11 Securing service-to-service APIs

This chapter covers

- Authenticating services with API keys and JWTs
- Using OAuth2 for authorizing service-to-service API calls
- TLS client certificate authentication and mutual TLS
- Credential and key management for services
- Making service calls in response to user requests

In previous chapters, authentication has been used to determine which user is accessing an API and what they can do. It's increasingly common for services to talk to other services without a user being involved at all. These service-to-service API calls can occur within a single organization, such as between microservices, or between organizations when an API is exposed to allow other businesses to access data or services. For example, an online retailer might provide an API for resellers to search products and place orders on behalf of customers. In both cases, it is the API client that needs to be authenticated rather than an end user. Sometimes this is needed for billing or to apply limits according to a service contract, but it's also essential for security when sensitive data or operations may be performed. Services are often granted wider access than individual users, so stronger protections may be required because the damage from compromise of a service account can be greater than any individual user account. In this chapter, you'll learn how to authenticate services and additional hardening that can be applied to better protect privileged accounts, using advanced features of OAuth2.

NOTE The examples in this chapter require a running Kubernetes installation configured according to the instructions in appendix B.

11.1 API keys and JWT bearer authentication

One of the most common forms of service authentication is an API key, which is a simple bearer token that identifies the service client. An API key is very similar to the tokens you've used for user authentication in previous chapters, except that an API key identifies a service or business

rather than a user and usually has a long expiry time. Typically, a user logs in to a website (known as a developer portal) and generates an API key that they can then add to their production environment to authenticate API calls, as shown in figure 11.1.

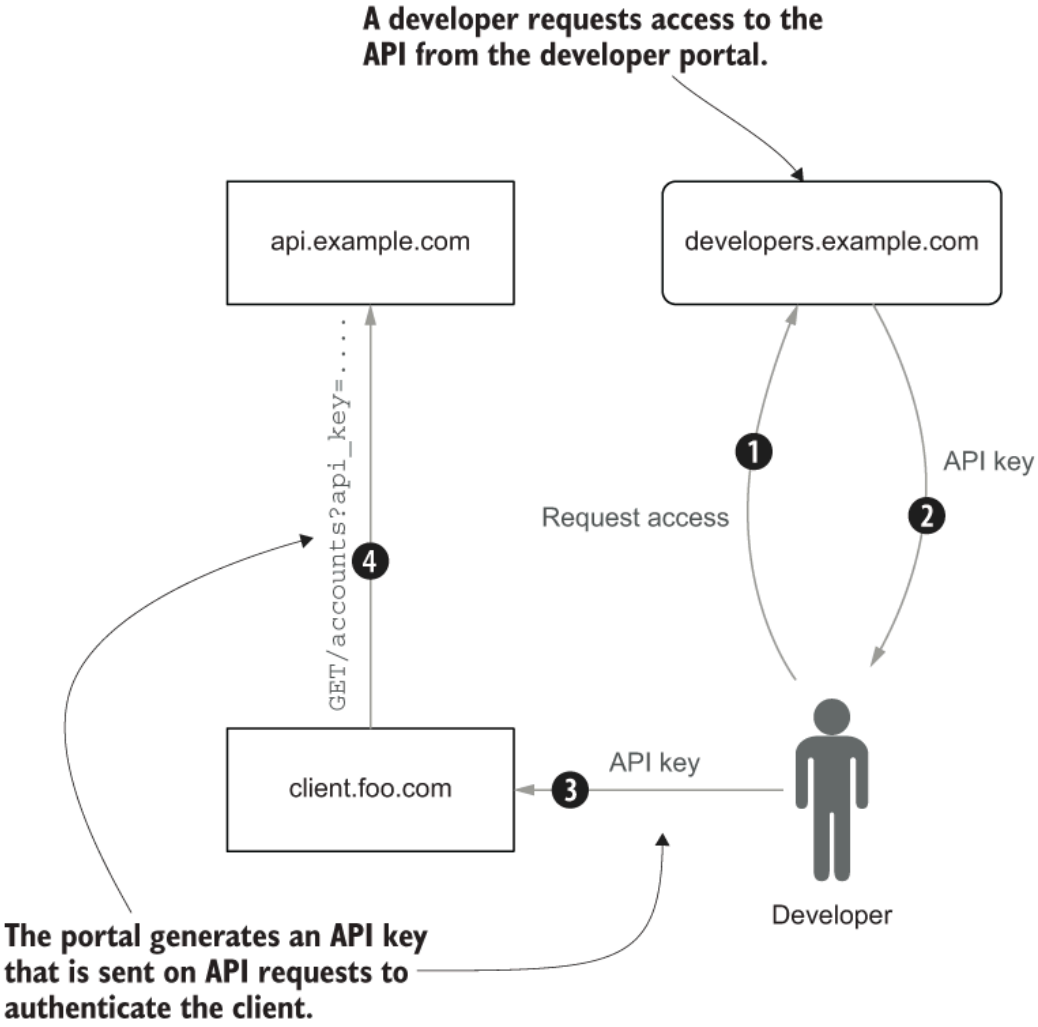

Figure 11.1 To gain access to an API, a representative of the organization logs into a developer portal and requests an API key. The portal generates the API key and returns it. The developer then includes the API key as a query parameter on requests to the API.

Section 11.5 covers techniques for securely deploying API keys and other credentials. The API key is added to each request as a request parameter or custom header.

DEFINITION An API key is a token that identifies a service client rather than a user. API keys are typically valid for a much longer time than a user token, often months or years.

Any of the token formats discussed in chapters 5 and 6 are suitable for generating API keys, with the username replaced by an identifier for the service or business that API usage should be associated with and the expiry time set to a few months or years in the future. Permissions or

scopes can be used to restrict which API calls can be called by which clients, and the resources they can read or modify, just as you've done for users in previous chapters--the same techniques apply.

An increasingly common choice is to replace ad hoc API key formats with standard JSON Web Tokens. In this case, the JWT is generated by the developer portal with claims describing the client and expiry time, and then either signed or encrypted with one of the symmetric authenticated encryption schemes described in chapter 6. This is known as JWT bearer authentication, because the JWT is acting as a pure bearer token: any client in possession of the JWT can use it to access the APIs it is valid for without presenting any other credentials. The JWT is usually passed to the API in the Authorization header using the standard Bearer scheme described in chapter 5.

DEFINITION In JWT bearer authentication, a client gains access to an API by presenting a JWT that has been signed by an issuer that the API trusts.

An advantage of JWTs over simple database tokens or encrypted strings is that you can use public key signatures to allow a single developer portal to generate tokens that are accepted by many different APIs. Only the developer portal needs to have access to the private key used to sign the JWTs, while each API server only needs access to the public key. Using public key signed JWTs in this way is covered in section 7.4.4, and the same approach can be used here, with a developer portal taking the place of the AS.

WARNING Although using JWTs for client authentication is more secure than client secrets, a signed JWT is still a bearer credential that can be used by anyone that captures it until it expires. A malicious or compromised API server could take the JWT and replay it to other APIs to impersonate the client. Use expiry, audience, and other standard JWT claims (chapter 6) to reduce the impact if a JWT is compromised.

11.2 The OAuth2 client credentials grant

Although JWT bearer authentication is appealing due to its apparent simplicity, you still need to develop the portal for generating JWTs, and you'll need to consider how to revoke tokens when a service is retired or a business partnership is terminated. The need to handle service-to-service API clients was anticipated by the authors of the OAuth2 specifications, and a dedicated grant type was added to support this case: the client credentials grant. This grant type allows an OAuth2 client to obtain an access token

using its own credentials without a user being involved at all. The access token issued by the authorization server (AS) can be used just like any other access token, allowing an existing OAuth2 deployment to be reused for service-to-service API calls. This allows the AS to be used as the developer portal and all the features of OAuth2, such as discoverable token revocation and introspection endpoints discussed in chapter 7, to be used for service calls.

WARNING If an API accepts calls from both end users and service clients, it's important to make sure that the API can tell which is which. Otherwise, users may be able to impersonate service clients or vice versa. The OAuth2 standards don't define a single way to distinguish these two cases, so you should consult the documentation for your AS vendor.

To obtain an access token using the client credentials grant, the client makes a direct HTTPS request to the token endpoint of the AS, specifying the `client_credentials` grant type and the scopes that it requires. The client authenticates itself using its own credentials. OAuth2 supports a range of different client authentication mechanisms, and you'll learn about several of them in this chapter. The simplest authentication method is known as `client_secret_basic`, in which the client presents its client ID and a secret value using HTTP Basic authentication.[1] For example, the following curl command shows how to use the client credentials grant to obtain an access token for a client with the ID `test` and secret value `password`:

```
$ curl -u test:password \                                    ❶
  -d 'grant_type=client_credentials&scope=a+b+c' \           ❷
  https://as.example.com/access_token
```

❶ Send the client ID and secret using Basic authentication.

❷ Specify the client_credentials grant.

Assuming the credentials are correct, and the client is authorized to obtain access tokens using this grant and the requested scopes, the response will be like the following:

```
{
  "access_token": "q4TNVUHUe9A9MilKIxZOCIs6fI0",
  "scope": "a b c",
  "token_type": "Bearer",
```

```
  "expires_in": 3599
}
```

NOTE OAuth2 client secrets are not passwords intended to be remembered by users. They are usually long random strings of high entropy that are generated automatically during client registration.

The access token can then be used to access APIs just like any other OAuth2 access token discussed in chapter 7. The API validates the access token in the same way that it would validate any other access token, either by calling a token introspection endpoint or directly validating the token if it is a JWT or other self-contained format.

TIP The OAuth2 spec advises AS implementations not to issue a refresh token when using the client credentials grant. This is because there is little point in the client using a refresh token when it can obtain a new access token by using the client credentials grant again.

11.2.1 Service accounts

As discussed in chapter 8, user accounts are often held in a LDAP directory or other central database, allowing APIs to look up users and determine their roles and permissions. This is usually not the case for OAuth2 clients, which are often stored in an AS-specific database as in figure 11.2. A consequence of this is that the API can validate the access token but then has no further information about who the client is to make access control decisions.

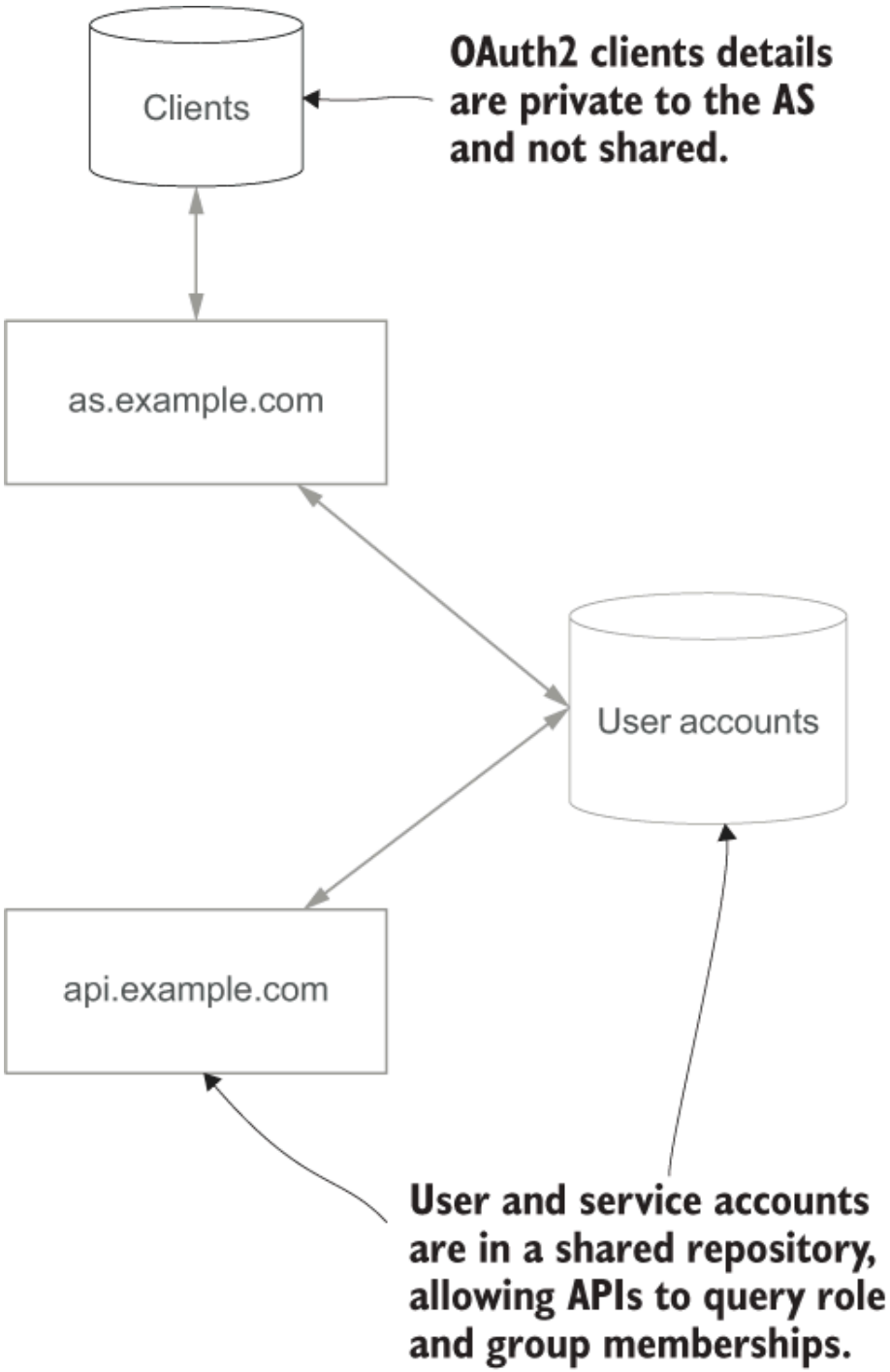

Figure 11.2 An authorization server (AS) typically stores client details in a private database, so these details are not accessible to APIs. A service account lives in the shared user repository, allowing APIs to look up identity details such as role or group membership.

One solution to this problem is for the API to make access control decisions purely based on the scope or other information related to the access token itself. In this case, access tokens act more like the capability tokens discussed in chapter 9, where the token grants access to resources on its own and the identity of the client is ignored. Fine-grained scopes can be used to limit the amount of access granted.

Alternatively, the client can avoid the client credentials grant and instead obtain an access token for a service account. A service account acts like a

regular user account and is created in a central directory and assigned permissions and roles just like any other account. This allows APIs to treat an access token issued for a service account the same as an access token issued for any other user, simplifying access control. It also allows administrators to use the same tools to manage service accounts that they use to manage user accounts. Unlike a user account, the password or other credentials for a service account should be randomly generated and of high entropy, because they don't need to be remembered by a human.

DEFINITION A service account is an account that identifies a service rather than a real user. Service accounts can simplify access control and account management because they can be managed with the same tools you use to manage users.

In a normal OAuth2 flow, such as the authorization code grant, the user's web browser is redirected to a page on the AS to login and consent to the authorization request. For a service account, the client instead uses a non-interactive grant type that allows it to submit the service account credentials directly to the token endpoint. The client must have access to the service account credentials, so there is usually a service account dedicated to each client. The simplest grant type to use is the Resource Owner Password Credentials (ROPC) grant type, in which the service account username and password are sent to the token endpoint as form fields:

```
$ curl -u test:password \                          ❶
  -d 'grant_type=password&scope=a+b+c' \
  -d 'username=serviceA&password=password' \       ❷
  https://as.example.com/access_token
```

❶ Send the client ID and secret using Basic auth.

❷ Pass the service account password in the form data.

This will result in an access token being issued to the `test` client with the service account `serviceA` as the resource owner.

