13 Securing IoT APIs

This chapter covers

- Authenticating devices to APIs
- Avoiding replay attacks in end-to-end device authentication
- Authorizing things with the OAuth2 device grant
- Performing local access control when a device is offline

In chapter 12, you learned how to secure communications between devices using Datagram TLS (DTLS) and end-to-end security. In this chapter, you'll learn how to secure access to APIs in Internet of Things (IoT) environments, including APIs provided by the devices themselves and cloud APIs the devices connect to. In its rise to become the dominant API security technology, OAuth2 is also popular for IoT applications, so you'll learn about recent adaptations of OAuth2 for constrained devices in section 13.3. Finally, we'll look at how to manage access control decisions when a device may be disconnected from other services for prolonged periods of time in section 13.4.

13.1 Authenticating devices

In consumer IoT applications, devices are often acting under the control of a user, but industrial IoT devices are typically designed to act autonomously without manual user intervention. For example, a system monitoring supply levels in a warehouse would be configured to automatically order new stock when levels of critical supplies become low. In these cases, IoT devices act under their own authority much like the service-to-service API calls in chapter 11. In chapter 12, you saw how to provision credentials to devices to secure IoT communications, and in this section, you'll see how to use those to authenticate devices to access APIs.

13.1.1 Identifying devices

To be able to identify clients and make access control decisions about them in your API, you need to keep track of legitimate device identifiers and other attributes of the devices and link those to the credentials that device uses to authenticate. This allows you to look up these device at-

tributes after authentication and use them to make access control decisions. The process is very similar to authentication for users, and you could reuse an existing user repository such as LDAP to also store device profiles, although it is usually safer to separate users from device accounts to avoid confusion. Where a user profile typically includes a hashed password and details such as their name and address, a device profile might instead include a pre-shared key for that device, along with manufacturer and model information, and the location of where that device is deployed.

The device profile can be generated at the point the device is manufactured, as shown in figure 13.1. Alternatively, the profile can be built when devices are first delivered to an organization, in a process known as onboarding.

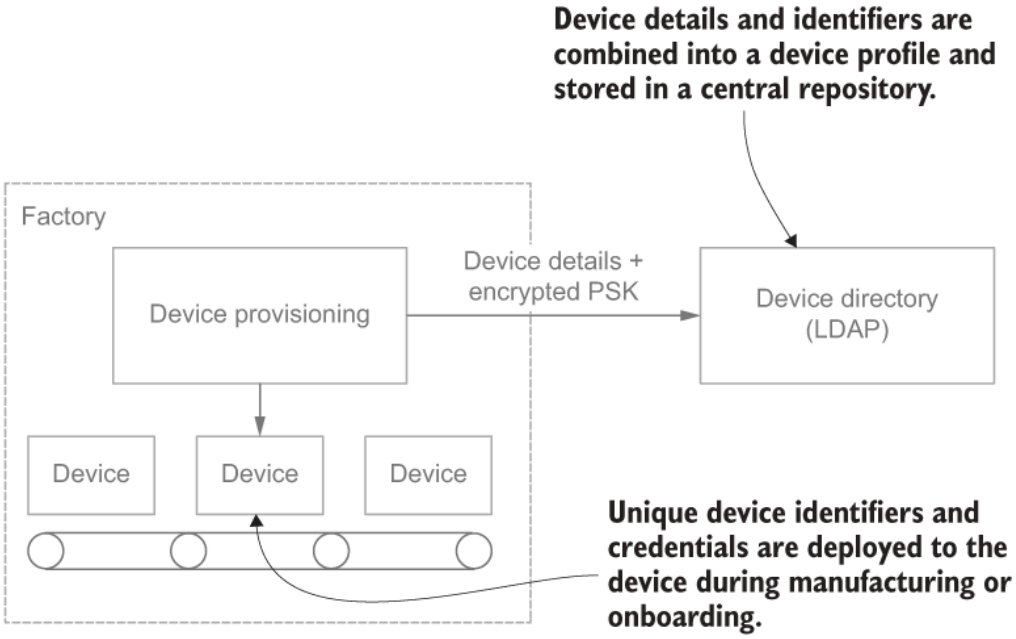

Figure 13.1 Device details and unique identifiers are stored in a shared repository where they can be accessed later.

DEFINITION Device onboarding is the process of deploying a device and registering it with the services and networks it needs to access.

Listing 13.1 shows code for a simple device profile with an identifier, basic model information, and an encrypted pre-shared key (PSK) that can be used to communicate with the device using the techniques in chapter 12. The PSK will be encrypted using the NaCl `SecretBox` class that you used in chapter 6, so you can add a method to decrypt the PSK with a secret key. Navigate to src/main/java/com/manning/ apisecurityinaction and create a new file named Device.java and copy in the contents of the listing.

Listing 13.1 A device profile

```
package com.manning.apisecurityinaction;

import org.dalesbred.Database;
import org.dalesbred.annotation.DalesbredInstantiator;
import org.h2.jdbcx.JdbcConnectionPool;
import software.pando.crypto.nacl.SecretBox;

import java.io.*;
import java.security.Key;
import java.util.Optional;

public class Device {
    final String deviceId;                                          (1)
    final String manufacturer;                                      (1)
    final String model;                                             (1)
    final byte[] encryptedPsk;                                      (1)

    @DalesbredInstantiator                                          (2)
    public Device(String deviceId, String manufacturer,
                  String model, byte[] encryptedPsk) {
        this.deviceId = deviceId;
        this.manufacturer = manufacturer;
        this.model = model;
        this.encryptedPsk = encryptedPsk;
    }

    public byte[] getPsk(Key decryptionKey) {                       (3)
        try (var in = new ByteArrayInputStream(encryptedPsk)) {     (3)
            var box = SecretBox.readFrom(in);                       (3)
            return box.decrypt(decryptionKey);                      (3)
        } catch (IOException e) {                                   (3)
            throw new RuntimeException("Unable to decrypt PSK", e); (3)
        }                                                           (3)
    }                                                               (3)
}
```

(1) Create fields for the device attributes.

(2) Annotate the constructor so that Dalesbred knows how to load a device from the database.

(3) Add a method to decrypt the device PSK using NaCl's SecretBox.

You can now populate the database with device profiles. Listing 13.2 shows how to initialize the database with an example device profile and encrypted PSK. Just like previous chapters you can use a temporary in-memory H2 database to hold the device details, because this makes it

easy to test. In a production deployment you would use a database server or LDAP directory. You can load the database into the Dalesbred library that you've used since chapter 2 to simplify queries. Then you should create the table to hold the device profiles, in this case with simple string attributes (`VARCHAR` in SQL) and a binary attribute to hold the encrypted PSK. You could extract these SQL statements into a separate schema.sql file as you did in chapter 2, but because there is only a single table, I've used string literals instead. Open the Device.java file again and add the new method from the listing to create the example device database.

Listing 13.2 Populating the device database

```
static Database createDatabase(SecretBox encryptedPsk) throws IOException {
    var pool = JdbcConnectionPool.create("jdbc:h2:mem:devices",     (1)
            "devices", "password");                                 (1)
    var database = Database.forDataSource(pool);                    (1)

    database.update("CREATE TABLE devices(" +                       (2)
            "device_id VARCHAR(30) PRIMARY KEY," +                  (2)
            "manufacturer VARCHAR(100) NOT NULL," +                 (2)
            "model VARCHAR(100) NOT NULL," +                        (2)
            "encrypted_psk VARBINARY(1024) NOT NULL)");             (2)

    var out = new ByteArrayOutputStream();                          (3)
    encryptedPsk.writeTo(out);                                      (3)
    database.update("INSERT INTO devices(" +                        (4)
            "device_id, manufacturer, model, encrypted_psk) " +     (4)
            "VALUES(?, ?, ?, ?)", "test", "example", "ex001",       (4)
            out.toByteArray());                                     (4)

    return database;
}
```

(1) Create and load the in-memory device database.

(2) Create a table to hold device details and encrypted PSKs.

(3) Serialize the example encrypted PSK to a byte array.

(4) Insert an example device into the database.

You'll also need a way to find a device by its device ID or other attributes. Dalesbred makes this quite simple, as shown in listing 13.3. The `findOptional` method can be used to search for a device; it will return an empty result if there is no matching device. You should select the fields of the de-

vice table in exactly the order they appear in the `Device` class constructor in listing 13.1. As described in chapter 2, use a bind parameter in the query to supply the device ID, to avoid SQL injection attacks.

Listing 13.3 Finding a device by ID

```
static Optional<Device> find(Database database, String deviceId) {
    return database.findOptional(Device.class,                          (1)
            "SELECT device_id, manufacturer, model, encrypted_psk " +   (2)
                    "FROM devices WHERE device_id = ?", deviceId);      (3)
}
```

(1) Use the findOptional method with your Device class to load devices.

(2) Select device attributes in the same order they appear in the constructor.

(3) Use a bind parameter to query for a device with the matching device_id.

Now that you have some device details, you can use them to authenticate devices and perform access control based on those device identities, which you'll do in sections 13.1.2 and 13.1.3.

13.1.2 Device certificates

An alternative to storing device details directly in a database is to instead provide each device with a certificate containing the same details, signed by a trusted certificate authority. Although traditionally certificates are used with public key cryptography, you can use the same techniques for constrained devices that must use symmetric cryptography instead. For example, the device can be issued with a signed JSON Web Token that contains device details and an encrypted PSK that the API server can decrypt, as shown in listing 13.4. The device treats the certificate as an opaque token and simply presents it to APIs that it needs to access. The API trusts the JWT because it is signed by a trusted issuer, and it can then decrypt the PSK to authenticate and communicate with the device.

Listing 13.4 Encrypted PSK in a JWT claims set

```
{
    "iss":"https://example.com/devices",    (1)
    "iat":1590139506,                       (1)
    "exp":1905672306,                       (1)
```

```
    "sub":"ada37d7b-e895-4d55-9571-4df602e60c27",                    ❶
    "psk":" jZvara1OnqqBZrz1HtvHBCNjXvCJptEuIAAAAJInAtaLFnYna9K0WxX4_  ❷
➥ IGPyztb8VUwo0CI_UmqDQgm"                                          ❷
}
```

❶ Include the usual JWT claims identifying the device.

❷ Add an encrypted PSK that can be used to communicate with the device.

This can be more scalable than a database if you have many devices, but makes it harder to update incorrect details or change keys. A middle ground is provided by the attestation techniques discussed in chapter 12, in which an initial certificate and key are used to prove the make and model of a device when it first registers on a network, and it then negotiates a device-specific key to use from then on.

13.1.3 Authenticating at the transport layer

If there is a direct connection between a device and the API it's accessing, then you can use authentication mechanisms provided by the transport layer security protocol. For example, the pre-shared key (PSK) cipher suites for TLS described in chapter 12 provide mutual authentication of both the client and the server. Client certificate authentication can be used by more capable devices just as you did in chapter 11 for service clients. In this section, we'll look at identifying devices using PSK authentication.

