12 Securing IoT communications

This chapter covers

- Securing IoT communications with Datagram TLS
- Choosing appropriate cryptographic algorithms for constrained devices
- Implementing end-to-end security for IoT APIs
- Distributing and managing device keys

So far, all the APIs you've looked at have been running on servers in the safe confines of a datacenter or server room. It's easy to take the physical security of the API hardware for granted, because the datacenter is a secure environment with restricted access and decent locks on the doors. Often only specially vetted staff are allowed into the server room to get close to the hardware. Traditionally, even the clients of an API could be assumed to be reasonably secure because they were desktop PCs installed in an office environment. This has rapidly changed as first laptops and then smartphones have moved API clients out of the office environment. The internet of things (IoT) widens the range of environments even further, especially in industrial or agricultural settings where devices may be deployed in remote environments with little physical protection or monitoring. These IoT devices talk to APIs in messaging services to stream sensor data to the cloud and provide APIs of their own to allow physical actions to be taken, such as adjusting machinery in a water treatment plant or turning off the lights in your home or office. In this chapter, you'll see how to secure the communications of IoT devices when talking to each other and to APIs in the cloud. In chapter 13, we'll discuss how to secure APIs provided by devices themselves.

DEFINITION The internet of things (IoT) is the trend for devices to be connected to the internet to allow easier management and communication. Consumer IoT refers to personal devices in the home being connected to the internet, such as a refrigerator that automatically orders more beer when you run low. IoT techniques are also applied in industry under the name industrial IoT (IIoT).

12.1 Transport layer security

In a traditional API environment, securing the communications between a client and a server is almost always based on TLS. The TLS connection between the two parties is likely to be end-to-end (or near enough) and using strong authentication and encryption algorithms. For example, a client making a request to a REST API can make a HTTPS connection directly to that API and then largely assume that the connection is secure. Even when the connection passes through one or more proxies, these typically just set up the connection and then copy encrypted bytes from one socket to another. In the IoT world, things are more complicated for many reasons:

- The IoT device may be constrained, reducing its ability to execute the public key cryptography used in TLS. For example, the device may have limited CPU power and memory, or may be operating purely on battery power that it needs to conserve.
- For efficiency, devices often use compact binary formats and low-level networking based on UDP rather than high-level TCP-based protocols such as HTTP and TLS.
- A variety of protocols may be used to transmit a single message from a device to its destination, from short-range wireless protocols such as Bluetooth Low Energy (BLE) or Zigbee, to messaging protocols like MQTT or XMPP. Gateway devices can translate messages from one protocol to another, as shown in figure 12.1, but need to decrypt the protocol messages to do so. This prevents a simple end-to-end TLS connection being used.
- Some commonly used cryptographic algorithms are difficult to implement securely or efficiently on devices due to hardware constraints or new threats from physical attackers that are less applicable to server-side APIs.

DEFINITION A constrained device has significantly reduced CPU power, memory, connectivity, or energy availability compared to a server or traditional API client machine. For example, the memory available to a device may be measured in kilobytes compared to the gigabytes often now available to most servers and even smartphones. RFC 7228 (**https://tools.ietf.org/html/rfc7228**) describes common ways that devices are constrained.

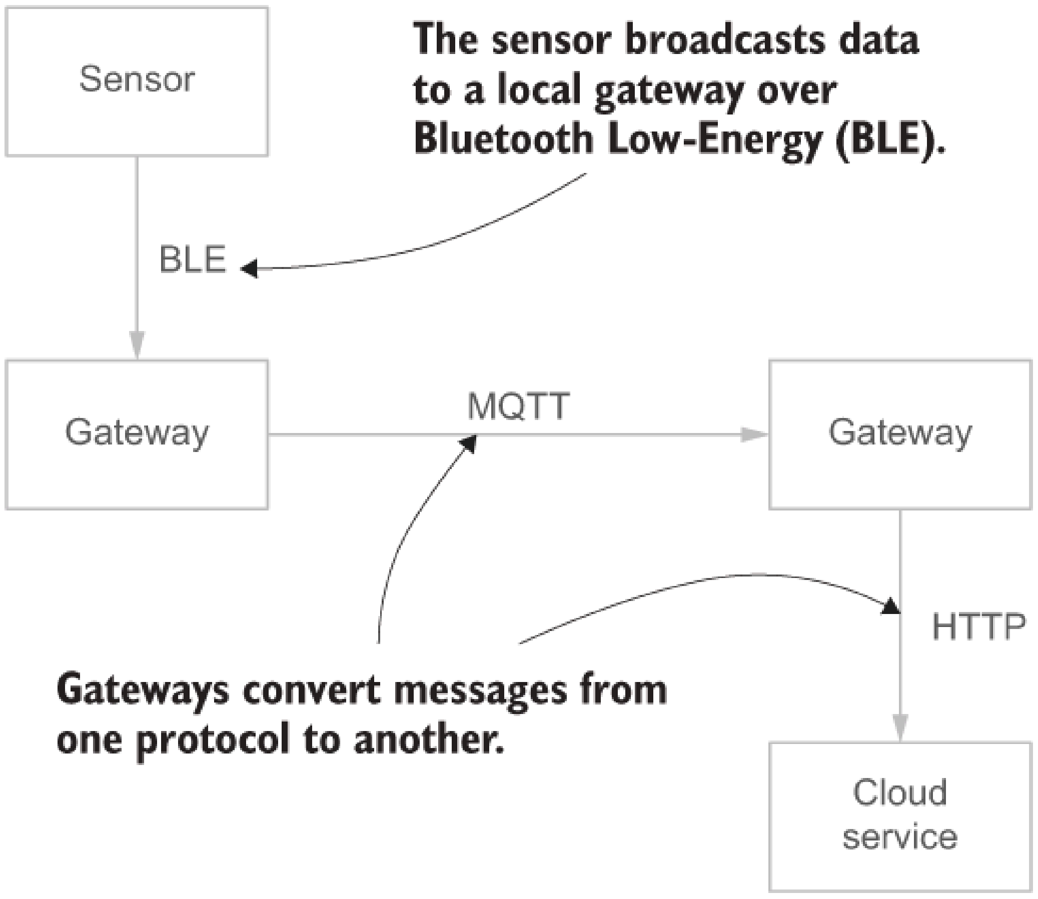

Sensor
The sensor broadcasts data to a local gateway over Bluetooth Low-Energy (BLE).
BLE
Gateway
MQTT
Gateway
HTTP
Gateways convert messages from one protocol to another.
Cloud service

Figure 12.1 Messages from IoT devices are often translated from one protocol to another. The original device may use low-power wireless networking such as Bluetooth Low-Energy (BLE) to communicate with a local gateway that retransmits messages using application protocols such as MQTT or HTTP.

In this section, you'll learn about how to secure IoT communications at the transport layer and the appropriate choice of algorithms for constrained devices.

TIP There are several TLS libraries that are explicitly designed for IoT applications, such as ARM's mbedTLS (**https://tls.mbed.org**), WolfSSL (**https://www .wolfssl.com**), and BearSSL (**https://bearssl.org**).

12.1.1 Datagram TLS

TLS is designed to secure traffic sent over TCP (Transmission Control Protocol), which is a reliable stream-oriented protocol. Most application protocols in common use, such as HTTP, LDAP, or SMTP (email), all use TCP and so can use TLS to secure the connection. But a TCP implementation has some downsides when used in constrained IoT devices, such as the following:

- A TCP implementation is complex and requires a lot of code to implement correctly. This code takes up precious space on the device, reducing the amount of code available to implement other functions.
- TCP's reliability features require the sending device to buffer messages until they have been acknowledged by the receiver, which increases storage requirements. Many IoT sensors produce continuous streams of real-time data, for which it doesn't make sense to retransmit lost messages because more recent data will already have replaced it.
- A standard TCP header is at least 16 bytes long, which can add quite a lot of overhead to short messages.
- TCP is unable to use features such as multicast delivery that allow a single message to be sent to many devices at once. Multicast can be much more efficient than sending messages to each device individually.
- IoT devices often put themselves into sleep mode to preserve battery power when not in use. This causes TCP connections to terminate and requires an expensive TCP handshake to be performed to re-establish the connection when the device wakes. Alternatively, the device can periodically send keep-alive messages to keep the connection open, at the cost of increased battery and bandwidth usage.

Many protocols used in the IoT instead opt to build on top of the lower-level User Datagram Protocol (UDP), which is much simpler than TCP but provides only connectionless and unreliable delivery of messages. For example, the Constrained Application Protocol (CoAP), provides an alternative to HTTP for constrained devices and is based on UDP. To protect these protocols, a variation of TLS known as Datagram TLS (DTLS) has been developed.[1]

DEFINITION Datagram Transport Layer Security (DTLS) is a version of TLS designed to work with connectionless UDP-based protocols rather than TCP-based ones. It provides the same protections as TLS, except that packets may be reordered or replayed without detection.

Recent DTLS versions correspond to TLS versions; for example, DTLS 1.2 corresponds to TLS 1.2 and supports similar cipher suites and extensions. At the time of writing, DTLS 1.3 is just being finalized, which corresponds to the recently standardized TLS 1.3.

QUIC

A middle ground between TCP and UDP is provided by Google's QUIC protocol (Quick UDP Internet Connections; https://en.wikipedia.org/wiki/QUIC), which will form the basis of the next version of HTTP: HTTP/3. QUIC layers on top of UDP but provides many of the same reliability and congestion control features as TCP. A key feature of QUIC is that it integrates TLS 1.3 directly into the transport protocol, reducing the overhead of the TLS handshake and ensuring that low-level protocol features also benefit from security protections. Google has already deployed QUIC into production, and around 7% of Internet traffic now uses the protocol.

QUIC was originally designed to accelerate Google's traditional web server HTTPS traffic, so compact code size was not a primary objective. However, the protocol can offer significant advantages to IoT devices in terms of reduced network usage and low-latency connections. Early experiments such as an analysis from Santa Clara University (http://mng.bz/X0WG) and another by NetApp (https://eggert.org/papers/ 2020-ndss-quic-iot.pdf) suggest that QUIC can provide significant savings in an IoT context, but the protocol has not yet been published as a final standard. Although not yet achieving widespread adoption in IoT applications, it's likely that QUIC will become increasingly important over the next few years.
Although Java supports DTLS, it only does so in the form of the low-level `SSLEngine` class, which implements the raw protocol state machine. There is no equivalent of the high-level `SSLSocket` class that is used by normal (TCP-based) TLS, so you must do some of the work yourself. Libraries for higher-level protocols such as CoAP will handle much of this for you, but because there are so many protocols used in IoT applications, in the next few sections you'll learn how to manually add DTLS to a UDP-based protocol.

NOTE The code examples in this chapter continue to use Java for consistency. Although Java is a popular choice on more capable IoT devices and gateways, programming constrained devices is more often performed in C or another language with low-level device support. The advice on secure configuration of DTLS and other protocols in this chapter is applicable to all languages and DTLS libraries. Skip ahead to section 12.1.2 if you are not using Java.

To begin a DTLS handshake in Java, you first create an `SSLContext` object, which indicates how to authenticate the connection. For a client connection, you initialize the context exactly like you did in section 7.4.2 when securing the connection to an OAuth2 authorization server, as shown in listing 12.1. First, obtain an `SSLContext` for DTLS by calling `SSLContext.getInstance("DTLS")`. This will return a context that allows DTLS connections with any supported protocol version (DTLS 1.0 and DTLS 1.2 in Java 11). You can then load the certificates of trusted certificate authorities (CAs) and use this to initialize a `TrustManagerFactory`, just as you've done in previous chapters. The `TrustManagerFactory` will be used by Java to determine if the server's certificate is trusted. In this, case you can use the as.example.com.ca.p12 file that you created in chapter 7 containing the mkcert CA certificate. The PKIX (Public Key Infrastructure with X.509) trust manager factory algorithm should be used. Finally, you can initialize the `SSLContext` object, passing in the trust managers from the factory, using the `SSLContext.init()` method. This method takes three arguments:

- An array of `KeyManager` objects, which are used if performing client certificate authentication (covered in chapter 11). Because this example doesn't use client certificates, you can leave this null.
- The array of `TrustManager` objects obtained from the `TrustManagerFactory`.
- An optional `SecureRandom` object to use when generating random key material and other data during the TLS handshake. You can leave this `null` in most cases to let Java choose a sensible default.

Create a new file named DtlsClient.java in the src/main/com/manning/apisecurityinaction folder and type in the contents of the listing.

NOTE The examples in this section assume you are familiar with UDP network programming in Java. See http://mng.bz/yr4G for an introduction.

Listing 12.1 The client SSLContext

```
package com.manning.apisecurityinaction;

import javax.net.ssl.*;
import java.io.FileInputStream;
import java.nio.file.*;
import java.security.KeyStore;
import org.slf4j.*;
import static java.nio.charset.StandardCharsets.UTF_8;

public class DtlsClient {
    private static final Logger logger =
        LoggerFactory.getLogger(DtlsClient.class);
    private static SSLContext getClientContext() throws Exception {
        var sslContext = SSLContext.getInstance("DTLS");                   1

        var trustStore = KeyStore.getInstance("PKCS12");                   2
        trustStore.load(new FileInputStream("as.example.com.ca.p12"),      2
                "changeit".toCharArray());                                 2

        var trustManagerFactory = TrustManagerFactory.getInstance(         3
                "PKIX");                                                   3
        trustManagerFactory.init(trustStore);                              3

        sslContext.init(null, trustManagerFactory.getTrustManagers(),      4
                null);                                                     4
        return sslContext;
    }
}
```

1 Create an SSLContext for DTLS.

2 Load the trusted CA certificates as a keystore.

3 Initialize a TrustManagerFactory with the trusted certificates.

4 Initialize the SSLContext with the trust manager.

After you've created the `SSLContext`, you can use the `createEngine()` method on it to create a new `SSLEngine` object. This is the low-level protocol implementation that is normally hidden by higher-level protocol libraries like the `HttpClient` class you used in chapter 7. For a client, you should pass the address and port of the server to the method when creating the engine and configure the engine to perform the client side of the DTLS handshake by calling `setUseClientMode(true )`, as shown in the following example.