WARNING Although the ROPC grant type is more secure for service accounts than for end users, there are better authentication methods available for service clients discussed in sections 11.3 and 11.4. The ROPC grant type may be deprecated or removed in future versions of OAuth.

The main downside of service accounts is the requirement for the client to manage two sets of credentials, one as an OAuth2 client and one for the service account. This can be eliminated by arranging for the same credentials to be used for both. Alternatively, if the client doesn't need to use features of the AS that require client credentials, it can be a public client and use only the service account credentials for access.

Pop quiz

1. Which of the following are differences between an API key and a user authentication token?
    1. API keys are more secure than user tokens.
    2. API keys can only be used during normal business hours.
    3. A user token is typically more privileged than an API key.
    4. An API key identifies a service or business rather than a user.
    5. An API key typically has a longer expiry time than a user token.
2. Which one of the following grant types is most easily used for authenticating a service account?
    1. PKCE
    2. Hugh Grant
    3. Implicit grant
    4. Authorization code grant
    5. Resource owner password credentials grant

The answers are at the end of the chapter.

11.3 The JWT bearer grant for OAuth2

NOTE To run the examples in this section, you'll need a running OAuth2 authorization server. Follow the instructions in appendix A to configure the AS and a test client before continuing with this section.

Authentication with a client secret or service account password is very simple, but suffers from several drawbacks:

- Some features of OAuth2 and OIDC require the AS to be able to access the raw bytes of the client secret, preventing the use of hashing. This increases the risk if the client database is ever compromised as an attacker may be able to recover all the client secrets.
- If communications to the AS are compromised, then an attacker can steal client secrets as they are transmitted. In section 11.4.6, you'll see how to harden access tokens against this possibility, but client secrets are inherently vulnerable to being stolen.

- It can be difficult to change a client secret or service account password, especially if it is shared by many servers.

For these reasons, it's beneficial to use an alternative authentication mechanism. One alternative supported by many authorization servers is the JWT Bearer grant type for OAuth2, defined in RFC 7523 (**https://tools.ietf.org/html/rfc7523**). This specification allows a client to obtain an access token by presenting a JWT signed by a trusted party, either to authenticate itself for the client credentials grant, or to exchange a JWT representing authorization from a user or service account. In the first case, the JWT is signed by the client itself using a key that it controls. In the second case, the JWT is signed by some authority that is trusted by the AS, such as an external OIDC provider. This can be useful if the AS wants to delegate user authentication and consent to a third-party service. For service account authentication, the client is often directly trusted with the keys to sign JWTs on behalf of that service account because there is a dedicated service account for each client. In section 11.5.3, you'll see how separating the duties of the client from the service account authentication can add an extra layer of security.

By using a public key signature algorithm, the client needs to supply only the public key to the AS, reducing the risk if the AS is ever compromised because the public key can only be used to verify signatures and not create them. Adding a short expiry time also reduces the risks when authenticating over an insecure channel, and some servers support remembering previously used JWT IDs to prevent replay.

Another advantage of JWT bearer authentication is that many authorization servers support fetching the client's public keys in JWK format from a HTTPS endpoint. The AS will periodically fetch the latest keys from the endpoint, allowing the client to change their keys regularly. This effectively bootstraps trust in the client's public keys using the web PKI: the AS trusts the keys because they were loaded from a URI that the client specified during registration and the connection was authenticated using TLS, preventing an attacker from injecting fake keys. The JWK Set format allows the client to supply more than one key, allowing it to keep using the old signature key until it is sure that the AS has picked up the new one (figure 11.3).

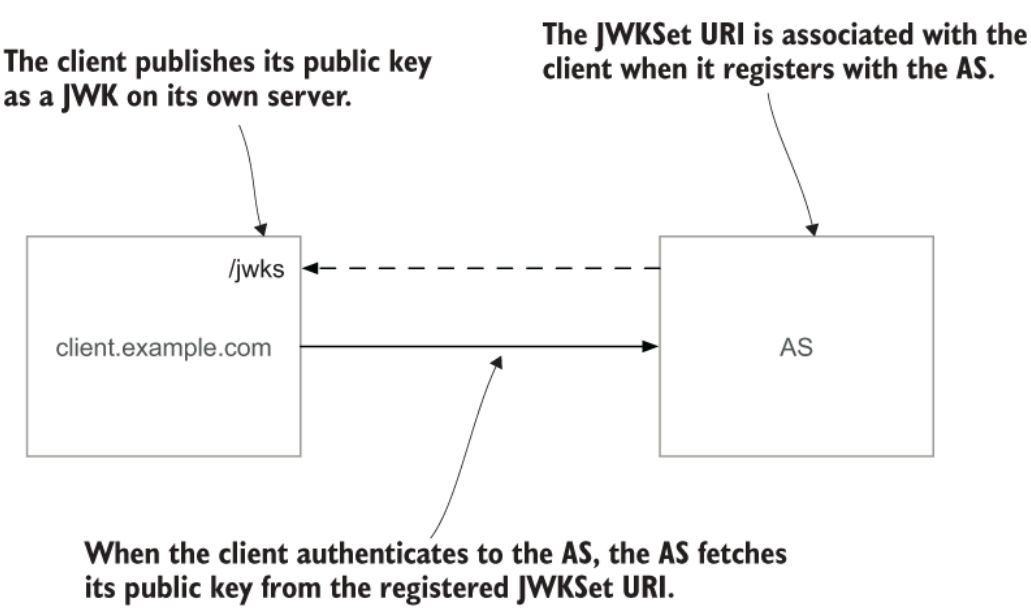

Figure 11.3 The client publishes its public key to a URI it controls and registers this URI with the AS. When the client authenticates, the AS will retrieve its public key over HTTPS from the registered URI. The client can publish a new public key whenever it wants to change the key.

11.3.1 Client authentication

To obtain an access token under its own authority, a client can use JWT bearer client authentication with the client credentials grant. The client performs the same request as you did in section 11.2, but rather than supplying a client secret using Basic authentication, you instead supply a JWT signed with the client's private key. When used for authentication, the JWT is also known as a client assertion.

DEFINITION An assertion is a signed set of identity claims used for authentication or authorization.

To generate the public and private key pair to use to sign the JWT, you can use `keytool` from the command line, as follows. Keytool will generate a certificate for TLS when generating a public key pair, so use the `-dname` option to specify the subject name. This is required even though you won't use the certificate. You'll be prompted for the keystore password.

```
keytool -genkeypair \
  -keystore keystore.p12 \                          ❶
  -keyalg EC -keysize 256 -alias es256-key \        ❷
  -dname cn=test                                    ❸
```

❶ Specify the keystore.

❷ Use the EC algorithm and 256-bit key size.

❸ Specify a distinguished name for the certificate.

TIP Keytool picks an appropriate elliptic curve based on the key size, and in this case happens to pick the correct P-256 curve required for the `ES256` algorithm. There are other 256-bit elliptic curves that are incompatible. In Java 12 and later you can use the `-groupname secp256r1` argument to explicitly specify the correct curve. For `ES384` the group name is `secp384r1` and for `ES512` it is `secp521r1` (note: 521 not 512). Keytool can't generate EdDSA keys at this time.

You can then load the private key from the keystore in the same way that you did in chapters 5 and 6 for the HMAC and AES keys. The JWT library requires that the key is cast to the specific `ECPrivateKey` type, so do that when you load it. Listing 11.1 shows the start of a `JwtBearerClient` class that you'll write to implement JWT bearer authentication. Navigate to src/main/java/com/manning/apisecurityinaction and create a new file named JwtBearerClient.java. Type in the contents of the listing and save the file. It doesn't do much yet, but you'll expand it next. The listing contains all the import statements you'll need to complete the class.

Listing 11.1 Loading the private key

```
package com.manning.apisecurityinaction;

import java.io.FileInputStream;
import java.net.URI;
import java.net.http.*;
import java.security.KeyStore;
import java.security.interfaces.ECPrivateKey;
import java.util.*;

import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.ECDSASigner;
import com.nimbusds.jose.jwk.*;
import com.nimbusds.jwt.*;

import static java.time.Instant.now;
import static java.time.temporal.ChronoUnit.SECONDS;
import static spark.Spark.*;

public class JwtBearerClient {
    public static void main(String... args) throws Exception {
        var password = "changeit".toCharArray();
```

```
        var keyStore = KeyStore.getInstance("PKCS12");
        keyStore.load(new FileInputStream("keystore.p12"),
                password);
        var privateKey = (ECPrivateKey)                          ❶
                keyStore.getKey("es256-key", password);          ❶
    }
}
```

❶ Cast the private key to the required type.

For the AS to be able to validate the signed JWT you send, it needs to know where to find the public key for your client. As discussed in the introduction to section 11.3, a flexible way to do this is to publish your public key as a JWK Set because this allows you to change your key regularly by simply publishing a new key to the JWK Set. The Nimbus JOSE+JWT library that you used in chapter 5 supports generating a JWK Set from a keystore using the `JWKSet.load` method, as shown in listing 11.2. After loading the JWK Set, use the `toPublicJWKSet` method to ensure that it only contains public key details and not the private keys. You can then use Spark to publish the JWK Set at a HTTPS URI using the standard `application/jwk-set+json` content type. Make sure that you turn on TLS support using the `secure` method so that the keys can't be tampered with in transit, as discussed in chapter 3. Open the JwtBearerClient.java file again and add the code from the listing to the main method, after the existing code.

WARNING Make sure you don't forget the `.toPublicJWKSet()` method call. Otherwise you'll publish your private keys to the internet!

Listing 11.2 Publishing a JWK Set

```
var jwkSet = JWKSet.load(keyStore, alias -> password)      ❶
       .toPublicJWKSet();                                  ❷

secure("localhost.p12", "changeit", null, null);           ❸
get("/jwks", (request, response) -> {                      ❸
    response.type("application/jwk-set+json");             ❸
    return jwkSet.toString();                              ❸
});                                                        ❸
```

❶ Load the JWK Set from the keystore.

❷ Ensure it contains only public keys.

❸ Publish the JWK Set to a HTTPS endpoint using Spark.

The Nimbus JOSE library requires the Bouncy Castle cryptographic library to be loaded to enable JWK Set support, so add the following dependency to the Maven pom.xml file in the root of the Natter API project:

```
    <dependency>
      <groupId>org.bouncycastle</groupId>
      <artifactId>bcpkix-jdk15on</artifactId>
      <version>1.66</version>
    </dependency>
```

You can now start the client by running the following command in the root folder of the Natter API project:

```
mvn clean compile exec:java \
  -Dexec.mainClass=com.manning.apisecurityinaction.JwtBearerClient
```

In a separate terminal, you can then test that the public keys are being published by running:

```
curl https://localhost:4567/jwks > jwks.txt
```

The result will be a JSON object containing a single `keys` field, which is an array of JSON Web Keys.

By default, the AS server running in Docker won't be able to access the URI that you've published the keys to, so for this example you can copy the JWK Set directly into the client settings. If you're using the ForgeRock Access Management software from appendix A, then log in to the admin console as amadmin as described in the appendix and carry out the following steps:

1. Navigate to the Top Level Realm and click on Applications in the left-hand menu and then OAuth2.0.
2. Click on the test client you registered when installing the AS.
3. Select the Signing and Encryption tab, and then copy and paste the contents of the jwks.txt file you just saved into the Json Web Key field.
4. Find the Token Endpoint Authentication Signing Algorithm field just above the JWK field and change it to ES256.

5. Change the Public Key Selector field to “JWKs” to ensure the keys you just configured are used.
6. Finally, scroll down and click Save Changes at the lower right of the screen.

11.3.2 Generating the JWT

A JWT used for client authentication must contain the following claims:

- The `sub` claim is the ID of the client.
- An `iss` claim that indicates who signed the JWT. For client authentication this is also usually the client ID.
- An `aud` claim that lists the URI of the token endpoint of the AS as the intended audience.
- An `exp` claim that limits the expiry time of the JWT. An AS may reject a client authentication JWT with an unreasonably long expiry time to reduce the risk of replay attacks.

Some authorization servers also require the JWT to contain a `jti` claim with a unique random value in it. The AS can remember the `jti` value until the JWT expires to prevent replay if the JWT is intercepted. This is very unlikely because client authentication occurs over a direct TLS connection between the client and the AS, but the use of a `jti` is required by the OpenID Connect specifications, so you should add one to ensure maximum compatibility. Listing 11.3 shows how to generate a JWT in the correct format using the Nimbus JOSE+JWT library that you used in chapter 6. In this case, you’ll use the `ES256` signature algorithm (ECDSA with SHA-256), which is widely implemented. Generate a JWT header indicating the algorithm and the key ID (which corresponds to the keystore alias). Populate the JWT claims set values as just discussed. Finally, sign the JWT to produce the assertion value. Open the JwtBearerClient.java file and type in the contents of the listing at the end of the `main` method.

Listing 11.3 Generating a JWT client assertion

```
var clientId = "test";
var as = "https://as.example.com:8080/oauth2/access_token";
var header = new JWSHeader.Builder(JWSAlgorithm.ES256)          (1)
        .keyID("es256-key")                                    (1)
        .build();                                              (1)
var claims = new JWTClaimsSet.Builder()
        .subject(clientId)                                     (2)
        .issuer(clientId)                                      (2)
        .expirationTime(Date.from(now().plus(30, SECONDS)))    (3)
```

```
        .audience(as)                                          4
        .jwtID(UUID.randomUUID().toString())                   5
        .build();
var jwt = new SignedJWT(header, claims);                       6
jwt.sign(new ECDSASigner(privateKey));                         6
var assertion = jwt.serialize();                               6
```

1 Create a header with the correct algorithm and key ID.

2 Set the subject and issuer claims to the client ID.

3 Add a short expiration time.

4 Set the audience to the AS token endpoint.

5 Add a random JWT ID claim to prevent replay.

6 Sign the JWT with the private key.

Once you've registered the JWK Set with the AS, you should then be able to generate an assertion and use it to authenticate to the AS to obtain an access token. Listing 11.4 shows how to format the client credentials request with the client assertion and send it to the AS an HTTP request. The JWT assertion is passed as a new `client_assertion` parameter, and the `client_assertion_type` parameter is used to indicate that the assertion is a JWT by specifying the value:

```
urn:ietf:params:oauth:client-assertion-type:jwt-bearer
```

The encoded form parameters are then POSTed to the AS token endpoint using the Java HTTP library. Open the JwtBearerClient.java file again and add the contents of the listing to the end of the `main` method.

Listing 11.4 Sending the request to the AS

```
var form = "grant_type=client_credentials&scope=create_space" +    1
        "&client_assertion_type=" +                                1
"urn:ietf:params:oauth:client-assertion-type:jwt-bearer" +         1
        "&client_assertion=" + assertion;                          1
var httpClient = HttpClient.newHttpClient();                       2
var request = HttpRequest.newBuilder()                             2
        .uri(URI.create(as))                                       2
        .header("Content-Type", "application/x-www-form-urlencoded")  2
```

```
        .POST(HttpRequest.BodyPublishers.ofString(form))      ❷
        .build();                                             ❷
var response = httpClient.send(request,                       ❸
        HttpResponse.BodyHandlers.ofString());               ❸
System.out.println(response.statusCode());
System.out.println(response.body());
```

❶ Build the form content with the assertion JWT.

❷ Create the POST request to the token endpoint.

❸ Send the request and parse the response.

Run the following Maven command to test out the client and receive an access token from the AS:

```
mvn -q clean compile exec:java \
  -Dexec.mainClass=com.manning.apisecurityinaction.JwtBearerClient
```

After the client flow completes, it will print out the access token response from the AS.

