

None of the three prevents XSS. The third turns a permanent leak into temporary abuse, and that is the only axis you can improve on without leaving the browser.

The BFF: the token never crosses the boundary

The backend-for-frontend pattern takes the idea to its conclusion. The SPA stops being an OAuth client. The client becomes a small backend on the same origin as the application, which runs the full flow, keeps the tokens on its side, and exposes nothing to the browser but an opaque session cookie.

The browser never sees an access token. There is nothing to exfiltrate. As a side effect the client becomes confidential: it can authenticate with a secret, which unlocks refresh token rotation with reuse detection and checks a public client simply cannot perform.

```
import secrets, hashlib, base64, json
from fastapi import FastAPI, Request, Response, HTTPException
from fastapi.responses import RedirectResponse
import httpx
```

```
app = FastAPI()
SESSIONS = redis_client   # in production: Redis with TTL, not a dict

def _pkce():
    verifier = base64.urlsafe_b64encode(secrets.token_bytes(64)).decode().rstrip("=")
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()).decode().rstrip("=")
    return verifier, challenge


@app.get("/auth/login")
async def login():
    verifier, challenge = _pkce()
    state = secrets.token_urlsafe(32)
    # Flow state lives on the server, tied to a short-lived cookie.
    await SESSIONS.setex(f"flow:{state}", 300, verifier)

    response = RedirectResponse(
        f"{ISSUER}/oauth2/authorize?response_type=code"
        f"&client_id={CLIENT_ID}&redirect_uri={REDIRECT_URI}"
        f"&scope=openid%20billing.read&state={state}"
        f"&code_challenge={challenge}&code_challenge_method=S256"
    )
    response.set_cookie("flow_state", state, httponly=True, secure=True,
                        samesite="lax", max_age=300, path="/auth")
    return response


@app.get("/auth/callback")
async def callback(request: Request, code: str, state: str):
    # The state in the URL must match the one in the cookie: this is what stops
    # somebody injecting THEIR authorization code into your session (login CSRF).
    if not secrets.compare_digest(state, request.cookies.get("flow_state", "")):
        raise HTTPException(400, "state mismatch")

    verifier = await SESSIONS.getdel(f"flow:{state}")
    if not verifier:
        raise HTTPException(400, "flow expired or replayed")

    async with httpx.AsyncClient() as c:
        r = await c.post(f"{ISSUER}/oauth2/token", data={
            "grant_type": "authorization_code", "code": code,
            "redirect_uri": REDIRECT_URI, "client_id": CLIENT_ID,
            "client_secret": CLIENT_SECRET,      # a genuinely confidential client
            "code_verifier": verifier,
        }, timeout=5)
    r.raise_for_status()
    tokens = r.json()

    # The tokens stay HERE. The browser only receives an opaque identifier.
    sid = secrets.token_urlsafe(32)
    await SESSIONS.setex(f"sess:{sid}", 28800, json.dumps(tokens))

    resp = RedirectResponse("/")
    resp.set_cookie("sid", sid, httponly=True, secure=True,
                    samesite="lax", path="/", max_age=28800)
    resp.delete_cookie("flow_state", path="/auth")
    return resp


@app.api_route("/api/{path:path}", methods=["GET", "POST", "PUT", "DELETE"])
async def proxy(path: str, request: Request):
    sid = request.cookies.get("sid")
    raw = await SESSIONS.get(f"sess:{sid}") if sid else None
    if not raw:
        raise HTTPException(401)
    tokens = json.loads(raw)
```

```
    # With SameSite=Lax a cross-site POST does not carry the cookie, but Lax does
    # not cover every case: require the Origin header on mutating methods.
    if request.method != "GET" and request.headers.get("origin") not in ALLOWED_ORIGINS:
        raise HTTPException(403, "origin not allowed")

    async with httpx.AsyncClient() as c:
        upstream = await c.request(
            request.method, f"{API_BASE}/{path}",
            headers={"authorization": f"Bearer {tokens['access_token']}"},
            content=await request.body(), timeout=10)
    return Response(upstream.content, upstream.status_code,
                    media_type=upstream.headers.get("content-type"))
```

What the BFF charges you

No pattern is free, and this one has a concrete bill worth reading before you adopt it:

- Server state is back. You have just reintroduced sessions, with their store, their expiry and their need to be shared across instances. If your frontend was static on a CDN, there is now a service to scale and watch.
- CSRF is back. Authenticating with a cookie instead of an Authorization header means the browser attaches it for you. SameSite=Lax covers most of it, but it is not a complete defence: you need an origin check on mutating methods, or an anti-CSRF token.
- The BFF is now the attack surface. A proxy that forwards arbitrary paths to an internal API with a privileged token attached is an SSRF waiting to happen unless you validate which paths may be forwarded.