NOTE You don't need to type in this example (and the other `SSLEngine` examples), because I have provided a wrapper class that hides some of this complexity and demonstrates correct use of the `SSLEngine`. See http://mng.bz/Mo27. You'll use that class in the example client and server shortly.

```
var address = InetAddress.getByName("localhost");
var engine = sslContext.createEngine(address, 54321);
engine.setUseClientMode(true);
```

You should then allocate buffers for sending and receiving network packets, and for holding application data. The `SSLSession` associated with an engine has methods that provide hints for the correct size of these buffers, which you can query to ensure you allocate enough space, as shown in the following example code (again, you don't need to type this in):

```
var session = engine.getSession();                                      ❶
var receiveBuffer =                                                     ❷
    ByteBuffer.allocate(session.getPacketBufferSize());                 ❷
var sendBuffer =                                                        ❷
    ByteBuffer.allocate(session.getPacketBufferSize());                 ❷
var applicationData =                                                   ❷
    ByteBuffer.allocate(session.getApplicationBufferSize());            ❷
```

❶ Retrieve the SSLSession from the engine.

❷ Use the session hints to correctly size the data buffers.

These initial buffer sizes are hints, and the engine will tell you if they need to be resized as you'll see shortly. Data is moved between buffers by using the following two method calls, also illustrated in figure 12.2:

- `sslEngine.wrap(appData, sendBuf)` causes the `SSLEngine` to consume any waiting application data from the `appData` buffer and write one or more DTLS packets into the network `sendBuf` that can then be sent to the other party.
- `sslEngine.unwrap(recvBuf, appData)` instructs the `SSLEngine` to consume received DTLS packets from the `recvBuf` and output any decrypted application data into the `appData` buffer.

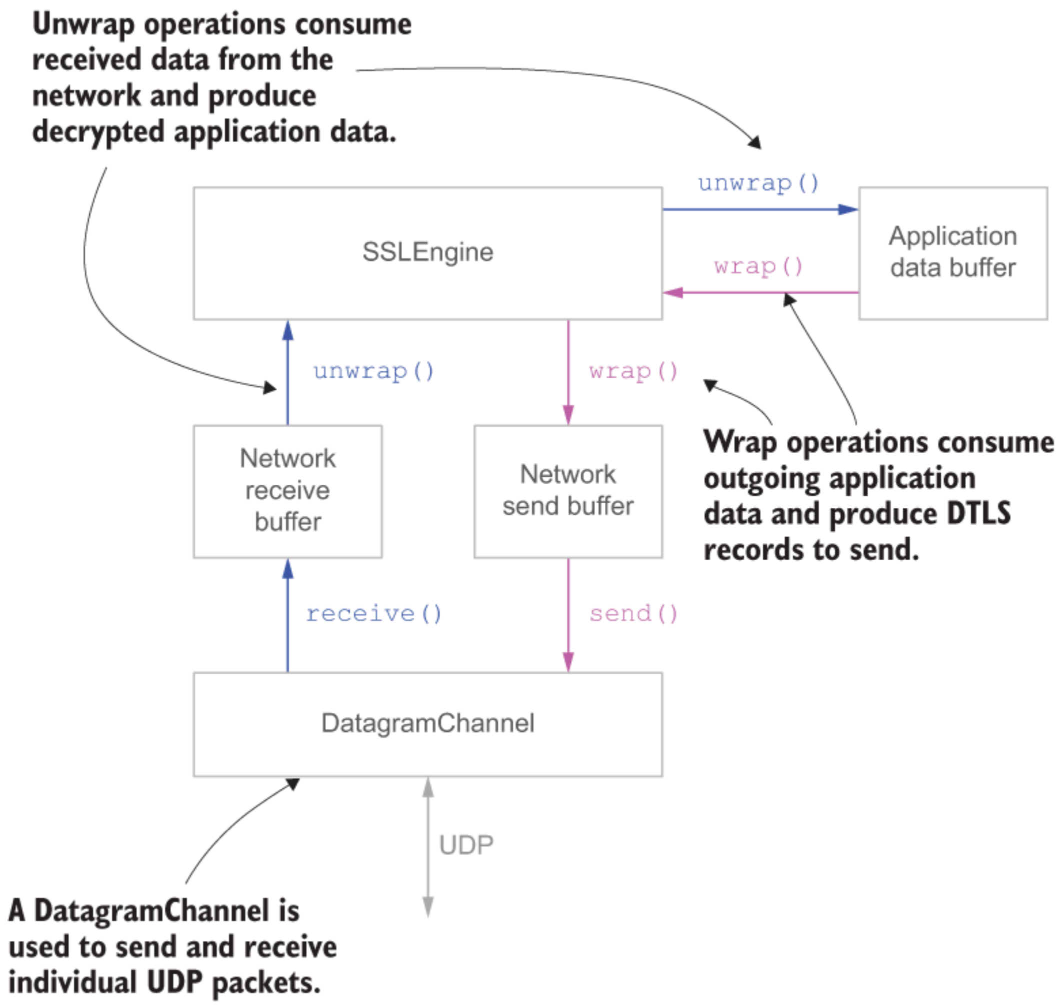

Unwrap operations consume received data from the network and produce decrypted application data.
unwrap()
SSLEngine
Application data buffer
wrap()
unwrap()
wrap()
Network receive buffer
Network send buffer
Wrap operations consume outgoing application data and produce DTLS records to send.
receive()
send()
DatagramChannel
UDP
A DatagramChannel is used to send and receive individual UDP packets.

Figure 12.2 The SSLEngine uses two methods to move data between the application and network buffers: `wrap()` consumes application data to send and writes DTLS packets into the send buffer, while `unwrap()` consumes data from the receive buffer and writes unencrypted application data back into the application buffer.

To start the DTLS handshake, call `sslEngine.beginHandshake ()`. Rather than blocking until the handshake is complete, this configures the engine to expect a new DTLS handshake to begin. Your application code is then responsible for polling the engine to determine the next action to take and sending or receiving UDP messages as indicated by the engine.

To poll the engine, you call the `sslEngine.getHandshakeStatus()` method, which returns one of the following values, as shown in figure 12.3:

- `NEED_UNWRAP` indicates that the engine is waiting to receive a new message from the server. Your application code should call the `receive()` method on its UDP `DatagramChannel` to receive a packet from the server, and then call the `SSLEngine.unwrap()` method passing in the data it received.

- `NEED_UNWRAP_AGAIN` indicates that there is remaining input that still needs to be processed. You should immediately call the `unwrap()` method again with an empty input buffer to process the message. This can happen if multiple DTLS records arrived in a single UDP packet.
- `NEED_WRAP` indicates that the engine needs to send a message to the server. The application should call the `wrap()` method with an output buffer that will be filled with the new DTLS message, which your application should then send to the server.
- `NEED_TASK` indicates that the engine needs to perform some (potentially expensive) processing, such as performing cryptographic operations. You can call the `getDelegatedTask()` method on the engine to get one or more `Runnable` objects to execute. The method returns null when there are no more tasks to run. You can either run these immediately, or you can run them using a background thread pool if you don't want to block your main thread while they complete.
- `FINISHED` indicates that the handshake has just finished, while `NOT_HANDSHAKING` indicates that no handshake is currently in progress (either it has already finished or has not been started). The `FINISHED` status is only generated once by the last call to `wrap()` or `unwrap ()` and then the engine will subsequently produce a `NOT_HANDSHAKING` status.

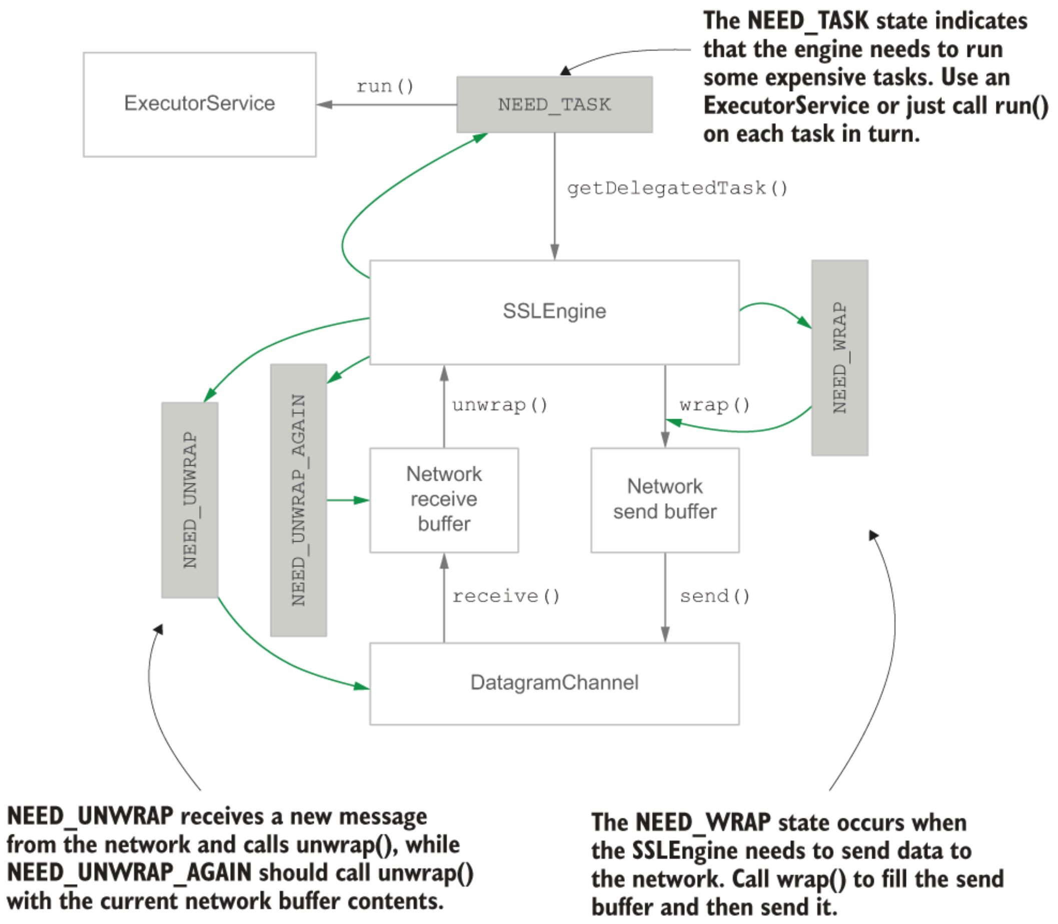

The NEED_TASK state indicates that the engine needs to run some expensive tasks. Use an ExecutorService or just call run() on each task in turn.
ExecutorService
run()
NEED_TASK
getDelegatedTask()
SSLEngine
NEED_WRAP
NEED_UNWRAP
NEED_UNWRAP_AGAIN
unwrap()
wrap()
Network receive buffer
Network send buffer
receive()
send()
DatagramChannel
NEED_UNWRAP receives a new message from the network and calls unwrap(), while NEED_UNWRAP_AGAIN should call unwrap() with the current network buffer contents.
The NEED_WRAP state occurs when the SSLEngine needs to send data to the network. Call wrap() to fill the send buffer and then send it.

Figure 12.3 The SSLEngine handshake state machine involves four main states. In the `NEED_UNWRAP` and `NEED_UNWRAP_AGAIN` states, you should use the `unwrap()` call to supply it with received network data. The `NEED_WRAP` state indicates that new DTLS packets should be retrieved with the `wrap()` call and then sent to the other party. The `NEED_TASK` state is used when the engine needs to execute expensive cryptographic functions.

Listing 12.2 shows the outline of how the basic loop for performing a DTLS handshake with `SSLEngine` is performed based on the handshake status codes.

NOTE This listing has been simplified compared to the implementation in the GitHub repository accompanying the book, but the core logic is correct.

Listing 12.2 SSLEngine handshake loop

```
engine.beginHandshake();                                          1

var handshakeStatus = engine.getHandshakeStatus();                2
while (handshakeStatus != HandshakeStatus.FINISHED) {             3
    SSLEngineResult result;
    switch (handshakeStatus) {
            case NEED_UNWRAP:                                             ❹
                if (recvBuf.position() == 0) {                            ❹
                    channel.receive(recvBuf);                             ❹
                }                                                         ❹
            case NEED_UNWRAP_AGAIN:                                       ❺
                result = engine.unwrap(recvBuf.flip(), appData);          ❻
                recvBuf.compact();                                        ❻
                checkStatus(result.getStatus());                          ❼
                handshakeStatus = result.getHandshakeStatus();            ❼
                break;
            case NEED_WRAP:
                result = engine.wrap(appData.flip(), sendBuf);            ❽
                appData.compact();                                        ❽
                channel.write(sendBuf.flip());                            ❽
                sendBuf.compact();                                        ❽
                checkStatus(result.getStatus());                          ❽
                handshakeStatus = result.getHandshakeStatus();            ❽
                break;                                                    ❽
            case NEED_TASK:
                Runnable task;                                            ❾
                while ((task = engine.getDelegatedTask()) != null) {      ❾
                    task.run();                                           ❾
                }                                                         ❾
                status = engine.getHandshakeStatus();                     ❾
            default:
                throw new IllegalStateException();
    }
```

❶ Trigger a new DTLS handshake.