11.3.3 Service account authentication

Authenticating a service account using JWT bearer authentication works a lot like client authentication. Rather than using the client credentials grant, a new grant type named

```
    urn:ietf:params:oauth:grant-type:jwt-bearer
```

is used, and the JWT is sent as the value of the `assertion` parameter rather than the `client_assertion` parameter. The following code snippet shows how to construct the form when using the JWT bearer grant type to authenticate using a service account:

```
var form = "grant_type=" +                                    ❶
      "urn:ietf:params:oauth:grant-type:jwt-bearer" +         ❶
      "&scope=create_space&assertion=" + assertion;           ❷
```

❶ Use the jwt-bearer grant type.

❷ Pass the JWT as the assertion parameter.

The claims in the JWT are the same as those used for client authentication, with the following exceptions:

- The `sub` claim should be the username of the service account rather than the client ID.
- The `iss` claim may also be different from the client ID, depending on how the AS is configured.

There is an important difference in the security properties of the two methods, and this is often reflected in how the AS is configured. When the client is using a JWT to authenticate itself, the JWT is a self-assertion of identity. If the authentication is successful, then the AS issues an access token authorized by the client itself. In the JWT bearer grant, the client is asserting that it is authorized to receive an access token on behalf of the given user, which may be a service account or a real user. Because the user is not present to consent to this authorization, the AS will usually enforce stronger security checks before issuing the access token. Otherwise, a client could ask for access tokens for any user it liked without the user being involved at all. For example, an AS might require separate registration of trusted JWT issuers with settings to limit which users and scopes they can authorize access tokens for.

An interesting aspect of JWT bearer authentication is that the issuer of the JWT and the client can be different parties. You'll use this capability in section 11.5.3 to harden the security of a service environment by ensuring that pods running in Kubernetes don't have direct access to privileged service credentials.

Pop quiz

3. Which one of the following is the primary reason for preferring a service account over the client credentials grant?
    1. Client credentials are more likely to be compromised.
    2. It's hard to limit the scope of a client credentials grant request.
    3. It's harder to revoke client credentials if the account is compromised.
    4. The client credentials grant uses weaker authentication than service accounts.
    5. Clients are usually private to the AS while service accounts can live in a shared repository.

4. Which of the following are reasons to prefer JWT bearer authentication over client secret authentication? (There may be multiple correct answers.)
    1. JWTs are simpler than client secrets.
    2. JWTs can be compressed and so are smaller than client secrets.
    3. The AS may need to store the client secret in a recoverable form.
    4. A JWT can have a limited expiry time, reducing the risk if it is stolen.
    5. JWT bearer authentication avoids sending a long-lived secret over the network.

The answers are at the end of the chapter.

11.4 Mutual TLS authentication

JWT bearer authentication is more secure than sending a client secret to the AS, but as you've seen in section 11.3.1, it can be significantly more complicated for the client. OAuth2 requires that connections to the AS are made using TLS, and you can use TLS for secure client authentication as well. In a normal TLS connection, only the server presents a certificate that authenticates who it is. As explained in chapter 10, this is all that is required to set up a secure channel as the client connects to the server, and the client needs to be assured that it has connected to the right server and not a malicious fake. But TLS also allows the client to optionally authenticate with a client certificate, allowing the server to be assured of the identity of the client and use this for access control decisions. You can use this capability to provide secure authentication of service clients. When both sides of the connection authenticate, this is known as mutual TLS (mTLS).

TIP Although it was once hoped that client certificate authentication would be used for users, perhaps even replacing passwords, it is very seldom used. The complexity of managing keys and certificates makes the user experience very poor and confusing. Modern user authentication methods such as WebAuthn (**https://webauthn.guide**) provide many of the same security benefits and are much easier to use.

11.4.1 How TLS certificate authentication works

The full details of how TLS certificate authentication works would take many chapters on its own, but a sketch of how the process works in the most common case will help you to understand the security properties provided. TLS communication is split into two phases:

1. An initial handshake, in which the client and the server negotiate which cryptographic algorithms and protocol extensions to use, optionally authenticate each other, and agree on shared session keys.
2. An application data transmission phase in which the client and server use the shared session keys negotiated during the handshake to exchange data using symmetric authenticated encryption.[2]

During the handshake, the server presents its own certificate in a TLS Certificate message. Usually this is not a single certificate, but a certificate chain, as described in chapter 10: the server's certificate is signed by a certificate authority (CA), and the CA's certificate is included too. The CA may be an intermediate CA, in which case another CA also signs its certificate, and so on until at the end of the chain is a root CA that is directly trusted by the client. The root CA certificate is usually not sent as part of the chain as the client already has a copy.

RECAP A certificate contains a public key and identity information of the subject the certificate was issued to and is signed by a certificate authority. A certificate chain consists of the server or client certificate followed by the certificates of one or more CAs. Each certificate is signed by the CA following it in the chain until a root CA is reached that is directly trusted by the recipient.

To enable client certificate authentication, the server sends a CertificateRequest message, which requests that the client also present a certificate, and optionally indicates which CAs it is willing to accept certificates signed by and the signature algorithms it supports. If the server doesn't send this message, then the client certificate authentication is disabled. The client then responds with its own Certificate message containing its certificate chain. The client can also ignore the certificate request, and the server can then choose whether to accept the connection or not.

NOTE The description in this section is of the TLS 1.3 handshake (simplified). Earlier versions of the protocol use different messages, but the process is equivalent.

If this was all that was involved in TLS certificate authentication, it would be no different to JWT bearer authentication, and the server could take the client's certificates and present them to other servers to impersonate the client, or vice versa. To prevent this, whenever the client or server present a Certificate message TLS requires them to also send a CertificateVerify message in which they sign a transcript of all previous messages exchanged during the handshake. This proves that the client (or

server) has control of the private key corresponding to their certificate and ensures that the signature is tightly bound to this specific handshake: there are unique values exchanged in the handshake, preventing the signature being reused for any other TLS session. The session keys used for authenticated encryption after the handshake are also derived from these unique values, ensuring that this one signature during the handshake effectively authenticates the entire session, no matter how much data is exchanged. Figure 11.4 shows the main messages exchanged in the TLS 1.3 handshake.

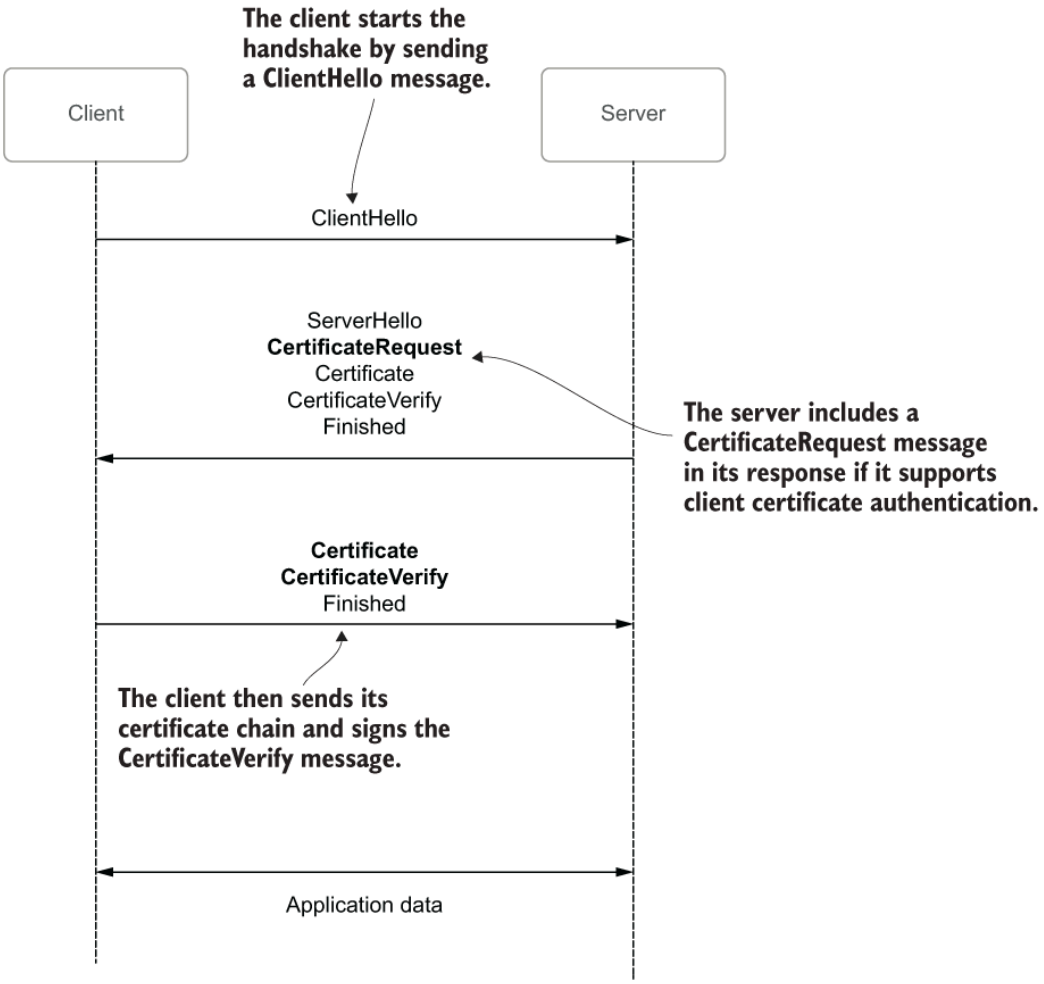

Figure 11.4 In the TLS handshake, the server sends its own certificate and can ask the client for a certificate using a CertificateRequest message. The client responds with a Certificate message containing the certificate and a CertificateVerify message proving that it owns the associated private key.

LEARN ABOUT IT We've only given a brief sketch of the TLS handshake process and certificate authentication. An excellent resource for learning more is Bulletproof SSL and TLS by Ivan Ristic´ (Feisty Duck, 2015).

Pop quiz

5. To request client certificate authentication, the server must send which one of the following messages?

1. Certificate
2. ClientHello
3. ServerHello
4. CertificateVerify
5. CertificateRequest

6. How does TLS prevent a captured CertificateVerify message being reused for a different TLS session? (Choose one answer.)
    1. The client's word is their honor.
    2. The CertificateVerify message has a short expiry time.
    3. The CertificateVerify contains a signature over all previous messages in the handshake.
    4. The server and client remember all CertificateVerify messages they've ever seen.

The answers are at the end of the chapter.

11.4.2 Client certificate authentication

To enable TLS client certificate authentication for service clients, you need to configure the server to send a CertificateRequest message as part of the handshake and to validate any certificate that it receives. Most application servers and reverse proxies support configuration options for requesting and validating client certificates, but these vary from product to product. In this section, you'll configure the Nginx ingress controller from chapter 10 to allow client certificates and verify that they are signed by a trusted CA.

To enable client certificate authentication in the Kubernetes ingress controller, you can add annotations to the ingress resource definition in the Natter project. Table 11.1 shows the annotations that can be used.

NOTE All annotation values must be contained in double quotes, even if they are not strings. For example, you must use `nginx.ingress.kubernetes.io/ auth-tls-verify-depth: "1"` to specify a maximum chain length of 1.

Table 11.1 Kubernetes Nginx ingress controller annotations for client cer-

tificate authentication

Annotation	Allowed values	Description
`nginx.ingress.kubernetes.io/auth-tls-verify-client`	`on`, `off`, `optional`, or `optional_no_ca`	Enables or disables client certificate authentication. If `on`, then a client certificate is required. The `optional` value requests a certificate and verifies it if the client presents one. The `optional_no_ca` option prompts the client for a certificate but doesn't verify it.
`nginx.ingress.kubernetes.io/auth-tls-secret`	The name of a Kubernetes secret in the form `namespace/secret-name`	The secret contains the set of trusted CAs to verify the client certificate against.
`nginx.ingress.kubernetes.io/auth-tls-verify-depth`	A positive integer	The maximum number of intermediate CA certificates allowed in the client's certificate chain.
`nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream`	`true` or `false`	If enabled, the client's certificate will be made available in the `ssl-client-cert` HTTP header to servers behind the ingress.

`nginx.ingress.kubernetes.io/auth-tls-error-page`	A URL	If certificate authentication fails, the client will be redirected to this error page.

To create the secret with the trusted CA certificates to verify any client certificates, you create a generic secret passing in a PEM-encoded certificate file. You can include multiple root CA certificates in the file by simply listing them one after the other. For the examples in this chapter, you can use client certificates generated by the `mkcert` utility that you've used since chapter 2. The root CA certificate for `mkcert` is installed into its CAROOT directory, which you can determine by running

```
mkcert -CAROOT
```

which will produce output like the following:

```
/Users/neil/Library/Application Support/mkcert
```

To import this root CA as a Kubernetes secret in the correct format, run the following command:

```
kubectl create secret generic ca-secret -n natter-api \
  --from-file=ca.crt="$(mkcert -CAROOT)/rootCA.pem"
```

Listing 11.5 shows an updated ingress configuration with support for optional client certificate authentication. Client verification is set to optional, so that the API can support service clients using certificate authentication and users performing password authentication. The TLS secret for the trusted CA certificates is set to `natter-api/ ca-secret` to match the secret you just created within the `natter-api` namespace. Finally, you can enable passing the certificate to upstream hosts so that you can extract the client identity from the certificate. Navigate to the kubernetes folder under the Natter API project and update the natter-ingress.yaml file to add the new annotations shown in bold in the following listing.

Listing 11.5 Ingress with optional client certificate authentication

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: api-ingress
  namespace: natter-api
  annotations:
    nginx.ingress.kubernetes.io/upstream-vhost:
        "$service_name.$namespace.svc.cluster.local:$service_port"
    nginx.ingress.kubernetes.io/auth-tls-verify-client: "optional"
    nginx.ingress.kubernetes.io/auth-tls-secret: "natter-api/ca-secret"
    nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"
    nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream:
        "true"
spec:
  tls:
    - hosts:
        - api.natter.local
      secretName: natter-tls
  rules:
    - host: api.natter.local
      http:
        paths:
          - backend:
              serviceName: natter-api-service
              servicePort: 4567
```

❶ Annotations to allow optional client certificate authentication

If you still have Minikube running from chapter 10, you can now update the ingress definition by running:

```
kubectl apply -f kubernetes/natter-ingress.yaml
```

TIP If changes to the ingress controller don't seem to be working, check the output of `kubectl describe ingress -n natter-api` to ensure the annotations are correct. For further troubleshooting tips, check the official documentation at **http://mng.bz/X0rG**.

11.4.3 Verifying client identity

The verification performed by NGINX is limited to checking that the client provided a certificate that was signed by one of the trusted CAs, and that any constraints specified in the certificates themselves are satisfied, such as the expiry time of the certificate. To verify the identity of the client and

apply appropriate permissions, the ingress controller sets several HTTP headers that you can use to check details of the client certificate, shown in table 11.2.

Table 11.2 HTTP headers set by NGINX

Header	Description
`ssl-client-verify`	Indicates whether a client certificate was presented and, if so, whether it was verified. The possible values are `NONE` to indicate no certificate was supplied, `SUCCESS` if a certificate was presented and is valid, or `FAILURE:<reason>` if a certificate was supplied but is invalid or not signed by a trusted CA.
`ssl-client-subject-dn`	The Subject Distinguished Name (DN) field of the certificate if one was supplied.
`ssl-client-issuer-dn`	The Issuer DN, which will match the Subject DN of the CA certificate.
`ssl-client-cert`	If `auth-tls-pass-certificate-to-upstream` is enabled, then this will contain the full client certificate in URL-encoded PEM format.

Figure 11.5 shows the overall process. The NGINX ingress controller terminates the client's TLS connection and verifies the client certificate during the TLS handshake. After the client has authenticated, the ingress controller forwards the request to the backend service and includes the verified client certificate in the `ssl-client-cert` header.

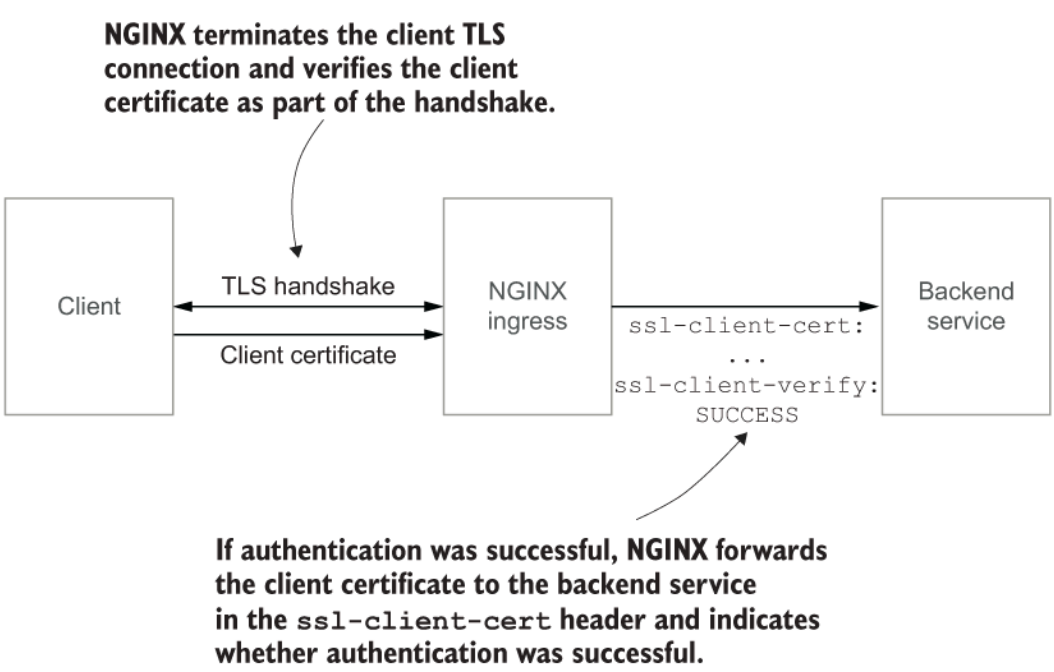

Figure 11.5 To allow client certificate authentication by external clients, you configure the NGINX ingress controller to request and verify the client certificate during the TLS handshake. NGINX then forwards the client certificate in the `ssl-client-cert` HTTP header.

The `mkcert` utility that you'll use for development in this chapter sets the client name that you specify as a Subject Alternative Name (SAN) extension on the certificate rather than using the Subject DN field. Because NGINX doesn't expose SAN values directly in a header, you'll need to parse the full certificate to extract it. Listing 11.5 shows how to parse the header supplied by NGINX into a `java.security.cert.X509Certificate` object using a `CertificateFactory`, from which you can then extract the client identifier from the SAN. Open the UserController.java file and add the new method from listing 11.6. You'll also need to add the following import statements to the top of the file:

```
import java.io.ByteArrayInputStream;
import java.net.URLDecoder;
import java.security.cert.*;
```

Listing 11.6 Parsing a certificate

```
public static X509Certificate decodeCert(String encodedCert) {
    var pem = URLDecoder.decode(encodedCert, UTF_8);
    try (var in = new ByteArrayInputStream(pem.getBytes(UTF_8))) {
        var certFactory = CertificateFactory.getInstance("X.509");
        return (X509Certificate) certFactory.generateCertificate(in);
    } catch (Exception e) {
        throw new RuntimeException(e);
```

```
    }
}
```

❶ Decode the URL-encoding added by NGINX.

❷ Parse the PEM-encoded certificate using a CertificateFactory.

There can be multiple SAN entries in a certificate and each entry can have a different type. Mkcert uses the DNS type, so the code looks for the first DNS SAN entry and returns that as the name. Java returns the SAN entries as a collection of two-element `List` objects, the first of which is the type (as an integer) and the second is the actual value (either a `String` or a byte array, depending on the type). DNS entries have type value 2. If the certificate contains a matching entry, you can set the client ID as the subject attribute on the request, just as you've done when authenticating users. Because the trusted CA issues client certificates, you can instruct the CA not to issue a certificate that clashes with the name of an existing user. Open the UserController.java file again and add the new constant and method definition from the following listing.

Listing 11.7 Parsing a client certificate

```
private static final int DNS_TYPE = 2;
void processClientCertificateAuth(Request request) {
    var pem = request.headers("ssl-client-cert");                    ❶
    var cert = decodeCert(pem);                                      ❶
    try {
        if (cert.getSubjectAlternativeNames() == null) {
            return;
        }
        for (var san : cert.getSubjectAlternativeNames()) {          ❷
            if ((Integer) san.get(0) == DNS_TYPE) {                  ❷
                var subject = (String) san.get(1);
                request.attribute("subject", subject);               ❸
                return;
            }
        }
    } catch (CertificateParsingException e) {
        throw new RuntimeException(e);
    }
}
```

❶ Extract the client certificate from the header and decode it.

❷ Find the first SAN entry with DNS type.

❸ Set the service account identity as the subject of the request.

To allow a service account to authenticate using a client certificate instead of username and password, you can add a case to the `UserController` `authenticate` method that checks if a client certificate was supplied. You should only trust the certificate if the ingress controller could verify it. As mentioned in table 11.2, NGINX sets the header `ssl-client-verify` to the value `SUCCESS` if the certificate was valid and signed by a trusted CA, so you can use this to decide whether to trust the client certificate.

WARNING If a client can set their own `ssl-client-verify` and `ssl-client-cert` headers, they can bypass the certificate authentication. You should test that your ingress controller strips these headers from any incoming requests. If your ingress controller supports using custom header names, you can reduce the risk by adding a random string to them, such as `ssl-client-cert-zOAGY18FHbAAljJV`. This makes it harder for an attacker to guess the correct header names even if the ingress is accidentally misconfigured.

You can now enable client certificate authentication by updating the `authenticate` method to check for a valid client certificate and extract the subject identifier from that instead. Listing 11.8 shows the changes required. Open the UserController.java file again, add the lines highlighted in bold from the listing to the `authenticate` method and save your changes.

Listing 11.8 Enabling client certificate authentication

```
public void authenticate(Request request, Response response) {
    if ("SUCCESS".equals(request.headers("ssl-client-verify"))) {    ❶
        processClientCertificateAuth(request);                       ❶
        return;                                                      ❶
    }
    var credentials = getCredentials(request);                       ❷
    if (credentials == null) return;

    var username = credentials[0];
    var password = credentials[1];

    var hash = database.findOptional(String.class,
            "SELECT pw_hash FROM users WHERE user_id = ?", username);
```

```
    if (hash.isPresent() && SCryptUtil.check(password, hash.get())) {
        request.attribute("subject", username);

        var groups = database.findAll(String.class,
            "SELECT DISTINCT group_id FROM group_members " +
                   "WHERE user_id = ?", username);
        request.attribute("groups", groups);
    }
}
```

❶ If certificate authentication was successful, then use the supplied certificate.

❷ Otherwise, use the existing password-based authentication.

You can now rebuild the Natter API service by running

```
eval $(minikube docker-env)
mvn clean compile jib:dockerBuild
```

in the root directory of the Natter project. Then restart the Natter API and database to pick up the changes,[3] by running:

```
kubectl rollout restart deployment \
    natter-api-deployment natter-database-deployment -n natter-api
```

After the pods have restarted (using `kubectl get pods -n natter-api` to check), you can register a new service user as if it were a regular user account:

```
curl -H 'Content-Type: application/json' \
  -d '{"username":"testservice","password":"password"}' \
  https://api.natter.local/users
```

Mini project

You still need to supply a dummy password to create the service account, and somebody could log in using that password if it's weak. Update the UserController registerUser method (and database schema) to allow the password to be missing, in which case password authentication is dis-

abled. The GitHub repository accompanying the book has a solution in the chapter11-end branch.
You can now use `mkcert` to generate a client certificate for this account, signed by the `mkcert` root CA that you imported as the `ca-secret`. Use the `-client` option to mkcert to generate a client certificate and specify the service account username:

```
mkcert -client testservice
```

This will generate a new certificate for client authentication in the file testservice-client.pem, with the corresponding private key in testservice-client-key.pem. You can now log in using the client certificate to obtain a session token:

```
curl -H 'Content-Type: application/json' -d '{}' \
    --key testservice-client-key.pem \              ❶
    --cert testservice-client.pem \                 ❷
    https://api.natter.local/sessions
```

❶ Use the --key option to specify the private key.

❷ Supply the certificate with --cert.

Because TLS certificate authentication effectively authenticates every request sent in the same TLS session, it can be more efficient for a client to reuse the same TLS session for many HTTP API requests. In this case, you can do without token-based authentication and just use the certificate.

Pop quiz

7. Which one of the following headers is used by the Nginx ingress controller to indicate whether client certificate authentication was successful?
    1. ssl-client-cert
    2. ssl-client-verify
    3. ssl-client-issuer-dn
    4. ssl-client-subject-dn
    5. ssl-client-naughty-or-nice

The answer is at the end of the chapter.

11.4.4 Using a service mesh

Although TLS certificate authentication is very secure, client certificates still must be generated and distributed to clients, and periodically renewed when they expire. If the private key associated with a certificate might be compromised, then you also need to have processes for handling revocation or use short-lived certificates. These are the same problems discussed in chapter 10 for server certificates, which is one of the reasons that you installed a service mesh to automate handling of TLS configuration within the network in section 10.3.2.

To support network authorization policies, most service mesh implementations already implement mutual TLS and distribute both server and client certificates to the service mesh proxies. Whenever an API request is made between a client and a server within the service mesh, that request is transparently upgraded to mutual TLS by the proxies and both ends authenticate to each other with TLS certificates. This raises the possibility of using the service mesh to authenticate service clients to the API itself. For this to work, the service mesh proxy would need to forward the client certificate details from the sidecar proxy to the underlying service as a HTTP header, just like you've configured the ingress controller to do. Istio supports this by default since the 1.1.0 release, using the `X-Forwarded-Client-Cert` header, but Linkerd currently doesn't have this feature.

Unlike NGINX, which uses separate headers for different fields extracted from the client certificate, Istio combines the fields into a single header like the following example:[4]

```
x-forwarded-client-cert: By=http://frontend.lyft.com;Hash=
➥ 468ed33be74eee6556d90c0149c1309e9ba61d6425303443c0748a
➥ 02dd8de688;Subject="CN=Test Client,OU=Lyft,L=San
➥ Francisco,ST=CA,C=US"
```

The fields for a single certificate are separated by semicolons, as in the example. The valid fields are given in table 11.3.

Table 11.3 Istio X-Forwarded-Client-Cert fields

Field	Description
`By`	The URI of the proxy that is forwarding the client details.
`Hash`	A hex-encoded SHA-256 hash of the full client certificate.
`Cert`	The client certificate in URL-encoded PEM format.
`Chain`	The full client certificate chain, in URL-encoded PEM format.
`Subject`	The Subject DN field as a double-quoted string.
`URI`	Any URI-type SAN entries from the client certificate. This field may be repeated if there are multiple entries.
`DNS`	Any DNS-type SAN entries. This field can be repeated if there's more than one matching SAN entry.

The behavior of Istio when setting this header is not configurable and depends on the version of Istio being used. The latest version sets the `By`, `Hash`, `Subject`, `URI`, and `DNS` fields when they are present in the client certificate used by the Istio sidecar proxy for mTLS. Istio's own certificates use a URI SAN entry to identify clients and servers, using a standard called SPIFFE (Secure Production Identity Framework for Everyone), which provides a way to name services in microservices environments. Figure 11.6 shows the components of a SPIFFE identifier, which consists of a trust domain and a path. In Istio, the workload identifier consists of the Kubernetes namespace and service account. SPIFFE allows Kubernetes services to be given stable IDs that can be included in a certificate without having to publish DNS entries for each one; Istio can use its knowledge of Kubernetes metadata to ensure that the SPIFFE ID matches the service a client is connecting to.

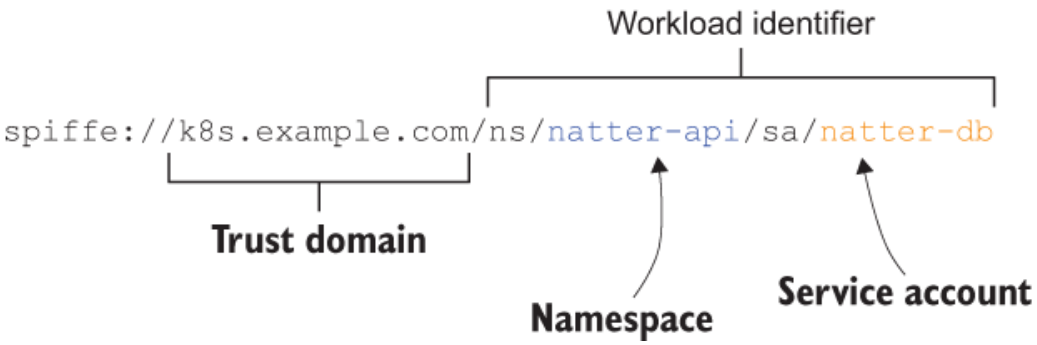

Figure 11.6 A SPIFFE identifier consists of a trust domain and a workload identifier. In Istio, the workload identifier is made up of the namespace and service account of the service.

DEFINITION SPIFFE stands for Secure Production Identity Framework for Everyone and is a standard URI for identifying services and workloads running in a cluster. See **https://spiffe.io** for more information.

NOTE Istio identities are based on Kubernetes service accounts, which are distinct from services. By default, there is only a single service account in each namespace, shared by all pods in that namespace. See **http://mng.bz/yrJG** for instructions on how to create separate service accounts and associate them with your pods.

Istio also has its own version of Kubernetes' ingress controller, in the form of the Istio Gateway. The gateway allows external traffic into the service mesh and can also be configured to process egress traffic leaving the service mesh.[5] The gateway can also be configured to accept TLS client certificates from external clients, in which case it will also set the `X-Forwarded-Client-Cert` header (and strip it from any incoming requests). The gateway sets the same fields as the Istio sidecar proxies, but also sets the Cert field with the full encoded certificate.

Because a request may pass through multiple Istio sidecar proxies as it is being processed, there may be more than one client certificate involved. For example, an external client might make a HTTPS request to the Istio Gateway using a client certificate, and this request then gets forwarded to a microservice over Istio mTLS. In this case, the Istio sidecar proxy's certificate would overwrite the certificate presented by the real client and the microservice would only ever see the identity of the gateway in the `X-Forwarded-Client-Cert` header. To solve this problem, Istio sidecar proxies don't replace the header but instead append the new certificate details to the existing header, separated by a comma. The microservice would then see a header with multiple certificate details in it, as in the following example:

```
X-Forwarded-Client-Cert: By=https://gateway.example.org;
➥ Hash=0d352f0688d3a686e56a72852a217ae461a594ef22e54cb
➥ 551af5ca6d70951bc,By=spiffe://api.natter.local/ns/          (1)
➥ natter-api/sa/natter-api-service;Hash=b26f1f3a5408f7
➥ 61753f3c3136b472f35563e6dc32fefd1ef97d267c43bcfdd1
```

(1) The comma separates the two certificate entries.

The original client certificate presented to the gateway is the first entry in the header, and the certificate presented by the Istio sidecar proxy is the

second. The gateway itself will strip any existing header from incoming requests, so the append behavior is only for internal sidecar proxies. The sidecar proxies also strip the header from new outgoing requests that originate inside the service mesh. These features allow you to use client certificate authentication in Istio without needing to generate or manage your own certificates. Within the service mesh, this is entirely managed by Istio, while external clients can be issued with certificates using an external CA.

11.4.5 Mutual TLS with OAuth2

OAuth2 can also support mTLS for client authentication through a new specification (RFC 8705 **https://tools.ietf.org/html/rfc8705**), which also adds support for certificate-bound access tokens, discussed in section 11.4.6. When used for client authentication, there are two modes that can be used:

- In self-signed certificate authentication, the client registers a certificate with the AS that is signed by its own private key and not by a CA. The client authenticates to the token endpoint with its client certificate and the AS checks that it exactly matches the certificate stored on the client's profile. To allow the certificate to be updated, the AS can retrieve the certificate as the `x5c` claim on a JWK from a HTTPS URL registered for the client.
- In the PKI (public key infrastructure) method, the AS establishes trust in the client's certificate through one or more trusted CA certificates. This allows the client's certificate to be issued and reissued independently without needing to update the AS. The client identity is matched to the certificate either through the Subject DN or SAN fields in the certificate.

Unlike JWT bearer authentication, there is no way to use mTLS to obtain an access token for a service account, but a client can get an access token using the client credentials grant. For example, the following curl command can be used to obtain an access token from an AS that supports mTLS client authentication:

```
curl -d 'grant_type=client_credentials&scope=create_space' \
    -d 'client_id=test' \                                    (1)
    --cert test-client.pem \                                 (2)
    --key test-client-key.pem \                              (2)
    https://as.example.org/oauth2/access_token
```

❶ Specify the client_id explicitly.

❷ Authenticate using the client certificate and private key.

The `client_id` parameter must be explicitly specified when using mTLS client authentication, so that the AS can determine the valid certificates for that client if using the self-signed method.

Alternatively, the client can use mTLS client authentication in combination with the JWT Bearer grant type of section 11.3.2 to obtain an access token for a service account while authenticating itself using the client certificate, as in the following curl example, which assumes that the JWT assertion has already been created and signed in the variable `$JWT`:

```
curl \
  -d 'grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer' \     ❶
  -d "assertion=$JWT&scope=a+b+c&client_id=test" \                  ❶
  --cert test-client.pem \                                          ❷
  --key test-client-key.pem \                                       ❷
  https://as.example.org/oauth2/access_token
```

❶ Authorize using a JWT bearer for the service account.

❷ Authenticate the client using mTLS.

The combination of mTLS and JWT bearer authentication is very powerful, as you'll see later in section 11.5.3.

11.4.6 Certificate-bound access tokens

Beyond supporting client authentication, the OAuth2 mTLS specification also describes how the AS can optionally bind an access token the TLS client certificate when it is issued, creating a certificate-bound access token. The access token then can be used to access an API only when the client authenticates to the API using the same client certificate and private key. This makes the access token no longer a simple bearer token because an attacker that steals the token can't use it without the associated private key (which never leaves the client).

DEFINITION A certificate-bound access token can't be used except over a TLS connection that has been authenticated with the same client certificate used when the access token was issued.

Proof-of-possession tokens

Certificate-bound access tokens are an example of proof-of-possession (PoP) tokens, also known as holder-of-key tokens, in which the token can't be used unless the client proves possession of an associated secret key. OAuth 1 supported PoP tokens using HMAC request signing, but the complexity of implementing this correctly was a factor in the feature being dropped in the initial version of OAuth2. Several attempts have been made to revive the idea, but so far, certificate-bound tokens are the only proposal to have become a standard.

Although certificate-bound access tokens are great when you have a working PKI, they can be difficult to deploy in some cases. They work poorly in single-page apps and other web applications. Alternative PoP schemes are being discussed, such as a JWT-based scheme known as DPoP (**https://tools.ietf.org/html/draft-fett-oauth-dpop-03**), but these are yet to achieve widespread adoption.

To obtain a certificate-bound access token, the client simply authenticates to the token endpoint with the client certificate when obtaining an access token. If the AS supports the feature, then it will associate a SHA-256 hash of the client certificate with the access token. The API receiving an access token from a client can check for a certificate binding in one of two ways:

- If using the token introspection endpoint (section 7.4.1 of chapter 7), the AS will return a new field of the form `"cnf": { "x5t#S256": "...hash..." }` where the hash is the Base64url-encoded certificate hash. The `cnf` claim communicates a confirmation key, and the `x5t#S256` part is the confirmation method being used.
- If the token is a JWT, then the same information will be included in the JWT claims set as a `"cnf"` claim with the same format.

DEFINITION A confirmation key communicates to the API how it can verify a constraint on who can use an access token. The client must confirm that it has access to the corresponding private key using the indicated confirmation method. For certificate-bound access tokens, the confirmation key is a SHA-256 hash of the client certificate and the client confirms possession of the private key by authenticating TLS connections to the API with the same certificate.

Figure 11.7 shows the process by which an API enforces a certificate-bound access token using token introspection. When the client accesses the API, it presents its access token as normal. The API introspects the to-

ken by calling the AS token introspection endpoint (chapter 7), which will return the `cnf` claim along with the other token details. The API can then compare the hash value in this claim to the client certificate associated with the TLS session from the client.

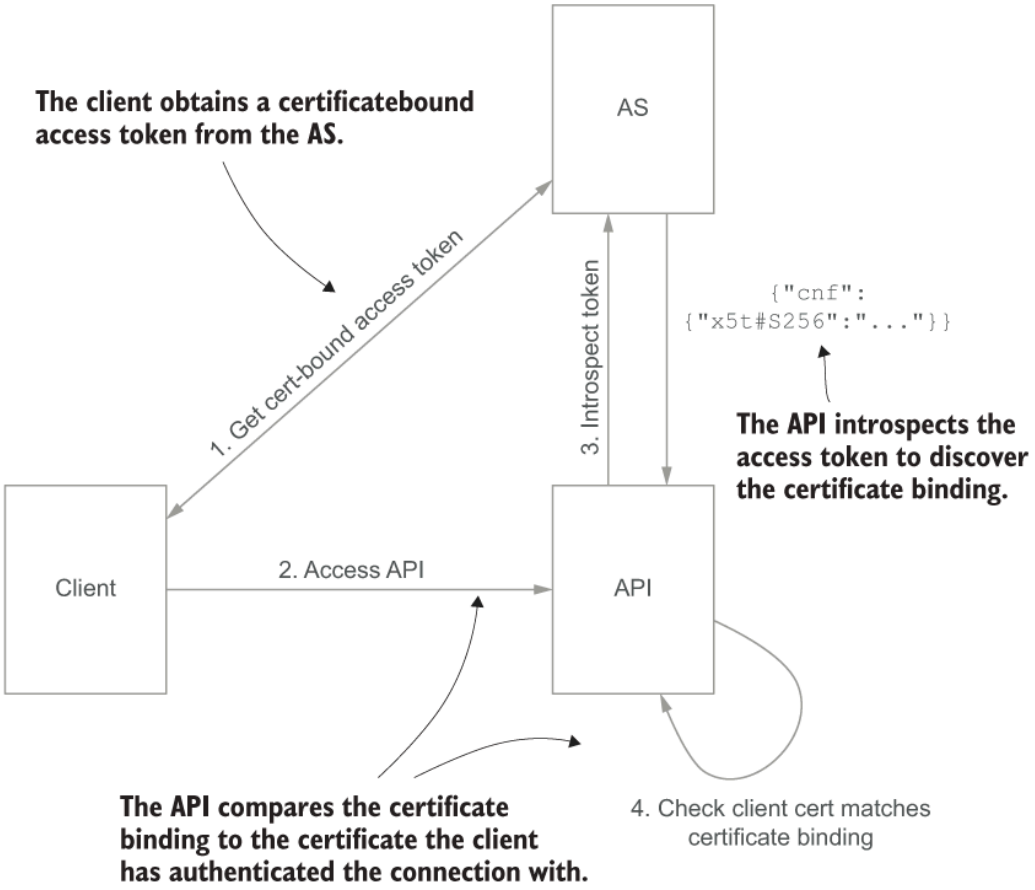

Figure 11.7 When a client obtains a certificate-bound access token and then uses it to access an API, the API can discover the certificate binding using token introspection. The introspection response will contain a `"cnf"` claim containing a hash of the client certificate. The API can then compare the hash to the certificate the client has used to authenticate the TLS connection to the API and reject the request if it is different.

In both cases, the API can check that the client has authenticated with the same certificate by comparing the hash with the client certificate used to authenticate at the TLS layer. Listing 11.9 shows how to calculate the hash of the certificate, known as a thumbprint in the JOSE specifications, using the `java.security.MessageDigest` class that you used in chapter 4. The hash should be calculated over the full binary encoding of the certificate, which is what the `certificate.getEncoded()` method returns. Open the OAuth2TokenStore.java file in your editor and add the `thumbprint` method from the listing.

DEFINITION A certificate thumbprint or fingerprint is a cryptographic hash of the encoded bytes of the certificate.

Listing 11.9 Calculating a certificate thumbprint

```
private byte[] thumbprint(X509Certificate certificate) {
    try {
        var sha256 = MessageDigest.getInstance("SHA-256");      ❶
        return sha256.digest(certificate.getEncoded());         ❷
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

❶ Use a SHA-256 MessageDigest instance.

❷ Hash the bytes of the entire certificate.

To enforce a certificate binding on an access token, you need to check the token introspection response for a `cnf` field containing a confirmation key. The confirmation key is a JSON object whose fields are the confirmation methods and the values are the determined by each method. Loop through the required confirmation methods as shown in listing 11.9 to ensure that they are all satisfied. If any aren't satisfied, or your API doesn't understand any of the confirmation methods, then you should reject the request so that a client can't access your API without all constraints being respected.

TIP The JWT specification for confirmation methods (RFC 7800, **https://tools .ietf.org/html/rfc7800**) requires only a single confirmation method to be specified. For robustness, you should check for other confirmation methods and reject the request if there are any that your API doesn't understand.

Listing 11.9 shows how to enforce a certificate-bound access token constraint by checking for an `x5t#S256` confirmation method. If a match is found, Base64url-decode the confirmation key value to obtain the expected hash of the client certificate. This can then be compared against the hash of the actual certificate the client has used to authenticate to the API. In this example, the API is running behind the Nginx ingress controller, so the certificate is extracted from the `ssl-client-cert` header.

CAUTION Remember to check the `ssl-client-verify` header to ensure the certificate authentication succeeded; otherwise, you shouldn't trust the certificate.

If the client had directly connected to the Java API server, then the certificate is available through a request attribute:

```
var cert = (X509Certificate) request.attributes(
        "javax.servlet.request.X509Certificate");
```

You can reuse the `decodeCert` method from the `UserController` to decode the certificate from the header and then compare the hash from the confirmation key to the certificate thumbprint using the `MessageDigest.isEqual` method. Open the OAuth2TokenStore.java file and update the `processResponse` method to enforce certificate-bound access tokens as shown in the following listing.

Listing 11.10 Verifying a certificate-bound access token

```
private Optional<Token> processResponse(JSONObject response,
        Request originalRequest) {
    var expiry = Instant.ofEpochSecond(response.getLong("exp"));
    var subject = response.getString("sub");

    var confirmationKey = response.optJSONObject("cnf");                1
    if (confirmationKey != null) {
        for (var method : confirmationKey.keySet()) {                   2
            if (!"x5t#S256".equals(method)) {                           3
                throw new RuntimeException(                             3
                        "Unknown confirmation method: " + method);      3
            }                                                           3
            if (!"SUCCESS".equals(                                      4
                   originalRequest.headers("ssl-client-verify"))) {     4
                return Optional.empty();                                4
            }                                                           4
            var expectedHash = Base64url.decode(                        5
                    confirmationKey.getString(method));                 5
            var cert = UserController.decodeCert(                       6
                    originalRequest.headers("ssl-client-cert"));        6
            var certHash = thumbprint(cert);                            6
            if (!MessageDigest.isEqual(expectedHash, certHash)) {       6
                return Optional.empty();                                6
            }                                                           6
        }
    }

    var token = new Token(expiry, subject);
    token.attributes.put("scope", response.getString("scope"));
    token.attributes.put("client_id",
```

```
                response.optString("client_id"));

        return Optional.of(token);
    }
```

❶ Check if a confirmation key is associated with the token.

❷ Loop through the confirmation methods to ensure all are satisfied.

❸ If there are any unrecognized confirmation methods, then reject the request.

❹ Reject the request if no valid certificate is provided.

❺ Extract the expected hash from the confirmation key.

❻ Decode the client certificate and compare the hash, rejecting if they don't match.

An important point to note is that an API can verify a certificate-bound access token purely by comparing the hash values, and doesn't need to validate certificate chains, check basic constraints, or even parse the certificate at all![6] This is because the authority to perform the API operation comes from the access token and the certificate is being used only to prevent that token being stolen and used by a malicious client. This significantly reduces the complexity of supporting client certificate authentication for API developers. Correctly validating an X.509 certificate is difficult and has historically been a source of many vulnerabilities. You can disable CA verification at the ingress controller by using the `optional_no_ca` option discussed in section 11.4.2, because the security of certificate-bound access tokens depends only on the client using the same certificate to access an API that it used when the token was issued, regardless of who issued that certificate.

TIP The client can even use a self-signed certificate that it generates just before calling the token endpoint, eliminating the need for a CA for issuing client certificates.

At the time of writing, only a few AS vendors support certificate-bound access tokens, but it's likely this will increase as the standard has been widely adopted in the financial sector. Appendix A has instructions on installing an evaluation version of ForgeRock Access Management 6.5.2, which supports the standard.

Certificate-bound tokens and public clients

An interesting aspect of the OAuth2 mTLS specification is that a client can request certificate-bound access tokens even if they don't use mTLS for client authentication. In fact, even a public client with no credentials at all can request certificate-bound tokens! This can be very useful for upgrading the security of public clients. For example, a mobile app is a public client because anybody who downloads the app could decompile it and extract any credentials embedded in it. However, many mobile phones now come with secure storage in the hardware of the phone. An app can generate a private key and self-signed certificate in this secure storage when it first starts up and then present this certificate to the AS when it obtains an access token to bind that token to its private key. The APIs that the mobile app then accesses with the token can verify the certificate binding based purely on the hash associated with the token, without the client needing to obtain a CA-signed certificate.

Pop quiz

8. Which of the following checks must an API perform to enforce a certificate-bound access token? Choose all essential checks.
    1. Check the certificate has not expired.
    2. Ensure the certificate has not expired.
    3. Check basic constraints in the certificate.
    4. Check the certificate has not been revoked.
    5. Verify that the certificate was issued by a trusted CA.
    6. Compare the x5t#S256 confirmation key to the SHA-256 of the certificate the client used when connecting.
9. True or False: A client can obtain certificate-bound access tokens only if it also uses the certificate for client authentication.

The answers are at the end of the chapter.

11.5 Managing service credentials

Whether you use client secrets, JWT bearer tokens, or TLS client certificates, the client will need access to some credentials to authenticate to other services or to retrieve an access token to use for service-to-service calls. In this section, you'll learn how to distribute credentials to clients securely. The process of distributing, rotating, and revoking credentials for service clients is known as secrets management. Where the secrets

are cryptographic keys, then it is alternatively known as key management.

DEFINITION Secrets management is the process of creating, distributing, rotating, and revoking credentials needed by services to access other services. Key management refers to secrets management where the secrets are cryptographic keys.

11.5.1 Kubernetes secrets

You've already used Kubernetes' own secrets management mechanism in chapter 10, known simply as secrets. Like other resources in Kubernetes, secrets have a name and live in a namespace, alongside pods and services. Each named secret can have any number of named secret values. For example, you might have a secret for database credentials containing a username and password as separate fields, as shown in listing 11.11. Just like other resources in Kubernetes, they can be created from YAML configuration files. The secret values are Base64-encoded, allowing arbitrary binary data to be included. These values were created using the UNIX echo and Base64 commands:

```
echo -n 'dbuser' | base64
```

TIP Remember to use the `-n` option to the echo command to avoid an extra newline character being added to your secrets.