During the handshake, the client provides a PSK identity to the server in the ClientKeyExchange message. The API can use this PSK ID to locate the correct PSK for that client. The server can look up the device profile for that device using the PSK ID at the same time that it loads the PSK, as shown in figure 13.2. Once the handshake has completed, the API is assured of the device identity by the mutual authentication that PSK cipher suites achieve.

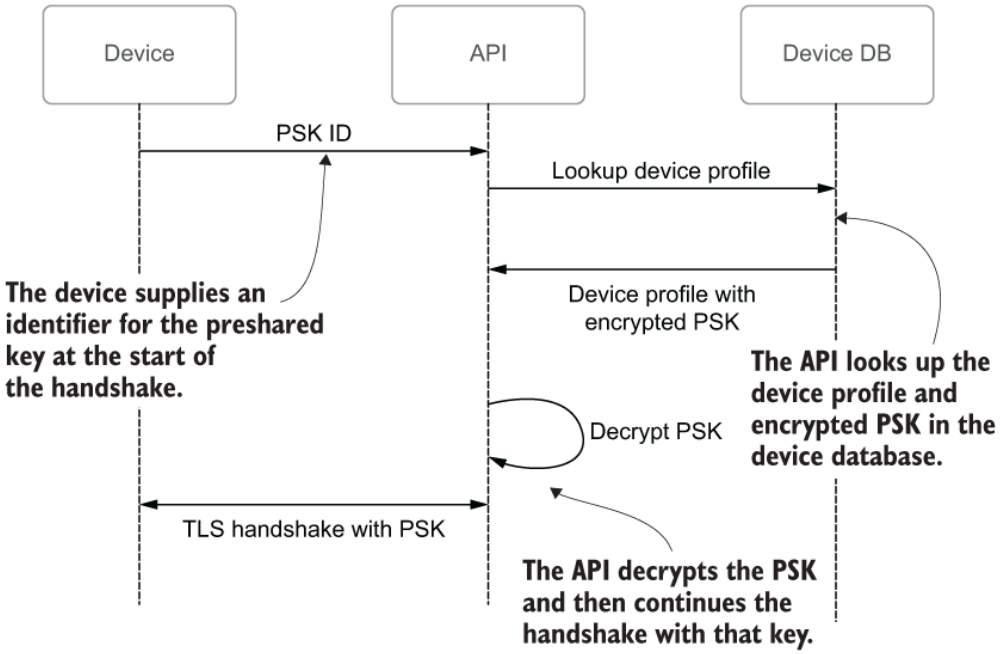

Figure 13.2 When the device connects to the API, it sends a PSK identifier in the TLS ClientKeyExchange message. The API can use this to find a matching device profile with an encrypted PSK for that device. The API decrypts the PSK and then completes the TLS handshake using the PSK to authenticate the device.

In this section, you'll adjust the `PskServer` from chapter 12 to look up the device profile during authentication. First, you need to load and initialize the device database. Open the PskServer.java file and add the following lines at the start of the `main()` method just after the PSK is loaded:

```
var psk = loadPsk(args[0].toCharArray());                    ❶
var encryptionKey = SecretBox.key();                         ❷
var deviceDb = Device.createDatabase(                        ❸
        SecretBox.encrypt(encryptionKey, psk));              ❸
```

❶ The existing line to load the example PSK

❷ Create a new PSK encryption key.

❸ Initialize the database with the encrypted PSK.

The client will present its device identifier as the PSK identity field during the handshake, which you can then use to find the associated device profile and encrypted PSK to use to authenticate the session. Listing 13.5 shows a new `DeviceIdentityManager` class that you can use with Bouncy Castle instead of the existing PSK identity manager. The new identity manager performs a lookup in the device database to find a device that matches the PSK identity supplied by the client. If a matching de-

vice is found, then you can decrypt the associated PSK from the device profile and use that to authenticate the TLS connection. Otherwise, return `null` to abort the connection. The client doesn't need any hint to determine its own identity, so you can also return `null` from the `getHint()` method to disable the ServerKeyExchange message in the handshake just as you did in chapter 12. Create a new file named DeviceIdentityManager.java in the same folder as the Device.java file you created earlier and add the contents of the listing.

Listing 13.5 The device IdentityManager

```
package com.manning.apisecurityinaction;
import org.bouncycastle.tls.TlsPSKIdentityManager;
import org.dalesbred.Database;
import java.security.Key;
import static java.nio.charset.StandardCharsets.UTF_8;
public class DeviceIdentityManager implements TlsPSKIdentityManager {
    private final Database database;
    private final Key pskDecryptionKey;

    public DeviceIdentityManager(Database database, Key pskDecryptionKey) {
        this.database = database;
        this.pskDecryptionKey = pskDecryptionKey;
    }
    @Override
    public byte[] getHint() {
        return null;
    }
    @Override
    public byte[] getPSK(byte[] identity) {
        var deviceId = new String(identity, UTF_8);
        return Device.find(database, deviceId)
                .map(device -> device.getPsk(pskDecryptionKey))
                .orElse(null);
    }
}
```

❶ Initialize the identity manager with the device database and PSK decryption key.

❷ Return a null identity hint to disable the ServerKeyExchange message.

❸ Convert the PSK identity hint into a UTF-8 string to use as the device identity.

❹ If the device exists, then decrypt the associated PSK.

❺ Otherwise, return null to abort the connection.

To use the new device identity manager, you need to update the PskServer class again. Open PskServer.java in your editor and change the lines of code that create the PSKTlsServer object to use the new class. I've highlighted the new code in bold:

```
var crypto = new BcTlsCrypto(new SecureRandom());
var server = new PSKTlsServer(crypto,
        new DeviceIdentityManager(deviceDb, encryptionKey)) {
```

You can delete the old `getIdentityManager()` method too because it is unused now. You also need to adjust the `PskClient` implementation to send the correct device ID during the handshake. If you recall from chapter 12, we used an SHA-512 hash of the PSK as the ID there, but the device database uses the ID `"test"` instead. Open PskClient.java and change the `pskId` variable at the top of the `main()` method to use the UTF-8 bytes of the correct device ID:

```
var pskId = "test".getBytes(UTF_8);
```

If you now run the `PskServer` and then the `PskClient` it will still work correctly, but now it is using the encrypted PSK loaded from the device database.

EXPOSING THE DEVICE IDENTITY TO THE API

Although you are now authenticating the device based on a PSK attached to its device profile, that device profile is not exposed to the API after the handshake completes. Bouncy Castle doesn't provide a public method to get the PSK identity associated with a connection, but it is easy to expose this yourself by adding a new method to the `PSKTlsServer`, as shown in listing 13.6. A protected variable inside the server contains the `TlsContext` class, which has information about the connection (the server supports only a single client at a time). The PSK identity is stored inside the `SecurityParameters` class for the connection. Open the PskServer.java file and add the new method highlighted in bold in the listing. You can then retrieve the device identity after receiving a message by calling:

```
var deviceId = server.getPeerDeviceIdentity();
```

CAUTION You should only trust the PSK identity returned from `getSecurityParametersConnection()`, which are the final parameters after the handshake completes. The similarly named `getSecurityParametersHandshake()` contains parameters negotiated during the handshake process before authentication has finished and may be incorrect.

Listing 13.6 Exposing the device identity

```
var server = new PSKTlsServer(crypto,
        new DeviceIdentityManager(deviceDb, encryptionKey)) {
    @Override
    protected ProtocolVersion[] getSupportedVersions() {
        return ProtocolVersion.DTLSv12.only();
    }
    @Override
    protected int[] getSupportedCipherSuites() {
        return new int[] {
                CipherSuite.TLS_PSK_WITH_AES_128_CCM,
                CipherSuite.TLS_PSK_WITH_AES_128_CCM_8,
                CipherSuite.TLS_PSK_WITH_AES_256_CCM,
                CipherSuite.TLS_PSK_WITH_AES_256_CCM_8,
                CipherSuite.TLS_PSK_WITH_AES_128_GCM_SHA256,
                CipherSuite.TLS_PSK_WITH_AES_256_GCM_SHA384,
                CipherSuite.TLS_PSK_WITH_CHACHA20_POLY1305_SHA256
        };
    }

    String getPeerDeviceIdentity() {                                    (1)
        return new String(context.getSecurityParametersConnection()    (2)
                .getPSKIdentity(), UTF_8);                              (2)
    }
};
```

(1) Add a new method to the PSKTlsServer to expose the client identity.

(2) Look up the PSK identity and decode it as a UTF-8 string.

The API server can then use this device identity to look up permissions for this device, using the same identity-based access control techniques used for users in chapter 8.

Pop quiz

1. True or False: A PSK ID is always a UTF-8 string.

2. Why should you only trust the PSK ID after the handshake completes?
    1. Before the handshake completes, the ID is encrypted.
    2. You should never trust anyone until you've shaken their hand.
    3. The ID changes after the handshake to avoid session fixation attacks.
    4. Before the handshake completes, the ID is unauthenticated so it could be fake.

The answers are at the end of the chapter.

13.2 End-to-end authentication

If the connection from the device to the API must pass through different protocols, as described in chapter 12, authenticating devices at the transport layer is not an option. In chapter 12, you learned how to secure end-to-end API requests and responses using authenticated encryption with Concise Binary Object Representation (CBOR) Object Signing and Encryption (COSE) or NaCl's `CryptoBox`. These encrypted message formats ensure that requests cannot be tampered with, and the API server can be sure that the request originated from the device it claims to be from. By adding a device identifier to the message as associated data.[1] which you'll recall from chapter 6 is authenticated but not encrypted, the API can look up the device profile to find the key to decrypt and authenticate messages from that device.

Unfortunately, this is not enough to ensure that API requests really did come from that device, so it is dangerous to make access control decisions based solely on the Message Authentication Code (MAC) used to authenticate the message. The reason is that API requests can be captured by an attacker and later replayed to perform the same action again at a later time, known as a replay attack. For example, suppose you are the leader of a clandestine evil organization intent on world domination. A monitoring device in your uranium enrichment plant sends an API request to increase the speed of a centrifuge. Unfortunately, the request is intercepted by a secret agent, who then replays the request hundreds of times, and the centrifuge spins too quickly, causing irreparable damage and delaying your dastardly plans by several years.

DEFINITION In a replay attack, an attacker captures genuine API requests and later replays them to cause actions that weren't intended by the original client. Replay attacks can cause disruption even if the message itself is authenticated.

To prevent replay attacks, the API needs to ensure that a request came from a legitimate client and is fresh. Freshness ensures that the message

is recent and hasn't been replayed and is critical to security when making access control decisions based on the identity of the client. The process of identifying who an API server is talking to is known as entity authentication.

DEFINITION Entity authentication is the process of identifying who requested an API operation to be performed. Although message authentication can confirm who originally authored a request, entity authentication additionally requires that the request is fresh and has not been replayed. The connection between the two kinds of authentication can be summed up as: entity authentication = message authentication + freshness.