❷ Allocate buffers for network and application data.

❸ Loop until the handshake is finished.

❹ In the NEED_UNWRAP state, you should wait for a network packet if not already received.

❺ Let the switch statement fall through to the NEED_UNWRAP_AGAIN case.

❻ Process any received DTLS packets by calling engine.unwrap().

❼ Check the result status of the unwrap() call and update the handshake state.

❽ In the NEED_WRAP state, call the wrap() method and then send the resulting DTLS packets.

❾ For NEED_TASK, just run any delegated tasks or submit them to a thread pool.

The `wrap ()` and `unwrap ()` calls return a status code for the operation as well as a new handshake status, which you should check to ensure that the operation completed correctly. The possible status codes are shown in table 12.1. If you need to resize a buffer, you can query the current `SSLSession` to determine the recommended application and network buffer sizes and compare that to the amount of space left in the buffer. If the buffer is too small, you should allocate a new buffer and copy any existing data into the new buffer. Then retry the operation again.

Table 12.1 SSLEngine operation status codes

Status code	Meaning
`OK`	The operation completed successfully.
`BUFFER_UNDERFLOW`	The operation failed because there was not enough input data. Check that the input buffer has enough space remaining. For an unwrap operation, you should receive another network packet if this status occurs.
`BUFFER_OVERFLOW`	The operation failed because there wasn't enough space in the output buffer. Check that the buffer is large enough and resize it if necessary.
`CLOSED`	The other party has indicated that they are closing the connection, so you should process any remaining packets and then close the `SSLEngine` too.

Using the `DtlsDatagramChannel` class from the GitHub repository accompanying the book, you can now implement a working DTLS client example application. The sample class requires that the underlying UDP channel is connected before the DTLS handshake occurs. This restricts the channel to send packets to only a single host and receive packets from only that host too. This is not a limitation of DTLS but just a simplification made to keep the sample code short. A consequence of this decision is that the server that you'll develop in the next section can only handle a single client at a time and will discard packets from other clients. It's not much harder to handle concurrent clients but you need to associate a unique `SSLEngine` with each client.

DEFINITION A UDP channel (or socket) is connected when it is restricted to only send or receive packets from a single host. Using connected channels simplifies programming and can be more efficient, but packets from other clients will be silently discarded. The `connect()` method is used to connect a Java `DatagramChannel`.

Listing 12.3 shows a sample client that connects to a server and then sends the contents of a text file line by line. Each line is sent as an individual UDP packet and will be encrypted using DTLS. After the packets are sent, the client queries the `SSLSession` to print out the DTLS cipher suite that was used for the connection. Open the DtlsClient.java file you created earlier and add the main method shown in the listing. Create a text file named test.txt in the root folder of the project and add some example text to it, such as lines from Shakespeare, your favorite quotes, or anything you like.

NOTE You won't be able to use this client until you write the server to accompany it in the next section.

Listing 12.3 The DTLS client

```
public static void main(String... args) throws Exception {
    try (var channel = new DtlsDatagramChannel(getClientContext());         ❶
         var in = Files.newBufferedReader(Paths.get("test.txt"))) {          ❷
        logger.info("Connecting to localhost:54321");
        channel.connect("localhost", 54321);                                 ❸

        String line;
        while ((line = in.readLine()) != null) {                             ❹
            logger.info("Sending packet to server: {}", line);               ❹
            channel.send(line.getBytes(UTF_8));                              ❹
        }

        logger.info("All packets sent");
        logger.info("Used cipher suite: {}",                                 ❺
                channel.getSession().getCipherSuite());                      ❺
    }
}
```

❶ Open the DTLS channel with the client SSLContext.

❷ Open a text file to send to the server.

❸ Connect to the server running on the local machine and port 54321.

❹ Send the lines of text to the server.

❺ Print details of the DTLS connection.

After the client completes, it will automatically close the `DtlsDatagramChannel`, which will trigger shutdown of the associated `SSLEngine` object. Closing a DTLS session is not as simple as just closing the UDP channel, because each party must send each other a close-notify alert message to signal that the DTLS session is being closed. In Java, the process is similar to the handshake loop that you saw earlier in listing 12.2. First, the client should indicate that it will not send any more packets by calling the `closeOutbound()` method on the engine. You should then call the `wrap()` method to allow the engine to produce the close-notify alert message and send that message to the server, as shown in listing 12.4. Once the alert has been sent, you should process incoming messages until you receive a corresponding close-notify from the server, at which point the `SSLEngine` will return true from the `isInboundDone()` method and you can close the underlying UDP `DatagramChannel`.

If the other side closes the channel first, then the next call to `unwrap ()` will return a `CLOSED` status. In this case, you should reverse the order of operations: first close the inbound side and process any received messages and then close the outbound side and send your own close-notify message.

Listing 12.4 Handling shutdown

```
public void close() throws IOException {
    sslEngine.closeOutbound();                          ❶
    sslEngine.wrap(appData.flip(), sendBuf);            ❷
    appData.compact();                                  ❷
    channel.write(sendBuf.flip());                      ❷
    sendBuf.compact();                                  ❷

    while (!sslEngine.isInboundDone()) {                ❸
        channel.receive(recvBuf);                       ❸
        sslEngine.unwrap(recvBuf.flip(), appData);      ❸
        recvBuf.compact();                              ❸
    }
    sslEngine.closeInbound();                           ❹
```

```
        channel.close();                                    ❹
    }
```

❶ Indicate that no further outbound application packets will be sent.

❷ Call wrap() to generate the close-notify message and send it to the server.

❸ Wait until a close-notify is received from the server.

❹ Indicate that the inbound side is now done too and close the UDP channel.

IMPLEMENTING A DTLS SERVER

Initializing a `SSLContext` for a server is similar to the client, except in this case you use a `KeyManagerFactory` to supply the server's certificate and private key. Because you're not using client certificate authentication, you can leave the `TrustManager` array as `null`. Listing 12.5 shows the code for creating a server-side DTLS context. Create a new file named DtlsServer.java next to the client and type in the contents of the listing.

Listing 12.5 The server SSLContext

```
package com.manning.apisecurityinaction;

import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.security.KeyStore;
import javax.net.ssl.*;
import org.slf4j.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class DtlsServer {
    private static SSLContext getServerContext() throws Exception {
        var sslContext = SSLContext.getInstance("DTLS");              ❶

        var keyStore = KeyStore.getInstance("PKCS12");                ❷
        keyStore.load(new FileInputStream("localhost.p12"),           ❷
                "changeit".toCharArray());                            ❷
        var keyManager = KeyManagerFactory.getInstance("PKIX");       ❸
        keyManager.init(keyStore, "changeit".toCharArray());          ❸
        sslContext.init(keyManager.getKeyManagers(), null, null);     ❹
        return sslContext;
    }
}
```

❶ Create a DTLS SSLContext again.

❷ Load the server’s certificate and private key from a keystore.

❸ Initialize the KeyManagerFactory with the keystore.

❹ Initialize the SSLContext with the key manager.

In this example, the server will be running on localhost, so use `mkcert` to generate a key pair and signed certificate if you don’t already have one, by running[2]

```
mkcert -pkcs12 localhost
```

in the root folder of the project. You can then implement the DTLS server as shown in listing 12.6. Just as in the client example, you can use the `DtlsDatagramChannel` class to simplify the handshake. Behind the scenes, the same handshake process will occur, but the order of `wrap ()` and `unwrap ()` operations will be different due to the different roles played in the handshake. Open the DtlsServer.java file you created earlier and add the `main` method shown in the listing.

NOTE The `DtlsDatagramChannel` provided in the GitHub repository accompanying the book will automatically connect the underlying `DatagramChannel` to the first client that it receives a packet from and discard packets from other clients until that client disconnects.

Listing 12.6 The DTLS server

```
public static void main(String... args) throws Exception {
    try (var channel = new DtlsDatagramChannel(getServerContext())) {
        channel.bind(54321);
        logger.info("Listening on port 54321");

        var buffer = ByteBuffer.allocate(2048);

        while (true) {
                channel.receive(buffer);
                buffer.flip();
                var data = UTF_8.decode(buffer).toString();
                logger.info("Received: {}", data);
                buffer.compact();
            }
        }
    }
```

❶ Create the DtlsDatagramChannel and bind to port 54321.

❷ Allocate a buffer for data received from the client.

❸ Receive decrypted UDP packets from the client.

❹ Print out the received data.

You can now start the server by running the following command:

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

This will produce many lines of output as it compiles and runs the code. You'll see the following line of output once the server has started up and is listening for UDP packets from clients:

```
[com.manning.apisecurityinaction.DtlsServer.main()] INFO
➥ com.manning.apisecurityinaction.DtlsServer - Listening on port
➥ 54321
```

You can now run the client in another terminal window by running:

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

TIP If you want to see details of the DTLS protocol messages being sent between the client and server, add the argument `-Djavax.net.debug=all` to the Maven command line. This will produce detailed logging of the handshake messages.

The client will start up, connect to the server, and send all of the lines of text from the input file to the server, which will receive them all and print them out. After the client has completed, it will print out the DTLS cipher suite that it used so that you can see what was negotiated. In the next section, you'll see how the default choice made by Java might not be appropriate for IoT applications and how to choose a more suitable replacement.

NOTE This example is intended to demonstrate the use of DTLS only and is not a production-ready network protocol. If you separate the client and server over a network, it is likely that some packets will get lost. Use a higher-level application protocol such as CoAP if your application requires reliable packet delivery (or use normal TLS over TCP).

12.1.2 Cipher suites for constrained devices

In previous chapters, you've followed the guidance from Mozilla[3] when choosing secure TLS cipher suites (recall from chapter 7 that a cipher suite is a collection of cryptographic algorithms chosen to work well together). This guidance is aimed at securing traditional web server applications and their clients, but these cipher suites are not always suitable for IoT use for several reasons:

- The size of code required to implement these suites securely can be quite large and require many cryptographic primitives. For example, the cipher suite `ECDHE-RSA-AES256-SHA384` requires implementing Elliptic Curve Diffie-Hellman (ECDH) key agreement, RSA signatures, AES encryption and decryption operations, and the SHA-384 hash function with HMAC!
- Modern recommendations heavily promote the use of AES in Galois/Counter Mode (GCM), because this is extremely fast and secure on modern Intel chips due to hardware acceleration. But it can be difficult to implement securely in software on constrained devices and fails catastrophically if misused.
- Some cryptographic algorithms, such as SHA-512 or SHA-384, are rarely hardware-accelerated and are designed to perform well when implemented in software on 64-bit architectures. There can be a performance penalty when implementing these algorithms on 32-bit architectures, which are very common in IoT devices. In low-power environments, 8-bit microcontrollers are still commonly used, which makes implementing such algorithms even more challenging.

- Modern recommendations concentrate on cipher suites that provide forward secrecy as discussed in chapter 7 (also known as perfect forward secrecy). This is a very important security property, but it increases the computational cost of these cipher suites. All of the forward secret cipher suites in TLS require implementing both a signature algorithm (such as RSA) and a key agreement algorithm (usually, ECDH), which increases the code size.[4]

Nonce reuse and AES-GCM in DTLS

The most popular symmetric authenticated encryption mode used in modern TLS applications is based on AES in Galois/Counter Mode (GCM). GCM requires that each packet is encrypted using a unique nonce and loses almost all security if the same nonce is used to encrypt two different packets. When GCM was first introduced for TLS 1.2, it required an 8-byte nonce to be explicitly sent with every record. Although this nonce could be a simple counter, some implementations decided to generate it randomly. Because 8 bytes is not large enough to safely generate randomly, these implementations were found to be susceptible to accidental nonce reuse. To prevent this problem, TLS 1.3 introduced a new scheme based on implicit nonces: the nonce for a TLS record is derived from the sequence number that TLS already keeps track of for each connection. This was a significant security improvement because TLS implementations must accurately keep track of the record sequence number to ensure proper operation of the protocol, so accidental nonce reuse will result in an immediate protocol failure (and is more likely to be caught by tests). You can read more about this development at **https://blog.cloudflare.com/tls-nonce-nse/**.

Due to the unreliable nature of UDP-based protocols, DTLS requires that record sequence numbers are explicitly added to all packets so that retransmitted or reordered packets can be detected and handled. Combined with the fact that DTLS is more lenient of duplicate packets, this makes accidental nonce reuse bugs in DTLS applications using AES GCM more likely. You should therefore prefer alternative cipher suites when using DTLS, such as those discussed in this section. In section 12.3.3, you'll learn about authenticated encryption algorithms you can use in your application that are more robust against nonce reuse.

Figure 12.4 shows an overview of the software components and algorithms that are required to support a set of TLS cipher suites that are commonly used for web connections. TLS supports a variety of key exchange algorithms used during the initial handshake, each of which needs different cryptographic primitives to be implemented. Some of these also require digital signatures to be implemented, again with several choices of algorithms. Some signature algorithms support different group parameters, such as elliptic curves used for ECDSA signatures, which require further code. After the handshake completes, there are several choices for cipher modes and MAC algorithms for securing application data. X.509 certificate authentication itself requires additional code. This can add up to a significant amount of code to include on a constrained device.

