详细讲解一下客户端的JWT认证

好的，我们来详细讲解一下在OAuth 2.0框架中，\*\*客户端如何使用JWT进行认证\*\*。这是一种比客户端密钥（client\_secret）更安全、更灵活的认证方式，特别适用于后端应用、微服务等无法安全存储密钥的场景。

---

### 一、核心概念：什么是客户端的JWT认证？

在标准的OAuth 2.0客户端认证中，客户端通常使用 `client\_id` 和 `client\_secret` 向授权服务器（AS）证明自己的身份。而\*\*JWT认证\*\*（又称\*\*客户端断言\*\*，Client Assertion）则是一种更安全的替代方案。

其核心思想是：\*\*客户端不再发送静态的密钥，而是创建一个包含自身身份信息且经过数字签名的JWT（即“客户端断言”），并将其发送给授权服务器。授权服务器通过验证JWT的签名和内容来确认客户端的合法性。\*\*

这种方式符合“\*\*永不传输密钥\*\*”的安全最佳实践，因为只有签名的行为需要私钥，私钥本身永远不会离开客户端。

---

### 二、JWT的结构（用于客户端认证）

一个用于客户端认证的JWT（通常是一个JWS，即经过签名的JWT）包含三个部分，对应用户在OAuth请求中 `client\_assertion` 参数的值：

\*\*示例：\*\*

`eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ6bWtfY2xpZW50X2lkIiwic3ViIjoiem1rX2NsaWVudF9pZCIsImF1ZCI6Imh0dHBzOi8vYXV0aC56bWsuY29tL29hdXRoL3YyL3Rva2VuIiwianRpIjoiM2RhZDYzMTAtNWU1MC00MWM1LWIyZDYtZjJkYmRiMzE3ZDE0IiwiZXhwIjoxNjU5ODQwNzAwfQ.Cc4rES2V7aY4...（签名数据）`

让我们解码并分析这个JWT：

#### 1. Header（头部）

\* \*\*功能\*\*：描述JWT的类型和使用的签名算法。

\* \*\*解码后示例\*\*：

```json

{

"alg": "RS256", // 签名算法：RSASSA-PKCS1-v1\_5 using SHA-256

"typ": "JWT" // 类型：JWT

}

```

\* \*\*说明\*\*：授权服务器根据这里的 `alg` 声明来选择对应的公钥来验证签名。常见的算法还有 `ES256`（ECDSA）。

#### 2. Payload（负载/声明）

这是JWT的核心，包含了客户端身份的声明。以下是一些\*\*必需和推荐\*\*的声明（Claims）：

\* `iss` (Issuer - 签发者)

\* \*\*必须\*\*。客户端的 `client\_id`。

\* \*\*示例\*\*：`"iss": "zmk\_client\_id"`

\* `sub` (Subject - 主题)

\* \*\*必须\*\*。同样应设置为客户端的 `client\_id`。

\* \*\*示例\*\*：`"sub": "zmk\_client\_id"`

\* `aud` (Audience - 受众)

\* \*\*必须\*\*。该JWT的接收方，即授权服务器令牌端点（token endpoint）的URL。

\* \*\*这是防御“受众注入攻击”的关键！\*\* 必须精确验证。

\* \*\*示例\*\*：`"aud": "https://auth.zmk.com/oauth/v2/token"`

\* `exp` (Expiration Time - 过期时间)

\* \*\*必须\*\*。JWT的过期时间（Unix时间戳）。此断言的生命周期应很短（例如，最多几分钟）。

\* \*\*示例\*\*：`"exp": 1659840700`

\* `iat` (Issued At - 签发时间)

\* \*\*推荐\*\*。JWT的签发时间（Unix时间戳）。服务器可用它来检查断言是否在合理的时间内被使用。

\* \*\*示例\*\*：`"iat": 1659840400`

\* `jti` (JWT ID - JWT唯一标识)