That said, for a web application that already has a server the arithmetic almost always comes out in favour. The honest comparison is not "pure SPA versus BFF", it is "a pure SPA with tokens in the browser and a permanent exfiltration window, versus one more service you already know how to operate". The second is the one you can explain in a security review without getting creative.

Authenticating the client without shared secrets: private_key_jwt and the audience that no longer allows ambiguity

Everything so far protects the token once it has been issued. But there is an earlier question almost nobody asks: how does your backend prove to the authorization server that it is who it claims to be? In most integrations the answer is a client_secret: a long string that lives in an environment variable, travels with every request to the token endpoint, and is also stored by the authorization server. For all practical purposes it is a shared password, with every problem a shared password has: it gets copied into CI, shows up in a configuration dump, ends up pasted into a support ticket and, once it leaks, there is no way to know who is using it.

The mature alternative is private_key_jwt (RFC 7523 and section 9 of OpenID Connect Core): the client keeps a private key that never leaves its process (or its KMS), publishes the public half in a JWKS and, on every request to the token endpoint, signs a short-lived assertion proving it holds that key. The authorization server stores no client secret at all, only the public key. If someone steals the authorization server's database, they walk away with nothing that lets them impersonate your clients. It is one of the two methods FAPI 2.0 requires (the other is mTLS) and the one I would default to for any new confidential client.

The assertion, field by field

- iss and sub: both are the client_id. It is not redundancy: the same format is used for assertion grants where sub is a user.
- aud: the authorization server's issuer. A little further down I explain why this is no longer a choice.
- jti: a unique identifier; the server must reject a jti it has already seen while the assertion is still valid.
- exp and iat: a lifetime of 60 seconds or less. A client assertion is never reused; you mint one per request.
- typ header: client-authentication+jwt, the explicit type introduced by the RFC 7523 updates.

```
import time, uuid
import httpx
import jwt  # PyJWT >= 2.8 with cryptography

ISSUER = "https://auth.example.com"          # the issuer validated during discovery
TOKEN_ENDPOINT = "https://auth.example.com/oauth2/token"
CLIENT_ID = "billing-service"

def client_assertion(private_key_pem: bytes, kid: str) -> str:
    now = int(time.time())
    claims = {
        "iss": CLIENT_ID,
        "sub": CLIENT_ID,
        "aud": ISSUER,               # the issuer, never the endpoint URL
        "jti": str(uuid.uuid4()),
        "iat": now,
        "exp": now + 60,
    }
    headers = {"kid": kid, "typ": "client-authentication+jwt"}
    return jwt.encode(claims, private_key_pem, algorithm="ES256", headers=headers)

def client_credentials_token(private_key_pem: bytes, kid: str, scope: str) -> dict:
    resp = httpx.post(
        TOKEN_ENDPOINT,
        data={
            "grant_type": "client_credentials",
            "scope": scope,
            "client_assertion_type": "urn:ietf:params:oauth:client-assertion-type:jwt-bearer",
            "client_assertion": client_assertion(private_key_pem, kid),
        },
        timeout=5.0,
    )
    resp.raise_for_status()
    return resp.json()
```

What it does and why it matters: every call mints a fresh assertion with a different jti and a one-minute lifetime, so an assertion captured in a log or a proxy expires before it is useful and, even if it did not, the server would reject it as a replay. Notice that the client sends no secret at all: the identity lives inside the signature.

The audience injection attack and why aud is no longer negotiable

RFC 7523 allowed aud to be either the issuer or the token endpoint URL, and many libraries used the endpoint because "that is where it is being sent". In January 2025, Pedram Hosseyni, Ralf Küsters and Tim Würtele (University of Stuttgart) disclosed to the OAuth working group a vulnerability born precisely from that ambiguity. The scenario: a client that talks to several authorization servers, one of them malicious or compromised. That server publishes metadata in which one of its endpoints (the PAR endpoint, for example) points at the honest server's token endpoint URL. The client builds an assertion whose aud is "the URL I am about to send this to", hands it to the attacker, and the attacker replays it against the honest server, which accepts it because aud matches its own endpoint. Result: the attacker authenticates as your client.