WARNING Base64 encoding is not encryption. Don't check secrets YAML files directly into a source code repository or other location where they can be easily read.

Listing 11.11 Kubernetes secret example

```
apiVersion: v1
kind: Secret                ❶
metadata:
  name: db-password         ❷
  namespace: natter-api     ❷
type: Opaque
data:
  username: ZGJ1c2Vy        ❸
  password: c2VrcmV0        ❸
```

❶ The kind field indicates this is a secret.

❷ Give the secret a name and a namespace.

❸ The secret has two fields with Base64-encoded values.

You can also define secrets at runtime using `kubectl`. Run the following command to define a secret for the Natter API database username and password:

```
kubectl create secret generic db-password -n natter-api \
    --from-literal=username=natter \
    --from-literal=password=password
```

TIP Kubernetes can also create secrets from files using the `--from-file=username.txt` syntax. This avoids credentials being visible in the history of your terminal shell. The secret will have a field named username.txt with the binary contents of the file.

Kubernetes defines three types of secrets:

- The most general are generic secrets, which are arbitrary sets of key-value pairs, such as the username and password fields in listing 11.11 and in the previous example. Kubernetes performs no special processing of these secrets and just makes them available to your pods.
- A TLS secret consists of a PEM-encoded certificate chain along with a private key. You used a TLS secret in chapter 10 to provide the server certificate and key to the Kubernetes ingress controller. Use `kubectl create secret tls` to create a TLS secret.
- A Docker registry secret is used to give Kubernetes credentials to access a private Docker container registry. You'd use this if your organization stores all images in a private registry rather than pushing them to a public registry like Docker Hub. Use `kubectl create secret docker-registry`.

For your own application-specific secrets, you should use the generic secret type.

Once you've defined a secret, you can make it available to your pods in one of two ways:

- As files mounted in the filesystem inside your pods. For example, if you mounted the secret defined in listing 11.11 under the path `/etc/secrets/db`, then you would end up with two files inside your pod: `/etc/secrets/db/username` and `/etc/secrets/db/password`. Your application can then read these files to get the secret values. The contents of the files will be the raw secret values, not the Base64-encoded ones stored in the YAML.
- As environment variables that are passed to your container processes when they first run. In Java you can then access these through the `System.getenv(String name)` method call.

TIP File-based secrets should be preferred over environment variables. It's easy to read the environment of a running process using `kubectl describe pod`, and you can't use environment variables for binary data such as keys. File-based secrets are also updated when the secret changes, while environment variables can only be changed by restarting the pod.

Listing 11.12 shows how to expose the Natter database username and password to the pods in the Natter API deployment by updating the natter-api-deployment.yaml file. A secret volume is defined in the `volumes` section of the pod spec, referencing the named secret to be exposed. In a `volumeMounts` section for the individual container, you can then mount the secret volume on a specific path in the filesystem. The new lines are highlighted in bold.

Listing 11.12 Exposing a secret to a pod

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: natter-api-deployment
  namespace: natter-api
spec:
  selector:
    matchLabels:
      app: natter-api
  replicas: 1
  template:
    metadata:
      labels:
        app: natter-api
    spec:
      securityContext:
        runAsNonRoot: true
      containers:
```

```
        - name: natter-api
          image: apisecurityinaction/natter-api:latest
          imagePullPolicy: Never
          volumeMounts:
            - name: db-password                          (1)
              mountPath: "/etc/secrets/database"         (2)
              readOnly: true
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop:
                - all
          ports:
            - containerPort: 4567
      volumes:
        - name: db-password                              (3)
          secret:
            secretName: db-password                      (4)
```

(1) The volumeMount name must match the volume name.

(2) Specify a mount path inside the container.

(3) The volumeMount name must match the volume name.

(4) Provide the name of the secret to expose.

You can now update the `Main` class to load the database username and password from these secret files rather than hard coding them. Listing 11.13 shows the updated code in the `main` method for initializing the database password from the mounted secret files. You'll need to import `java.nio.file.*` at the top of the file. Open the Main .java file and update the method according to the listing. The new lines are highlighted in bold.

Listing 11.13 Loading Kubernetes secrets

```
var secretsPath = Paths.get("/etc/secrets/database");                    (1)
var dbUsername = Files.readString(secretsPath.resolve("username"));      (1)
var dbPassword = Files.readString(secretsPath.resolve("password"));      (1)

var jdbcUrl = "jdbc:h2:tcp://natter-database-service:9092/mem:natter";
var datasource = JdbcConnectionPool.create(
```

```
    jdbcUrl, dbUsername, dbPassword);
createTables(datasource.getConnection());
```

❶ Load secrets as files from the filesystem.

❷ Use the secret values to initialize the JDBC connection.

You can rebuild the Docker image by running[7]

```
mvn clean compile jib:dockerBuild
```

then reload the deployment configuration to ensure the secret is mounted:

```
kubectl apply -f kubernetes/natter-api-deployment.yaml
```

Finally, you can restart Minikube to pick up the latest changes:

```
minikube stop && minikube start
```

Use `kubectl get pods -n natter-api --watch` to verify that all pods start up correctly after the changes.

Managing Kubernetes secrets

Although you can treat Kubernetes secrets like other configuration and store them in your version control system, this is not a wise thing to do for several reasons:

- Credentials should be kept secret and distributed to as few people as possible. Storing secrets in a source code repository makes them available to all developers with access to that repository. Although encryption can help, it is easy to get wrong, especially with complex command-line tools such as GPG.
- Secrets should be different in each environment that the service is deployed to; the database password should be different in a development environment compared to your test or production environments. This is the opposite requirement to source code, which should be identical (or close to it) between environments.

- There is almost no value in being able to view the history of secrets. Although you may want to revert the most recent change to a credential if it causes an outage, nobody ever needs to revert to the database password from two years ago. If a mistake is made in the encryption of a secret that is hard to change, such as an API key for a third-party service, it's difficult to completely delete the exposed value from a distributed version control system.

A better solution is to either manually manage secrets from the command line, or else use a templating system to generate secrets specific to each environment. Kubernetes supports a templating system called Kustomize, which can generate per-environment secrets based on templates. This allows the template to be checked into version control, but the actual secrets are added during a separate deployment step. See **http://mng.bz/Mov7** for more details.

SECURITY OF KUBERNETES SECRETS

Although Kubernetes secrets are easy to use and provide a level of separation between sensitive credentials and other source code and configuration data, they have some drawbacks from a security perspective:

- Secrets are stored inside an internal database in Kubernetes, known as etcd. By default, etcd is not encrypted, so anyone who gains access to the data storage can read the values of all secrets. You can enable encryption by following the instructions in **http://mng.bz/awZz**.

WARNING The official Kubernetes documentation lists `aescbc` as the strongest encryption method supported. This is an unauthenticated encryption mode and potentially vulnerable to padding oracle attacks as you'll recall from chapter 6. You should use the kms encryption option if you can, because all modes other than kms store the encryption key alongside the encrypted data, providing only limited security. This was one of the findings of the Kubernetes security audit conducted in 2019 (**https://github .com/trailofbits/audit-kubernetes**).

- Anybody with the ability to create a pod in a namespace can use that to read the contents of any secrets defined in that namespace. System administrators with root access to nodes can retrieve all secrets from the Kubernetes API.

- Secrets on disk may be vulnerable to exposure through path traversal or file exposure vulnerabilities. For example, Ruby on Rails had a recent vulnerability in its template system that allowed a remote attacker to view the contents of any file by sending specially-crafted HTTP headers (**https://nvd.nist.gov/vuln/detail/ CVE-2019-5418**).

DEFINITION A file exposure vulnerability occurs when an attacker can trick a server into revealing the contents of files on disk that should not be accessible externally. A path traversal vulnerability occurs when an attacker can send a URL to a webserver that causes it to serve a file that was intended to be private. For example, an attacker might ask for the file /public/../../../etc/secrets/db-password. Such vulnerabilities can reveal Kubernetes secrets to attackers.

11.5.2 Key and secret management services

An alternative to Kubernetes secrets is to use a dedicated service to provide credentials to your application. Secrets management services store credentials in an encrypted database and make the available to services over HTTPS or a similar secure protocol. Typically, the client needs an initial credential to access the service, such as an API key or client certificate, which can be made available via Kubernetes secrets or a similar mechanism. All other secrets are then retrieved from the secrets management service. Although this may sound no more secure than using Kubernetes secrets directly, it has several advantages:

- The storage of the secrets is encrypted by default, providing better protection of secret data at rest.
- The secret management service can automatically generate and update secrets regularly. For example, Hashicorp Vault (**https://www.vaultproject.io**) can automatically create short-lived database users on the fly, providing a temporary username and password. After a configurable period, Vault will delete the account again. This can be useful to allow daily administration tasks to run without leaving a highly privileged account enabled at all times.
- Fine-grained access controls can be applied, ensuring that services only have access to the credentials they need.
- All access to secrets can be logged, leaving an audit trail. This can help to establish what happened after a breach, and automated systems can analyze these logs and alert if unusual access requests are noticed.

When the credentials being accessed are cryptographic keys, a Key Management Service (KMS) can be used. A KMS, such as those provided

by the main cloud providers, securely stores cryptographic key material. Rather than exposing that key material directly, a client of a KMS sends cryptographic operations to the KMS; for example, requesting that a message is signed with a given key. This ensures that sensitive keys are never directly exposed, and allows a security team to centralize cryptographic services, ensuring that all applications use approved algorithms.

DEFINITION A Key Management Service (KMS) stores keys on behalf of applications. Clients send requests to perform cryptographic operations to the KMS rather than asking for the key material itself. This ensures that sensitive keys never leave the KMS.

To reduce the overhead of calling a KMS to encrypt or decrypt large volumes of data, a technique known as envelope encryption can be used. The application generates a random AES key and uses that to encrypt the data locally. The local AES key is known as a data encryption key (DEK). The DEK is then itself encrypted using the KMS. The encrypted DEK can then be safely stored or transmitted alongside the encrypted data. To decrypt, the recipient first decrypts the DEK using the KMS and then uses the DEK to decrypt the rest of the data.

DEFINITION In envelope encryption, an application encrypts data with a local data encryption key (DEK). The DEK is then encrypted (or wrapped) with a key encryption key (KEK) stored in a KMS or other secure service. The KEK itself might be encrypted with another KEK creating a key hierarchy.

For both secrets management and KMS, the client usually interacts with the service using a REST API. Currently, there is no common standard API supported by all providers. Some cloud providers allow access to a KMS using the standard PKCS#11 API used by hardware security modules. You can access a PKCS#11 API in Java through the Java Cryptography Architecture, as if it was a local keystore, as shown in listing 11.14. (This listing is just to show the API; you don't need to type it in.) Java exposes a PKCS#11 device, including a remote one such as a KMS, as a `KeyStore` object with the type `"PKCS11"`.[8] You can load the keystore by calling the `load()` method, providing a null `InputStream` argument (because there is no local keystore file to open) and passing the KMS password or other credential as the second argument. After the PKCS#11 keystore has been loaded, you can then load keys and use them to initialize `Signature` and `Cipher` objects just like any other local key. The difference is that the `Key` object returned by the PKCS#11 keystore has no key material inside

it. Instead, Java will automatically forward cryptographic operations to the KMS via the PKCS#11 API.

TIP Java's built-in PKCS#11 cryptographic provider only supports a few algorithms, many of which are old and no longer recommended. A KMS vendor may offer their own provider with support for more algorithms.

Listing 11.14 Accessing a KMS through PKCS#11

```
var keyStore = KeyStore.getInstance("PKCS11");                    (1)
var keyStorePassword = "changeit".toCharArray();                  (1)
keyStore.load(null, keyStorePassword);                            (1)

var signingKey = (PrivateKey) keyStore.getKey("rsa-key",          (2)
        keyStorePassword);                                        (2)

