**DPop**

好的，我们来详细、深入地讲解一下 DPoP（Demonstrating Proof-of-Possession，演示持有证明）头部。

DPoP 是一种现代的安全机制，旨在增强 OAuth 2.0 框架的安全性，特别是解决 Access Token（访问令牌）被盗用后的滥用问题。它被定义在 https://www.rfc-editor.org/rfc/rfc9449.html 中。

一、DPoP 要解决的核心问题：Bearer Token 的缺陷

在传统的 OAuth 2.0 中，最常用的令牌类型是 Bearer Token（持有者令牌）。它的安全模型非常简单：任何一方（Bearer）只要持有这个令牌，就可以使用它访问受保护的资源。这就是所谓的“持有即所有”。

致命缺陷： 如果一个攻击者通过中间人攻击、恶意软件或不安全的日志等方式窃取了一个 Bearer Token，他就可以在完全不同的设备和网络环境下冒充用户，直到该令牌过期。服务器无法区分合法的客户端和窃取了令牌的攻击者。

DPoP 的目标： 将令牌与发起请求的特定客户端绑定，证明调用者不仅“知道”令牌（拥有令牌），还“持有”一个特定的加密密钥（拥有密钥）。这样，即使令牌被窃，攻击者没有对应的私钥也无法使用它。

二、DPoP 的核心思想

DPoP 的核心思想是使用非对称密码学（公钥-私钥对）为每个 HTTP 请求创建一个数字签名，以证明客户端拥有一个特定的密钥。

1. 客户端生成一对非对称密钥（例如 RSA 或 ECC）。

2. 在向授权服务器请求令牌时，客户端会使用私钥创建一个 DPoP Proof JWT，并将其发送给授权服务器。

3. 授权服务器验证 Proof JWT 的签名，确认客户端拥有对应的私钥。然后，它在颁发的 Access Token 中嵌入一个对该客户端公钥的指纹（thumbprint）。我们说这个令牌被“绑定”到了这个公钥上。

4. 客户端随后使用这个 Access Token 访问资源服务器时，必须在每次请求中都创建一个新的 DPoP Proof JWT。

5. 资源服务器同时验证两件事：

◦ Access Token 本身是否有效（签名、过期时间等）。

◦ DPoP Proof JWT 的签名是否有效，并且其内部包含的公钥指纹是否与 Access Token 中绑定的指纹匹配。

通过这个流程，资源服务器可以确信：正在使用这个 Access Token 的客户端，正是最初向授权服务器申请该令牌的那个客户端。

三、DPoP HTTP 请求/响应流程详解

让我们通过一个具体的序列图来理解整个过程：

sequenceDiagram

participant C as Client

participant AS as Authorization Server

participant RS as Resource Server

Note over C: 生成密钥对 (public\_key, private\_key)

C->>AS: 1. Token Request + DPoP Proof JWT

Note right of C: Proof JWT 中包含公钥指纹 (jkt)

AS->>AS: 验证Proof签名，生成jkt

AS->>C: 2. Access Token (内含jkt claim)

C->>RS: 3. API Request + Access Token + New DPoP Proof JWT

RS->>RS: 并行验证：

RS->>RS: - Access Token 有效且未过期

RS->>RS: - Proof JWT签名有效且未过期

RS->>RS: - Token中的jkt == Proof中的jkt

RS->>RS: - Proof中的htm、htu与本次请求匹配

RS->>C: 4. Protected Resource

1. 客户端生成密钥对

客户端（如Web应用、手机App）在启动时生成一个非对称密钥对（如 ES256）。私钥安全存储，绝不被发送；公钥可以共享。

2. 向授权服务器申请令牌

客户端向授权服务器的令牌端点 (/token) 发送请求时，必须构造一个 DPoP Proof JWT 并将其放在 DPoP 头中。

DPoP Proof JWT 的Payload结构：

{

"jti": "7a38a078-7ffe-46ad-b467", // JWT唯一标识符，用于防重放

"htm": "POST", // HTTP Method（必须大写）

"htu": "https://api.example.com/token", // HTTP Target URI（不含查询参数和片段）

"iat": 1686142800, // 签发时间（Unix时间戳）

"ath": "cRZgfe…fQaOA" // (可选) 请求体的SHA-256哈希值（base64url编码）

}

客户端使用其私钥对该 JWT 进行签名，然后将它作为 DPoP 头的值发送：

POST /token HTTP/1.1

Host: auth.example.com

Content-Type: application/x-www-form-urlencoded

DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik...

grant\_type=authorization\_code&

code=abc123&

...