In previous chapters, you've relied on TLS or authentication protocols such as OpenID Connect (OIDC; see chapter 7) to ensure freshness, but end-to-end API requests need to ensure this property for themselves. There are three general ways to ensure freshness:

- API requests can include timestamps that indicate when the request was generated. The API server can then reject requests that are too old. This is the weakest form of replay protection because an attacker can still replay requests until they expire. It also requires the client and server to have access to accurate clocks that cannot be influenced by an attacker.
- Requests can include a unique nonce (number-used-once). The server remembers these nonces and rejects requests that attempt to reuse one that has already been seen. To reduce the storage requirements on the server, this is often combined with a timestamp, so that used nonces only have to be remembered until the associated request expires. In some cases, you may be able to use a monotonically increasing counter as the nonce, in which case the server only needs to remember the highest value it has seen so far and reject requests that use a smaller value. If multiple clients or servers share the same key, it can be difficult to synchronize the counter between them all.
- The most secure method is to use a challenge-response protocol shown in figure 13.3, in which the server generates a random challenge value (a nonce) and sends it to the client. The client then includes the challenge value in the API request, proving that the request was generated after the challenge. Although more secure, this adds overhead because the client must talk to the server to obtain a challenge before they can send any requests.

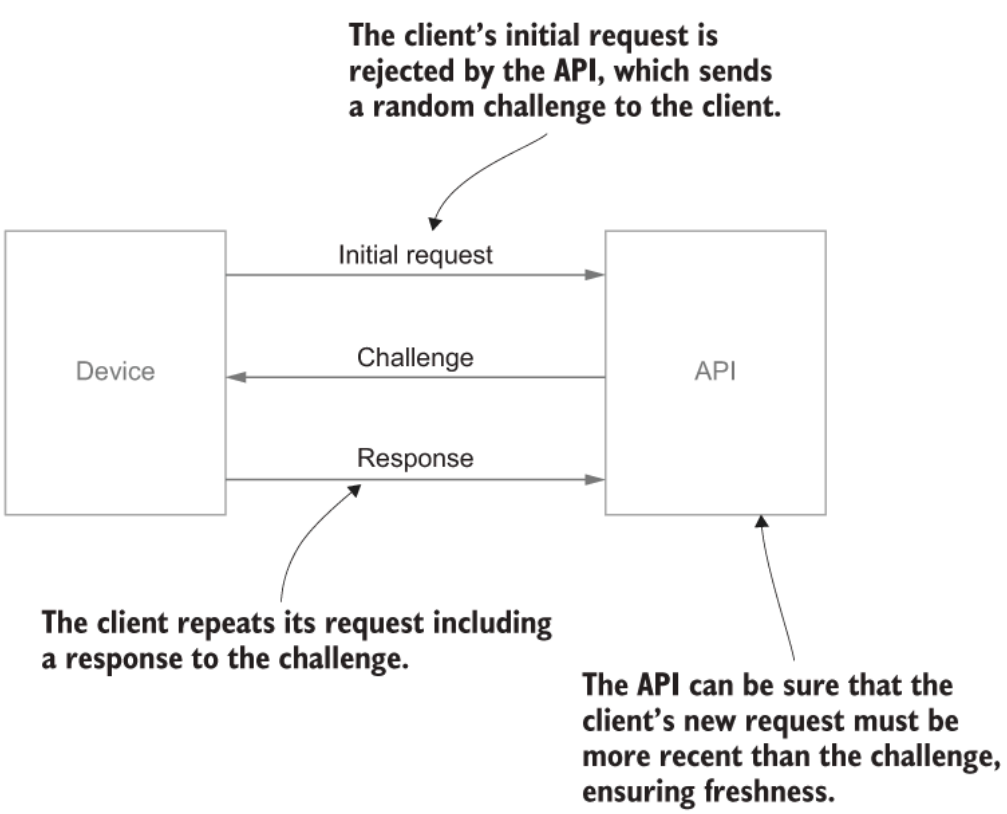

Figure 13.3 A challenge-response protocol ensures that an API request is fresh and has not been replayed by an attacker. The client's first API request is rejected, and the API generates a random challenge value that it sends to the client and stores locally. The client retries its request, including a response to the challenge. The server can then be sure that the request has been freshly generated by the genuine client and is not a replay attack.

DEFINITION A monotonically increasing counter is one that only ever increases and never goes backward and can be used as a nonce to prevent replay of API requests. In a challenge-response protocol, the server generates a random challenge that the client includes in a subsequent request to ensure freshness.

Both TLS and OIDC employ challenge-response protocols for authentication. For example, in OIDC the client includes a random nonce in the authentication request and the identity provider includes the same nonce in the generated ID token to ensure freshness. However, in both cases the challenge is only used to ensure freshness of an initial authentication request and then other methods are used from then on. In TLS, the challenge response happens during the handshake, and afterward a monotonically increasing sequence number is added to every message. If either side sees the sequence number go backward, then they abort the connection and a new handshake (and new challenge response) needs to be performed. This relies on the fact that TLS is a stateful protocol between a single client and a single server, but this can't generally be guaranteed for

an end-to-end security protocol where each API request may go to a different server.

Attacks from delaying, reordering, or blocking messages

Replay attacks are not the only way that an attacker may interfere with API requests and responses. They may also be able to block or delay messages from being received, which can cause security issues in some cases, beyond simple denial of service. For example, suppose a legitimate client sends an authenticated “unlock” request to a door-lock device. If the request includes a unique nonce or other mechanism described in this section, then an attacker won’t be able to replay the request later. However, they can prevent the original request being delivered immediately and then send it to the device later, when the legitimate user has given up and walked away. This is not a replay attack because the original request was never received by the API; instead, the attacker has merely delayed the request and delivered it at a later time than was intended. **http://mng.bz/nzYK** describes a variety of attacks against CoAP that don’t directly violate the security properties of DTLS, TLS, or other secure communication protocols. These examples illustrate the importance of good threat modeling and carefully examining assumptions made in device communications. A variety of mitigations for CoAP are described in **http://mng.bz/v9oM**, including a simple challenge-response “Echo” option that can be used to prevent delay attacks, ensuring a stronger guarantee of freshness.

13.2.1 OSCORE

Object Security for Constrained RESTful Environments (OSCORE; **https://tools.ietf .org/html/rfc8613**) is designed to be an end-to-end security protocol for API requests in IoT environments. OSCORE is based on the use of pre-shared keys between the client and server and makes use of CoAP (Constrained Application Protocol) and COSE (CBOR Object Signing and Encryption) so that cryptographic algorithms and message formats are suitable for constrained devices.

NOTE OSCORE can be used either as an alternative to transport layer security protocols such as DTLS or in addition to them. The two approaches are complimentary, and the best security comes from combining both. OSCORE doesn’t encrypt all parts of the messages being exchanged so TLS or DTLS provides additional protection, while OSCORE ensures end-to-end security.

To use OSCORE, the client and server must maintain a collection of state, known as the security context, for the duration of their interactions with each other. The security context consists of three parts, shown in figure 13.4:

- A Common Context, which describes the cryptographic algorithms to be used and contains a Master Secret (the PSK) and an optional Master Salt. These are used to derive keys and nonces used to encrypt and authenticate messages, such as the Common IV, described later in this section.
- A Sender Context, which contains a Sender ID, a Sender Key used to encrypt messages sent by this device, and a Sender Sequence Number. The sequence number is a nonce that starts at zero and is incremented every time the device sends a message.
- A Recipient Context, which contains a Recipient ID, a Recipient Key, and a Replay Window, which is used to detect replay of received messages.

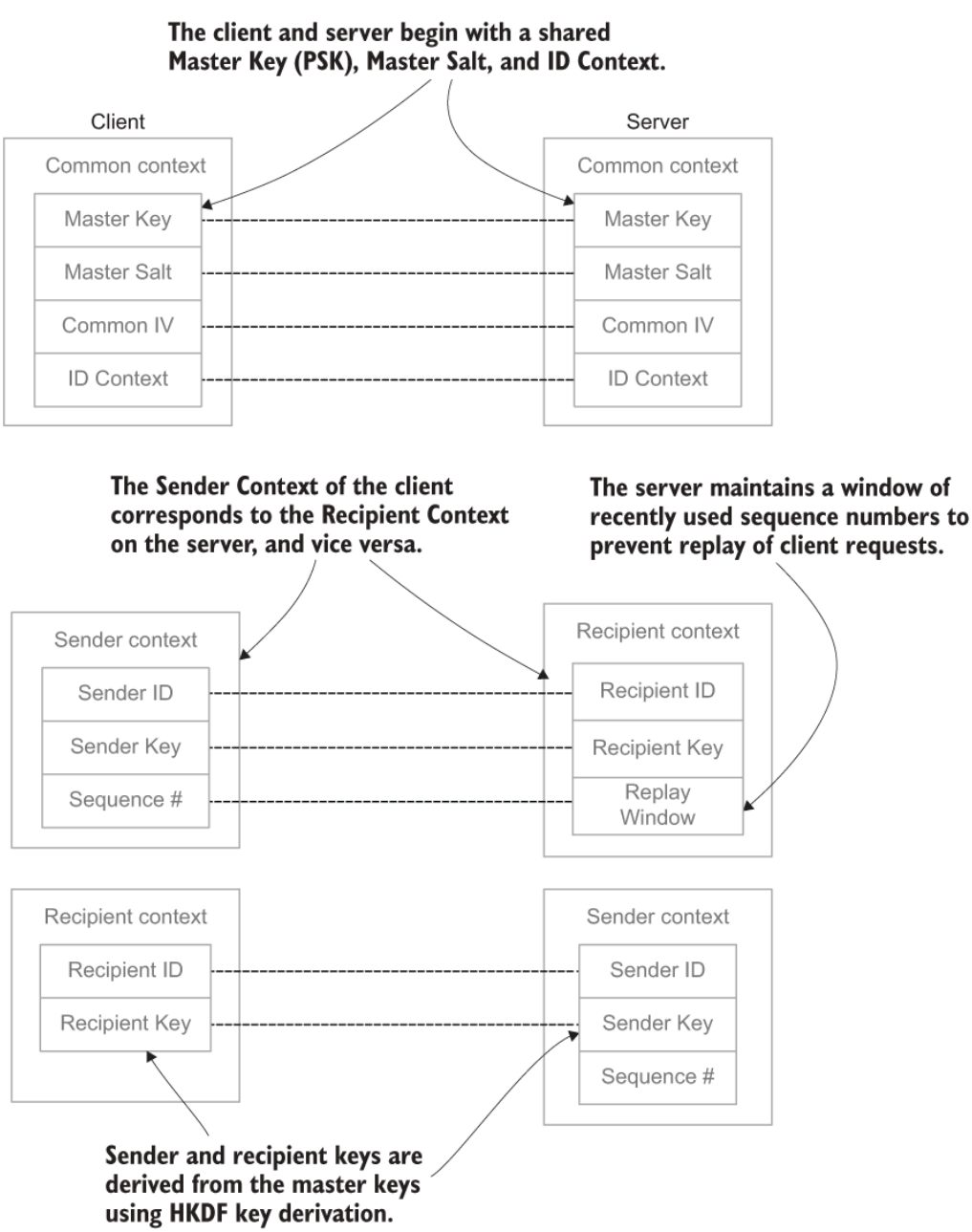

Figure 13.4 The OSCORE context is maintained by the client and server and consists of three parts: a common context contains a Master Key, Master Salt, and Common IV component. Sender and Recipient Contexts are derived from this common context and IDs for the sender and recipient. The context on the server mirrors that on the client, and vice versa.