\* \*\*推荐\*\*。一个唯一的随机值，用于防止JWT重放攻击。服务器可以记录已使用的 `jti` 并在短时间内拒绝重复的 `jti`。

\* \*\*示例\*\*：`"jti": "3dad6310-5e50-41c5-b2d6-f2dbdb317d14"`

#### 3. Signature（签名）

\* \*\*功能\*\*：客户端使用其\*\*私钥\*\*对编码后的Header和Payload进行签名（根据Header中指定的算法，如RS256）。

\* \*\*验证\*\*：授权服务器使用该客户端事先注册的\*\*公钥\*\*来验证此签名。如果验证通过，则证明此JWT确实是由拥有对应私钥的客户端创建的，消息未被篡改。

---

### 三、工作流程（以PAR流程为例）

以下序列图展示了客户端在Pushed Authorization Request (PAR)流程中使用JWT进行认证的完整过程：

```mermaid

sequenceDiagram

participant User

participant Client

participant AS as Authorization Server (AS)

Note over Client: 预备：拥有私钥，<br>且公钥已注册到AS

User->>Client: 访问客户端应用

Client->>AS: 1. 发现配置<br>GET /.well-known/openid-configuration

AS-->>Client: 返回元数据，含token\_endpoint等URL

Note over Client: 2. 创建并签名JWT断言<br>（包含iss, sub, aud, exp, jti等）

Client->>AS: 3. POST /par<br>Content-Type: application/x-www-form-urlencoded

Note right of Client: 表单数据包含：<br>client\_id={cid}<br>client\_assertion={JWT}<br>client\_assertion\_type=<br>urn:ietf:params:oauth:...:jwt-bearer<br>+ 其他请求参数

AS->>AS: 4. 验证JWT：<br>a. 检查签名（用注册的公钥）<br>b. 检查aud是否为token端点<br>c. 检查exp是否有效<br>d. 检查jti是否未使用<br>e. 检查iss/sub是否为已注册的client\_id

AS-->>Client: 5. 返回request\_uri（若验证成功）

```

图中第4步的验证细节是安全的关键，授权服务器必须严格执行所有检查，尤其是签名和受众(`aud`)的验证，才能有效抵御之前提到的受众注入等攻击。

---

### 四、最佳实践与安全考量

1. \*\*保护私钥\*\*：客户端的私钥必须被极其安全地存储（例如，使用HSM、云密钥管理服务或安全密码库）。私钥泄露意味着客户端身份被盗用。

2. \*\*使用强算法\*\*：优先使用非对称算法（如 `RS256`, `ES256`），而非对称算法（如 `HS256`，它需要共享密钥，降低了安全性）。

3. \*\*保持超短有效期\*\*：JWT断言的有效期（`exp` - `iat`）应非常短（例如30-120秒），以减少被盗用的风险。

4. \*\*强制使用 `jti`\*\*：服务器应维护一个短期缓存，拒绝重复的 `jti` 以防止重放攻击。

5. \*\*精确验证 `aud`\*\*：服务器必须严格验证 `aud` 声明是否与自己的令牌端点URL\*\*完全匹配\*\*。这是防止“受众注入攻击”最根本的防御措施。

6. \*\*使用正确的 `client\_assertion\_type`\*\*：在OAuth请求中，除了 `client\_assertion` 参数，还必须包含 `client\_assertion\_type` 参数，其值固定为：`urn:ietf:params:oauth:client-assertion-type:jwt-bearer`。

总之，客户端的JWT认证提供了一种基于密码学而非共享秘密的强大认证机制，当正确实施时，可以显著提升OAuth整体体系的安全性。

JWTClientValidator类

- 全限定名 ： org.keycloak.authentication.authenticators.client.JWTClientValidator

- 作用 ：提供JWT客户端认证的通用验证逻辑，支持 private\_key\_jwt 和 client\_secret\_jwt 两种方式

- 核心职责 ：参数验证、JWT解析、客户端身份验证、签名验证、令牌声明校验、重放攻击防护

- 作者 ：Marek Posolda（Red Hat）

clientAssertionParametersValidation()：负责对客户端的断言参数进行合法性检查

readJws()：负责解析客户端提交的JWT断言字符串，提取JWS结构

ValidateClient(): 是 JWTClientValidator 类的核心验证方法，负责验证JWT断言中的客户端身份信息与Keycloak服务器中的客户端配置是否匹配，是客户端身份认证的关键环节。