var signature = Signature.getInstance("SHA256WithRSA");           (3)
signature.initSign(signingKey);                                   (3)
signature.update("Hello!".getBytes(UTF_8));                       (3)
var sig = signature.sign();                                       (3)
```

(1) Load the PKCS11 keystore with the correct password.

(2) Retrieve a key object from the keystore.

(3) Use the key to sign a message.

PKCS#11 and hardware security modules

PKCS#11, or Public Key Cryptography Standard 11, defines a standard API for interacting with hardware security modules (HSMs). An HSM is a hardware device dedicated to secure storage of cryptographic keys. HSMs range in size from tiny USB keys that support just a few keys, to rack-mounted network HSMs that can handle thousands of requests per second (and cost tens of thousands of dollars). Just like a KMS, the key material can't normally be accessed directly by clients and they instead send cryptographic requests to the device after logging in. The API defined by PKCS#11, known as Cryptoki, provides operations in the C programming language for logging into the HSM, listing available keys, and performing cryptographic operations.

Unlike a purely software KMS, an HSM is designed to offer protection against an attacker with physical access to the device. For example, the

circuitry of the HSM may be encased in tough resin with embedded sensors that can detect anybody trying to tamper with the device, in which case the secure memory is wiped to prevent compromise. The US and Canadian governments certify the physical security of HSMs under the FIPS 140-2 certification program, which offers four levels of security: level 1 certified devices offer only basic protection of key material, while level 4 offers protection against a wide range of physical and environmental threats. On the other hand, FIPS 140-2 offers very little validation of the quality of implementation of the algorithms running on the device, and some HSMs have been found to have serious software security flaws. Some cloud KMS providers can be configured to use FIPS 140-2 certified HSMs for storage of keys, usually at an increased cost. However, most such services are already running in physically secured data centers, so the additional physical protection is usually unnecessary.

A KMS can be used to encrypt credentials that are then distributed to services using Kubernetes secrets. This provides better protection than the default Kubernetes configuration and enables the KMS to be used to protect secrets that aren't cryptographic keys. For example, a database connection password can be encrypted with the KMS and then the encrypted password is distributed to services as a Kubernetes secret. The application can then use the KMS to decrypt the password after loading it from the disk.

Pop quiz

10. Which of the following are ways that a Kubernetes secret can be exposed to pods?
    1. As files
    2. As sockets
    3. As named pipes
    4. As environment variables
    5. As shared memory buffers
11. What is the name of the standard that defines an API for talking to hardware security modules?
    1. PKCS#1
    2. PKCS#7
    3. PKCE
    4. PKCS#11
    5. PKCS#12

The answers are at the end of the chapter.

11.5.3 Avoiding long-lived secrets on disk

Although a KMS or secrets manager can be used to protect secrets against theft, the service will need an initial credential to access the KMS itself. While cloud KMS providers often supply an SDK that transparently handles this for you, in many cases the SDK is just reading its credentials from a file on the filesystem or from another source in the environment that the SDK is running in. There is therefore still a risk that an attacker could compromise these credentials and then use the KMS to decrypt the other secrets.

TIP You can often restrict a KMS to only allow your keys to be used from clients connecting from a virtual private cloud (VPC) that you control. This makes it harder for an attacker to use compromised credentials because they can't directly connect to the KMS over the internet.

A solution to this problem is to use short-lived tokens to grant access to the KMS or secrets manager. Rather than deploying a username and password or other static credential using Kubernetes secrets, you can instead generate a temporary credential with a short expiry time. The application uses this credential to access the KMS or secrets manager at startup and decrypt the other secrets it needs to operate. If an attacker later compromises the initial token, it will have expired and can't be used. For example, Hashicorp Vault (**https://vaultproject.io**) supports generating tokens with a limited expiry time which a client can then use to retrieve other secrets from the vault.

CAUTION The techniques in this section are significantly more complex than other solutions. You should carefully weigh the increased security against your threat model before adopting these approaches.

If you primarily use OAuth2 for access to other services, you can deploy a short-lived JWT that the service can use to obtain access tokens using the JWT bearer grant described in section 11.3. Rather than giving clients direct access to the private key to create their own JWTs, a separate controller process generates JWTs on their behalf and distributes these short-lived bearer tokens to the pods that need them. The client then uses the JWT bearer grant type to exchange the JWT for a longer-lived access token (and optionally a refresh token too). In this way, the JWT bearer grant type can be used to enforce a separation of duties that allows the private key to be kept securely away from pods that service user requests. When combined with certificate-bound access tokens of section 11.4.6, this pat-

tern can result in significantly increased security for OAuth2-based microservices.

The main problem with short-lived credentials is that Kubernetes is designed for highly dynamic environments in which pods come and go, and new service instances can be created to respond to increased load. The solution is to have a controller process register with the Kubernetes API server and watch for new pods being created. The controller process can then create a new temporary credential, such as a fresh signed JWT, and deploy it to the pod before it starts up. The controller process has access to long-lived credentials but can be deployed in a separate namespace with strict network policies to reduce the risk of it being compromised, as shown in figure 11.8.

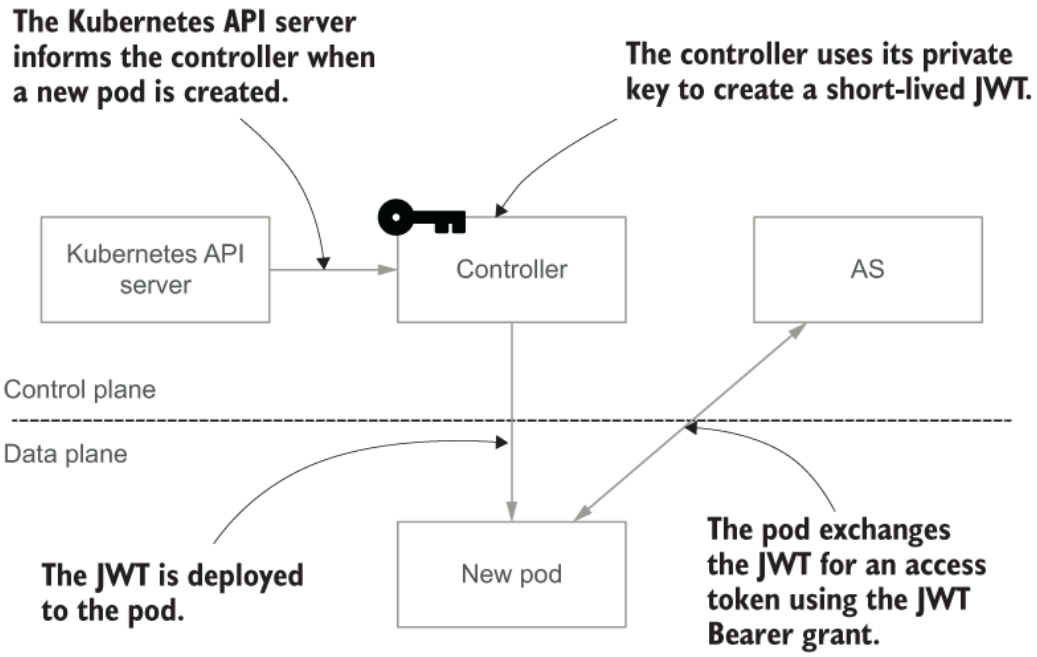

Figure 11.8 A controller process running in a separate control plane namespace can register with the Kubernetes API to watch for new pods. When a new pod is created, the controller uses its private key to sign a short-lived JWT, which it then deploys to the new pod. The pod can then exchange the JWT for an access token or other long-lived credentials.

A production-quality implementation of this pattern is available, again for Hashicorp Vault, as the Boostport Kubernetes-Vault integration project (**https://github.com/ Boostport/kubernetes-vault**). This controller can inject unique secrets into each pod, allowing the pod to connect to Vault to retrieve its other secrets. Because the initial secrets are unique to a pod, they can be restricted to allow only a single use, after which the token becomes invalid. This ensures that the credential is valid for the shortest possible time. If an attacker somehow managed to compromise the token before the pod used it, then the pod will noisily fail to start up

when it fails to connect to Vault, providing a signal to security teams that something unusual has occurred.

11.5.4 Key derivation

A complementary approach to secure distribution of secrets is to reduce the number of secrets your application needs in the first place. One way to achieve this is to derive cryptographic keys for different purposes from a single master key, using a key derivation function (KDF). A KDF takes the master key and a context argument, which is typically a string, and returns one or more new keys as shown in figure 11.9. A different context argument results in completely different keys and each key is indistinguishable from a completely random key to somebody who doesn't know the master key, making them suitable as strong cryptographic keys.

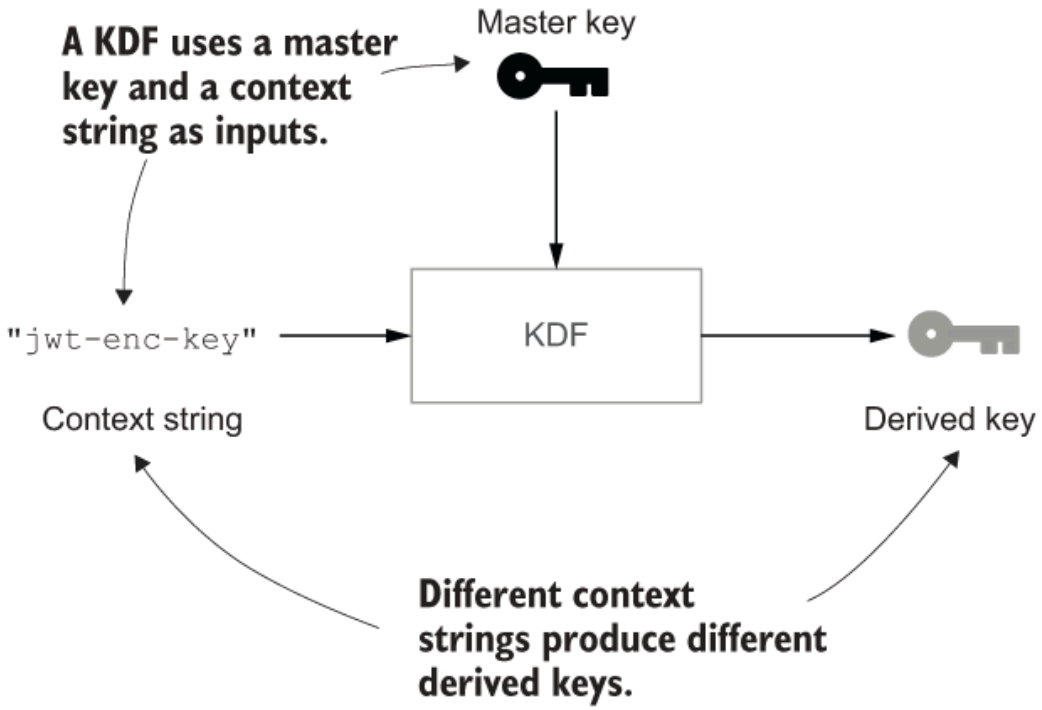

Figure 11.9 A key derivation function (KDF) takes a master key and context string as inputs and produces derived keys as outputs. You can derive an almost unlimited number of strong keys from a single high-entropy master key.

If you recall from chapter 9, macaroons work by treating the HMAC tag of an existing token as a key when adding a new caveat. This works because HMAC is a secure pseudorandom function, which means that its outputs appear completely random if you don't know the key. This is exactly what we need to build a KDF, and in fact HMAC is used as the basis for a widely used KDF called HKDF (HMAC-based KDF, **https://tools.ietf.org/html/rfc5869**). HKDF consists of two related functions:

- HKDF-Extract takes as input a high-entropy input that may not be suitable for direct use as a cryptographic key and returns a HKDF master key. This function is useful in some cryptographic protocols but can be skipped if you already have a valid HMAC key. You won't use HKDF-Extract in this book.
- HKDF-Expand takes the master key and a context and produces an output key of any requested size.

DEFINITION HKDF is a HMAC-based KDF based on an extract-and-expand method. The expand function can be used on its own to generate keys from a master HMAC key.

Listing 11.15 shows an implementation of HKDF-Expand using HMAC-SHA-256. To generate the required amount of output key material, HKDF-Expand performs a loop. Each iteration of the loop runs HMAC to produce a block of output key material with the following inputs:

1. The HMAC tag from the last time through the loop unless this is the first loop.
2. The context string.
3. A block counter byte, which starts at 1 and is incremented each time.

With HMAC-SHA-256 each iteration of the loop generates 32 bytes of output key material, so you'll typically only need one or two loops to generate a big enough key for most algorithms. Because the block counter is a single byte, and cannot be 0, you can only loop a maximum of 255 times, which gives a maximum key size of 8,160 bytes. Finally, the output key material is converted into a `Key` object using the `javax.crypto.spec.SecretKeySpec` class. Create a new file named HKDF.java in the src/main/ java/com/manning/apisecurityinaction folder with the contents of the file.

TIP If the master key lives in a HSM or KMS then it is much more efficient to combine the inputs into a single byte array rather than making multiple calls to the `update()` method.

Listing 11.15 HKDF-Expand

```
package com.manning.apisecurityinaction;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.security.*;
```

```
import static java.nio.charset.StandardCharsets.UTF_8;
import static java.util.Objects.checkIndex;

public class HKDF {
    public static Key expand(Key masterKey, String context,
                             int outputKeySize, String algorithm)
            throws GeneralSecurityException {
        checkIndex(outputKeySize, 255*32);                    (1)

        var hmac = Mac.getInstance("HmacSHA256");             (2)
        hmac.init(masterKey);                                 (2)

        var output = new byte[outputKeySize];
        var block = new byte[0];
        for (int i = 0; i < outputKeySize; i += 32) {         (3)
            hmac.update(block);                               (4)
            hmac.update(context.getBytes(UTF_8));             (5)
            hmac.update((byte) ((i / 32) + 1));               (5)
            block = hmac.doFinal();                           (6)
            System.arraycopy(block, 0, output, i,             (6)
                    Math.min(outputKeySize - i, 32));         (6)
        }

        return new SecretKeySpec(output, algorithm);
    }
}
```

(1) Ensure the caller didn't ask for too much key material.

(2) Initialize the Mac with the master key.

(3) Loop until the requested output size has been generated.

(4) Include the output block of the last loop in the new HMAC.

(5) Include the context string and the current block counter.

(6) Copy the new HMAC tag to the next block of output.

You can now use this to generate as many keys as you want from an initial HMAC key. For example, you can open the Main.java file and replace the code that loads the AES encryption key from the keystore with the following code that derives it from the HMAC key instead as shown in the bold line here:

```
var macKey = keystore.getKey("hmac-key", "changeit".toCharArray());
var encKey = HKDF.expand(macKey, "token-encryption-key",
        32, "AES");
```

WARNING A cryptographic key should be used for a single purpose. If you use a HMAC key for key derivation, you should not use it to also sign messages. You can use HKDF to derive a second HMAC key to use for signing.

You can generate almost any kind of symmetric key using this method, making sure to use a distinct context string for each different key. Key pairs for public key cryptography generally can't be generated in this way, as the keys are required to have some mathematical structure that is not present in a derived random key. However, the Salty Coffee library used in chapter 6 contains methods for generating key pairs for public key encryption and for digital signatures from a 32-byte seed, which can be used as follows:

```
var seed = HKDF.expand(macKey, "nacl-signing-key-seed",      (1)
        32, "NaCl");                                         (1)
var keyPair = Crypto.seedSigningKeyPair(seed.getEncoded());  (2)
```

(1) Use HKDF to generate a seed.

(2) Derive a signing keypair from the seed.

CAUTION The algorithms used by Salty Coffee, X25519 and Ed25519, are designed to safely allow this. The same is not true of other algorithms.

Although generating a handful of keys from a master key may not seem like much of a savings, the real value comes from the ability to generate keys programmatically that are the same on all servers. For example, you can include the current date in the context string and automatically derive a fresh encryption key each day without needing to distribute a new key to every server. If you include the context string in the encrypted data, for example as the `kid` header in an encrypted JWT, then you can quickly re-derive the same key whenever you need without storing previous keys.

Facebook CATs

As you might expect, Facebook needs to run many services in production with numerous clients connecting to each service. At the huge scale they are running at, public key cryptography is deemed too expensive, but they still want to use strong authentication between clients and services. Every request and response between a client and a service is authenticated with HMAC using a key that is unique to that client-service pair. These signed HMAC tokens are known as Crypto Auth Tokens, or CATs, and are a bit like signed JWTs.

To avoid storing, distributing, and managing thousands of keys, Facebook uses key derivation heavily. A central key distribution service stores a master key. Clients and services authenticate to the key distribution service to get keys based on their identity. The key for a service with the name "AuthService" is calculated using `KDF(masterKey, "AuthService")`, while the key for a client named "Test" to talk to the auth service is calculated as `KDF(KDF(masterKey, "AuthService"), "Test")`. This allows Facebook to quickly generate an almost unlimited number of client and service keys from the single master key. You can read more about Facebook's CATs at **https://eprint.iacr.org/2018/413**.

Pop quiz

12. Which HKDF function is used to derive keys from a HMAC master key?
    1. HKDF-Extract
    2. HKDF-Expand
    3. HKDF-Extrude
    4. HKDF-Exhume
    5. HKDF-Exfiltrate

The answer is at the end of the chapter.

11.6 Service API calls in response to user requests

When a service makes an API call to another service in response to a user request, but uses its own credentials rather than the user's, there is an opportunity for confused deputy attacks like those discussed in chapter 9. Because service credentials are often more privileged than normal users, an attacker may be able to trick the service to performing malicious actions on their behalf.

Kubernetes critical API server vulnerability

In 2018, the Kubernetes project itself reported a critical vulnerability allowing this kind of confused deputy attack (https://rancher.com/blog/2018/2018-12-04-k8s-cve/). In the attack, a user made an initial request to the Kubernetes API server, which authenticated the request and applied access control checks. It then made its own connection to a backend service to fulfill the request. This API request to the backend service used highly privileged Kubernetes service account credentials, providing administrator-level access to the entire cluster. The attacker could trick Kubernetes into leaving the connection open, allowing the attacker to send their own commands to the backend service using the service account. The default configuration permitted even unauthenticated users to exploit the vulnerability to execute any commands on backend servers. To make matters worse, Kubernetes audit logging filtered out all activity from system accounts so there was no trace that an attack had taken place.

You can avoid confused deputy attacks in service-to-service calls that are carried out in response to user requests by ensuring that access control decisions made in backend services include the context of the original request. The simplest solution is for frontend services to pass along the username or other identifier of the user that made the original request. The backend service can then make an access control decision based on the identity of this user rather than solely on the identity of the calling service. Service-to-service authentication is used to establish that the request comes from a trusted source (the frontend service), and permission to perform the action is determined based on the identity of the user indicated in the request.

TIP As you'll recall from chapter 9, capability-based security can be used to systematically eliminate confused deputy attacks. If the authority to perform an operation is encapsulated as a capability, this can be passed from the user to all backend services involved in implementing that operation. The authority to perform an operation comes from the capability rather than the identity of the service making a request, so an attacker can't request an operation they don't have a capability for.

11.6.1 The phantom token pattern

Although passing the username of the original user is simple and can avoid confused deputy attacks, a compromised frontend service can easily impersonate any user by simply including their username in the request. An alternative would be to pass down the token originally presented by the user, such as an OAuth2 access token or JWT. This allows

backend services to check that the token is valid, but it still has some drawbacks:

- If the access token requires introspection to check validity, then a network call to the AS has to be performed at each microservice that is involved in processing a request. This can add a lot of overhead and additional delays.
- On the other hand, backend microservices have no way of knowing if a long-lived signed token such as a JWT has been revoked without performing an introspection request.
- A compromised microservice can take the user's token and use it to access other services, effectively impersonating the user. If service calls cross trust boundaries, such as when calls are made to external services, the risk of exposing the user's token increases.

The first two points can be addressed through an OAuth2 deployment pattern implemented by some API gateways, shown in figure 11.10. In this pattern, users present long-lived access tokens to the API gateway which performs a token introspection call to the AS to ensure the token is valid and hasn't been revoked. The API gateway then takes the contents of the introspection response, perhaps augmented with additional information about the user (such as roles or group memberships) and produces a short-lived JWT signed with a key trusted by all the microservices behind the gateway. The gateway then forwards the request to the target microservices, replacing the original access token with this short-lived JWT. This is sometimes referred to as the phantom token pattern. If a public key signature is used for the JWT then microservices can validate the token but not create their own.

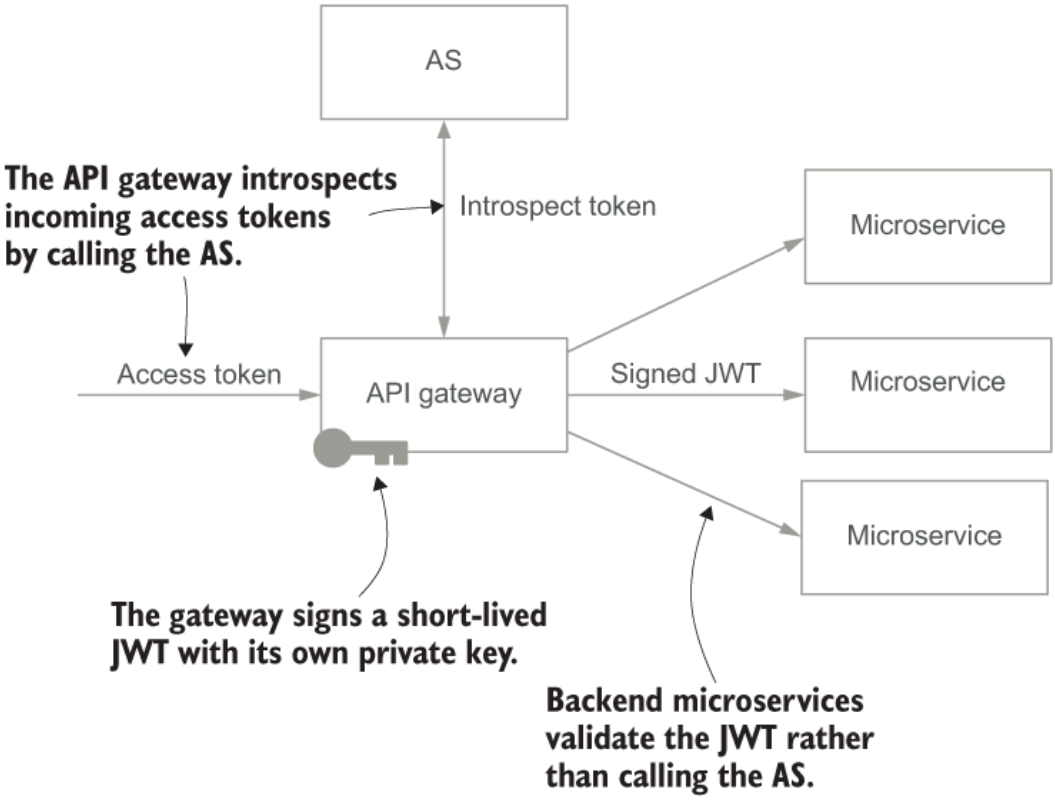

Figure 11.10 In the phantom token pattern, an API gateway introspects access tokens arriving from external clients. It then replaces the access token with a short-lived signed JWT containing the same information. Microservices can then examine the JWT without having to call the AS to introspect themselves.

DEFINITION In the phantom token pattern, a long-lived opaque access token is validated and then replaced with a short-lived signed JWT at an API gateway. Microservices behind the gateway can examine the JWT without needing to perform an expensive introspection request.

The advantage of the phantom token pattern is that microservices behind the gateway don't need to perform token introspection calls themselves. Because the JWT is short-lived, typically with an expiry time measured in seconds or minutes at most, there is no need for those microservices to check for revocation. The API gateway can examine the request and reduce the scope and audience of the JWT, limiting the damage that would be done if any backend microservice has been compromised. In principle, if the gateway needs to call five different microservices to satisfy a request, it can create five separate JWTs with scope and audience appropriate to each request. This ensures the principle of least privilege is respected and reduces the risk if any one of those services is compromised, but is rarely done due to the extra overhead of creating new JWTs, especially if public key signatures are used.

TIP A network roundtrip within the same datacenter takes a minimum of 0.5ms plus the processing time required by the AS (which may involve database network requests). Verifying a public key signature varies from about 1/10th of this time (RSA-2048 using OpenSSL) to roughly 10 times as long (ECDSA P-521 using Java's SunEC provider). Verifying a signature also generally requires more CPU power than making a network call, which may impact costs.

The phantom token pattern is a neat balance of the benefits and costs of opaque access tokens compared to self-contained token formats like JWTs. Self-contained tokens are scalable and avoid extra network roundtrips, but are hard to revoke, while the opposite is true of opaque tokens.

PRINCIPLE Prefer using opaque access tokens and token introspection when tokens cross trust boundaries to ensure timely revocation. Use self-contained short-lived tokens for service calls within a trust boundary, such as between microservices.

11.6.2 OAuth2 token exchange

The token exchange extension of OAuth2 (**https://www.rfc-editor.org/rfc/rfc8693.html**) provides a standard way for an API gateway or other client to exchange an access token for a JWT or other security token. As well as allowing the client to request a new token, the AS may also add an `act` claim to the resulting token that indicates that the service client is acting on behalf of the user that is identified as the subject of the token. A backend service can then identify both the service client and the user that initiated the request originally from a single access token.

DEFINITION Token exchange should primarily be used for delegation semantics, in which one party acts on behalf of another but both are clearly identified. It can also be used for impersonation, in which the backend service is unable to tell that another party is impersonating the user. You should prefer delegation whenever possible because impersonation leads to misleading audit logs and loss of accountability.

To request a token exchange, the client makes a HTTP POST request to the AS's token endpoint, just as for other authorization grants. The `grant_type` parameter is set to `urn:ietf:params:oauth:grant-type:token-exchange`, and the client passes a token representing the user's initial authority as the `subject_token` parameter, with a `subject_token_type` parameter describing the type of token (token ex-

change allows a variety of tokens to be used, not just access tokens). The client authenticates to the token endpoint using its own credentials and can provide several optional parameters shown in table 11.4. The AS will make an authorization decision based on the supplied information and the identity of the subject and the client and then either return a new access token or reject the request.

TIP Although token exchange is primarily intended for service clients, the `actor_token` parameter can reference another user. For example, you can use token exchange to allow administrators to access parts of other users' accounts without giving them the user's password. While this can be done, it has obvious privacy implications for your users.

Table 11.4 Token exchange optional parameters

Parameter	Description
`resource`	The URI of the service that the client intends to access on the user's behalf.
`audience`	The intended audience of the token. This is an alternative to the `resource` parameter where the identifier of the target service is not a URI.
`scope`	The desired scope of the new access token.
`requested_token_type`	The type of token the client wants to receive.
`actor_token`	A token that identifies the party that is acting on behalf of the user. If not specified, the identity of the client will be used.
`actor_token_type`	The type of the `actor_token` parameter.

The `requested_token_type` attribute allows the client to request a specific type of token in the response. The value `urn:ietf:params:oauth:token-type:access_token` indicates that the client wants an access token, in whatever token format the AS prefers, while `urn:ietf:params:oauth:token-type:jwt` can be used to request a JWT specifically. There are other values defined in the specification, permitting the client to ask for other security token types. In this way, OAuth2 token exchange can be seen as a limited form of security token service.

DEFINITION A security token service (STS) is a service that can translate security tokens from one format to another based on security policies. An STS can be used to bridge security systems that expect different token formats.

When a backend service introspects the exchanged access token, they may see a nested chain of `act` claims, as shown in listing 11.15. As with other access tokens, the `sub` claim indicates the user on whose behalf the request is being made. Access control decisions should always be made primarily based on the user indicated in this claim. Other claims in the token, such as roles or permissions, will be about that user. The first `act` claim indicates the calling service that is acting on behalf of the user. An `act` claim is itself a JSON claims set that may contain multiple identity attributes about the calling service, such as the issuer of its identity, which may be needed to uniquely identify the service. If the token has passed through multiple services, then there may be further `act` claims nested inside the first one, indicating the previous services that also acted as the same user in servicing the same request. If the backend service wants to take the service account into consideration when making access control decisions, it should limit this to just the first (outermost) `act` identity. Any previous `act` identities are intended only for ensuring a complete audit record.

NOTE Nested `act` claims don't indicate that service77 is pretending to be service16 pretending to be Alice! Think of it as a mask being passed from actor to actor, rather than a single actor wearing multiple layers of masks.

Listing 11.16 An exchanged access token introspection response

```
{
   "aud":"https://service26.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904100,
   "nbf":1443904000,
   "sub":"alice@example.com",                    (1)
   "act":                                        (2)
   {
     "sub":"https://service16.example.com",      (2)
     "act":                                      (3)
     {
       "sub":"https://service77.example.com"     (3)
     }
   }
 }
```

(1) The effective user of the token

❷ The service that is acting on behalf of the user

❸ A previous service that also acted on behalf of the user in the same request

Token exchange introduces an additional network roundtrip to the AS to exchange the access token at each hop of servicing a request. It can therefore be more expensive than the phantom token pattern and introduce additional latency in a microservices architecture. Token exchange is more compelling when service calls cross trust boundaries and latency is less of a concern. For example, in healthcare, a patient may enter the healthcare system and be treated by multiple healthcare providers, each of which needs some level of access to the patient's records. Token exchange allows one provider to hand off access to another provider without repeatedly asking the patient for consent. The AS decides an appropriate level of access for each service based on configured authorization policies.

NOTE When multiple clients and organizations are granted access to user data based on a single consent flow, you should ensure that this is indicated to the user in the initial consent screen so that they can make an informed decision.

Macaroons for service APIs

If the scope or authority of a token only needs to be reduced when calling other services, a macaroon-based access token (chapter 9) can be used as an alternative to token exchange. Recall that a macaroon allows any party to append caveats to the token, restricting what it can be used for. For example, an initial broad-scoped token supplied by a user granting access to their patient records can be restricted with caveats before calling external services, perhaps only to allow access to notes from the last 24 hours. The advantage is that this can be done locally (and efficiently) without having to call the AS to exchange the token.

A common use of service credentials is for a frontend API to make calls to a backend database. The frontend API typically has a username and password that it uses to connect, with privileges to perform a wide range of operations. If instead the database used macaroons for authorization, it could issue a broadly privileged macaroon to the frontend service. The frontend service can then append caveats to the macaroon and reissue it to its own API clients and ultimately to users. For example, it might append a caveat `user = "mary"` to a token issued to Mary so that she can

only read her own data, and an expiry time of 5 minutes. These constrained tokens can then be passed all the way back to the database, which can enforce the caveats. This was the approach adopted by the Hyperdex database (**http://mng.bz/gg1l**). Very few databases support macaroons today, but in a microservice architecture you can use the same techniques to allow more flexible and dynamic access control.

Pop quiz

13. In the phantom token pattern, the original access token is replaced by which one of the following?
    1. A macaron
    2. A SAML assertion
    3. A short-lived signed JWT
    4. An OpenID Connect ID token
    5. A token issued by an internal AS
14. In OAuth2 token exchange, which parameter is used to communicate a token that represents the user on whose behalf the client is operating?
    1. The `scope` parameter
    2. The `resource` parameter
    3. The `audience` parameter
    4. The `actor_token` parameter
    5. The `subject_token` parameter

The answers are at the end of the chapter.

Answers to pop quiz questions

1. d and e. API keys identify services, external organizations, or businesses that need to call your API. An API key may have a long expiry time or never expire, while user tokens typically expire after minutes or hours.
2. e.
3. e. Client credentials and service account authentication can use the same mechanisms; the primary benefit of using a service account is that clients are often stored in a private database that only the AS has access to. Service accounts live in the same repository as other users and so APIs can query identity details and role/group memberships.
4. c, d, and e.
5. e. The CertificateRequest message is sent to request client certificate authentication. If it's not sent by the server, then the client can't use a certificate.

6. c. The client signs all previous messages in the handshake with the private key. This prevents the message being reused for a different handshake.
7. b.
8. f. The only check required is to compare the hash of the certificate. The AS performs all other checks when it issues the access token. While an API can optionally implement additional checks, these are not required for security.
9. False. A client can request certificate-bound access tokens even if it uses a different client authentication method. Even a public client can request certificate-bound access tokens.
10. a and d.
11. d.
12. a. HKDF-Expand. HKDF-Extract is used to convert non-uniform input key material into a uniformly random master key.
13. c.
14. e.

Summary

- API keys are often used to authenticate service-to-service API calls. A signed or encrypted JWT is an effective API key. When used to authenticate a client, this is known as JWT bearer authentication.
- OAuth2 supports service-to-service API calls through the client credentials grant type that allows a client to obtain an access token under its own authority.
- A more flexible alternative to the client credentials grant is to create service accounts which act like regular user accounts but are intended for use by services. Service accounts should be protected with strong authentication mechanisms because they often have elevated privileges compared to normal accounts.
- The JWT bearer grant type can be used to obtain an access token for a service account using a JWT. This can be used to deploy short-lived JWTs to services when they start up that can then be exchanged for access and refresh tokens. This avoids leaving long-lived, highly-privileged credentials on disk where they might be accessed.
- TLS client certificates can be used to provide strong authentication of service clients. Certificate-bound access tokens improve the security of OAuth2 and prevent token theft and misuse.

- Kubernetes includes a simple method for distributing credentials to services, but it suffers from some security weaknesses. Secret vaults and key management services provide better security but need an initial credential to access. A short-lived JWT can provide this initial credential with the least risk.
- When service-to-service API calls are made in response to user requests, care should be taken to avoid confused deputy attacks. To avoid this, the original user identity should be communicated to backend services. The phantom token pattern provides an efficient way to achieve this in a microservice architecture, while OAuth2 token exchange and macaroons can be used across trust boundaries.

---

1. OAuth2 Basic authentication requires additional URL-encoding if the client ID or secret contain non-ASCII characters. See https://tools.ietf.org/html/rfc6749#section-2.3.1 for details.

2. There are additional sub-protocols that are used to change algorithms or keys after the initial handshake and to signal alerts, but you don't need to understand these.

3. The database must be restarted because the Natter API tries to recreate the schema on startup and will throw an exception if it already exists.

4. The Istio sidecar proxy is based on Envoy, which is developed by Lyft, in case you're wondering about the examples!

5. The Istio Gateway is not just a Kubernetes ingress controller. An Istio service mesh may involve only part of a Kubernetes cluster, or may span multiple Kubernetes clusters, while a Kubernetes ingress controller always deals with external traffic coming into a single cluster.

6. The code in listing 11.9 does parse the certificate as a side effect of decoding the header with a CertificateFactory, but you could avoid this if you wanted to.

7. Remember to run `eval $(minikube docker-env)` if this is a new terminal session.

8. If you're using the IBM JDK, use the name "PKCS11IMPLKS" instead.