WARNING Keys and nonces are derived deterministically in OSCORE, so if the same security context is used more than once, then catastrophic nonce reuse can occur. Devices must either reliably store the context state for the life of the Master Key (including across device restarts) or else negotiate fresh random parameters for each session.

DERIVING THE CONTEXT

The Sender ID and Recipient ID are short sequences of bytes and are typically only allowed to be a few bytes long, so they can't be globally unique names. Instead, they are used to distinguish the two parties involved in

the communication. For example, some OSCORE implementations use a single 0 byte for the client, and a single 1 byte for the server. An optional ID Context string can be included in the Common Context, which can be used to map the Sender and Recipient IDs to device identities, for example in a lookup table.

The Master Key and Master Salt are combined using the HKDF key derivation function that you first used in chapter 11. Previously, you've only used the HKDF-Expand function, but this combination is done using the HKDF-Extract method that is intended for inputs that are not uniformly random. HKDF-Extract is shown in listing 13.7 and is just a single application of HMAC using the Master Salt as the key and the Master Key as the input. Open the HKDF.java file and add the `extract` method to the existing code.

Listing 13.7 HKDF-Extract

```
public static Key extract(byte[] salt, byte[] inputKeyMaterial)    (1)
        throws GeneralSecurityException {
    var hmac = Mac.getInstance("HmacSHA256");
    if (salt == null) {                                            (2)
        salt = new byte[hmac.getMacLength()];                      (2)
    }                                                              (2)
    hmac.init(new SecretKeySpec(salt, "HmacSHA256"));              (3)
    return new SecretKeySpec(hmac.doFinal(inputKeyMaterial),       (3)
            "HmacSHA256");
}
```

(1) HKDF-Extract takes a random salt value and the input key material.

(2) If a salt is not provided, then an all-zero salt is used.

(3) The result is the output of HMAC using the salt as the key and the key material as the input.

The HKDF key for OSCORE can then be calculated from the Master Key and Master Salt as follows:

```
var hkdfKey = HKDF.extract(masterSalt, masterKey);
```

The sender and recipient keys are then derived from this master HKDF key using the HKDF-Expand function from chapter 10, as shown in listing

13.8. A context argument is generated as a CBOR array, containing the following items in order:

- The Sender ID or Recipient ID, depending on which key is being derived.
- The ID Context parameter, if specified, or a zero-length byte array otherwise.
- The COSE algorithm identifier for the authenticated encryption algorithm being used.
- The string “Key” encoded as a CBOR binary string in ASCII.
- The size of the key to be derived, in bytes.

This is then passed to the `HKDF.expand()` method to derive the key. Create a new file named Oscore.java and copy the listing into it. You’ll need to add the following imports at the top of the file:

```
import COSE.*;
import com.upokecenter.cbor.CBORObject;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import java.nio.*;
import java.security.*;
```

Listing 13.8 Deriving the sender and recipient keys

```
private static Key deriveKey(Key hkdfKey, byte[] id,
    byte[] idContext, AlgorithmID coseAlgorithm)
        throws GeneralSecurityException {

    int keySizeBytes = coseAlgorithm.getKeySize() / 8;
    CBORObject context = CBORObject.NewArray();              (1)
    context.Add(id);                                         (1)
    context.Add(idContext);                                  (1)
    context.Add(coseAlgorithm.AsCBOR());                     (1)
    context.Add(CBORObject.FromObject("Key"));               (1)
    context.Add(keySizeBytes);                               (1)

    return HKDF.expand(hkdfKey, context.EncodeToBytes(),     (2)
            keySizeBytes, "AES");                            (2)
}
```

(1) The context is a CBOR array containing the ID, ID context, algorithm identifier, and key size.

(2) HKDF-Expand is used to derive the key from the master HKDF key.

The Common IV is derived in almost the same way as the sender and recipient keys, as shown in listing 13.9. The label "IV" is used instead of "Key," and the length of the IV or nonce used by the COSE authenticated encryption algorithm is used instead of the key size. For example, the default algorithm is AES_CCM_16_64_128, which requires a 13-byte nonce, so you would pass 13 as the `ivLength` argument. Because our HKDF implementation returns a `Key` object, you can use the `getEncoded()` method to convert that into the raw bytes needed for the Common IV. Add this method to the `Oscore` class you just created.

Listing 13.9 Deriving the Common IV

```
private static byte[] deriveCommonIV(Key hkdfKey,
    byte[] idContext, AlgorithmID coseAlgorithm, int ivLength)
        throws GeneralSecurityException {
    CBORObject context = CBORObject.NewArray();
    context.Add(new byte[0]);
    context.Add(idContext);
    context.Add(coseAlgorithm.AsCBOR());
    context.Add(CBORObject.FromObject("IV"));                    (1)
    context.Add(ivLength);                                       (1)

    return HKDF.expand(hkdfKey, context.EncodeToBytes(),         (2)
            ivLength, "dummy").getEncoded();                     (2)
}
```

(1) Use the label "IV" and the length of the required nonce in bytes.

(1) Use HKDF-Expand but return the raw bytes rather than a Key object.

Listing 13.10 shows an example of deriving the sender and recipient keys and Common IV based on the test case from appendix C of the OSCORE specification (**https://tools.ietf.org/html/rfc8613#appendix-C.1.1**). You can run the code to verify that you get the same answers as the RFC. You can use `org.apache.commons.codec` `.binary.Hex` to print the keys and IV in hexadecimal to check the test outputs.

WARNING Don't use this master key and master salt in a real application! Fresh keys should be generated for each device.

Listing 13.10 Deriving OSCORE keys and IV

```
public static void main(String... args) throws Exception {
    var algorithm = AlgorithmID.AES_CCM_16_64_128;
    var masterKey = new byte[] {
```

```
            0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
            0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10
    };
    var masterSalt = new byte[] {
            (byte) 0x9e, 0x7c, (byte) 0xa9, 0x22, 0x23, 0x78,
            0x63, 0x40
    };
    var hkdfKey = HKDF.extract(masterSalt, masterKey);
    var senderId = new byte[0];
    var recipientId = new byte[] { 0x01 };

    var senderKey = deriveKey(hkdfKey, senderId, null, algorithm);
    var recipientKey = deriveKey(hkdfKey, recipientId, null, algorithm);
    var commonIv = deriveCommonIV(hkdfKey, null, algorithm, 13);
}
```

❶ The default algorithm used by OSCORE

❷ The Master Key and Master Salt from the OSCORE test case

❸ Derive the HKDF master key.

❹ The Sender ID is an empty byte array, and the Recipient ID is a single 1 byte.

❺ Derive the keys and Common IV.

GENERATING NONCES

The Common IV is not used directly to encrypt data because it is a fixed value, so would immediately result in nonce reuse vulnerabilities. Instead the nonce is derived from a combination of the Common IV, the sequence number (called the Partial IV), and the ID of the sender, as shown in listing 13.11. First the sequence number is checked to make sure it fits in 5 bytes, and the Sender ID is checked to ensure it will fit in the remainder of the IV. This puts significant constraints on the maximum size of the Sender ID. A packed binary array is generated consisting of the following items, in order:

- The length of the Sender ID as a single byte
- The sender ID itself, left-padded with zero bytes until it is 6 bytes less than the total IV length
- The sequence number encoded as a 5-byte big-endian integer

The resulting array is then combined with the Common IV using bitwise XOR, using the following method:

```
private static byte[] xor(byte[] xs, byte[] ys) {
    for (int i = 0; i < xs.length; ++i)             ❶
        xs[i] ^= ys[i];                             ❶
    return xs;                                      ❷
}
```

❶ XOR each element of the second array (ys) into the corresponding element of the first array (xs).

❷ Return the updated result.

Add the `xor()` method and the `nonce()` method from listing 13.11 to the `Oscore` class.

NOTE Although the generated nonce looks random due to being XORed with the Common IV, it is in fact a deterministic counter that changes predictably as the sequence number increases. The encoding is designed to reduce the risk of accidental nonce reuse.

Listing 13.11 Deriving the per-message nonce

```
private static byte[] nonce(int ivLength, long sequenceNumber,
                            byte[] id, byte[] commonIv) {
    if (sequenceNumber > (1L << 40))
        throw new IllegalArgumentException(
            "Sequence number too large");
    int idLen = ivLength - 6;
    if (id.length > idLen)
        throw new IllegalArgumentException("ID is too large");

    var buffer = ByteBuffer.allocate(ivLength).order(ByteOrder.BIG_ENDIAN)
    buffer.put((byte) id.length);
    buffer.put(new byte[idLen - id.length]);
    buffer.put(id);
    buffer.put((byte) ((sequenceNumber >>> 32) & 0xFF));
    buffer.putInt((int) sequenceNumber);
    return xor(buffer.array(), commonIv);
}
```

❶ Check the sequence number is not too large.

❷ Check the Sender ID fits in the remaining space.

❸ Encode the Sender ID length followed by the Sender ID left-padded to 6 less than the IV length.

❹ Encode the sequence number as a 5-byte big-endian integer.

❺ XOR the result with the Common IV to derive the final nonce.

ENCRYPTING A MESSAGE

Once you've derived the per-message nonce, you can encrypt an OSCORE message, as shown in listing 13.12, which is based on the example in section C.4 of the OSCORE specification. OSCORE messages are encoded as `COSE_Encrypt0` structures, in which there is no explicit recipient information. The Partial IV and the Sender ID are encoded into the message as unprotected headers, with the Sender ID using the standard COSE Key ID (KID) header. Although marked as unprotected, those values are actually authenticated because OSCORE requires them to be included in a COSE external additional authenticated data structure, which is a CBOR array with the following elements:

- An OSCORE version number, currently always set to 1
- The COSE algorithm identifier
- The Sender ID
- The Partial IV
- An options string. This is used to encode CoAP headers but is blank in this example.

The COSE structure is then encrypted with the sender key.

DEFINITION COSE allows messages to have external additional authenticated data, which are included in the message authentication code (MAC) calculation but not sent as part of the message itself. The recipient must be able to independently recreate this external data otherwise decryption will fail.

Listing 13.12 Encrypting the plaintext

```
long sequenceNumber = 20L;
byte[] nonce = nonce(13, sequenceNumber, senderId, commonIv);      ❶
byte[] partialIv = new byte[] { (byte) sequenceNumber };           ❶

var message = new Encrypt0Message();
```

```
message.addAttribute(HeaderKeys.Algorithm,                    ❷
        algorithm.AsCBOR(), Attribute.DO_NOT_SEND);           ❷
message.addAttribute(HeaderKeys.IV,                           ❷
        nonce, Attribute.DO_NOT_SEND);                        ❷
message.addAttribute(HeaderKeys.PARTIAL_IV,                   ❸
        partialIv, Attribute.UNPROTECTED);                    ❸
message.addAttribute(HeaderKeys.KID,                          ❸
        senderId, Attribute.UNPROTECTED);                     ❸
message.SetContent(                                           ❹
    new byte[] { 0x01, (byte) 0xb3, 0x74, 0x76, 0x31});       ❹

var associatedData = CBORObject.NewArray();                   ❺
associatedData.Add(1);                                        ❺
associatedData.Add(algorithm.AsCBOR());                       ❺
associatedData.Add(senderId);                                 ❺
associatedData.Add(partialIv);                                ❺
associatedData.Add(new byte[0]);                              ❺
message.setExternal(associatedData.EncodeToBytes());          ❺

Security.addProvider(new BouncyCastleProvider());             ❻
message.encrypt(senderKey.getEncoded());                      ❻
```

❶ Generate the nonce and encode the Partial IV.

