8 客户端认证方式 之 private\_key\_jwt

这篇文章来讲讲 private\_key\_jwt 方式。

private\_key\_jwt

private\_key\_jwt 方式就是利用 JWT 进行认证。请求方拥有自己的公私钥（密钥对），使用私钥对 JWT 加签，并将公钥暴露给 授权服务器。授权服务器通过请求方的公钥验证 JWT。也能达到客户端认证的目的。

请求方 维护了一对公私钥，通过 RSA算法，使用私钥 将客户端信息加签生成 JWT ；另外还通过接口暴露 公钥 给 授权服务器；

授权服务器 使用 请求方 的 公钥 对请求方的 JWT 进行验签以认证客户端。

如果你是从上一篇文章过来的，你会发现 private\_key\_jwt 和 client\_secret\_jwt 比较类似，都是通过jwt来做认证。不同之处在于，client\_secret\_jwt 使用 client\_secret+对称签名算法 对JWT加签；而 private\_key\_jwt 使用 公私钥+非对称签名算法 对JWT加签。在代码层面，两者的处理类是一样的，只是走了不同分支，这在源码分析中我们将会详细说说。

了解 JWT 的签名算法

示例

请求方传参：

client\_id：客户端id

client\_assertion\_type：固定值 urn:ietf:params:oauth:client-assertion-type:jwt-bearer

client\_assertion：使用私钥加签的jwt

环境准备

请求方

请求方本身需要暴露 公钥 的获取接口，所以，此请求方本身也是一个服务，我们也是通过SpringBoot快速创建一个Web服务。

生成公私钥（密钥对）

执行命令：keytool -genkeypair -keystore my.jks -storepass 123456 -alias my-key -keypass 654321 -keyalg RSA -keysize 2048 -sigalg SHA256withRSA -validity 365 -v

填写必要的信息，最后确认 Y。生成一个 my.jks 密钥对文件。将它放到请求方项目 src/main/resources 中。

创建web服务，并暴露公钥获取接口

@RestController

public class JwksController {

@GetMapping("/jwks")

public String jwkSet() throws JOSEException {

KeyPair keyPair = loadRsaKey();

RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();

RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();

RSAKey rsaKey = new RSAKey.Builder(publicKey)

.privateKey(privateKey)

.keyID(UUID.randomUUID().toString())

.build();

JWKSet jwkSet = new JWKSet(rsaKey);

return jwkSet.toString();

}

/\*\*

\* 加载自定义的 密钥对

\*/

public static KeyPair loadRsaKey(){

KeyPair keyPair;

try {

ClassPathResource resource = new ClassPathResource("my.jks");

KeyStore ks = KeyStore.getInstance("jks");

ks.load(resource.getInputStream(), "123456".toCharArray());

PrivateKey priKey = (PrivateKey)ks.getKey("my-key", "654321".toCharArray());

PublicKey pubKey = ks.getCertificate("my-key").getPublicKey();

keyPair = new KeyPair(pubKey, priKey);

} catch (Exception e) {

throw new IllegalStateException(e);

}

return keyPair;

}

}

启动服务。

另外，生成jwt的Java代码如下：

public class ClientJwtTest {

public static void main(String[] args) throws JOSEException {

String clientId = "client4";

String clientSecret = "01234567890123456789012345678912";

// 至少以下四项信息

JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()

// 主体：固定clientId

.subject(clientId)

// 发行者：固定clientId

.issuer(clientId)

// 授权中心的地址

.audience("http://localhost:9000")

// 过期时间 24h

.expirationTime(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 24))

.build();

String jwt = rsaSign(claimsSet);

System.out.println(jwt);

// eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJjbGllbnQ0Iiwic3ViIjoiY2xpZW50NCIsImF1ZCI6Imh0dHA6XC9cL2xvY2FsaG9zdDo5MDAwIiwiZXhwIjoxNjY0NTAyNTQ0fQ.dcgZNb\_pSRGmA36SoU2EeI1ZcwYjSfLNvs\_xy14m3m9kU6yA4Q2-mny8CqXwPW9JVPd0UlExV6lizJsjpf2WUBIAbwjnifAwX7bGPp7rlwAgQQm5CCClE\_G5KpRWAtsjeLH6hu-AYP5vkGg\_5ErhMR23YvEQBf41N\_c8UDa9kn6Ti0PH86ZjFBssZmlxaKXQXu6gmQiwEE1JpDo-hWvro62TEyYS0vCcMJGdJE5CMQ-sPweKOnAC1D3mdbFqeop\_LCtfpshbbirbFiUKXK3C4hR9TbHiaFuW4myt\_yx2RdPOGUs5IC2l-QDMxwbE8s4PAx-5uhHNvA\_nQCtMbUkmOg