1. 提取客户端ID
2. 签发者与主体一致性检查，检查issuer
3. 检查请求参数：检查请求中的client\_id与JWT中的客户端id是否一致
4. 查找客户端：通过客户端ID在当前领域中查找客户端
5. 客户端启用状态检查：防止使用已禁用的客户端进行认证
6. 认证方法一致性检查
7. 检查客户端配置的认证器类型是否与当前使用的认证器匹配

validateSignatureAlgorithm()：检查客户端提交的JWT断言使用了预配置的签名算法

validateToken()：用于验证令牌有效性，负责校验令牌的时间有效性、必要声明存在性和防重放攻击基础检查

1. 检查JWT令牌是否在有效期内
2. 验证JWT令牌的ID是否存在

calculateLifespanInSeconds()：计算JWT令牌的生命周期

validateTokenReuse()：用于确保JWT令牌没有重复使用

validateTokenAudience()：用于确保令牌仅被授权的目标服务使用

详细讲解一下这个项目中如何控制是否允许多受众

查看ClientModel接口和JWTClientValidator类中是否存在控制多受众的配置参数

toolName: view\_files

status: success

filePath: g:\Master-2\OAuth\Keycloak\keycloak\server-spi\src\main\java\org\keycloak\models\ClientModel.java

g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientValidator.java

Keycloak通过全局配置控制JWT客户端认证中的多受众（Audiences）允许策略，核心实现位于`JWTClientValidator`类的`validateTokenAudience`方法中，具体控制逻辑如下：

### \*\*1. 多受众检查触发条件\*\*

在`validateTokenAudience`方法中，存在双重验证逻辑：

```java

// 1. 验证受众是否包含预期值（如issuer URL、token端点等）

if (!token.hasAnyAudience(expectedAudiences)) { ... }

// 2. 检查是否允许多受众

if (!isAllowMultipleAudiencesForJwtClientAuthentication(context) && token.getAudience().length > 1) {

throw new RuntimeException("Multiple audiences not allowed...");

}

```

- \*\*第一重\*\*：确保令牌受众包含至少一个预期值（如Keycloak issuer URL、令牌端点URL等）。

- \*\*第二重\*\*：根据配置决定是否允许令牌包含多个受众。

### \*\*2. 配置开关：`isAllowMultipleAudiencesForJwtClientAuthentication`\*\*

多受众允许策略通过OIDC协议配置控制，具体实现：

```java

private boolean isAllowMultipleAudiencesForJwtClientAuthentication(ClientAuthenticationFlowContext context) {

OIDCLoginProtocol loginProtocol = (OIDCLoginProtocol) context.getSession().getProvider(LoginProtocol.class, OIDCLoginProtocol.LOGIN\_PROTOCOL);

OIDCProviderConfig config = loginProtocol.getConfig();

return config.isAllowMultipleAudiencesForJwtClientAuthentication();

}

```

- \*\*配置来源\*\*：`OIDCProviderConfig`类封装了Keycloak的OIDC协议全局配置，其中`isAllowMultipleAudiencesForJwtClientAuthentication()`方法返回布尔值，表示是否允许JWT客户端认证包含多个受众。

- \*\*默认行为\*\*：若未显式配置，Keycloak默认\*\*禁止\*\*多受众（即仅允许单个受众）。

### \*\*3. 配置方式\*\*

通过Keycloak管理控制台或领域配置文件（`realm.json`）修改OIDC协议配置：

```json

// realm.json 示例

{

"oidcConfig": {

"allowMultipleAudiencesForJwtClientAuthentication": true

}

}

```

- \*\*控制台路径\*\*：领域设置 → 登录 → OpenID Connect 配置 → 勾选「允许JWT客户端认证包含多个受众」。

### \*\*4. 异常场景处理\*\*

当配置禁止多受众且令牌包含多个`aud`值时，验证将抛出异常：

```

Multiple audiences not allowed in the JWT token for client authentication