❷ Configure the algorithm and nonce.

❸ Set the Partial IV and Sender ID as unprotected headers.

❹ Set the content field to the plaintext to encrypt.

❺ Encode the external associated data.

❻ Ensure Bouncy Castle is loaded for AES-CCM support, then encrypt the message.

The encrypted message is then encoded into the application protocol, such as CoAP or HTTP and sent to the recipient. Details of this encoding are given in section 6 of the OSCORE specification. The recipient can recreate the nonce from its own recipient security context, together with the Partial IV and Sender ID encoded into the message.

The recipient is responsible for checking that the Partial IV has not been seen before to prevent replay attacks. When OSCORE is transmitted over a reliable protocol such as HTTP, this can be achieved by keeping track of the last Partial IV received and ensuring that any new messages always

use a larger number. For unreliable protocols such as CoAP over UDP, where messages may arrive out of order, you can use the algorithm from RFC 4303 (http://mng.bz/4BjV). This approach maintains a window of allowed sequence numbers between a minimum and maximum value that the recipient will accept and explicitly records which values in that range have been received. If the recipient is a cluster of servers, such as a typical cloud-hosted API, then this state must be synchronized between all servers to prevent replay attacks. Alternatively, sticky load balancing can be used to ensure requests from the same device are always delivered to the same server instance, shown in figure 13.5, but this can be problematic in environments where servers are frequently added or removed. Section 13.1.5 discusses an alternative approach to preventing replay attacks that can be effective to REST APIs.

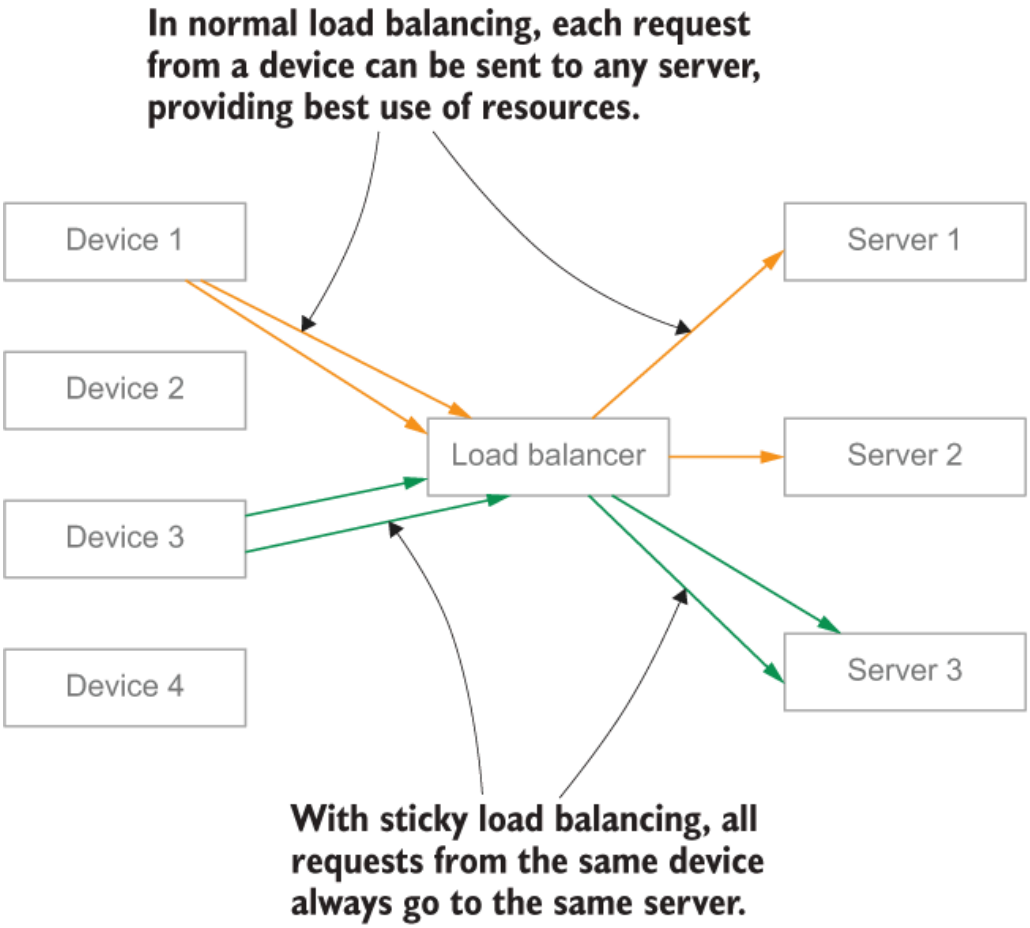

Figure 13.5 In sticky load balancing, all requests from one device are always handled by the same server. This simplifies state management but reduces scalability and can cause problems if that server restarts or is removed from the cluster.

DEFINITION Sticky load balancing is a setting supported by most load balancers that ensures that API requests from a device or client are always delivered to the same server instance. Although this can help with stateful connections, it can harm scalability and is generally discouraged.

13.2.2 Avoiding replay in REST APIs

All solutions to message replay involve the client and server maintaining some state. However, in some cases you can avoid the need for per-client state to prevent replay. For example, requests that only read data are harmless if replayed, so long as they do not require significant processing on the server and the responses are kept confidential. Some requests that perform operations are also harmless to replay if the request is idempotent.

DEFINITION An operation is idempotent if performing it multiple times has the same effect as performing it just once. Idempotent operations are important for reliability because if a request fails because of a network error, the client can safely retry it.

The HTTP specification requires the read-only methods GET, HEAD, and OPTIONS, along with PUT and DELETE requests, to all be idempotent. Only the POST and PATCH methods are not generally idempotent.

WARNING Even if you stick to PUT requests instead of POST, this doesn't mean that your requests are always safe from replay.

The problem is that the definition of idempotency says nothing about what happens if another request occurs in between the original request and the replay. For example, suppose you send a PUT request updating a page on a website, but you lose your network connection and do not know if the request succeeded or not. Because the request is idempotent, you send it again. Unknown to you, one of your colleagues in the meantime sent a DELETE request because the document contained sensitive information that shouldn't have been published. Your replayed PUT request arrives afterwards, and the document is resurrected, sensitive data and all. An attacker can replay requests to restore an old version of a resource, even though all the operations were individually idempotent.

Thankfully, there are several mechanisms you can use to ensure that no other request has occurred in the meantime. Many updates to a resource follow the pattern of first reading the current version and then sending an updated version. You can ensure that nobody has changed the resource since you read it using one of two standard HTTP mechanisms:

- The server can return a Last-Modified header when reading a resource that indicates the date and time when it was last modified. The client can then send an If-Unmodified-Since header in its update request with the same timestamp. If the resource has changed in the meantime, then the request will be rejected with a 412 Precondition Failed status.2 The main downside of Last-Modified headers is that they are limited to the nearest second, so are unable to detect changes occurring more frequently.
- Alternatively, the server can return an ETag (Entity Tag) header that should change whenever the resource changes as shown in figure 13.6. Typically, the ETag is either a version number or a cryptographic hash of the contents of the resource. The client can then send an If-Matches header containing the expected ETag when it performs an update. If the resource has changed in the meantime, then the ETag will be different and the server will respond with a 412 status-code and reject the request.

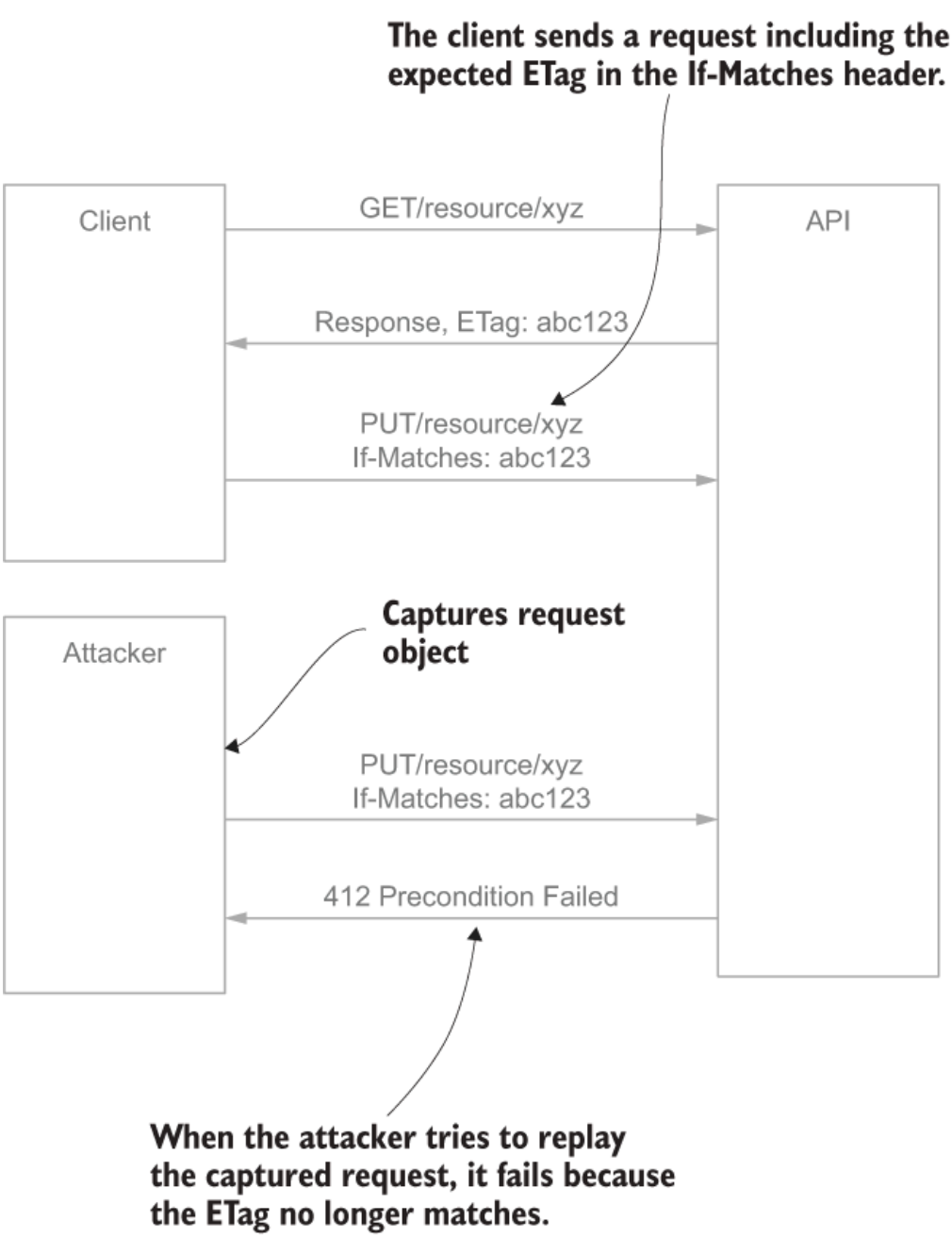

Figure 13.6 A client can prevaent replay of authenticated request objects by including an If-Matches header with the expected ETag of the resource. The update will modify the resource and cause the ETag to change, so if an attacker tries to replay the request, it will fail with a 412 Precondition Failed error.