}

/\*\*

\* 使用 RSA 算法加签生成jwt

\*/

private static String rsaSign(JWTClaimsSet claimsSet) throws JOSEException {

KeyPair keyPair = JwksController.loadRsaKey();

RSASSASigner signer = new RSASSASigner(keyPair.getPrivate());

SignedJWT signedJWT = new SignedJWT(new JWSHeader(JWSAlgorithm.RS256), claimsSet);

signedJWT.sign(signer);

String token = signedJWT.serialize();

return token;

}

}

授权服务器

同样的，基于 快速搭建一个授权服务器 文章中的示例，修改 SecurityConfiguration 中 registeredClientRepository() 方法，如下：

@Bean

public RegisteredClientRepository registeredClientRepository() {

RegisteredClient registeredClient4 = RegisteredClient.withId(UUID.randomUUID().toString())

.clientId("client4")

// 利用公私钥验证，可以不需要 client\_secret

// .clientSecret("01234567890123456789012345678912")

.clientAuthenticationMethod(ClientAuthenticationMethod.PRIVATE\_KEY\_JWT)

.clientSettings(ClientSettings.builder()

// JWT 方式必须配置，确定jwt的签名算法（PRIVATE\_KEY\_JWT 方式使用 SignatureAlgorithm（非对称加密算法，公私钥加密））

.tokenEndpointAuthenticationSigningAlgorithm(SignatureAlgorithm.RS256)

// 配置公钥获取地址，公私钥加密需要获取公钥来验证 jwt（验签）

.jwkSetUrl("http://localhost:8088/jwks")

.build())

.authorizationGrantType(AuthorizationGrantType.CLIENT\_CREDENTIALS)

.build();

return new InMemoryRegisteredClientRepository(registeredClient4);

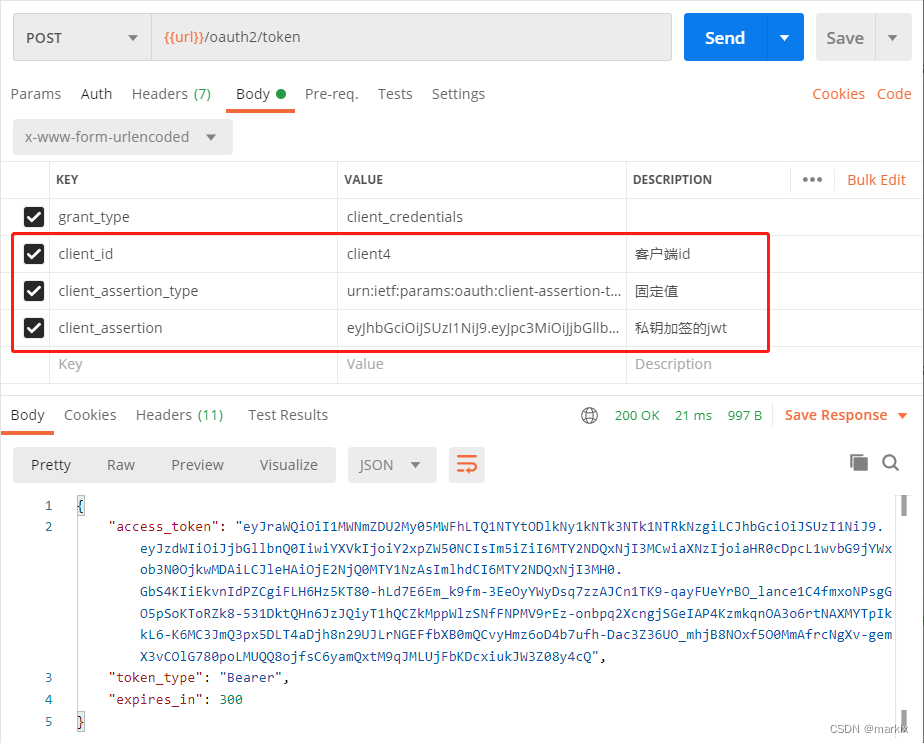
}

测试

生成 JWT

使用上文的Java代码，生成一个jwt

使用Postman测试，在 Body栏，填入’client\_id、client\_assertion\_type、client\_assertion 和grant\_type’，发送请求。



可以看到，使用此方式能成功获取到 access\_token，说明授权服务器确实支持此认证方式。

相应的curl命令如下：

curl --location --request POST 'localhost:9000/oauth2/token' \

--header 'Content-Type: application/x-www-form-urlencoded' \

--data-urlencode 'grant\_type=client\_credentials' \

--data-urlencode 'client\_id=client4' \

--data-urlencode 'client\_assertion\_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer' \