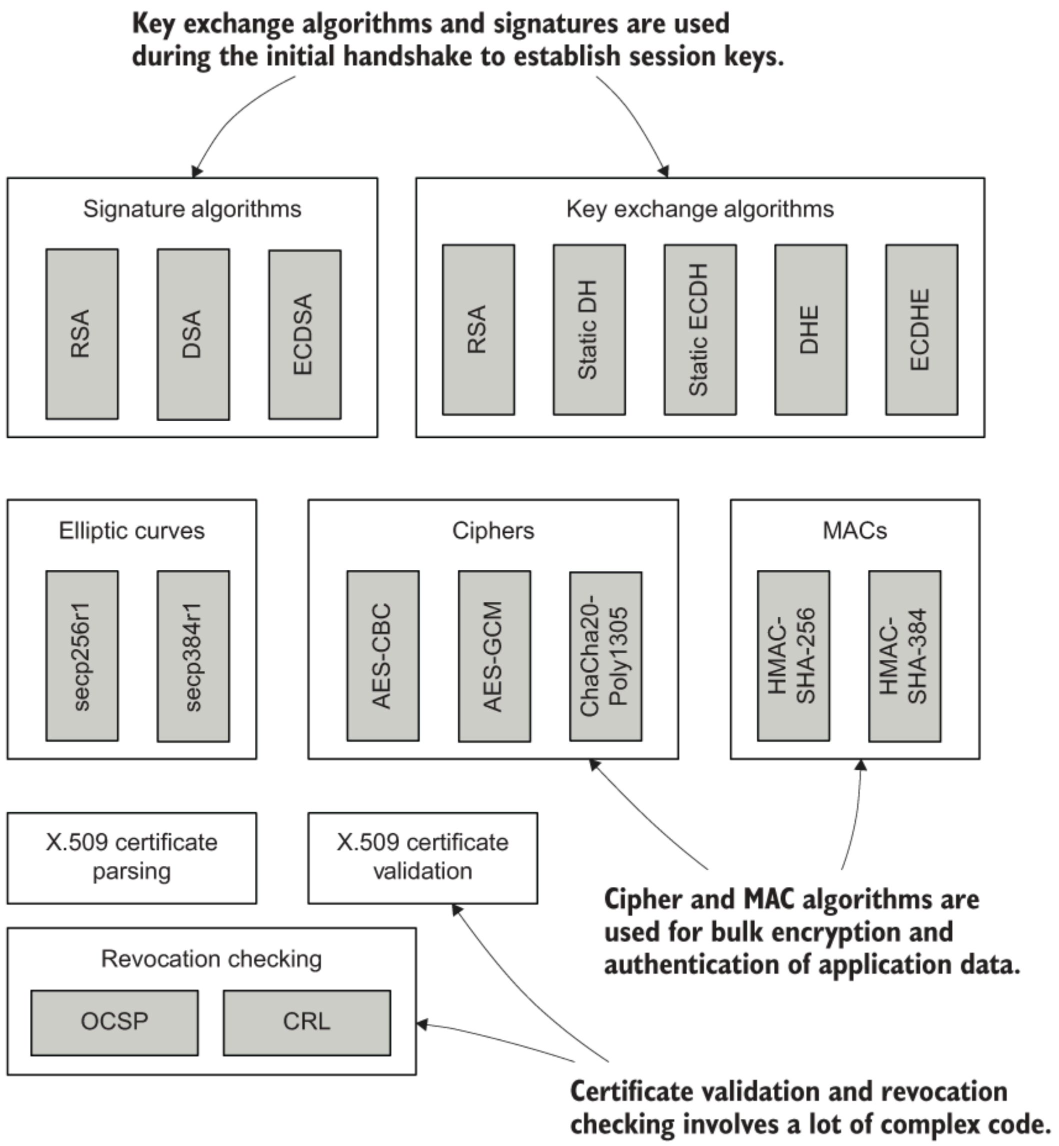

Key exchange algorithms and signatures are used during the initial handshake to establish session keys.
Signature algorithms
RSA
DSA
ECDSA
Key exchange algorithms
RSA
Static DH
Static ECDH
DHE
ECDHE
Elliptic curves
secp256r1
secp384r1
Ciphers
AES-CBC
AES-GCM
ChaCha20-Poly1305
MACs
HMAC-SHA-256
HMAC-SHA-384
X.509 certificate parsing
X.509 certificate validation
Cipher and MAC algorithms are used for bulk encryption and authentication of application data.
Revocation checking
OCSP
CRL
Certificate validation and revocation checking involves a lot of complex code.

Figure 12.4 A cross-section of algorithms and components that must be implemented to support common TLS web connections. Key exchange and signature algorithms are used during the initial handshake, and then cipher modes and MACs are used to secure application data once a session has been established. X.509 certificates require a lot of complex code for parsing, validation, and checking for revoked certificates.

For these reasons, other cipher suites are often popular in IoT applications. As an alternative to forward secret cipher suites, there are older cipher suites based on either RSA encryption or static Diffie-Hellman key agreement (or the elliptic curve variant, ECDH). Unfortunately, both algorithm families have significant security weaknesses, not directly related to their lack of forward secrecy. RSA key exchange uses an old mode of encryption (known as PKCS#1 version 1.5) that is very hard to implement securely and has resulted in many vulnerabilities in TLS implementations. Static ECDH key agreement has potential security weaknesses of its own, such as invalid curve attacks that can reveal the server's long-term private key; it is rarely implemented. For these reasons, you should prefer forward secret cipher suites whenever possible, as they provide better protection against common cryptographic vulnerabilities. TLS 1.3 has completely removed these older modes due to their insecurity.

DEFINITION An invalid curve attack is an attack on elliptic curve cryptographic keys. An attacker sends the victim a public key on a different (but related) elliptic curve to the victim's private key. If the victim's TLS library doesn't validate the received public key carefully, then the result may leak information about their private key. Although ephemeral ECDH cipher suites (those with ECDHE in the name) are also vulnerable to invalid curve attacks, they are much harder to exploit because each private key is only used once.

Even if you use an older cipher suite, a DTLS implementation is required to include support for signatures in order to validate certificates that are presented by the server (and optionally by the client) during the handshake. An extension to TLS and DTLS allows certificates to be replaced with raw public keys (https://tools.ietf.org/html/ rfc7250). This allows the complex certificate parsing and validation code to be eliminated, along with support for many signature algorithms, resulting in a large reduction in code size. The downside is that keys must instead be manually distributed to all devices, but this can be a viable approach in some environments. Another alternative is to use pre-shared keys, which you'll learn more about in section 12.2.

DEFINITION Raw public keys can be used to eliminate the complex code required to parse and verify X.509 certificates and verify signatures over those certificates. A raw public key must be manually distributed to devices over a secure channel (for example, during manufacture).

The situation is somewhat better when you look at the symmetric cryptography used to secure application data after the TLS handshake and key exchange has completed. There are two alternative cryptographic algorithms that can be used instead of the usual AES-GCM and AES-CBC modes:

- Cipher suites based on AES in CCM mode provide authenticated encryption using only an AES encryption circuit, providing a reduction in code size compared to CBC mode and is a bit more robust compared to GCM. CCM has become widely adopted in IoT applications and standards, but it has some undesirable features too, as discussed in a critique of the mode by Phillip Rogaway and David Wagner (**https://web.cs.ucdavis.edu/~rogaway/papers/ccm.pdf**).
- The ChaCha20-Poly1305 cipher suites can be implemented securely in software with relatively little code and good performance on a range of CPU architectures. Google adapted these cipher suites for TLS to provide better performance and security on mobile devices that lack AES hardware acceleration.

DEFINITION AES-CCM (Counter with CBC-MAC) is an authenticated encryption algorithm based solely on the use of an AES encryption circuit for all operations. It uses AES in Counter mode for encryption and decryption, and a Message Authentication Code (MAC) based on AES in CBC mode for authentication. ChaCha20-Poly1305 is a stream cipher and MAC designed by Daniel Bernstein that is very fast and easy to implement in software.

Both of these choices have fewer weaknesses compared to either AES-GCM or the older AES-CBC modes when implemented on constrained devices.[5] If your devices have hardware support for AES, for example in a dedicated secure element chip, then CCM can be an attractive choice. In most other cases, ChaCha20-Poly1305 can be easier to implement securely. Java has support for ChaCha20-Poly1305 cipher suites since Java 12. If you have Java 12 installed, you can force the use of ChaCha20-Poly1305 by specifying a custom `SSLParameters` object and passing it to the `setSSLParameters()` method on the `SSLEngine`. Listing 12.7 shows how to configure the parameters to only allow ChaCha20-Poly1305-based cipher suites. If you have Java 12, open the DtlsClient.java file and add the new method to the class. Otherwise, skip this example.

TIP If you need to support servers or clients running older versions of DTLS, you should add the `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` marker cipher suite. Otherwise Java may be unable to negotiate a connection with some older software. This cipher suite is enabled by default so be sure to re-enable it when specifying custom cipher suites.

Listing 12.7 Forcing use of ChaCha20-Poly1305

```
private static SSLParameters sslParameters() {
    var params = DtlsDatagramChannel.defaultSslParameters();        ❶
    params.setCipherSuites(new String[] {                            ❷
            "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",         ❷
            "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256",           ❷
            "TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256",             ❷
            "TLS_EMPTY_RENEGOTIATION_INFO_SCSV"                      ❸
    });
    return params;
}
```

❶ Use the defaults from the DtlsDatagramChannel.

❷ Enable only cipher suites that use ChaCha20-Poly1305.

❸ Include this cipher suite if you need to support multiple DTLS versions.

After adding the new method, you can update the call to the `DtlsDatagramChannel` constructor in the same file to pass the custom parameters:

```
try (var channel = new DtlsDatagramChannel(getClientContext(),
    sslParameters());
```

If you make that change and re-run the client, you'll see that the connection now uses ChaCha20-Poly1305, so long as both the client and server are using Java 12 or later.

WARNING The example in listing 12.7 uses the default parameters from the DtlsDatagramChannel class. If you create your own parameters, ensure that you set an endpoint identification algorithm. Otherwise, Java won't validate that the server's certificate matches the hostname you have connected to and the connection may be vulnerable to man-in-the-middle attacks. You can set the identification algorithm by calling `"params.setEndpointIdenticationAlgorithm("HTTPS")"`.

AES-CCM is not yet supported by Java, although work is in progress to add support. The Bouncy Castle library (**https://www.bouncycastle.org/java.html**) supports CCM cipher suites with DTLS, but only through a different API and not the standard `SSLEngine` API. There's an example using the Bouncy Castle DTLS API with CCM in section 12.2.1.

The CCM cipher suites come in two variations:

- The original cipher suites, whose names end in _CCM, use a 128-bit authentication tag.
- Cipher suites ending in _CCM_8, which use a shorter 64-bit authentication tag. This can be useful if you need to save every byte in network messages but provides much weaker protections against message forgery and tampering.

You should therefore prefer using the variants with a 128-bit authentication tag unless you have other measures in place to prevent message forgery, such as strong network protections, and you know that you need to reduce network overheads. You should apply strict rate-limiting to API endpoints where there is a risk of brute force attacks against authentication tags; see chapter 3 for details on how to apply rate-limiting.

Pop quiz

1. Which SSLEngine handshake status indicates that a message needs to be sent across the network?
   1. NEED_TASK
   2. NEED_WRAP
   3. NEED_UNWRAP
   4. NEED_UNWRAP_AGAIN
2. Which one of the following is an increased risk when using AES-GCM cipher suites for IoT applications compared to other modes?
   1. A breakthrough attack on AES
   2. Nonce reuse leading to a loss of security
   3. Overly large ciphertexts causing packet fragmentation
   4. Decryption is too expensive for constrained devices

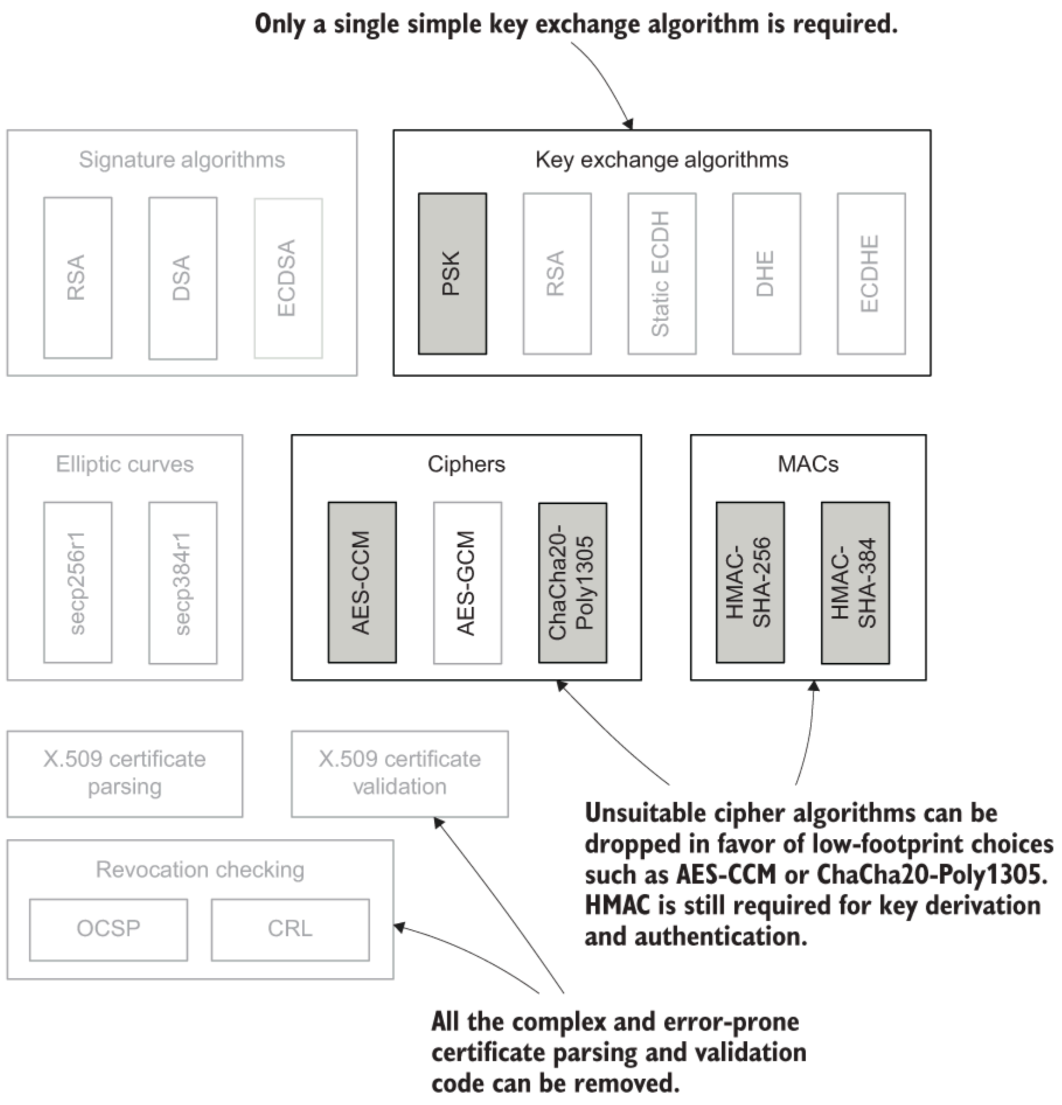
The answers are at the end of the chapter.