3. 授权服务器颁发绑定的令牌

授权服务器：

1. 收到 DPoP 头。

2. 验证 DPoP Proof JWT：签名是否有效、htm/htu 是否与当前请求匹配、iat 是否在合理时间窗口内、jti 是否未被使用过（防重放）。

3. 验证通过后，从 Proof JWT 的头部提取公钥（jwk），计算其指纹（jkt，即 JWK Thumbprint）。

4. 颁发 Access Token，并在 Token 的 payload 中包含一个 cnf（confirmation）声明，记录公钥指纹。

// Access Token 的内容示例

{

"sub": "user123",

"iss": "https://auth.example.com",

"exp": 1686146400,

"cnf": {

"jkt": "0ZcacI7Cced2X7ZNZC2TQo6oWzZbHjVcFnmY4AhNnks" // 绑定的公钥指纹

}

}

4. 使用 DPoP 绑定的令牌访问资源

客户端现在要访问资源服务器 API。它必须为每一个请求都创建一个新的 DPoP Proof JWT。

新的 Proof JWT Payload：

{

"jti": "f091a435-1c5d-4c33", // 新的唯一ID

"htm": "GET", // 当前请求的方法

"htu": "https://api.example.com/protected-resource", // 当前请求的URL

"iat": 1686143000 // 新的签发时间

// 注意：通常访问资源的GET请求没有请求体，所以没有 `ath`

}

客户端使用同一个私钥签名，然后发送请求：

GET /protected-resource HTTP/1.1

Host: api.example.com

Authorization: DPoP eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9... // 绑定的Access Token

DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik... // 新的Proof

5. 资源服务器验证

资源服务器必须执行以下双重验证：

1. 验证 Access Token：检查签名、过期时间、发行者等。

2. 验证 DPoP Proof：

◦ 验证 Proof JWT 的签名是否有效。

◦ 检查 htm（例如 GET）和 htu（例如 https://api.example.com/protected-resource）是否与当前请求完全匹配。

◦ 检查 iat 时间是否在可接受的时间窗口内（例如 ±60秒），防止重放。

◦ 检查 jti 是否未被使用过（需要服务器短期缓存以防重放）。

3. 验证绑定关系：从验证通过的 Proof JWT 中提取公钥，计算其指纹（jkt），然后与 Access Token 的 cnf.jkt 声明值进行比对。两者必须完全一致。

只有所有验证都通过，资源服务器才会返回请求的资源。

四、DPoP 头的语法和格式

DPoP 头的值就是一个 DPoP Proof JWT，其本身也是一个经过编码和签名的 JWT。

格式： DPoP <JWT>

JWT 头部示例：

{

"typ": "dpop+jwt", // 类型，表明这是一个DPoP Proof JWT

"alg": "ES256", // 签名算法，必须是非对称算法（RS256, ES256, EdDSA等）

"jwk": { // 用于验证签名的公钥（JSON Web Key）

"kty": "EC",

"crv": "P-256",

"x": "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",

"y": "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM"

}

}

注意：在最终规范中，为了减少头部大小和避免潜在的攻击面，jwk 头参数已被废弃，推荐使用 jkt（ thumbprint）来在令牌中确认密钥。公钥本身通常不再放在每个请求的头部中。上述流程描述已反映这一最新实践。