The draft-ietf-oauth-rfc7523bis draft (version 06, March 2026) closes the door in both directions:

- The client must use the authorization server's issuer as aud, and only that value. The token endpoint URL is explicitly forbidden.
- The authorization server must reject any assertion whose sole aud is not its own issuer, compared as an exact string.
- The explicit client-authentication+jwt typ is recommended for clients, but servers are advised not to reject assertions that lack it, so clients that already comply on audience do not break.

The practical lesson is simple: the issuer you put in aud is the one you validated during discovery (RFC 8414 requires checking that the metadata issuer matches the one you requested), not whatever URL happens to appear in the metadata. And if you run the authorization server, check your product: several vendors shipped patches in 2025 to stop accepting the token endpoint as an audience.

```
# Authorization server side: strict validation of the client assertion
import jwt
from jwt import PyJWKClient

ISSUER = "https://auth.example.com"

def authenticate_client(assertion: str, client, replay_cache) -> str:
    jwks = PyJWKClient(client.jwks_uri, cache_keys=True)
    key = jwks.get_signing_key_from_jwt(assertion).key
    claims = jwt.decode(
        assertion,
        key,
        algorithms=["ES256", "PS256"],      # closed list, never the header
        audience=ISSUER,
        options={"require": ["iss", "sub", "aud", "exp", "jti"]},
        leeway=5,
    )
    aud = claims["aud"]
    if isinstance(aud, list) and aud != [ISSUER]:
        raise PermissionError("aud must contain only the issuer")
    if claims["iss"] != client.client_id or claims["sub"] != client.client_id:
        raise PermissionError("iss/sub do not match the client")
    if claims["exp"] - claims.get("iat", claims["exp"]) > 300:
        raise PermissionError("assertion lifetime too long")
    # SET NX with an expiry equal to the assertion's remaining lifetime
    if not replay_cache.add(f"jti:{client.client_id}:{claims['jti']}", expires_at=claims["exp"]):
        raise PermissionError("replayed assertion")
    return client.client_id
```

PyJWT already checks that ISSUER is in aud, but it accepts lists with several values; the explicit list check closes the aud: [issuer, attacker] case. The jti cache with SET NX is the same building block we used for DPoP: if you already have it, reuse it.

Rotating client keys without downtime

The operational advantage of private_key_jwt over a secret is that rotation needs no coordination. The pattern: generate the new key, add it to the published JWKS next to the old one, wait for the authorization server's cache to expire (usually minutes; ask your vendor or look at the JWKS Cache-Control header), start signing with the new key using its kid and, after a safety margin, remove the old one. If you keep the key in a KMS or HSM, the private key never exists outside the module and signing is an API call; the cost is single- or double-digit milliseconds per token request, irrelevant with 5 to 15 minute tokens you already cache.

PAR: taking the authorization request out of the browser

In the classic flow, every authorization parameter travels in the URL the user's browser follows: client_id, redirect_uri, scope, state, code_challenge and, if you use rich authorization requests, a whole encoded JSON document. That URL goes through browser history, extensions, corporate proxies and the logs of anything that touches it, and the user (or an attacker) can modify it before it reaches the server. Pushed Authorization Requests (RFC 9126) flip the order: the client's backend first sends the parameters over an authenticated channel to the PAR endpoint, gets back an opaque, single-use, short-lived request_uri (typically 60 seconds), and the browser only carries that identifier.

What you buy with this is more than it looks:

- Integrity: nobody can change scope or redirect_uri in transit, because they are no longer in the URL.
- Early client authentication: the authorization server knows who is asking before showing anything to the user, so it can reject malformed requests without exposing a consent screen.
- Confidentiality: parameters do not appear in browser or proxy logs.
- Short URLs: the practical URL length limit stops being a problem with large authorization_details.

```
import base64, hashlib, secrets
from urllib.parse import urlencode
import httpx

PAR_ENDPOINT = "https://auth.example.com/oauth2/par"
AUTHZ_ENDPOINT = "https://auth.example.com/oauth2/authorize"

def start_login(session, private_key_pem, kid) -> str:
    verifier = secrets.token_urlsafe(64)
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()
    ).rstrip(b"=").decode()
    state = secrets.token_urlsafe(32)
    session["pkce_verifier"], session["oauth_state"] = verifier, state

    resp = httpx.post(PAR_ENDPOINT, data={
        "response_type": "code",
        "client_id": CLIENT_ID,
        "redirect_uri": "https://app.example.com/callback",
        "scope": "openid profile invoices:read",
        "state": state,
        "code_challenge": challenge,
        "code_challenge_method": "S256",
        "client_assertion_type": "urn:ietf:params:oauth:client-assertion-type:jwt-bearer",
        "client_assertion": client_assertion(private_key_pem, kid),  # aud = issuer
    }, timeout=5.0)
    resp.raise_for_status()
    request_uri = resp.json()["request_uri"]   # urn:ietf:params:oauth:request_uri:...
    # The browser only carries client_id and request_uri
    return AUTHZ_ENDPOINT + "?" + urlencode({"client_id": CLIENT_ID, "request_uri": request_uri})
```

Note the aud comment: the assertion you send to the PAR endpoint carries the issuer as its audience, not the PAR URL. That is exactly the situation the audience injection attack exploited, and the reason rfc7523bis also updates RFC 9126.

If you run the authorization server, publish require_pushed_authorization_requests: true in the metadata (server-wide or per client) once all your confidential clients support it; while it stays optional, an attacker can keep crafting tampered authorization URLs with your client_id. PAR is mandatory in the FAPI 2.0 Security Profile, which pairs it with PKCE and with sender-constrained tokens via DPoP or mTLS.

PAR and JAR are not the same thing

JAR (RFC 9101) means signing the authorization request as a JWT, the so-called request object. PAR solves integrity through the channel; JAR solves it with a signature. If you already authenticate the client at PAR, JAR mostly adds non-repudiation: a signed proof of exactly what the client asked for, useful in regulated environments. For most applications, PAR plus private_key_jwt is the combination with the best ratio of complexity to security.

Step-up: when the token is valid but the authentication is not enough

An access token says what the client may do on the user's behalf, but it does not say how that user authenticated or when. To view invoices, a password login from six hours ago is fine; to change the payout bank account, you want a second factor from the last five minutes. Until RFC 9470 (September 2023), every API solved this its own way, with invented 403 errors clients could not interpret. RFC 9470 standardizes it in three pieces:

- The authorization server includes in the token (or in the introspection response) the acr claim, the authentication level reached, and auth_time, when it happened.
- The resource server, if the level or the age is insufficient for that operation, answers 401 with WWW-Authenticate: Bearer error="insufficient_user_authentication" and the acr_values and/or max_age parameters it needs.
- The client repeats the authorization flow with those parameters, gets a new token and retries the operation.

```
import time
from fastapi import Depends, HTTPException

STRONG_ACR = "urn:example:acr:mfa"   # agree on the values with your authorization server

def require_recent_mfa(max_age: int = 300):
    def dependency(claims: dict = Depends(verified_access_token)):
        acr_ok = claims.get("acr") == STRONG_ACR
        auth_time = claims.get("auth_time")
        fresh = auth_time is not None and time.time() - auth_time <= max_age
        if not (acr_ok and fresh):
            raise HTTPException(
                status_code=401,
                headers={"WWW-Authenticate": (
                    'Bearer error="insufficient_user_authentication", '
                    'error_description="Recent MFA required", '
                    f'acr_values="{STRONG_ACR}", max_age="{max_age}"'
                )},
            )
        return claims
    return dependency

@app.put("/billing/payout-account")
def change_payout_account(body: PayoutAccount, claims=Depends(require_recent_mfa(300))):
    ...
```

Two details make the difference in production. First: acr_values is advisory, while max_age is enforceable; OpenID Connect requires the authorization server to re-authenticate when max_age is exceeded, but only asks it to try to satisfy acr_values. That is why the resource server must never assume the new token complies: it checks acr and auth_time again on every request, as the dependency above does. Second: if you validate tokens through introspection, make sure your server

returns acr and auth_time in the response; some products only include them in the ID token which, as we saw, is not an access token.

On the client, handling it is a bounded loop: if a call returns this error, start an authorization with the parameters received and retry exactly once. An unbounded loop turns a configuration mismatch between the resource server and the authorization server (an acr the server never issues) into a user trapped in endless login screens.