12.2 Pre-shared keys

In some particularly constrained environments, devices may not be capable of carrying out the public key cryptography required for a TLS handshake. For example, tight constraints on available memory and code size may make it hard to support public key signature or key-agreement algorithms. In these environments, you can still use TLS (or DTLS) by using cipher suites based on pre-shared keys (PSK) instead of certificates for authentication. PSK cipher suites can result in a dramatic reduction in the amount of code needed to implement TLS, as shown in figure 12.5, because the certificate parsing and validation code, along with the signatures and public key exchange modes can all be eliminated.

DEFINITION A pre-shared key (PSK) is a symmetric key that is directly shared with the client and server ahead of time. A PSK can be used to avoid the overheads of public key cryptography on constrained devices.

In TLS 1.2 and DTLS 1.2, a PSK can be used by specifying dedicated PSK cipher suites such as `TLS_PSK_WITH_AES_128_CCM`. In TLS 1.3 and the upcoming DTLS 1.3, use of a PSK is negotiated using an extension that the client sends in the initial ClientHello message. Once a PSK cipher suite has been selected, the server and client derive session keys from the PSK and random values that they each contribute during the handshake, ensuring that unique keys are still used for every session. The session key is used to compute a HMAC tag over all of the handshake messages, providing authentication of the session: only somebody with access to the PSK could derive the same HMAC key and compute the correct authentication tag.



Only a single simple key exchange algorithm is required.
Signature algorithms
RSA
DSA
ECDSA
Key exchange algorithms
PSK
RSA
Static ECDH
DHE
ECDHE
Elliptic curves
secp256r1
secp384r1
Ciphers
AES-CCM
AES-GCM
ChaCha20-Poly1305
MACs
HMAC-SHA-256
HMAC-SHA-384
X.509 certificate parsing
X.509 certificate validation
Revocation checking
OCSP
CRL
Unsuitable cipher algorithms can be dropped in favor of low-footprint choices such as AES-CCM or ChaCha20-Poly1305. HMAC is still required for key derivation and authentication.
All the complex and error-prone certificate parsing and validation code can be removed.

Figure 12.5 Use of pre-shared key (PSK) cipher suites allows implementations to remove a lot of complex code from a TLS implementation. Signature algorithms are no longer needed at all and can be removed, as can most key exchange algorithms. The complex X.509 certificate parsing and validation logic can be deleted too, leaving only the basic symmetric cryptography primitives.

CAUTION Although unique session keys are generated for each session, the basic PSK cipher suites lack forward secrecy: an attacker that compromises the PSK can easily derive the session keys for every previous session if they captured the handshake messages. Section 12.2.4 discusses PSK cipher suites with forward secrecy.

Because PSK is based on symmetric cryptography, with the client and server both using the same key, it provides mutual authentication of both parties. Unlike client certificate authentication, however, there is no name associated with the client apart from an opaque identifier for the PSK, so a server must maintain a mapping between PSKs and the associated client or rely on another method for authenticating the client's identity.

WARNING Although TLS allows the PSK to be any length, you should only use a PSK that is cryptographically strong, such as a 128-bit value from a secure random number generator. PSK cipher suites are not suitable for use with passwords because an attacker can perform an offline dictionary or brute-force attack after seeing one PSK handshake.

12.2.1 Implementing a PSK server

Listing 12.8 shows how to load a PSK from a keystore. For this example, you can load the existing HMAC key that you created in chapter 6, but it is good practice to use distinct keys for different uses within an application even if they happen to use the same algorithm. A PSK is just a random array of bytes, so you can call the `getEncoded()` method to get the raw bytes from the `Key` object. Create a new file named PskServer.java under src/main/java/com/manning/apisecurityinaction and copy in the contents of the listing. You'll flesh out the rest of the server in a moment.

Listing 12.8 Loading a PSK

```
package com.manning.apisecurityinaction;

import static java.nio.charset.StandardCharsets.UTF_8;
import java.io.FileInputStream;
import java.net.*;
import java.security.*;
import org.bouncycastle.tls.*;
import org.bouncycastle.tls.crypto.impl.bc.BcTlsCrypto;

public class PskServer {
    static byte[] loadPsk(char[] password) throws Exception {
        var keyStore = KeyStore.getInstance("PKCS12");
        keyStore.load(new FileInputStream("keystore.p12"), password);
        return keyStore.getKey("hmac-key", password).getEncoded();
    }
}
```

❶ Load the keystore.

❷ Load the key and extract the raw bytes.

Listing 12.9 shows a basic DTLS server with pre-shared keys written using the Bouncy Castle API. The following steps are used to initialize the server and perform a PSK handshake with the client:

- First load the PSK from the keystore.
- Then you need to initialize a `PSKTlsServer` object, which requires two arguments: a `BcTlsCrypto` object and a `TlsPSKIdentityManager`, that is used to look up the PSK for a given client. You'll come back to the identity manager shortly.
- The `PSKTlsServer` class only advertises support for normal TLS by default, although it supports DTLS just fine. Override the `getSupportedVersions()` method to ensure that DTLS 1.2 support is enabled; otherwise, the handshake will fail. The supported protocol versions are communicated during the handshake and some clients may fail if there are both TLS and DTLS versions in the list.
- Just like the `DtlsDatagramChannel` you used before, Bouncy Castle requires the UDP socket to be connected before the DTLS handshake occurs. Because the server doesn't know where the client is located, you can wait until a packet is received from any client and then call `connect ()` with the socket address of the client.
- Create a `DTLSServerProtocol` and `UDPTransport` objects, and then call the accept method on the protocol object to perform the DTLS handshake. This returns a `DTLSTransport` object that you can then use to send and receive encrypted and authenticated packets with the client.

TIP Although the Bouncy Castle API is straightforward when using PSKs, I find it cumbersome and hard to debug if you want to use certificate authentication, and I prefer the `SSLEngine` API.

Listing 12.9 DTLS PSK server

```
public static void main(String[] args) throws Exception {
    var psk = loadPsk(args[0].toCharArray());                                1
    var crypto = new BcTlsCrypto(new SecureRandom());
    var server = new PSKTlsServer(crypto, getIdentityManager(psk)) {         2
        @Override                                                            2
        protected ProtocolVersion[] getSupportedVersions() {                 2
            return ProtocolVersion.DTLSv12.only();                           2
        }                                                                    2
    };                                                                       2
    var buffer = new byte[2048];
    var serverSocket = new DatagramSocket(54321);
    var packet = new DatagramPacket(buffer, buffer.length);
    serverSocket.receive(packet);                                            3
    serverSocket.connect(packet.getSocketAddress());                         3
```

```
        var protocol = new DTLSServerProtocol();                              4
        var transport = new UDPTransport(serverSocket, 1500);                 4
        var dtls = protocol.accept(server, transport);                        4

        while (true) {                                                        5
            var len = dtls.receive(buffer, 0, buffer.length, 60000);          5
            if (len == -1) break;                                             5
            var data = new String(buffer, 0, len, UTF_8);                     5
            System.out.println("Received: " + data);                          5
        }                                                                     5
    }
```

❶ Load the PSK from the keystore.

❷ Create a new PSKTlsServer and override the supported versions to allow DTLS.

❸ BouncyCastle requires the socket to be connected before the handshake.

❹ Create a DTLS protocol and perform the handshake using the PSK.

❺ Receive messages from the client and print them out.

The missing part of the puzzle is the PSK identity manager, which is responsible for determining which PSK to use with each client. Listing 12.10 shows a very simple implementation of this interface for the example, which returns the same PSK for every client. The client sends an identifier as part of the PSK handshake, so a more sophisticated implementation could look up different PSKs for each client. The server can also provide a hint to help the client determine which PSK it should use, in case it has multiple PSKs. You can leave this `null` here, which instructs the server not to send a hint. Open the PskServer.java file and add the method from listing 12.10 to complete the server implementation.

TIP A scalable solution would be for the server to generate distinct PSKs for each client from a master key using HKDF, as discussed in chapter 11.

Listing 12.10 The PSK identity manager

```
    static TlsPSKIdentityManager getIdentityManager(byte[] psk) {
        return new TlsPSKIdentityManager() {
            @Override
        public byte[] getHint() {                   ❶
            return null;                            ❶
        }                                           ❶

        @Override
        public byte[] getPSK(byte[] identity) {     ❷
            return psk;                             ❷
        }                                           ❷
    };
}
```

❶ Leave the PSK hint unspecified.

❷ Return the same PSK for all clients.

12.2.2 The PSK client

The PSK client is very similar to the server, as shown in listing 12.11. As before, you create a new `BcTlsCrypto` object and use that to initialize a `PSKTlsClient` object. In this case, you pass in the PSK and an identifier for it. If you don't have a good identifier for your PSK already, then a secure hash of the PSK works well. You can use the `Crypto.hash()` method from the Salty Coffee library from chapter 6, which uses SHA-512. As for the server, you need to override the `getSupportedVersions()` method to ensure DTLS support is enabled. You can then connect to the server and perform the DTLS handshake using the `DTLSClientProtocol` object. The `connect()` method returns a `DTLSTransport` object that you can then use to send and receive encrypted packets with the server.

Create a new file named PskClient.java alongside the server class and type in the contents of the listing to create the server. If your editor doesn't automatically add them, you'll need to add the following imports to the top of the file:

```
import static java.nio.charset.StandardCharsets.UTF_8;
import java.io.FileInputStream;
import java.net.*;
import java.security.*;
import org.bouncycastle.tls.*;
import org.bouncycastle.tls.crypto.impl.bc.BcTlsCrypto;
```

Listing 12.11 The PSK client

```
package com.manning.apisecurityinaction;
public class PskClient {
    public static void main(String[] args) throws Exception {
        var psk = PskServer.loadPsk(args[0].toCharArray());          ❶
        var pskId = Crypto.hash(psk);                                 ❶

        var crypto = new BcTlsCrypto(new SecureRandom());            ❷
        var client = new PSKTlsClient(crypto, pskId, psk) {          ❷
            @Override
            protected ProtocolVersion[] getSupportedVersions() {     ❸
                return ProtocolVersion.DTLSv12.only();               ❸
            }                                                         ❸
        };

        var address = InetAddress.getByName("localhost");
        var socket = new DatagramSocket();
        socket.connect(address, 54321);                               ❹
        socket.send(new DatagramPacket(new byte[0], 0));              ❹
        var transport = new UDPTransport(socket, 1500);               ❺
        var protocol = new DTLSClientProtocol();                      ❺
        var dtls = protocol.connect(client, transport);               ❺

        try (var in = Files.newBufferedReader(Paths.get("test.txt"))) {
            String line;
            while ((line = in.readLine()) != null) {
                System.out.println("Sending: " + line);
                var buf = line.getBytes(UTF_8);
                dtls.send(buf, 0, buf.length);                        ❻
            }
        }
    }
}
```

❶ Load the PSK and generate an ID for it.

❷ Create a PSKTlsClient with the PSK.

❸ Override the supported versions to ensure DTLS support.

❹ Connect to the server and send a dummy packet to start the handshake.

❺ Create the DTLSClientProtocol instance and perform the handshake over UDP.

6 Send encrypted packets using the returned DTLSTransport object.

You can now test out the handshake by running the server and client in separate terminal windows. Open two terminals and change to the root directory of the project in both. Then run the following in the first one:

```
mvn clean compile exec:java \
  -Dexec.mainClass=com.manning.apisecurityinaction.PskServer \
  -Dexec.args=changeit                                          1
```

1 Specify the keystore password as an argument.

This will compile and run the server class. If you've changed the keystore password, then supply the correct value on the command line. Open the second terminal window and run the client too:

```
mvn exec:java \
  -Dexec.mainClass=com.manning.apisecurityinaction.PskClient \
  -Dexec.args=changeit
```

After the compilation has finished, you'll see the client sending the lines of text to the server and the server receiving them.