```

此异常会中断认证流程，客户端需重新生成符合单受众要求的JWT。

### \*\*5. 设计意义\*\*

- \*\*安全性\*\*：默认禁止多受众可防止客户端意外包含无关受众，降低令牌被滥用风险。

- \*\*兼容性\*\*：允许开启多受众以支持复杂场景（如跨服务认证），符合OIDC规范灵活性要求。

### \*\*总结\*\*

Keycloak通过`OIDCProviderConfig`的`allowMultipleAudiencesForJwtClientAuthentication`配置项控制多受众策略，默认禁用。在`validateTokenAudience`方法中，系统会先验证受众合法性，再根据配置检查数量，最终决定是否允许认证通过。此设计兼顾了安全性与灵活性，支持管理员根据业务需求调整受众验证策略。

ValidateTokenAudience

public void validateTokenAudience(ClientAuthenticationFlowContext context, RealmModel realm, JsonWebToken token) {

        List<String> expectedAudiences = getExpectedAudiences(context, realm);

        if (!token.hasAnyAudience(expectedAudiences)) {

            throw new RuntimeException("Token audience doesn't match domain. Expected audiences are any of " + expectedAudiences

                    + " but audience from token is '" + Arrays.asList(token.getAudience()) + "'");

        }

        if (!isAllowMultipleAudiencesForJwtClientAuthentication(context) && token.getAudience().length > 1) {

            throw new RuntimeException("Multiple audiences not allowed in the JWT token for client authentication");

        }

    }

获取预期受众列表

List<String> expectedAudiences = getExpectedAudiences(context, realm);

getExpectedAudiences源码

private List<String> getExpectedAudiences(ClientAuthenticationFlowContext context, RealmModel realm) {

        String issuerUrl = Urls.realmIssuer(context.getUriInfo().getBaseUri(), realm.getName());

        String tokenUrl = OIDCLoginProtocolService.tokenUrl(context.getUriInfo().getBaseUriBuilder()).build(realm.getName()).toString();

        String tokenIntrospectUrl = OIDCLoginProtocolService.tokenIntrospectionUrl(context.getUriInfo().getBaseUriBuilder()).build(realm.getName()).toString();

        String parEndpointUrl = ParEndpoint.parUrl(context.getUriInfo().getBaseUriBuilder()).build(realm.getName()).toString();

        List<String> expectedAudiences = new ArrayList<>(Arrays.asList(issuerUrl, tokenUrl, tokenIntrospectUrl, parEndpointUrl));

        String backchannelAuthenticationUrl = CibaGrantType.authorizationUrl(context.getUriInfo().getBaseUriBuilder()).build(realm.getName()).toString();

        expectedAudiences.add(backchannelAuthenticationUrl);

        return expectedAudiences;

    }

通过OIDC协议进行配置控制是否允许多audience

if (!isAllowMultipleAudiencesForJwtClientAuthentication(context) && token.getAudience().length > 1) {

            throw new RuntimeException("Multiple audiences not allowed in the JWT token for client authentication");

        }

OIDCProviderConfig.java

isAllowMultipleAudiencesForJwtClientAuthentication方法获取是否允许多audience

    public OIDCProviderConfig(Config.Scope config) {

        this.additionalReqParamsMaxNumber = config.getInt(OIDCLoginProtocolFactory.CONFIG\_OIDC\_REQ\_PARAMS\_MAX\_NUMBER, DEFAULT\_ADDITIONAL\_REQ\_PARAMS\_MAX\_NUMBER);

        this.additionalReqParamsMaxSize = config.getInt(OIDCLoginProtocolFactory.CONFIG\_OIDC\_REQ\_PARAMS\_MAX\_SIZE, DEFAULT\_ADDITIONAL\_REQ\_PARAMS\_MAX\_SIZE);

        this.additionalReqParamsMaxOverallSize = config.getInt(OIDCLoginProtocolFactory.CONFIG\_OIDC\_REQ\_PARAMS\_MAX\_OVERALL\_SIZE, DEFAULT\_ADDITIONAL\_REQ\_PARAMS\_MAX\_OVERALL\_SIZE);