Native apps, CLIs and devices without a keyboard

Loopback instead of custom schemes

A desktop app or a CLI cannot keep secrets: anyone can extract them from the binary. It is a public client, and RFC 8252 describes how it should run the flow: open the system browser (never an embedded WebView, which would let the app read the user's password) and receive the code through a redirect to a loopback interface. The authorization server must accept any port in loopback redirect URIs, because the app picks a free port on every run; it is the only legitimate exception to the exact matching we argued for earlier, and it is limited to 127.0.0.1 and [::1]. Use the literal IP, not localhost: the name can resolve to another interface or be tampered with through the hosts file, and OAuth 2.1 discourages it.

```
import base64, hashlib, secrets, webbrowser
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlencode, urlparse, parse_qs

def cli_login(authz_endpoint: str, client_id: str, scope: str) -> dict:
    verifier = secrets.token_urlsafe(64)
    challenge = base64.urlsafe_b64encode(
        hashlib.sha256(verifier.encode()).digest()).rstrip(b"=").decode()
    state = secrets.token_urlsafe(24)
    result: dict = {}

    class Handler(BaseHTTPRequestHandler):
        def do_GET(self):
            qs = parse_qs(urlparse(self.path).query)
            if qs.get("state", [None])[0] != state:
                self.send_response(400); self.end_headers(); return
            result["code"] = qs.get("code", [None])[0]
            self.send_response(200); self.end_headers()
            self.wfile.write("You can close this window.".encode())
        def log_message(self, *args):  # do not print the code to the terminal
            pass

    server = HTTPServer(("127.0.0.1", 0), Handler)   # free port chosen by the OS
    redirect_uri = f"http://127.0.0.1:{server.server_port}/callback"
    webbrowser.open(authz_endpoint + "?" + urlencode({
        "response_type": "code", "client_id": client_id, "scope": scope,
        "redirect_uri": redirect_uri, "state": state,
        "code_challenge": challenge, "code_challenge_method": "S256",
    }))
    server.timeout = 120
    server.handle_request()          # serves a single request and returns
    server.server_close()
    if not result.get("code"):
        raise RuntimeError("login cancelled or invalid state")
    return {"code": result["code"], "verifier": verifier, "redirect_uri": redirect_uri}
```

The server listens only on 127.0.0.1, serves a single request and disappears; state protects against another local app trying to inject a code, and PKCE makes an intercepted code useless. The exchange at the token endpoint then uses the same redirect_uri and the verifier, with no client secret. If you store

the refresh token, put it in the operating system keychain (Keychain, Credential Manager or Secret Service), never in a plain-text file in the user's home directory.

Device flow: useful, and the phishing vector of the moment

The Device Authorization Grant (RFC 8628) handles login on TVs, consoles or browserless terminals: the device shows a short code, the user enters it on another screen, and the device polls the token endpoint until it receives the token. The problem is that the screen where the user types the code has no idea which device asked for it. An attacker can start the flow themselves, send the code to the victim with a believable pretext ("enter this code to open the shared document") and receive the tokens when the victim signs in on the provider's legitimate page, MFA included. Campaigns such as Storm-2372 against Microsoft 365 tenants since 2024, and the sale of the complete kit as a service in 2026, have made it a common technique; Salesforce went as far as removing device flow entirely in September 2025.

If you need it, treat it as an exceptional permission:

- Enable it only for the client_ids that truly need it, never as a grant available by default.
- On the code entry screen, show prominently which application is asking and from where (approximate IP and location), and warn that a code received by email or message must never be entered.
- Short lifetimes for the device_code (five minutes or less) and a cap on user_code attempts, so the attacker's window is small.
- Do not issue long-lived refresh tokens to device flow clients without sender constraint, and alert when the IP polling the token endpoint is in a different country from the person approving.