WARNING Although a cryptographic hash can be appealing as an ETag, it does mean that the ETag will revert to a previous value if the content does. This allows an attacker to replay any old requests with a matching ETag. You can prevent this by including a counter or timestamp in the ETag calculation so that the ETag is always different even if the content is the same.

Listing 13.13 shows an example of updating a resource using a simple monotonic counter as the ETag. In this case, you can use an `AtomicInteger` class to hold the current ETag value, using the atomic `compareAndSet` method to increment the value if the If-Matches header

in the request matches the current value. Alternatively, you can store the ETag values for resources in the database alongside the data for a resource and update them in a transaction. If the If-Matches header in the request doesn't match the current value, then a 412 Precondition Failed header is returned; otherwise, the resource is updated and a new ETag is returned.

Listing 13.13 Using ETags to prevent replay

```
var etag = new AtomicInteger(42);
put("/test", (request, response) -> {
    var expectedEtag = parseInt(request.headers("If-Matches"));    ❶

    if (!etag.compareAndSet(expectedEtag, expectedEtag + 1)) {    ❶
        response.status(412);                                     ❷
        return null;                                              ❷
    }                                                             ❷

    System.out.println("Updating resource with new content: " +
        request.body());

    response.status(200);                                         ❸
    response.header("ETag", String.valueOf(expectedEtag + 1));    ❸
    response.type("text/plain");
    return "OK";
});
```

❶ Check the current ETag matches the one in the request.

❷ If not, return a 412 Precondition Failed response.

❸ Otherwise, return the new ETag after updating the resource.

The ETag mechanism can also be used to prevent replay of a PUT request that is intended to create a resource that doesn't yet exist. Because the resource doesn't exist, there is no existing ETag or Last-Modified date to include. An attacker could replay this message to overwrite a later version of the resource with the original content. To prevent this, you can include an If-None-Match header with the special value `*`, which tells the server to reject the request if there is any existing version of this resource at all.

TIP The Constrained Application Protocol (CoAP), often used for implementing REST APIs in constrained environments, doesn't support the Last-Modified or If-Unmodified-Since headers, but it does support ETags

along with If-Matches and If-None-Match. In CoAP, headers are known as options.

ENCODING HEADERS WITH END-TO-END SECURITY

As explained in chapter 12, in an end-to-end IoT application, a device may not be able to directly talk to the API in HTTP (or CoAP) but must instead pass an authenticated message through multiple intermediate proxies. Even if each proxy supports HTTP, the client may not trust those proxies not to interfere with the message if there isn't an end-to-end TLS connection. The solution is to encode the HTTP headers along with the request data into an encrypted request object, as shown in listing 13.14.

DEFINITION A request object is an API request that is encapsulated as a single data object that can be encrypted and authenticated as one element. The request object captures the data in the request as well as headers and other metadata required by the request.

In this example, the headers are encoded as a CBOR map, which is then combined with the request body and an indication of the expected HTTP method to create the overall request object. The entire object is then encrypted and authenticated using NaCl's `CryptoBox` functionality. OSCORE, discussed in section 13.1.4, is an example of an end-to-end protocol using request objects. The request objects in OSCORE are CoAP messages encrypted with COSE.

TIP Full source code for this example is provided in the GitHub repository accompanying the book at **http://mng.bz/QxWj**.

Listing 13.14 Encoding HTTP headers into a request object

```
var revisionEtag = "42";                                          (1)
var headers = CBORObject.NewMap()                                 (1)
        .Add("If-Matches", revisionEtag);                         (1)
var body = CBORObject.NewMap()
        .Add("foo", "bar")
        .Add("data", 12345);
var request = CBORObject.NewMap()
        .Add("method", "PUT")                                     (2)
        .Add("headers", headers)                                  (2)
        .Add("body", body);                                       (2)
var sent = CryptoBox.encrypt(clientKeys.getPrivate(),             (3)
        serverKeys.getPublic(), request.EncodeToBytes());         (3)
```

❶ Encode any required HTTP headers into CBOR.

❷ Encode the headers and body, along with the HTTP method, as a single object.

❸ Encrypt and authenticate the entire request object.

To validate the request, the API server should decrypt the request object and then verify that the headers and HTTP request method match those specified in the object. If they don't match, then the request should be rejected as invalid.

CAUTION You should always ensure the actual HTTP request headers match the request object rather than replacing them. Otherwise, an attacker can use the request object to bypass security filtering performed by Web Application Firewalls and other security controls. You should never let a request object change the HTTP method because many security checks in web browsers rely on it.

Listing 13.15 shows how to validate a request object in a filter for the Spark HTTP framework you've used in earlier chapters. The request object is decrypted using NaCl. Because this is authenticated encryption, the decryption process will fail if the request has been faked or tampered with. You should then verify that the HTTP method of the request matches the method included in the request object, and that any headers listed in the request object are present with the expected values. If any details don't match, then you should reject the request with an appropriate error code and message. Finally, if all checks pass, then you can store the decrypted request body in an attribute so that it can easily be retrieved without having to decrypt the message again.

Listing 13.15 Validating a request object

```
before((request, response) -> {
    var encryptedRequest = CryptoBox.fromString(request.body());     ❶
    var decrypted = encryptedRequest.decrypt(                        ❶
            serverKeys.getPrivate(), clientKeys.getPublic());        ❶
    var cbor = CBORObject.DecodeFromBytes(decrypted);                ❶
    if (!cbor.get("method").AsString()                               ❷
            .equals(request.requestMethod())) {                      ❷
        halt(403);                                                   ❷
    }                                                                ❷

    var expectedHeaders = cbor.get("headers");                       ❸
    for (var headerName : expectedHeaders.getKeys()) {               ❸
```

```
        if (!expectedHeaders.get(headerName).AsString()                    ❸
                .equals(request.headers(headerName.AsString()))) {         ❸
            halt(403);                                                     ❸
        }                                                                  ❸
    }                                                                      ❸

    request.attribute("decryptedRequest", cbor.get("body"));               ❹
});
```

❶ Decrypt the request object and decode it.

❷ Check that the HTTP method matches the request object.

❸ Check that any headers in the request object have their expected values.

❹ If all checks pass, then store the decrypted request body.

Pop quiz

3. Entity authentication requires which additional property on top of message authentication?
    1. Fuzziness
    2. Friskiness
    3. Funkiness
    4. Freshness
4. Which of the following are ways of ensuring authentication freshness? (There are multiple correct answers.)
    1. Deodorant
    2. Timestamps
    3. Unique nonces
    4. Challenge-response protocols
    5. Message authentication codes
5. Which HTTP header is used to ensure that the ETag of a resource matches an expected value?
    1. If-Matches
    2. Cache-Control
    3. If-None-Matches
    4. If-Unmodified-Since

The answers are at the end of the chapter.

13.3 OAuth2 for constrained environments

Throughout this book, OAuth2 has cropped up repeatedly as a common approach to securing APIs in many different environments. What started as a way to do delegated authorization in traditional web applications has expanded to encompass mobile apps, service-to-service APIs, and microservices. It should therefore come as little surprise that it is also being applied to securing APIs in the IoT. It's especially suited to consumer IoT applications in the home. For example, a smart TV may allow users to log in to streaming services to watch films or listen to music, or to view updates from social media streams. These are well-suited to OAuth2, because they involve a human delegating part of their authority to a device for a well-defined purpose.

DEFINITION A smart TV (or connected TV) is a television that is capable of accessing services over the internet, such as music or video streaming or social media APIs. Many other home entertainment devices are also now capable of accessing the internet and APIs are powering this transformation.

But the traditional approaches to obtain authorization can be difficult to use in an IoT environment for several reasons:

- The device may lack a screen, keyboard, or other capabilities needed to let a user interact with the authorization server to approve consent. Even on a more capable device such as a smart TV, typing in long usernames or passwords on a small remote control can be time-consuming and annoying for users. Section 13.2.1 discusses the device authorization grant that aims to solve this problem.
- Token formats and security mechanisms used by authorization servers are often heavily focused on web browser clients or mobile apps and are not suitable for more constrained devices. The ACE-OAuth framework discussed in section 13.2.2 is an attempt to adapt OAuth2 for such constrained environments.

DEFINITION ACE-OAuth (Authorization for Constrained Environments using OAuth2) is a framework specification that adapts OAuth2 for constrained devices.

13.3.1 The device authorization grant

The OAuth2 device authorization grant (RFC 8628, https://tools.ietf.org/html/rfc8628) allows devices that lack normal input and output capabilities to obtain access tokens from users. In the normal

OAuth2 flows discussed in chapter 7, the OAuth2 client would redirect the user to a web page on the authorization server (AS), where they can log in and approve access. This is not possible on many IoT devices because they have no display to show a web browser, and no keyboard, mouse, or touchscreen to let the user enter their details. The device authorization grant, or device flow as it is often called, solves this problem by letting the user complete the authorization on a second device, such as a laptop or mobile phone. Figure 13.7 shows the overall flow, which is described in more detail in the rest of this section.