--data-urlencode 'client\_assertion=eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJjbGllbnQ0Iiwic3ViIjoiY2xpZW50NCIsImF1ZCI6Imh0dHA6XC9cL2xvY2FsaG9zdDo5MDAwIiwiZXhwIjoxNjY0NTAyNTQ0fQ.dcgZNb\_pSRGmA36SoU2EeI1ZcwYjSfLNvs\_xy14m3m9kU6yA4Q2-mny8CqXwPW9JVPd0UlExV6lizJsjpf2WUBIAbwjnifAwX7bGPp7rlwAgQQm5CCClE\_G5KpRWAtsjeLH6hu-AYP5vkGg\_5ErhMR23YvEQBf41N\_c8UDa9kn6Ti0PH86ZjFBssZmlxaKXQXu6gmQiwEE1JpDo-hWvro62TEyYS0vCcMJGdJE5CMQ-sPweKOnAC1D3mdbFqeop\_LCtfpshbbirbFiUKXK3C4hR9TbHiaFuW4myt\_yx2RdPOGUs5IC2l-QDMxwbE8s4PAx-5uhHNvA\_nQCtMbUkmOg'

源码分析

private\_key\_jwt 认证的处理类和 client\_secret\_jwt 认证的处理类一样，只是某些逻辑在不同的认证方式会走不同的分支。下面分析时没特别说明的话，表示逻辑和 client\_secret\_jwt 一致。

JwtClientAssertionAuthenticationConverter

从请求中解析出 client\_id、client\_assertion\_type、client\_assertion 参数。

JwtClientAssertionAuthenticationProvider

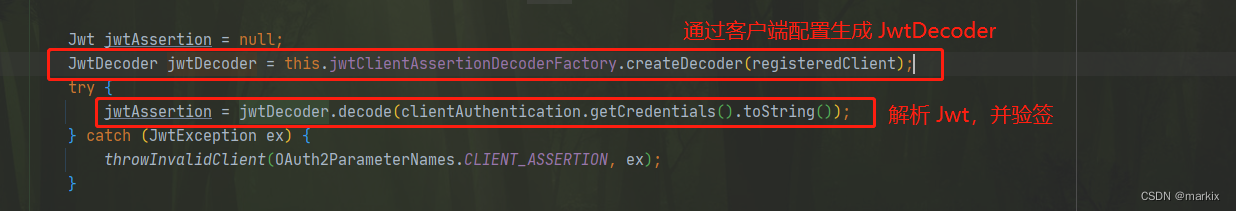
JwtClientAssertionAuthenticationProvider 的核心就是对JWT进行解析和验证。

核心流程如下：

使用请求携带的 clientId 查询客户端信息，若不存在则直接抛出异常。

创建解析 JWT 的核心类 JwtDecoder

解析 JWT，并验签



解析：

得益于框架优秀的设计，实际上 private\_key\_jwt 和 client\_secret\_jwt 的处理逻辑只是在 步骤2 中有些许差异，其他步骤处理逻辑都是一样的，我们下面重点分析步骤2。其他步骤的分析，读者可以回看上一篇文章客户端认证方式 之 client\_secret\_jwt。