Tests that prove your integration withstands attacks

Tests for an OAuth integration usually check that login works. The ones that matter check that what must not work, does not. They are cheap to write against a test authorization server (Keycloak in a container, or your vendor's development environment) and they catch regressions no code review sees, such as a library upgrade that starts accepting alg=none again or a new setting that relaxes redirect_uri validation.

```
import pytest

def test_redirect_uri_with_suffix_rejected(authz):
    r = authz.authorize(redirect_uri="https://app.example.com/callback/../evil")
    assert r.status_code == 400 and "redirect_uri" in r.text

def test_code_without_verifier_rejected(authz, code_with_pkce):
    r = authz.token(code=code_with_pkce, code_verifier=None)
    assert r.status_code == 400 and r.json()["error"] == "invalid_grant"

def test_code_is_single_use(authz, code_with_pkce, verifier):
    assert authz.token(code=code_with_pkce, code_verifier=verifier).status_code == 200
    assert authz.token(code=code_with_pkce, code_verifier=verifier).status_code == 400

def test_refresh_reuse_revokes_family(authz, tokens):
    rt1 = tokens["refresh_token"]
    rt2 = authz.refresh(rt1).json()["refresh_token"]
    authz.refresh(rt1)                              # reuse: theft signal
    assert authz.refresh(rt2).status_code == 400    # the whole family goes down

def test_assertion_with_endpoint_aud_rejected(authz, make_assertion):
    bad = make_assertion(aud=authz.token_endpoint)
    assert authz.client_credentials(client_assertion=bad).status_code == 401
```

```
@pytest.mark.parametrize("alg", ["none", "HS256"])
def test_resource_server_rejects_disallowed_algs(api, forge_token, alg):
    assert api.get("/invoices", token=forge_token(alg=alg)).status_code == 401
```

Each test maps to a section of this guide: exact redirect_uri, mandatory PKCE, single-use codes, family-based reuse detection, strict audience in client authentication and a closed algorithm list on the resource server. The fixtures (authz, api, forge_token) are thin httpx wrappers pointing at your test environment. Run them in CI against the same authorization server version and configuration you use in production; a test that passes against a different configuration only gives you false comfort.

Observability: the signals that warn you before the incident

A healthy OAuth system produces few errors, and almost all of them are boring. The interesting ones tell you someone is probing your defenses or that a defense has broken. These are the metrics worth exporting from the authorization server, the BFF or the resource servers:

- oauth_refresh_reuse_detected_total: every reuse of an already-rotated refresh token. An isolated spike may be the race we discussed; a trend is token theft.
- oauth_token_errors_total{error} by type: invalid_grant, invalid_client, use_dpop_nonce. A jump in invalid_client after a deploy is usually a botched key rotation or a wrong aud.
- oauth_dpop_replay_rejected_total: proofs with a repeated jti. Under normal conditions it should be practically zero.
- oauth_step_up_challenges_total and its completion rate: if users abandon step-up, the protected operation is secure but useless.
- oauth_device_code_approvals_total{country_mismatch}: device flow approvals where the polling country does not match the user's.
- oauth_jwks_refresh_failures_total: if the resource server cannot refresh the keys, it will fail the moment the server rotates.

```
groups:
  - name: oauth-security
    rules:
      - alert: OAuthRefreshReuseSpike
        expr: sum(rate(oauth_refresh_reuse_detected_total[10m])) > 0.05
        for: 10m
        labels: { severity: page }
        annotations:
          summary: "Refresh token reuse above baseline: possible theft"
      - alert: OAuthInvalidClientAfterDeploy
        expr: sum(rate(oauth_token_errors_total{error="invalid_client"}[5m])) > 1
        for: 5m
        labels: { severity: ticket }
        annotations:
          summary: "Clients failing to authenticate: check key rotation or aud"
      - alert: OAuthJWKSRefreshFailing
        expr: increase(oauth_jwks_refresh_failures_total[15m]) > 3
        labels: { severity: page }
        annotations:
          summary: "Resource server cannot refresh the JWKS"
```

Also record in your audit logs the client_id, the sub, the DPoP jkt when present and the refresh token family identifier on every issuance. When the question "what did this stolen token do?" arrives, those four fields are the difference between answering in ten minutes or in three days. And never log the token itself: log its jti or a truncated hash.