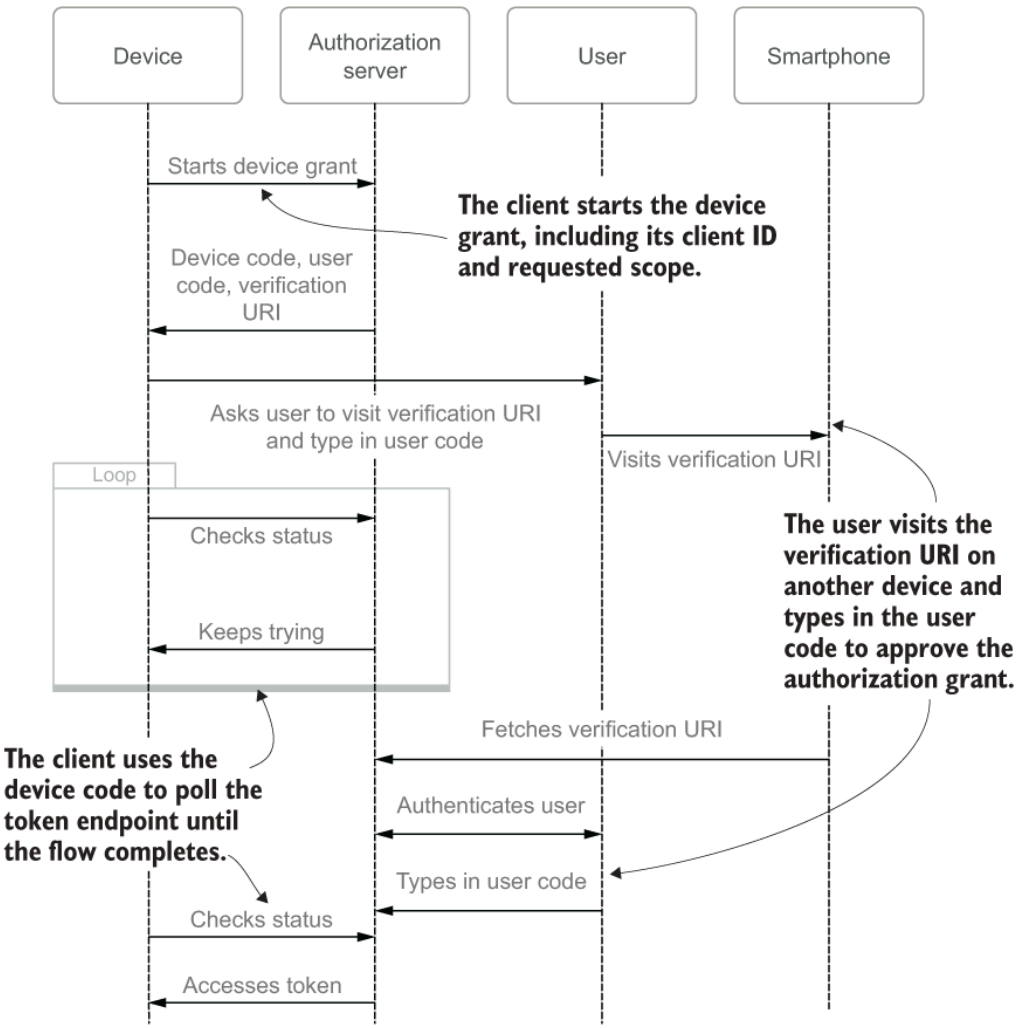

Figure 13.7 In the OAuth2 device authorization grant, the device first calls an endpoint on the AS to start the flow and receives a device code and short user code. The device asks the user to navigate to the AS on a separate device, such as a smartphone. After the user authenticates, they type in the user code and approve the request. The device polls the AS in the background using the device code until the flow completes. If the user approved the request, then the device receives an access token the next time it polls the AS.

To initiate the flow, the device first makes a POST request to a new device authorization endpoint at the AS, indicating the scope of the access token

it requires and authenticating using its client credentials. The AS returns three details in the response:

- A device code, which is a bit like an authorization code from chapter 7 and will eventually be exchanged for an access token after the user authorizes the request. This is typically an unguessable random string.
- A user code, which is a shorter code designed to be manually entered by the user when they approve the authorization request.
- A verification URI where the user should go to type in the user code to approve the request. This will typically be a short URI if the user will have to manually type it in on another device.

Listing 13.16 shows how to begin a device grant authorization request from Java. In this example, the device is a public client and so you only need to supply the `client_id` and `scope` parameters on the request. If your device is a confidential client, then you would also need to supply client credentials using HTTP Basic authentication or another client authentication method supported by your AS. The parameters are URL-encoded as they are for other OAuth2 requests. The AS returns a 200 OK response if the request is successful, with the device code, user code, and verification URI in JSON format. Navigate to src/main/java/com/manning/apisecurityinaction and create a new file named DeviceGrantClient.java. Create a new public class in the file with the same name and add the method from listing 13.16 to the file. You'll need the following imports at the top of the file:

```
import org.json.JSONObject;
import java.net.*;
import java.net.http.*;
import java.net.http.HttpRequest.BodyPublishers;
import java.net.http.HttpResponse.BodyHandlers;
import java.util.concurrent.TimeUnit;
import static java.nio.charset.StandardCharsets.UTF_8;
```

Listing 13.16 Starting a device authorization grant flow

```
private static final HttpClient httpClient = HttpClient.newHttpClient();

private static JSONObject beginDeviceAuthorization(
        String clientId, String scope) throws Exception {
    var form = "client_id=" + URLEncoder.encode(clientId, UTF_8) +      1
            "&scope=" + URLEncoder.encode(scope, UTF_8);                1
    var request = HttpRequest.newBuilder()                              1
            .header("Content-Type",                                     1
```

```
                "application/x-www-form-urlencoded")                    ❶
            .uri(URI.create(                                            ❶
                "https://as.example.com/device_authorization"))         ❶
            .POST(BodyPublishers.ofString(form))                        ❶
            .build();                                                   ❶
    var response = httpClient.send(request, BodyHandlers.ofString());   ❶

    if (response.statusCode() != 200) {                                 ❷
        throw new RuntimeException("Bad response from AS: " +           ❷
            response.body());                                           ❷
    }                                                                   ❷
    return new JSONObject(response.body());                             ❸
}
```

❶ Encode the client ID and scope as form parameters and POST them to the device endpoint.

❷ If the response is not 200 OK, then an error occurred.

❸ Otherwise, parse the response as JSON.

The device that initiated the flow communicates the verification URI and user code to the user but keeps the device code secret. For example, the device might be able to display a QR code (figure 13.8) that the user can scan on their phone to open the verification URI, or the device might communicate directly with the user's phone over a local Bluetooth connection. To approve the authorization, the user opens the verification URI on their other device and logs in. They then type in the user code and can either approve or deny the request after seeing details of the scopes requested.

Figure 13.8 A QR code is a way to encode a URI that can be easily scanned by a mobile phone with a camera. This can be used to display the verification URI used in the OAuth2 device authorization grant. If you scan this QR code on your phone, it will take you to the home page for this book.

TIP The AS may also return a `verification_uri_complete` field that combines the verification URI with the user code. This allows the user to just follow the link without needing to manually type in the code.

The original device that requested authorization is not notified that the flow has completed. Instead, it must periodically poll the access token endpoint at the AS, passing in the device code it received in the initial request as shown in listing 13.17. This is the same access token endpoint used in the other OAuth2 grant types discussed in chapter 7, but you set the `grant_type` parameter to

```
urn:ietf:params:oauth:grant-type:device_code
```

to indicate that the device authorization grant is being used. The client also includes its client ID and the device code itself. If the client is confidential, it must also authenticate using its client credentials, but this example is using a public client. Open the DeviceGrantClient.java file again and add the method from the following listing.

Listing 13.17 Checking status of the authorization request

```
private static JSONObject pollAccessTokenEndpoint(
        String clientId, String deviceCode) throws Exception {
```

```
    var form = "client_id=" + URLEncoder.encode(clientId, UTF_8) +      ❶
        "&grant_type=urn:ietf:params:oauth:grant-type:device_code" +    ❶
        "&device_code=" + URLEncoder.encode(deviceCode, UTF_8);         ❶

    var request = HttpRequest.newBuilder()                              ❷
            .header("Content-Type",                                     ❷
                    "application/x-www-form-urlencoded")                ❷
            .uri(URI.create("https://as.example.com/access_token"))     ❷
            .POST(BodyPublishers.ofString(form))                        ❷
            .build();                                                   ❷
    var response = httpClient.send(request, BodyHandlers.ofString());   ❷
    return new JSONObject(response.body());                             ❸
}
```

❶ Encode the client ID and device code along with the device_code grant type URI.

❷ Post the parameters to the access token endpoint at the AS.

❸ Parse the response as JSON.

If the user has already approved the request, then the AS will return an access token, optional refresh token, and other details as it does for other access token requests you learned about in chapter 7. Otherwise, the AS returns one of the following status codes:

- `authorization_pending` indicates that the user hasn't yet approved or denied the request and the device should try again later.
- `slow_down` indicates that the device is polling the authorization endpoint too frequently and should increase the interval between requests by 5 seconds. An AS may revoke authorization if the device ignores this code and continues to poll too frequently.
- `access_denied` indicates that the user refused the request.
- `expired_token` indicates that the device code has expired without the request being approved or denied. The device will have to initiate a new flow to obtain a new device code and user code.

Listing 13.18 shows how to handle the full authorization flow in the client building on the previous methods. Open the DeviceGrantClient.java file again and add the main method from the listing.

TIP If you want to test the client, the ForgeRock Access Management (AM) product supports the device authorization grant. Follow the instructions in appendix A to set up the server and then the instructions in **http://mng.bz/ X0W6** to configure the device authorization grant. AM im-

plements an older draft version of the standard and requires an extra `response_type=device_code` parameter on the initial request to begin the flow.

Listing 13.18 The full device authorization grant flow

```
public static void main(String... args) throws Exception {
    var clientId = "deviceGrantTest";
    var scope = "a b c";

    var json = beginDeviceAuthorization(clientId, scope);          (1)
    var deviceCode = json.getString("device_code");                (1)
    var interval = json.optInt("interval", 5);                     (1)
    System.out.println("Please open " +                            (2)
        json.getString("verification_uri"));                       (2)
    System.out.println("And enter code:\n\t" +                     (2)
        json.getString("user_code"));                              (2)

    while (true) {                                                 (3)
        Thread.sleep(TimeUnit.SECONDS.toMillis(interval));         (3)
        json = pollAccessTokenEndpoint(clientId, deviceCode);      (3)
        var error = json.optString("error", null);
        if (error != null) {
            switch (error) {
                case "slow_down":                                  (4)
                    System.out.println("Slowing down");            (4)
                    interval += 5;                                 (4)
                    break;
                case "authorization_pending":                      (5)
                    System.out.println("Still waiting!");          (5)
                    break;
                default:
                    System.err.println("Authorization failed: " + error);
                    System.exit(1);
                    break;
            }
        } else {
            System.out.println("Access token: " +                  (6)
                json.getString("access_token"));                   (6)
            break;
        }
    }
}
```

❶ Start the authorization process and store the device code and poll interval.

❷ Display the verification URI and user code to the user.

❸ Poll the access token endpoint with the device code according to the poll interval.

❹ If the AS tells you to slow down, then increase the poll interval by 5 seconds.

❺ Otherwise, keep waiting until a response is received.

❻ The AS will return an access token when the authorization is complete.

13.3.2 ACE-OAuth

The Authorization for Constrained Environments (ACE) working group at the IETF is working to adapt OAuth2 for IoT applications. The main output of this group is the definition of the ACE-OAuth framework (**http://mng.bz/yr4q**), which describes how to perform OAuth2 authorization requests over CoAP instead of HTTP and using CBOR instead of JSON for requests and responses. COSE is used as a standard format for access tokens and can also be used as a proof of possession (PoP) scheme to secure tokens against theft (see section 11.4.6 for a discussion of PoP tokens). COSE can also be used to protect API requests and responses themselves, using the OSCORE framework you saw in section 13.1.4.

At the time of writing, the ACE-OAuth specifications are still under development but are approaching publication as standards. The main framework describes how to adapt OAuth2 requests and responses to use CBOR, including support for the authorization code, client credentials, and refresh token grants.[3] The token introspection endpoint is also supported, using CBOR over CoAP, providing a standard way for resource servers to check the status of an access token.