NOTE As in previous examples, this sample code makes no attempt to handle lost packets after the handshake has completed.

12.2.3 Supporting raw PSK cipher suites

By default, Bouncy Castle follows the recommendations from the IETF and only enables PSK cipher suites combined with ephemeral Diffie-Hellman key agreement to provide forward secrecy. These cipher suites are discussed in section 12.1.4. Although these are more secure than the raw PSK cipher suites, they are not suitable for very constrained devices that can't perform public key cryptography. To enable the raw PSK cipher suites, you have to override the `getSupportedCipherSuites()` method in both the client and the server. Listing 12.12 shows how to override this method for the server, in this case providing support for just a single PSK cipher suite using AES-CCM to force its use. An identical change can be made to the `PSKTlsClient` object.

Listing 12.12 Enabling raw PSK cipher suites

```
var server = new PSKTlsServer(crypto, getIdentityManager(psk)) {
    @Override
    protected ProtocolVersion[] getSupportedVersions() {
        return ProtocolVersion.DTLSv12.only();
    }
    @Override
    protected int[] getSupportedCipherSuites() {      ❶
        return new int[] {                            ❶
                CipherSuite.TLS_PSK_WITH_AES_128_CCM  ❶
        };                                            ❶
    }                                                 ❶
};
```

❶ Override the getSupportedCipherSuites method to return raw PSK suites.

Bouncy Castle supports a wide range of raw PSK cipher suites in DTLS 1.2, shown in table 12.2. Most of these also have equivalents in TLS 1.3. I haven't listed the older variants using CBC mode or those with unusual ciphers such as Camellia (the Japanese equivalent of AES); you should generally avoid these in IoT applications.

Table 12.2 Raw PSK cipher suites

Cipher suite	Description
`TLS_PSK_WITH_AES_128_CCM`	AES in CCM mode with a 128-bit key and 128-bit authentication tag
`TLS_PSK_WITH_AES_128_CCM_8`	AES in CCM mode with 128-bit keys and 64-bit authentication tags
`TLS_PSK_WITH_AES_256_CCM`	AES in CCM mode with 256-bit keys and 128-bit authentication tags
`TLS_PSK_WITH_AES_256_CCM_8`	AES in CCM mode with 256-bit keys and 64-bit authentication tags
`TLS_PSK_WITH_AES_128_GCM_SHA256`	AES in GCM mode with 128-bit keys
`TLS_PSK_WITH_AES_256_GCM_SHA384`	AES in GCM mode with 256-bit keys
`TLS_PSK_WITH_CHACHA20_POLY1305_SHA256`	ChaCha20-Poly1305 with 256-bit keys

12.2.4 PSK with forward secrecy

I mentioned in section 12.1.3 that the raw PSK cipher suites lack forward secrecy: if the PSK is compromised, then all previously captured traffic can be easily decrypted. If confidentiality of data is important to your application and your devices can support a limited amount of public key cryptography, you can opt for PSK cipher suites combined with ephemeral Diffie-Hellman key agreement to ensure forward secrecy. In these cipher suites, authentication of the client and server is still guaranteed by the PSK, but both parties generate random public-private key-pairs and swap the public keys during the handshake, as shown in figure 12.6. The output of a Diffie-Hellman key agreement between each side's ephemeral private key and the other party's ephemeral public key is then mixed into the derivation of the session keys. The magic of Diffie-Hellman ensures that the session keys can't be recovered by an attacker that observes the handshake messages, even if they later recover the PSK. The ephemeral private keys are scrubbed from memory as soon as the handshake completes.

Custom protocols and the Noise protocol framework

Although for most IoT applications TLS or DTLS should be perfectly adequate for your needs, you may feel tempted to design your own cryptographic protocol that is a custom fit for your application. This is almost always a mistake, because even experienced cryptographers have made serious mistakes when designing protocols. Despite this widely repeated advice, many custom IoT security protocols have been developed, and new ones continue to be made. If you feel that you must develop a custom protocol for your application and can't use TLS or DTLS, the Noise protocol framework (https://noiseprotocol.org) can be used as a starting point. Noise describes how to construct a secure protocol from a few basic building blocks and describes a variety of handshakes that achieve different security goals. Most importantly, Noise is designed and reviewed by experts and has been used in real-world applications, such as the WireGuard VPN protocol (https://www.wireguard.com).

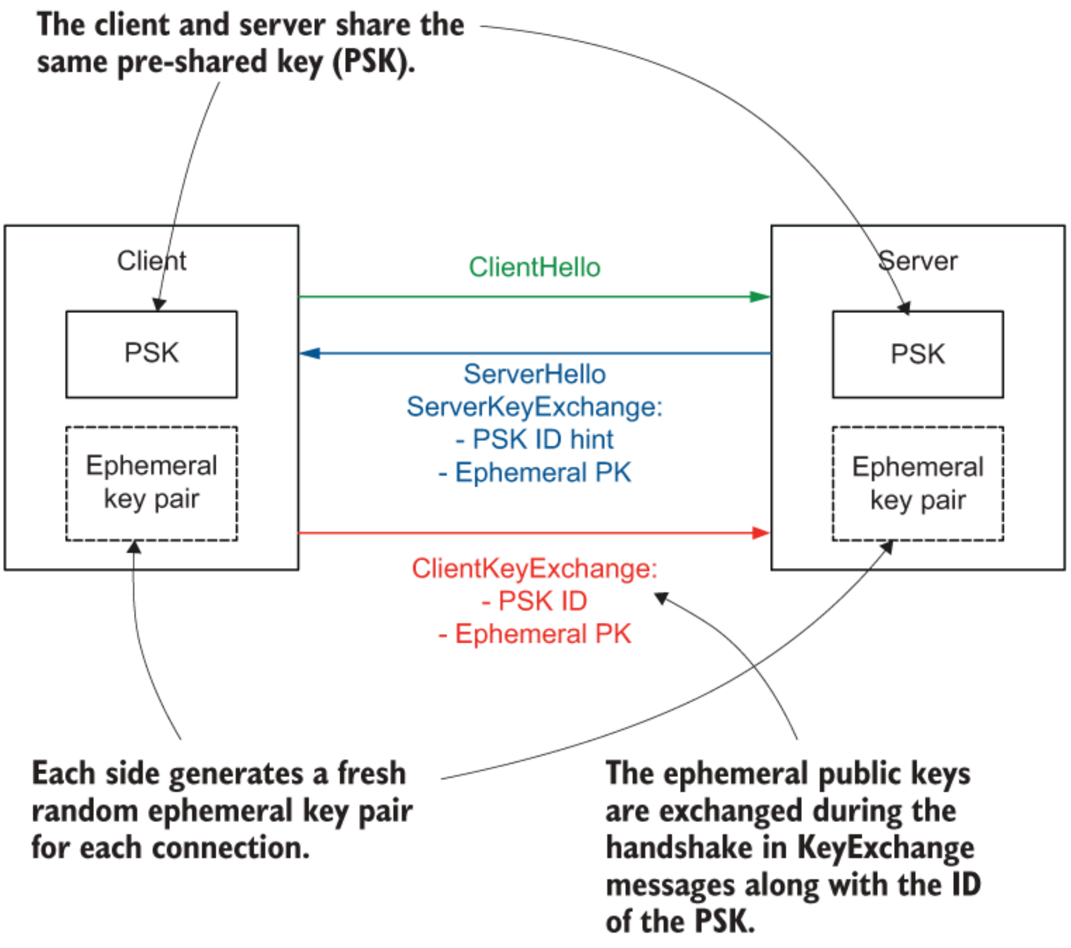

The client and server share the same pre-shared key (PSK).
Client
PSK
Ephemeral key pair
ClientHello
ServerHello
ServerKeyExchange:
- PSK ID hint
- Ephemeral PK
ClientKeyExchange:
- PSK ID
- Ephemeral PK
Server
PSK
Ephemeral key pair
Each side generates a fresh random ephemeral key pair for each connection.
The ephemeral public keys are exchanged during the handshake in KeyExchange messages along with the ID of the PSK.

Figure 12.6 PSK cipher suites with forward secrecy use ephemeral key pairs in addition to the PSK. The client and server swap ephemeral public keys in key exchange messages during the TLS handshake. A Diffie-Hellman key agreement is then performed between each side's ephemeral private key and the received ephemeral public key, which produces an identical secret value that is then mixed into the TLS key derivation process.

Table 12.3 shows some recommended PSK cipher suites for TLS or DTLS 1.2 that provide forward secrecy. The ephemeral Diffie-Hellman keys can be based on either the original finite-field Diffie-Hellman, in which case the suite names contain DHE, or on elliptic curve Diffie-Hellman, in which case they contain ECDHE. In general, the ECDHE variants are better-suited to constrained devices because secure parameters for DHE require large key sizes of 2048 bits or more. The newer X25519 elliptic curve is efficient and secure when implemented in software, but it has only recently been standardized for use in TLS 1.3.[6] The secp256r1 curve (also known as prime256v1 or P-256) is commonly implemented by low-cost secure element microchips and is a reasonable choice too.

Table 12.3 PSK cipher suites with forward secrecy

Cipher suite	Description
`TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256`	PSK with ECDHE followed by AES-CCM with 128-bit keys and 128-bit authentication tags. SHA-256 is used for key derivation and handshake authentication.
`TLS_DHE_PSK_WITH_AES_128_CCM` `TLS_DHE_PSK_WITH_AES_256_CCM`	PSK with DHE followed by AES-CCM with either 128-bit or 256-bit keys. These also use SHA-256 for key derivation and handshake authentication.
`TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256` `TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256`	PSK with either DHE or ECDHE followed by ChaCha20-Poly1305.

All of the CCM cipher suites also come in a CCM_8 variant that uses a short 64-bit authentication tag. As previously discussed, these variants should only be used if you need to save every byte of network use and you are confident that you have alternative measures in place to ensure authenticity of network traffic. AES-GCM is also supported by PSK cipher suites, but I would not recommend it in constrained environments due to the increased risk of accidental nonce reuse.

Pop quiz

3. True or False: PSK cipher suites without forward secrecy derive the same encryption keys for every session.
4. Which one of the following cryptographic primitives is used to ensure forward secrecy in PSK cipher suites that support this?
    1. RSA encryption
    2. RSA signatures
    3. HKDF key derivation
    4. Diffie-Hellman key agreement
    5. Elliptic curve digital signatures

The answers are at the end of the chapter.

12.3 End-to-end security

TLS and DTLS provide excellent security when an API client can talk directly to the server. However, as mentioned in the introduction to section 12.1, in a typical IoT application messages may travel over multiple different protocols. For example, sensor data produced by devices may be sent over low-power wireless networks to a local gateway, which then puts them onto a MQTT message queue for transmission to another service, which aggregates the data and performs a HTTP POST request to a cloud REST API for analysis and storage. Although each hop on this journey can be secured using TLS, messages are available unencrypted while being processed at the intermediate nodes. This makes these intermediate nodes an attractive target for attackers because, once compromised, they can view and manipulate all data flowing through that device.

The solution is to provide end-to-end security of all data, independent of the transport layer security. Rather than relying on the transport protocol to provide encryption and authentication, the message itself is encrypted and authenticated. For example, an API that expects requests with a JSON payload (or an efficient binary alternative) can be adapted to accept data that has been encrypted with an authenticated encryption algorithm, which it then manually decrypts and verifies as shown in figure 12.7. This ensures that an API request encrypted by the original client can only be decrypted by the destination API, no matter how many different network protocols are used to transport the request from the client to its destination.

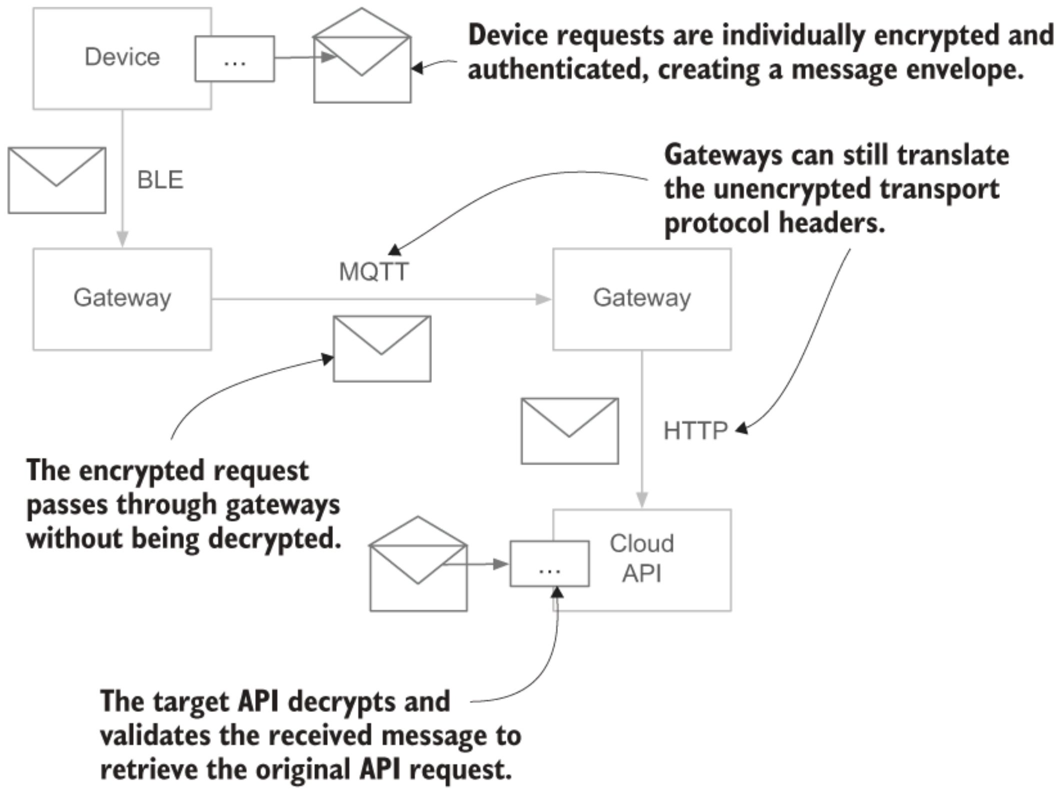

Device
...
Device requests are individually encrypted and authenticated, creating a message envelope.
BLE
Gateways can still translate the unencrypted transport protocol headers.
Gateway
MQTT
Gateway
HTTP
The encrypted request passes through gateways without being decrypted.
...
Cloud API
The target API decrypts and validates the received message to retrieve the original API request.