Eight mistakes you will see in production

1. Storing the code_verifier in localStorage. It outlives the tab and is readable by any script on the page, including the third-party ones marketing added. Use sessionStorage or, better, the server-side session.
2. Comparing redirect_uri by prefix or after normalizing. We saw how that ends. Byte equality against a closed set.
3. Not validating aud on the resource server. The most widespread mistake and the most expensive: it turns any token in your ecosystem into a master key.
4. Accepting the alg that arrives in the JWT header. The eternal classic. Explicit allowlist in the decode call, and never none or symmetric algorithms where you expect asymmetric ones.
5. Refresh token rotation with no grace window. Works beautifully in staging over fiber and fails with real users on the subway. It shows up as "the app keeps logging me out" in your reviews.
6. Multi-hour access tokens "because revocation is hard". That is circular: they are hard to revoke because they are long. Ten-minute tokens plus refresh rotation give you a real revocation window without introspecting on every request.
7. Including the query string in the DPoP proof's htu. Produces an unexplained 401. The spec is clear: origin and path only.
8. Token exchange without narrowing permissions. If the outgoing token has the same scope and audience as the incoming one, you have added a network call and zero security.

Production checklist

- PKCE with S256 on every client, plus a boot-time check that the server advertises S256.
- redirect_uri validated by exact equality against a closed list; no redirects in error responses.
- iss validation (RFC 9207) if the client talks to more than one issuer.
- Access tokens of 5 to 15 minutes, with aud narrowed via resource (RFC 8707).
- Refresh tokens rotated, stored as hashes, with families, reuse detection, and a 15–60 second grace window.
- DPoP on public clients: non-extractable key, ath always, a jti cache shared across replicas, and a cnf.jkt check.
- Token exchange with actor_token, the next hop's audience, and a narrowed scope on every exchange.
- A revocation endpoint (RFC 7009) implemented and called on logout. Clearing a cookie revokes nothing.
- Metrics: invalid_grant rate, revocations from detected reuse, DPoP rejections by reason, and p99 age of access tokens in use.
- Alerts on spikes in per-family revocations: a sudden rise is usually a deploy that broke retry handling, but it could be token theft in progress.

FAQ

Do I need PKCE if my client is a backend with a secret? Yes. The client secret authenticates the client to the authorization server; PKCE binds the code to a specific transaction. They are different properties

and OAuth 2.1 requires both. The cost is ten lines.

Can I store tokens in localStorage if my SPA uses DPoP? It improves the situation a great deal —the token is worthless without the private key— but it does not fix it. An active XSS can ask the non-extractable key for proofs while the page is open. The correct answer is still the BFF pattern: tokens in the backend, an HttpOnly session cookie to the browser.

Rotation or DPoP for refresh tokens? OAuth 2.1 asks for one of the two on public clients. Rotation is easier to deploy and detects theft after it happens; DPoP prevents it but requires authorization server support. If you can, do both: they are orthogonal.

Do I have to migrate right now if I use the implicit grant? Yes, and not for compliance reasons. A token in the URL fragment shows up in history, can leak via Referer, and is reachable by any injected script. Migrating to authorization code with PKCE is mechanical and very well documented.

What do I do about clock skew and the proof's iat? Sync with NTP and use a 60-second window. If you see skew rejections, measure before widening it: widening the window widens the replay surface, and the problem is almost always a container without NTP rather than the spec.

Glossary

- PKCE — Proof Key for Code Exchange (RFC 7636). A per-transaction ephemeral secret binding the authorization code to the client that requested it.
- Authorization code injection — An attack in which a stolen code is redeemed from the attacker's session.
- DPoP — Demonstrating Proof-of-Possession (RFC 9449). Binds tokens to a key the client holds.
- cnf.jkt — Confirmation claim carrying the SHA-256 thumbprint (RFC 7638) of the public key bound to the token.
- ath — Hash of the access token inside the DPoP proof; prevents reusing a proof with a different token.
- Refresh token family — The chain of tokens descending from a single login; revoked as a whole when reuse is detected.
- Token exchange — RFC 8693. Trading one token for another scoped to a destination and a narrower permission set.
- act claim — Identifies the actor operating on the subject's behalf; nestable to represent the delegation chain.
- Confused deputy — A privileged service that takes orders from someone it shouldn't, typically because it never validates aud.
- Sender-constrained token — A token usable only by whoever holds a specific key or certificate (DPoP or mTLS).

Where to start tomorrow

If you can only do one thing this week, validate aud on every resource server and confirm that your verifier's algorithms is an explicit list. Those are the two most common failures and the two that turn any token into full access.

If you can do two, add PKCE with S256 wherever it is missing and cut access token lifetime to ten minutes with refresh rotation. From there, DPoP and token exchange are incremental improvements with

something solid to build on.