Unlike the original OAuth2, which used bearer tokens exclusively and has only recently started supporting proof-of-possession (PoP) tokens, ACE-OAuth has been designed around PoP from the start. Issued access tokens are bound to a cryptographic key and can only be used by a client that can prove possession of this key. This can be accomplished with either symmetric or public key cryptography, providing support for a wide range of device capabilities. APIs can discover the key associated with a device either through token introspection or by examining the access to-

ken itself, which is typically in CWT format. When public key cryptography is used, the token will contain the public key of the client, while for symmetric key cryptography, the secret key will be present in COSE-encrypted form, as described in RFC 8747 (**https://datatracker.ietf.org/doc/html/rfc8747**).

13.4 Offline access control

Many IoT applications involve devices operating in environments where they may not have a permanent or reliable connection to central authorization services. For example, a connected car may be driven through long tunnels or to remote locations where there is no signal. Other devices may have limited battery power and so want to avoid making frequent network requests. It's usually not acceptable for a device to completely stop functioning in this case, so you need a way to perform security checks while the device is disconnected. This is known as offline authorization. Offline authorization allows devices to continue accepting and producing API requests to other local devices and users until the connection is restored.

DEFINITION Offline authorization allows a device to make local security decisions when it is disconnected from a central authorization server.

Allowing offline authorization often comes with increased risks. For example, if a device can't check with an OAuth2 authorization server whether an access token is valid, then it may accept a token that has been revoked. This risk must be balanced against the costs of downtime if devices are offline and the appropriate level of risk determined for your application. You may want to apply limits to what operations can be performed in offline mode or enforce a time limit for how long devices will operate in a disconnected state.

13.4.1 Offline user authentication

Some devices may never need to interact with a user at all, but for some IoT applications this is a primary concern. For example, many companies now operate smart lockers where goods ordered online can be delivered for later collection. The user arrives at a later time and uses an app on their smartphone to send a request to open the locker. Devices used in industrial IoT deployments may work autonomously most of the time, but occasionally need servicing by a human technician. It would be frustrating for the user if they couldn't get their latest purchase because the locker can't connect to a cloud service to authenticate them, and a techni-

cian is often only involved when something has gone wrong, so you shouldn't assume that network services will be available in this situation.

The solution is to make user credentials available to the device so that it can locally authenticate the user. This doesn't mean that the user's password hash should be transmitted to the device, because this would be very dangerous: an attacker that intercepted the hash could perform an offline dictionary attack to try to recover the password. Even worse, if the attacker compromised the device, then they could just intercept the password directly as the user types it. Instead, the credential should be short-lived and limited to just the operations needed to access that device. For example, a user can be sent a one-time code that they can display on their smartphone as a QR code that the smart locker can scan. The same code is hashed and sent to the device, which can then compare the hash to the QR code and if they match, it opens the locker, as shown in figure 13.9.

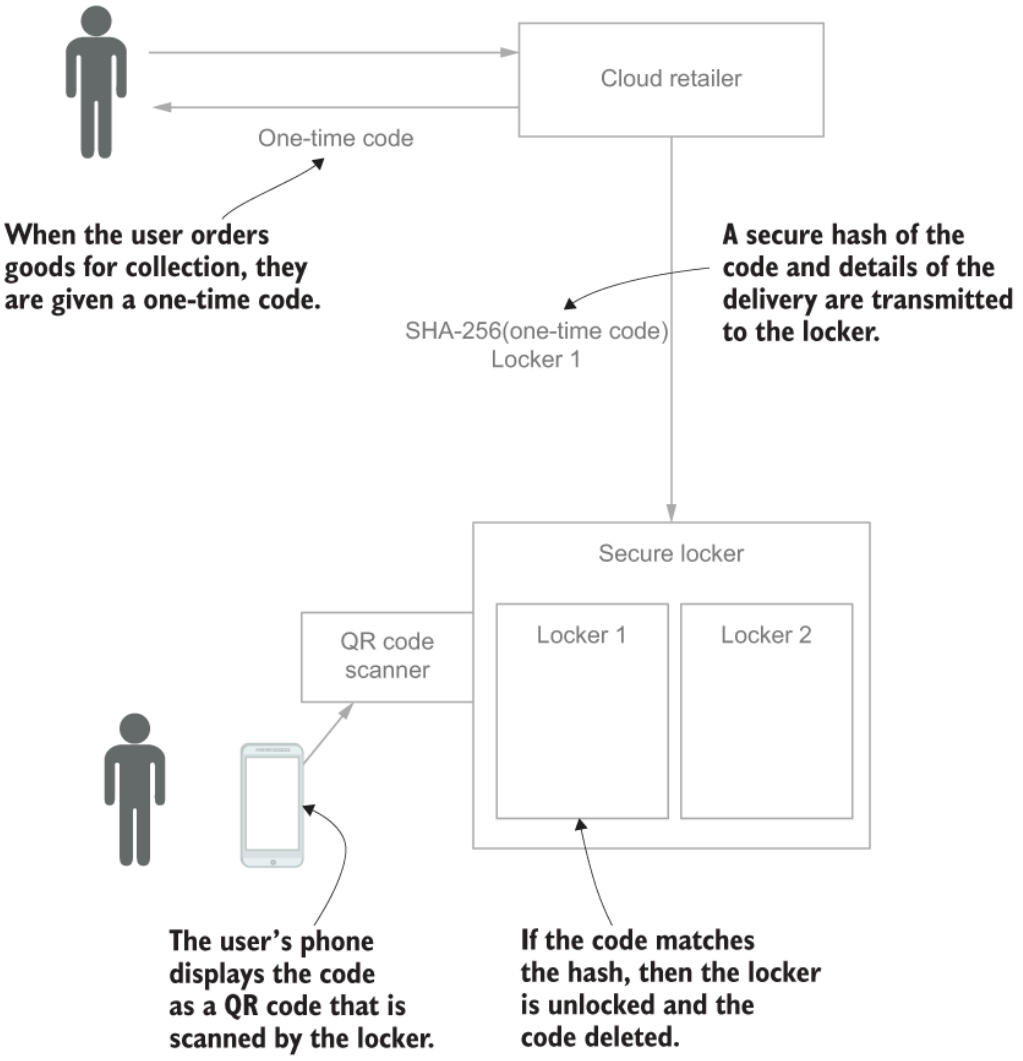

Figure 13.9 One-time codes can be periodically sent to an IoT device such as a secure locker. A secure hash of the code is stored locally, allowing the locker to authenticate users even if it cannot contact the cloud service at that time.

For this approach to work, the device must be online periodically to download new credentials. A signed, self-contained token format can overcome this problem. Before leaving to service a device in the field, the technician can authenticate to a central authorization server and receive an OAuth2 access token or OpenID Connect ID token. This token can include a public key or a temporary credential that can be used to locally authenticate the user. For example, the token can be bound to a TLS client certificate as described in chapter 11, or to a key using CWT PoP tokens mentioned in section 13.3.2. When the technician arrives to service the device, they can present the access token to access device APIs over a local connection, such as Bluetooth Low-Energy (BLE). The device API can verify the signature on the access token and check the scope, issuer, audience, expiry time, and other details. If the token is valid, then the embedded credentials can be used to authenticate the user locally to allow access according to the conditions attached to the token.

13.4.2 Offline authorization

Offline authentication solves the problem of identifying users without a direct connection to a central authentication service. In many cases, device access control decisions are simple enough to be hard-coded based on pre-existing trust relationships. For example, a device may allow full access to any user that has a credential issued by a trusted source and deny access to everybody else. But not all access control policies are so simple, and access may depend on a range of dynamic factors and changing conditions. Updating complex policies for individual devices becomes difficult as the number of devices grows. As you learned in chapter 8, access control policies can be centralized using a policy engine that is accessed via its own API. This simplifies management of device policies, but again can lead to problems if the device is offline.

The solutions are similar to the solutions to offline authentication described in the last section. The most basic solution is for the device to periodically download the latest policies in a standard format such as XACML, discussed in chapter 8. The device can then make local access control decisions according to the policies. XACML is a complex XML-based format, so you may want to consider a more lightweight policy language encoded in CBOR or another compact format, but I am not aware of any standards for such a language.

Self-contained access token formats can also be used to permit offline authorization. A simple example is the scope included in an access token, which allows an offline device to determine which API operations a client should be allowed to call. More complex conditions can be encoded as

caveats using a macaroon token format, discussed in chapter 9. Suppose that you used your smartphone to book a rental car. An access token in macaroon format is sent to your phone, allowing you to unlock the car by transmitting the token to the car over BLE just like in the example at the end of section 13.4.1. You later drive the car to an evening event at a luxury hotel in a secluded location with no cellular network coverage. The hotel offers valet parking, but you don't trust the attendant, so you only want to allow them limited ability to drive the expensive car you hired. Because your access token is a macaroon, you can simply append caveats to it restricting the token to expire in 10 minutes and only allow the car to be driven in a quarter-mile radius of the hotel.

Macaroons are a great solution for offline authorization because caveats can be added by devices at any time without any coordination and can then be locally verified by devices without needing to contact a central service. Third-party caveats can also work well in an IoT application, because they require the client to obtain proof of authorization from the third-party API. This authorization can be obtained ahead of time by the client and then verified by the device by checking the discharge macaroon, without needing to directly contact the third party.

Pop quiz

6. Which OAuth authorization grant can be used on devices that lack user input features?
    1. The client credentials grant
    2. The authorization code grant
    3. The device authorization grant
    4. The resource owner password grant

The answer is at the end of the chapter.

Answers to pop quiz questions

1. False. The PSK can be any sequence of bytes and may not be a valid string.
2. d. the ID is authenticated during the handshake so you should only trust it after the handshake completes.
3. d. Entity authentication requires that messages are fresh and haven't been replayed.
4. b, c, and d.
5. a.
6. c. The device authorization grant.

Summary

- Devices can be identified using credentials associated with a device profile. These credentials could be an encrypted pre-shared key or a certificate containing a public key for the device.
- Device authentication can be done at the transport layer, using facilities in TLS, DTLS, or other secure protocols. If there is no end-to-end secure connection, then you'll need to implement your own authentication protocol.
- End-to-end device authentication must ensure freshness to prevent replay attacks. Freshness can be achieved with timestamps, nonces, or challenge-response protocols. Preventing replay requires storing per-device state, such as a monotonically increasing counter or recently used nonces.
- REST APIs can prevent replay by making use of authenticated request objects that contain an ETag that identifies a specific version of the resource being acted on. The ETag should change whenever the resource changes to prevent replay of previous requests.
- The OAuth2 device grant can be used by devices with no input capability to obtain access tokens authorized by a user. The ACE-OAuth working group at the IETF is developing specifications that adapt OAuth2 for use in constrained environments.
- Devices may not always be able to connect to central cloud services. Offline authentication and access control allow devices to continue to operate securely when disconnected. Self-contained token formats can include credentials and policies to ensure authority isn't exceeded, and proof-of-possession (PoP) constraints can be used to provide stronger security guarantees.

---

1. One of the few drawbacks of the NaCl `CryptoBox` and `SecretBox` APIs is that they don't allow authenticated associated data.

2. If the server can determine that the current state of the resource happens to match the requested state, then it can also return a success status code as if the request succeeded in this case. But in this case the request is really idempotent anyway.

3. Strangely, the device authorization grant is not yet supported.