Figure 12.7 In end-to-end security, API requests are individually encrypted and authenticated by the client device. These encrypted requests can then traverse multiple transport protocols without being decrypted. The API can then decrypt the request and verify it hasn't been tampered with before processing the API request.

NOTE End-to-end security is not a replacement for transport layer security. Transport protocol messages contain headers and other details that are not protected by end-to-end encryption or authentication. You should aim to include security at both layers of your architecture.

End-to-end security involves more than simply encrypting and decrypting data packets. Secure transport protocols, such as TLS, also ensure that both parties are adequately authenticated, and that data packets cannot be reordered or replayed. In the next few sections you'll see how to ensure the same protections are provided when using end-to-end security.

12.3.1 COSE

If you wanted to ensure end-to-end security of requests to a regular JSON-based REST API, you might be tempted to look at the JOSE (JSON Object Signing and Encryption) standards discussed in chapter 6. For IoT applications, JSON is often replaced by more efficient binary encodings that make better use of constrained memory and network bandwidth and that have compact software implementations. For example, numeric data such as sensor readings is typically encoded as decimal strings in JSON, with only 10 possible values for each byte, which is wasteful compared to a packed binary encoding of the same data.

Several binary alternatives to JSON have become popular in recent years to overcome these problems. One popular choice is Concise Binary Object Representation (CBOR), which provides a compact binary format that roughly follows the same model as JSON, providing support for objects consisting of key-value fields, arrays, text and binary strings, and integer and floating-point numbers. Like JSON, CBOR can be parsed and processed without a schema. On top of CBOR, the CBOR Object Signing and Encryption (COSE; **https://tools.ietf.org/html/rfc8152**) standards provide similar cryptographic capabilities as JOSE does for JSON.

DEFINITION CBOR (Concise Binary Object Representation) is a binary alternative to JSON. COSE (CBOR Object Signing and Encryption) provides encryption and digital signature capabilities for CBOR and is loosely based on JOSE.

Although COSE is loosely based on JOSE, it has diverged quite a lot, both in the algorithms supported and in how messages are formatted. For example, in JOSE symmetric MAC, algorithms like HMAC are part of JWS (JSON Web Signatures) and treated as equivalent to public key signature algorithms. In COSE, MACs are treated more like authenticated encryption algorithms, allowing the same key agreement and key wrapping algorithms to be used to transmit a per-message MAC key.

In terms of algorithms, COSE supports many of the same algorithms as JOSE, and adds additional algorithms that are more suited to constrained devices, such as AES-CCM and ChaCha20-Poly1305 for authenticated encryption, and truncated version of HMAC-SHA-256 that produces a smaller 64-bit authentication tag. It also removes some algorithms with perceived weaknesses, such as RSA with PKCS#1 v1.5 padding and AES in CBC mode with a separate HMAC tag. Unfortunately, dropping support for CBC mode means that all of the COSE authenticated encryption algorithms require nonces that are too small to generate randomly. This is a problem, because when implementing end-to-end encryption, there are no session keys or record sequence numbers that can be used to safely implement a deterministic nonce.

Thankfully, COSE has a solution in the form of HKDF (hash-based key derivation function) that you used in chapter 11. Rather than using a key to directly encrypt a message, you can instead use the key along with a random nonce to derive a unique key for every message. Because nonce reuse problems only occur if you reuse a nonce with the same key, this reduces the risk of accidental nonce reuse considerably, assuming that your devices have access to an adequate source of random data (see section 12.3.2 if they don't).

To demonstrate the use of COSE for encrypting messages, you can use the Java reference implementation from the COSE working group. Open the pom.xml file in your editor and add the following lines to the dependencies section:7

```
<dependency>
  <groupId>com.augustcellars.cose</groupId>
  <artifactId>cose-java</artifactId>
  <version>1.1.0</version>
</dependency>
```

Listing 12.13 shows an example of encrypting a message with COSE using HKDF to derive a unique key for the message and AES-CCM with a 128-bit key for the message encryption, which requires installing Bouncy Castle as a cryptography provider. For this example, you can reuse the PSK from the examples in section 12.2.1. COSE requires a `Recipient` object to be created for each recipient of a message and the HKDF algorithm is specified at this level. This allows different key derivation or wrapping algorithms to be used for different recipients of the same message, but in this example, there's only a single recipient. The algorithm is specified by adding an attribute to the recipient object. You should add these attributes to the `PROTECTED` header region, to ensure they are authenticated. The random nonce is also added to the recipient object, as the `HKDF_Context_PartyU_nonce` attribute; I'll explain the `PartyU` part shortly. You then create an `EncryptMessage` object and set some content for the message. Here I've used a simple string, but you can also pass any array of bytes. Finally, you specify the content encryption algorithm as an attribute of the message (a variant of AES-CCM in this case) and then encrypt it.

Listing 12.13 Encrypting a message with COSE HKDF

```
Security.addProvider(new BouncyCastleProvider());                    ❶
var keyMaterial = PskServer.loadPsk("changeit".toCharArray());       ❷

var recipient = new Recipient();                                     ❸
var keyData = CBORObject.NewMap()                                        ❸
        .Add(KeyKeys.KeyType.AsCBOR(), KeyKeys.KeyType_Octet)            ❸
        .Add(KeyKeys.Octet_K.AsCBOR(), keyMaterial);                     ❸
recipient.SetKey(new OneKey(keyData));                                   ❸
recipient.addAttribute(HeaderKeys.Algorithm,                             ❹
        AlgorithmID.HKDF_HMAC_SHA_256.AsCBOR(),                          ❹
        Attribute.PROTECTED);                                            ❹
var nonce = new byte[16];                                                ❺
new SecureRandom().nextBytes(nonce);                                     ❺
recipient.addAttribute(HeaderKeys.HKDF_Context_PartyU_nonce,             ❺
        CBORObject.FromObject(nonce), Attribute.PROTECTED);              ❺

var message = new EncryptMessage();                                      ❻
message.SetContent("Hello, World!");                                     ❻
message.addAttribute(HeaderKeys.Algorithm,                               ❻
        AlgorithmID.AES_CCM_16_128_128.AsCBOR(),                         ❻
        Attribute.PROTECTED);                                            ❻
message.addRecipient(recipient);                                         ❻

message.encrypt();                                                       ❼
System.out.println(Base64url.encode(message.EncodeToBytes()));           ❼
```

❶ Install Bouncy Castle to get AES-CCM support.

❷ Load the key from the keystore.

❸ Encode the key as a COSE key object and add to the recipient.

❹ The KDF algorithm is specified as an attribute of the recipient.

❺ The nonce is also set as an attribute on the recipient.

❻ Create the message and specify the content encryption algorithm.

❼ Encrypt the message and output the encoded result.

The HKDF algorithm in COSE supports specifying several fields in addition to the PartyU nonce, as shown in table 12.4, which allows the derived key to be bound to several attributes, ensuring that distinct keys are derived for different uses. Each attribute can be set for either Party U or Party V, which are just arbitrary names for the participants in a communication protocol. In COSE, the convention is that the sender of a message is Party U and the recipient is Party V. By simply swapping the Party U and Party V roles around, you can ensure that distinct keys are derived for each direction of communication, which provides a useful protection against reflection attacks. Each party can contribute a nonce to the KDF, as well as identity information and any other contextual information. For example, if your API can receive many different types of requests, you could include the request type in the context to ensure that different keys are used for different types of requests.

DEFINITION A reflection attack occurs when an attacker intercepts a message from Alice to Bob and replays that message back to Alice. If symmetric message authentication is used, Alice may be unable to distinguish this from a genuine message from Bob. Using distinct keys for messages from Alice to Bob than messages from Bob to Alice prevents these attacks.

Table 12.4 COSE HKDF context fields

Field	Purpose
PartyU identity PartyV identity	An identifier for party U and V. This might be a username or domain name or some other application-specific identifier.
PartyU nonce PartyV nonce	Nonces contributed by either or both parties. These can be arbitrary random byte arrays or integers. Although these could be simple counters it's best to generate them randomly in most cases.
PartyU other PartyV other	Any application-specific additional context information that should be included in the key derivation.

HKDF context fields can either be explicitly communicated as part of the message, or they can be agreed on by parties ahead of time and be included in the KDF computation without being included in the message. If a random nonce is used, then this obviously needs to be included in the message; otherwise, the other party won't be able to guess it. Because the fields are included in the key derivation process, there is no need to separately authenticate them as part of the message: any attempt to tamper with them will cause an incorrect key to be derived. For this reason, you can put them in an `UNPROTECTED` header which is not protected by a MAC.

Although HKDF is designed for use with hash-based MACs, COSE also defines a variant of it that can use a MAC based on AES in CBC mode, known as HKDF-AES-MAC (this possibility was explicitly discussed in Appendix D of the original HKDF proposal, see https://eprint.iacr.org/2010/264.pdf).

This eliminates the need for a hash function implementation, saving some code size on constrained devices. This can be particularly important on low-power devices because some secure element chips provide hardware support for AES (and even public key cryptography) but have no support for SHA-256 or other hash functions, requiring devices to fall back on slower and less efficient software implementations.

NOTE You'll recall from chapter 11 that HKDF consists of two functions: an extract function that derives a master key from some input key material, and an expand function that derives one or more new keys from the master key. When used with a hash function, COSE's HKDF performs both functions. When used with AES it only performs the expand phase; this is fine because the input key is already uniformly random as explained in chapter 11.8

In addition to symmetric authenticated encryption, COSE supports a range of public key encryption and signature options, which are mostly very similar to JOSE, so I won't cover them in detail here. One public key algorithm in COSE that is worth highlighting in the context of IoT applications is support for elliptic curve Diffie-Hellman (ECDH) with static keys for both the sender and receiver, known as ECDH-SS. Unlike the ECDH-ES encryption scheme supported by JOSE, ECDH-SS provides sender authentication, avoiding the need for a separate signature over the contents of each message. The downside is that ECDH-SS always derives the same key for the same pair of sender and receiver, and so can be vulnerable to replay attacks and reflection attacks, and lacks any kind of forward secrecy. Nevertheless, when used with HKDF and making use of the context fields in table 12.4 to bind derived keys to the context in which they are used, ECDH-SS can be a very useful building block in IoT applications.

12.3.2 Alternatives to COSE

Although COSE is in many ways better designed than JOSE and is starting to see wide adoption in standards such as FIDO 2 for hardware security keys (https://fidoalliance .org/fido2/), it still suffers from the same problem of trying to do too much. It supports a wide variety of cryptographic algorithms, with varying security goals and qualities. At the time of writing, I counted 61 algorithm variants registered in the COSE algorithms registry (http://mng.bz/awDz), the vast majority of which are marked as recommended. This desire to cover all bases can make it hard for developers to know which algorithms to choose and while many of them are fine algorithms, they can lead to security issues when misused, such as the accidental nonce reuse issues you've learned about in the last few sections.

SHA-3 and STROBE

The US National Institute of Standards and Technology (NIST) recently completed an international competition to select the algorithm to become SHA-3, the successor to the widely used SHA-2 hash function family. To protect against possible future weaknesses in SHA-2, the winning algorithm (originally known as Keccak) was chosen partly because it is very different in structure to SHA-2. SHA-3 is based on an elegant and flexible cryptographic primitive known as a sponge construction. Although SHA-3 is relatively slow in software, it is well-suited to efficient hardware implementations. The Keccak team have subsequently implemented a wide variety of cryptographic primitives based on the same core sponge construction: other hash functions, MACs, and authenticated encryption algorithms. See **https://keccak.team** for more details.

Mike Hamburg's STROBE framework (**https://strobe.sourceforge.io**) builds on top of the SHA-3 work to create a framework for cryptographic protocols for IoT applications. The design allows a single small core of code to provide a wide variety of cryptographic protections, making a compelling alternative to AES for constrained devices. If hardware support for the Keccak core functions becomes widely available, then frameworks like STROBE may become very attractive.