五、DPoP 的优势与注意事项

优势：

1. 缓解令牌泄露：即使 Access Token 被泄露，攻击者没有对应的私钥也无法使用它。

2. 无需服务器端状态管理：与传统的 Proof-of-Possession 不同，DPoP 利用密码学签名，资源服务器无需维护会话状态。

3. 适用于单页应用（SPA）和原生应用：这些环境可以安全地生成和存储密钥对。

注意事项：

1. 时钟同步：iat 检查要求客户端和服务器时钟大致同步。

2. 重放保护：服务器需要缓存近期看到的 jti 或使用时间窗口来防止重放攻击。

3. 密钥管理：客户端必须安全地存储其私钥。丢失私钥意味着无法证明所有权。对于浏览器，通常使用非可导出密钥的 Web Crypto API。

4. 性能影响：每个请求都需要进行 JWT 签名和验证，会增加少量计算开销。

总结

DPoP 头是一种强大的安全增强机制，它将 OAuth 2.0 的 Bearer Token 模型升级为 Proof-of-Possession Token 模型。通过要求客户端在每次请求中使用私钥进行签名，它将令牌的有效性与特定的客户端实例绑定，极大地降低了令牌被盗所带来的风险，是构建现代安全应用的重要实践。

clientSession中的clientId和client中的ClientId是如何一步一步得到的

### clientSession和client中clientId的获取流程详解 一、client对象的clientId获取流程

1. 1.

客户端认证入口 在 AuthorizeClientUtil.authorizeClient() 方法中，通过 AuthenticationProcessor.authenticateClient() 启动客户端认证流程 `AuthorizeClientUtil.java` 。

2. 2.

认证流程调度 AuthenticationProcessor 创建 ClientAuthenticationFlow 实例，遍历执行客户端认证器链。默认使用 ClientIdAndSecretAuthenticator 进行客户端身份验证 `ClientAuthenticationFlow.java` 。

3. 3.

客户端ID提取 从请求中提取客户端标识：

- Basic认证 ：解析 Authorization: Basic 头获取client\_id

- 表单提交 ：从 client\_id 表单参数提取

```

// ClientIdAndSecretAuthenticator核心逻辑

client\_id = formData.getFirst

(OAuth2Constants.CLIENT\_ID);

```

`ClientIdAndSecretAuthenticator.java`

4. 4.

ClientModel加载 通过客户端ID查询数据库获取 ClientModel ：

```

ClientModel client = session.clients().

getClientByClientId(realm, client\_id);

```

验证客户端状态（是否启用、是否公开客户端等）后，通过 context.setClient(client) 绑定到认证上下文 `ClientIdAndSecretAuthenticator.java` 。 二、clientSession的clientId获取流程

1. 1.

授权码生成阶段 在授权码发放时， OAuth2CodeParser.persistCode() 将客户端ID编码到授权码中：

```

code = codeUUID + "." + userSessionId + "."

+ clientSession.getClient().getId();

``` `OAuth2CodeParser.java`

2.\*\* 授权码解析阶段\*\*

处理token请求时， OAuth2CodeParser.parseCode() 解析授权码获取客户端UUID：

```

String[] parts = code.split("\\.");

String clientUUID = parts[2];

```

通过UUID查询客户端会话：

```

ClientSessionModel clientSession =

userSession.

getAuthenticatedClientSessionByClient

(clientUUID);

```

`OAuth2CodeParser.java`

3.\*\* clientSession的clientId来源

clientSession中的clientId来自创建会话时关联的客户端：

- 在 InfinispanUserSessionProvider.createClientSession() 中初始化

- 通过 clientSession.setClient(client) 绑定客户端信息

- 最终通过 clientSession.getClient().getClientId() 暴露客户端ID 三、关键验证点（AuthorizationCodeGrantType.java:137）

```

if (!client.getClientId().equals

(clientSession.getClient().getClientId())) {

    throw new OAuthErrorException

    (OAuthErrorException.INVALID\_GRANT);

}

```

此验证确保：

1.\*\* 身份一致性 ：请求的客户端与授权码颁发的客户端必须一致

2. 会话有效性 ：通过clientId查询的客户端会话必须处于有效状态

3. 防跨客户端攻击 \*\*：防止恶意客户端伪造其他客户端的授权码

四、数据流向总结

```

客户端请求 → 提取client\_id → 查询ClientModel → 绑

定client对象

                    ↓

授权码生成 → 编码clientId → 存储到clientSession

→ 解析授权码获取clientSession

                    ↓

                    → 比较client.getClientId()

                    与clientSession.getClient

                    ().getClientId()

```

通过这种双重验证机制，Keycloak确保了客户端身份的一致性和授权码的安全性，有效防止跨客户端伪造授权码的攻击。