        this.additionalReqParamsFailFast = config.getBoolean(OIDCLoginProtocolFactory.CONFIG\_OIDC\_REQ\_PARAMS\_FAIL\_FAST, DEFAULT\_ADDITIONAL\_REQ\_PARAMS\_FAIL\_FAST);

        this.allowMultipleAudiencesForJwtClientAuthentication = config.getBoolean(OIDCLoginProtocolFactory.CONFIG\_OIDC\_ALLOW\_MULTIPLE\_AUDIENCES\_FOR\_JWT\_CLIENT\_AUTHENTICATION, DEFAULT\_ALLOW\_MULTIPLE\_AUDIENCES\_FOR\_JWT\_CLIENT\_AUTHENTICATION);

    }

    public int getAdditionalReqParamsMaxNumber() {

        return additionalReqParamsMaxNumber;

    }

    public int getAdditionalReqParamsMaxSize() {

        return additionalReqParamsMaxSize;

    }

    public boolean isAdditionalReqParamsFailFast() {

        return additionalReqParamsFailFast;

    }

    public int getAdditionalReqParamsMaxOverallSize() {

        return additionalReqParamsMaxOverallSize;

    }

    public boolean isAllowMultipleAudiencesForJwtClientAuthentication() {

        return allowMultipleAudiencesForJwtClientAuthentication;

    }

JWT从请求中获取audience

我需要查看JWTClientAuthenticator.java中从请求提取JWT的代码，特别是第142行附近的内容，以了解Keycloak如何从请求中获取audience。

toolName: view\_files

status: success

filePath: g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientAuthenticator.java

当前查看的JWTClientAuthenticator.java代码片段仅包含元数据方法，未涉及请求处理逻辑。需要查看该文件中处理客户端认证的核心方法（如authenticateClient），以定位从请求提取JWT及audience的代码实现。

toolName: view\_files

status: success

filePath: g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientAuthenticator.java

### Keycloak从请求中获取JWT Audience的实现流程

#### 一、请求参数提取

Keycloak通过`<mcsymbol name="clientAssertionParametersValidation" filename="JWTClientValidator.java" path="g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientValidator.java" startline="137" type="function"></mcsymbol>`方法从请求中提取JWT令牌：

```java

// 从请求参数获取JWT断言

String clientAssertion = context.getHttpRequest().getDecodedFormParameters().getFirst(OAuth2Constants.CLIENT\_ASSERTION);

```

该方法验证请求中是否包含`client\_assertion`和`client\_assertion\_type`参数，确保符合OIDC规范要求。

#### 二、JWT解析过程

在`<mcsymbol name="authenticateClient" filename="JWTClientAuthenticator.java" path="g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientAuthenticator.java" startline="65" type="function"></mcsymbol>`方法中完成JWT解析：

1. 通过`validator.readJws()`读取并解析JWT结构

2. 通过`JsonWebToken token = validator.getToken()`获取令牌对象

3. 令牌对象包含所有标准声明，包括`aud`（受众）

#### 三、Audience提取与验证

1. \*\*提取aud声明\*\*：通过JWT令牌对象的`getAudience()`方法获取客户端提供的受众列表

```java

List<String> tokenAudiences = token.getAudience();

```

2. \*\*验证逻辑\*\*：在`<mcsymbol name="validateTokenAudience" filename="JWTClientValidator.java" path="g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientValidator.java" startline="261" type="function"></mcsymbol>`方法中实现：

- 调用`<mcsymbol name="getExpectedAudiences" filename="JWTClientValidator.java" path="g:\Master-2\OAuth\Keycloak\keycloak\services\src\main\java\org\keycloak\authentication\authenticators\client\JWTClientValidator.java" startline="279" type="function"></mcsymbol>`生成预期受众列表