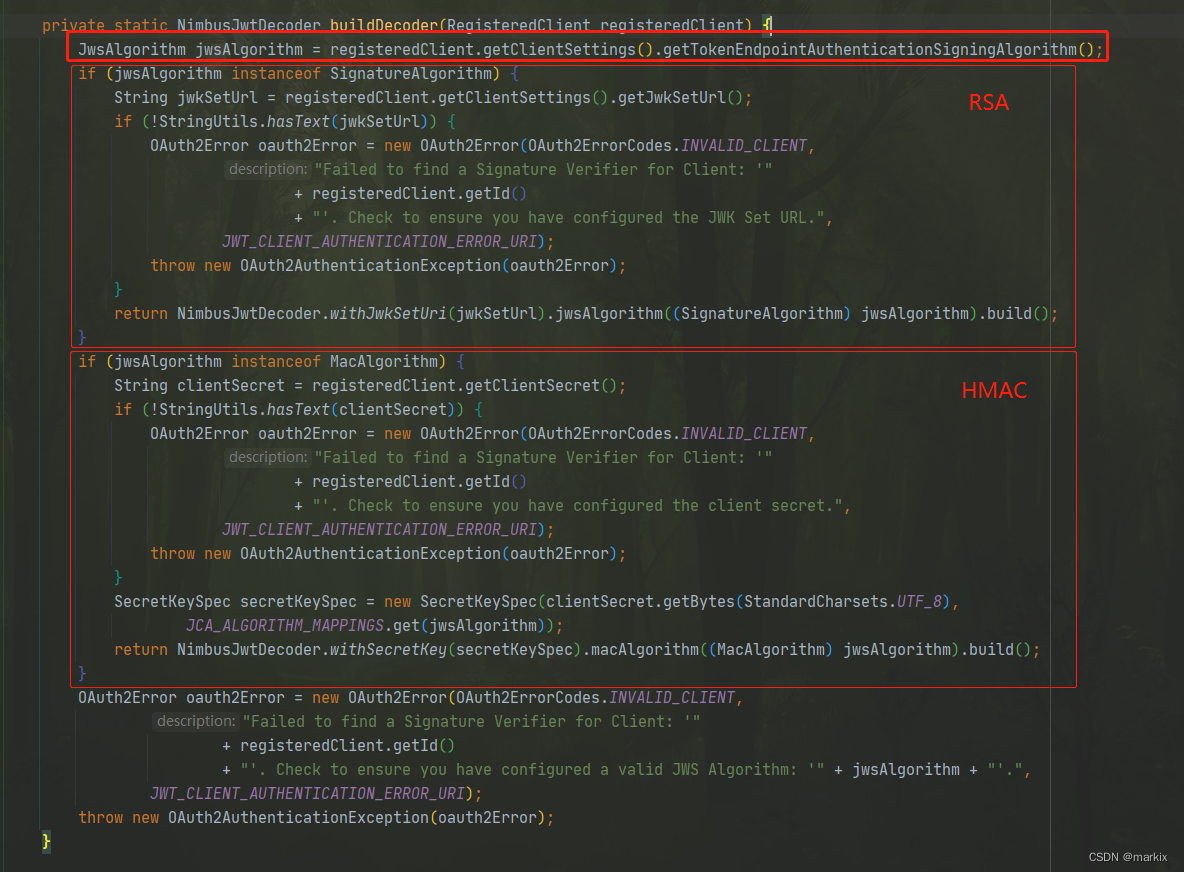
归根结底，两者的差异就在于 JwtDecoder 的创建过程，我们知道JwtDecoder 的创建过程需要设置签名算法和SecretKey（密钥）。

对于 client\_secret\_jwt 来说，签名算法就是 HMAC，密钥则是 client\_secret。

对于 private\_key\_jwt 来说，签名算法就是 RAS，密钥则是 请求方提供的公钥（需要通过API去获取）。

如何区分当前要以哪种方式创建 JwtDecoder 呢？

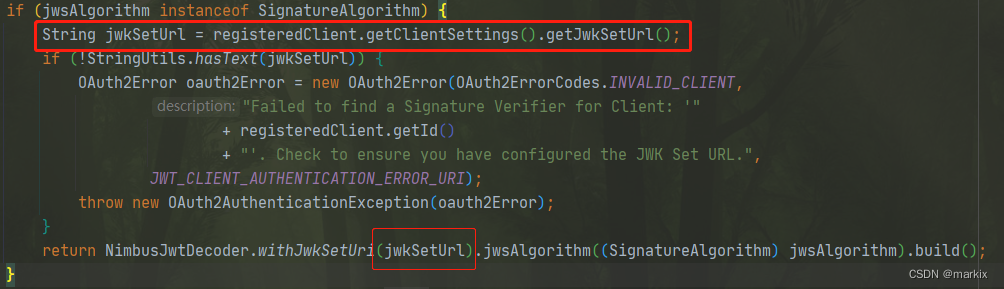
答案就是通过客户端的配置，有图有真相：



通过客户端的 TokenEndpointAuthenticationSigningAlgorithm 配置来确定要以哪种方式创建 JwtDecoder，从这也能看出对于同个客户端，不能同时支持 client\_secret\_jwt 认证方式和 private\_key\_jwt 认证方式。

ok，上一篇已经分析过client\_secret\_jwt认证的创建过程，本文的主角是private\_key\_jwt，我们便来深挖它是怎么创建的。前面我们只给客户端配置了 公钥的API，那它是如何获取到公钥的呢？

答案就在 NimbusJwtDecoder 之中。



它将 url 配置进NimbusJwtDecoder中，当需要对jwt验证时，会通过url去获取公钥。

创建过程：NimbusJwtDecoder#build() -> NimbusJwtDecoder#processor() -> NimbusJwtDecoder#jwkSource() -> 构建RemoteJWKSet

当需要验证时，获取公钥的流程：RemoteJWKSet#get() -> RemoteJWKSet#updateJWKSetFromURL() -> ResourceRetriever#retrieveResource(jwkSetURL)

至此，完成了配置中url的使命，获取到公钥，获取完成后也会缓存起来，并不是每次验证都会重新发送请求。

以上，便是 private\_key\_jwt 主干流程。

下集预告：客户端认证方式 之 none，敬请期待。

end