- 验证客户端提供的`aud`是否与预期受众匹配

- 支持多受众场景，只要匹配其中一个预期值即通过验证

#### 四、核心代码流程

```java

// 1. 创建验证器实例

JWTClientValidator validator = new JWTClientValidator(context, getId());

// 2. 验证并读取JWT

validator.clientAssertionParametersValidation();

validator.readJws();

// 3. 获取解析后的令牌对象

JsonWebToken token = validator.getToken();

// 4. 验证受众

validator.validateTokenAudience(context, realm, token);

```

#### 五、关键技术特点

- \*\*防御性编程\*\*：对缺失`aud`声明或空受众列表的情况进行严格校验

- \*\*灵活匹配\*\*：支持JWT中包含多个受众值（数组形式）

- \*\*上下文感知\*\*：根据当前请求的URI信息动态生成预期受众列表

- \*\*规范兼容\*\*：完全符合RFC 7519和OIDC客户端认证规范要求

ClientAuthenticationFlowContext

JWTClientValidator

是 Keycloak 中负责 JWT 客户端认证的核心验证器，支持 private\_key\_jwt 和 client\_secret\_jwt 两种认证方式，实现了从请求参数提取到令牌完整验证的全流程。

- context : 客户端认证上下文，提供请求信息和领域模型

- realm : 当前领域模型

- currentTime : 验证时的时间戳（用于令牌时效判断）

- clientAssertion : 从请求提取的 JWT 断言字符串

- jws : 解析后的 JWS 对象

- token : 解析后的 JWT 令牌对象

- client : 匹配到的客户端模型

方法

readJws：

使用JWSInput解析JWT字符串

提取 JsonWebToken 对象供后续验证

validateClient

- 验证 JWT sub 与 iss 一致（客户端 ID）

- 检查 client\_id 参数（若提供）与 JWT 中的客户端 ID 匹配

- 验证客户端存在、启用且认证方式匹配

validateSignatureAlgorithm

- 验证 JWT 签名算法与客户端配置的 token.endpoint.auth.signing.alg 一致

- 支持 HS256/RS256/ES256 等标准算法

validateToken

- 检查令牌 exp （过期时间）和 iat （签发时间）

- 允许 15 秒时钟偏差（ ALLOWED\_CLOCK\_SKEW ）

- 处理缺失 exp 时的特殊逻辑（默认 10 秒有效期）

validateTokenReuse

- 使用 SingleUseObjectProvider 缓存令牌 ID

- 若令牌 ID 已存在则判定为重放攻击

- 缓存时长基于令牌有效期动态计算

validateTokenAudience

- 通过 `getExpectedAudiences` 生成预期受众列表（发行者 URL、令牌端点、内省端点等）

- 验证 JWT aud 声明包含至少一个预期受众

- 控制是否允许多受众（通过 allowMultipleAudiencesForJwtClientAuthentication 配置）

受众验证

`validateTokenAudience`

- 通过 `getExpectedAudiences` 生成预期受众列表（发行者 URL、令牌端点、内省端点等）

- 验证 JWT aud 声明包含至少一个预期受众

- 控制是否允许多受众（通过 allowMultipleAudiencesForJwtClientAuthentication 配置）

authenticateClient

    public void authenticateClient(ClientAuthenticationFlowContext context) {

        JWTClientValidator validator = new JWTClientValidator(context, getId());

        if (!validator.clientAssertionParametersValidation()) return;

        try {

            validator.readJws();

            if (!validator.validateClient()) return;

            if (!validator.validateSignatureAlgorithm()) return;

            RealmModel realm = validator.getRealm();

            ClientModel client = validator.getClient();

            JWSInput jws = validator.getJws();

            JsonWebToken token = validator.getToken();

            String clientAssertion = validator.getClientAssertion();

            // Get client key and validate signature

            PublicKey clientPublicKey = getSignatureValidationKey(client, context, jws);

            if (clientPublicKey == null) {

                // Error response already set to context

                return;

            }

            boolean signatureValid;

            try {

                JsonWebToken jwt = context.getSession().tokens().decodeClientJWT(clientAssertion, client, (jose, validatedClient) -> {

                    DEFAULT\_VALIDATOR.accept(jose, validatedClient);

                    String signatureAlgorithm = jose.getHeader().getRawAlgorithm();

                    ClientSignatureVerifierProvider signatureProvider = context.getSession().getProvider(ClientSignatureVerifierProvider.class, signatureAlgorithm);

                    if (signatureProvider == null) {

                        throw new RuntimeException("Algorithm not supported");

                    }

                    if (!signatureProvider.isAsymmetricAlgorithm()) {

                        throw new RuntimeException("Algorithm is not asymmetric");

                    }

                }, JsonWebToken.class);

                signatureValid = jwt != null;

            } catch (RuntimeException e) {

                Throwable cause = e.getCause() != null ? e.getCause() : e;

                throw new RuntimeException("Signature on JWT token failed validation", cause);

            }

            if (!signatureValid) {

                throw new RuntimeException("Signature on JWT token failed validation");

            }

            validator.validateTokenAudience(context, realm, token);

            validator.validateToken();

            validator.validateTokenReuse();

            context.success();

        } catch (Exception e) {

            ServicesLogger.LOGGER.errorValidatingAssertion(e);

            Response challengeResponse = ClientAuthUtil.errorResponse(Response.Status.BAD\_REQUEST.getStatusCode(), OAuthErrorException.INVALID\_CLIENT, "Client authentication with signed JWT failed: " + e.getMessage());

            context.failure(AuthenticationFlowError.INVALID\_CLIENT\_CREDENTIALS, challengeResponse);

        }

    }

初始化JWT认证器

        JWTClientValidator validator = new JWTClientValidator(context, getId());

验证客户端的请求是否有断言参数

        if (!validator.clientAssertionParametersValidation()) return;

读取并解析JWS（签名）格式的令牌

            validator.readJws();

            if (!validator.validateClient()) return;

            if (!validator.validateSignatureAlgorithm()) return;

获取客户端公钥进行签名验证

            PublicKey clientPublicKey = getSignatureValidationKey(client, context, jws);

            if (clientPublicKey == null) {

                // Error response already set to context

                return;

            }

JWT令牌解码与签名验证

            boolean signatureValid;

            try {

                JsonWebToken jwt = context.getSession().tokens().decodeClientJWT(clientAssertion, client, (jose, validatedClient) -> {

                    DEFAULT\_VALIDATOR.accept(jose, validatedClient);

                    String signatureAlgorithm = jose.getHeader().getRawAlgorithm();

                    ClientSignatureVerifierProvider signatureProvider = context.getSession().getProvider(ClientSignatureVerifierProvider.class, signatureAlgorithm);

                    if (signatureProvider == null) {

                        throw new RuntimeException("Algorithm not supported");

                    }

                    if (!signatureProvider.isAsymmetricAlgorithm()) {

                        throw new RuntimeException("Algorithm is not asymmetric");

                    }

                }, JsonWebToken.class);

                signatureValid = jwt != null;

            } catch (RuntimeException e) {

                Throwable cause = e.getCause() != null ? e.getCause() : e;

                throw new RuntimeException("Signature on JWT token failed validation", cause);

            }

            if (!signatureValid) {

                throw new RuntimeException("Signature on JWT token failed validation");

            }

验证签名

JsonWebToken jwt = context.getSession().tokens().decodeClientJWT(clientAssertion, client, (jose, validatedClient) -> {

                    DEFAULT\_VALIDATOR.accept(jose, validatedClient);

                    String signatureAlgorithm = jose.getHeader().getRawAlgorithm();

                    ClientSignatureVerifierProvider signatureProvider = context.getSession().getProvider(ClientSignatureVerifierProvider.class, signatureAlgorithm);

                    if (signatureProvider == null) {

                        throw new RuntimeException("Algorithm not supported");

                    }

                    if (!signatureProvider.isAsymmetricAlgorithm()) {

                        throw new RuntimeException("Algorithm is not asymmetric");

                    }

                }, JsonWebToken.class);

验证audience 验证token

            validator.validateTokenAudience(context, realm, token);

            validator.validateToken();

            validator.validateTokenReuse();

JWSInput