If you need standards-based interoperability with other software, the COSE can be a fine choice for an IoT ecosystem, so long as you approach it with care. In many cases, however, interoperability is not a requirement because you control all of the software and devices being deployed. In this a simpler approach can be adopted, such as using NaCl (the Networking and Cryptography Library; **https://nacl.cr.yp.to**) to encrypt and authenticate a packet of data just as you did in chapter 6. You can still use CBOR or another compact binary encoding for the data itself, but NaCl (or a rewrite of it, like libsodium) takes care of choosing appropriate cryptographic algorithms, vetted by genuine experts. Listing 12.14 shows how easy it is to encrypt a CBOR object using NaCl's `SecretBox` functionality (in this case through the pure Java Salty Coffee library you used in chapter 6), which is roughly equivalent to the COSE example from the previous section. First you load or generate the secret key, and then you encrypt your CBOR data using that key.

Listing 12.14 Encrypting CBOR with NaCl

```
var key = SecretBox.key();                                        ❶
var cborMap = CBORObject.NewMap()                                 ❷
        .Add("foo", "bar")                                        ❷
        .Add("data", 12345);                                      ❷
var box = SecretBox.encrypt(key, cborMap.EncodeToBytes());        ❸
System.out.println(box);
```

❶ Create or load a key.

❷ Generate some CBOR data.

❸ Encrypt the data.

NaCl's secret box is relatively well suited to IoT applications for several reasons:

- It uses a 192-bit per-message nonce, which minimizes the risk of accidental nonce reuse when using randomly generated values. This is the maximum size of nonce, so you can use a shorter value if you absolutely need to save space and pad it with zeroes before decrypting. Reducing the size increases the risk of accidental nonce reuse, so you should avoid reducing it to much less than 128 bits.
- The XSalsa20 cipher and Poly1305 MAC used by NaCl can be compactly implemented in software on a wide range of devices. They are particularly suited to 32-bit architectures, but there are also fast implementations for 8-bit microcontrollers. They therefore make a good choice on platforms without hardware AES support.
- The 128-bit authentication tag use by Poly1305 is a good trade-off between security and message expansion. Although stronger MAC algorithms exist, the authentication tag only needs to remain secure for the lifetime of the message (until it expires, for example), whereas the contents of the message may need to remain secret for a lot longer.

If your devices are capable of performing public key cryptography, then NaCl also provides convenient and efficient public key authenticated encryption in the form the `CryptoBox` class, shown in listing 12.15. The `CryptoBox` algorithm works a lot like COSE's ECDH-SS algorithm in that it performs a static key agreement between the two parties. Each party has their own key pair along with the public key of the other party (see section 12.4 for a discussion of key distribution). To encrypt, you use your own private key and the recipient’s public key, and to decrypt, the recipient uses their private key and your public key. This shows that even public key cryptography is not much more work when you use a well-designed library like NaCl.

WARNING Unlike COSE’s HKDF, the key derivation performed in NaCl’s crypto box doesn’t bind the derived key to any context material. You should make sure that messages themselves contain the identities of the sender and recipient and sufficient context to avoid reflection or replay attacks.

Listing 12.15 Using NaCl’s CryptoBox

```
var senderKeys = CryptoBox.keyPair();                               ❶
var recipientKeys = CryptoBox.keyPair();                            ❶
var cborMap = CBORObject.NewMap()
        .Add("foo", "bar")
        .Add("data", 12345);
var sent = CryptoBox.encrypt(senderKeys.getPrivate(),               ❷
        recipientKeys.getPublic(), cborMap.EncodeToBytes());        ❷

var recvd = CryptoBox.fromString(sent.toString());
var cbor = recvd.decrypt(recipientKeys.getPrivate(),                ❸
        senderKeys.getPublic());                                    ❸
System.out.println(CBORObject.DecodeFromBytes(cbor));
```

❶ The sender and recipient each have a key pair.

❷ Encrypt using your private key and the recipient’s public key.

❸ The recipient decrypts with their private key and your public key.

12.3.3 Misuse-resistant authenticated encryption

Although NaCl and COSE can both be used in ways that minimize the risk of nonce reuse, they only do so on the assumption that a device has access to some reliable source of random data. This is not always the case for constrained devices, which often lack access to good sources of entropy or even reliable clocks that could be used for deterministic nonces. Pressure to reduce the size of messages may also result in developers using nonces that are too small to be randomly generated safely. An attacker may also be able to influence conditions to make nonce reuse more likely, such as by tampering with the clock, or exploiting weaknesses in network protocols, as occurred in the KRACK attacks against WPA2 (**https://www.krackattacks .com**). In the worst case, where a nonce is reused for many messages, the algorithms in NaCl and COSE both fail catastrophically, enabling an attacker to recover a lot of information about the encrypted data and in some cases to tamper with that data or construct forgeries.

To avoid this problem, cryptographers have developed new modes of operation for ciphers that are much more resistant to accidental or malicious nonce reuse. These modes of operation achieve a security goal called misuse-resistant authenticated encryption (MRAE). The most well-known such algorithm is SIV-AES, based on a mode of operation known as Synthetic Initialization Vector (SIV; **https://tools.ietf.org/html/rfc5297**). In normal use with unique nonces, SIV mode provides the same guarantees as any other authenticated encryption cipher. But if a nonce is reused, a MRAE mode doesn't fail as catastrophically: an attacker could only tell if the exact same message had been encrypted with the same key and nonce. No loss of authenticity or integrity occurs at all. This makes SIV-AES and other MRAE modes much safer to use in environments where it might be hard to guarantee unique nonces, such as IoT devices.

DEFINITION A cipher provides misuse-resistant authenticated encryption (MRAE) if accidental or deliberate nonce reuse results in only a small loss of security. An attacker can only learn if the same message has been encrypted twice with the same nonce and key and there is no loss of authenticity. Synthetic Initialization Vector (SIV) mode is a well-known MRAE mode, and SIV-AES the most common use of it.

SIV mode works by computing the nonce (also known as an Initialization Vector or IV) using a pseudorandom function (PRF) rather than using a purely random value or counter. Many MACs used for authentication are also PRFs, so SIV reuses the MAC used for authentication to also provide the IV, as shown in figure 12.8.

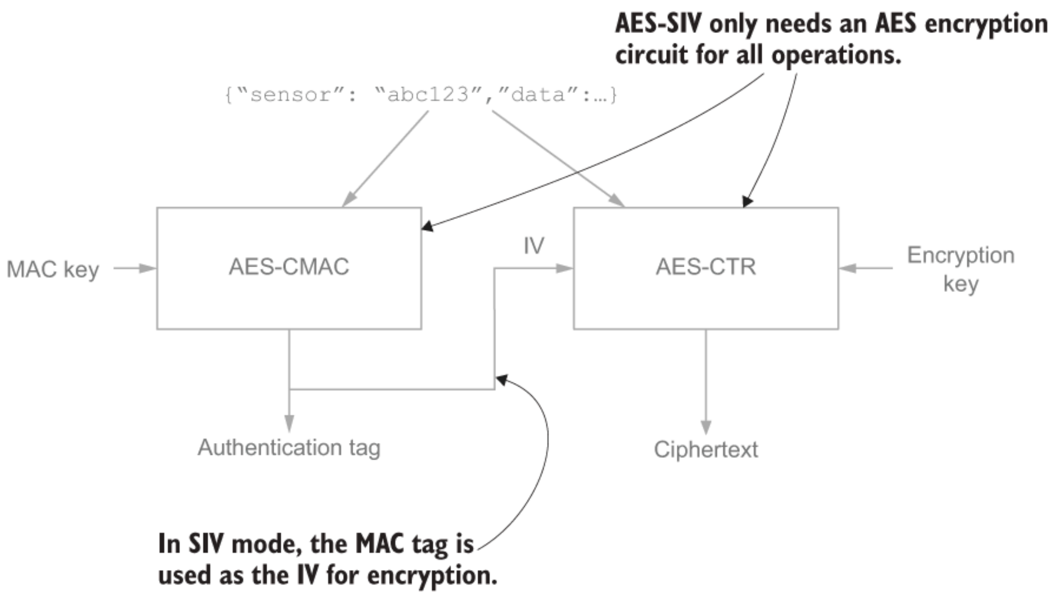

AES-SIV only needs an AES encryption circuit for all operations.
{"sensor": "abc123","data":…}
MAC key
AES-CMAC
IV
AES-CTR
Encryption key
Authentication tag
Ciphertext
In SIV mode, the MAC tag is used as the IV for encryption.

Figure 12.8 SIV mode uses the MAC authentication tag as the IV for encryption. This ensures that the IV will only repeat if the message is identical, eliminating nonce reuse issues that can cause catastrophic security failures. SIV-AES is particularly suited to IoT environments because it only needs an AES encryption circuit to perform all operations (even decryption).

CAUTION Not all MACs are PRFs so you should stick to standard implementations of SIV mode rather than inventing your own.

The encryption process works by making two passes over the input:

1. First, a MAC is computed over the plaintext input and any associated data.[9] The MAC tag is known as the Synthetic IV, or SIV.
2. Then the plaintext is encrypted using a different key using the MAC tag from step 1 as the nonce.

The security properties of the MAC ensure that it is extremely unlikely that two different messages will result in the same MAC tag, and so this ensures that the same nonce is not reused with two different messages. The SIV is sent along with the message, just as a normal MAC tag would be. Decryption works in reverse: first the ciphertext is decrypted using the SIV, and then the correct MAC tag is computed and compared with the SIV. If the tags don't match, then the message is rejected.

WARNING Because the authentication tag can only be validated after the message has been decrypted, you should be careful not to process any decrypted data before this crucial authentication step has completed.

In SIV-AES, the MAC is AES-CMAC, which is an improved version of the AES-CBC-MAC used in COSE. Encryption is performed using AES in CTR mode. This means that SIV-AES has the same nice property as AES-CCM: it requires only an AES encryption circuit for all operations (even decryption), so can be compactly implemented.

Side-channel and fault attacks

Although SIV mode protects against accidental or deliberate misuse of nonces, it doesn't protect against all possible attacks in an IoT environment. When an attacker may have direct physical access to devices, especially where there is limited physical protection or surveillance, you may also need to consider other attacks. A secure element chip can provide some protection against tampering and attempts to read keys directly from memory, but keys and other secrets may also leak though many side channels. A side channel occurs when information about a secret can be deduced by measuring physical aspects of computations using that secret, such as the following:

- The timing of operations may reveal information about the key. Modern cryptographic implementations are designed to be constant time to avoid leaking information about the key in this way. Many software implementations of AES are not constant time, so alternative ciphers such as ChaCha20 are often preferred for this reason.
- The amount of power used by a device may vary depending on the value of secret data it is processing. Differential power analysis can be used to recover secret data by examining how much power is used when processing different inputs.
- Emissions produced during processing, including electromagnetic radiation, heat, or even sounds have all been used to recover secret data from cryptographic computations.

As well as passively observing physical aspects of a device, an attacker may also directly interfere with a device in an attempt to recover secrets. In a fault attack, an attacker disrupts the normal functioning of a device in the hope that the faulty operation will reveal some information about secrets it is processing. For example, tweaking the power supply (known as a glitch) at a well-chosen moment might cause an algorithm to reuse a nonce, leaking information about messages or a private key. In some cases, deterministic algorithms such as SIV-AES can actually make fault attacks easier for an attacker.

Protecting against side-channel and fault attacks is well beyond the scope of this book. Cryptographic libraries and devices will document if they have been designed to resist these attacks. Products may be certified against standards such as FIPS 140-2 or Commons Criteria, which both provide some assurance that the device will resist some physical attacks, but you need to read the fine print to determine exactly which threats have been tested.
So far, the mode I've described will always produce the same nonce and the same ciphertext whenever the same plaintext message is encrypted. If you recall from chapter 6, such an encryption scheme is not secure because an attacker can easily tell if the same message has been sent multiple times. For example, if you have a sensor sending packets of data containing sensor readings in a small range of values, then an observer may be able to work out what the encrypted sensor readings are after seeing enough of them. This is why normal encryption modes add a unique nonce or random IV in every message: to ensure that different ciphertext is produced even if the same message is encrypted. SIV mode solves this problem by allowing you to include a random IV in the associated data that accompanies the message. Because this associated data is also included in the MAC calculation, it ensures that the calculated SIV will be different even if the message is the same. To make this a bit easier, SIV mode allows more than one associated data block to be provided to the cipher--up to 126 blocks in SIV-AES.

Listing 12.16 shows an example of encrypting some data with SIV-AES in Java using an open source library that implements the mode using AES primitives from Bouncy Castle.10 To include the library, open the pom.xml file and add the following lines to the dependencies section:

```
<dependency>
  <groupId>org.cryptomator</groupId>
  <artifactId>siv-mode</artifactId>
  <version>1.3.2</version>
</dependency>
```

SIV mode requires two separate keys: one for the MAC and one for encryption and decryption. The specification that defines SIV-AES (https://tools.ietf.org/html/rfc5297) describes how a single key that is twice as long as normal can be split into two, with the first half becoming the MAC key and the second half the encryption key. This is demonstrated in listing 12.16 by splitting the existing 256-bit PSK key into two 128-bit keys. You could also derive the two keys from a single master key using

HKDF, as you learned in chapter 11. The library used in the listing provides `encrypt ()` and `decrypt()` methods that take the encryption key, the MAC key, the plaintext (or ciphertext for decryption), and then any number of associated data blocks. In this example, you'll pass in a header and a random IV. The SIV specification recommends that any random IV should be included as the last associated data block.

TIP The `SivMode` class from the library is thread-safe and designed to be reused. If you use this library in production, you should create a single instance of this class and reuse it for all calls.

Listing 12.16 Encrypting data with SIV-AES

```
var psk = PskServer.loadPsk("changeit".toCharArray());                      ❶
var macKey = new SecretKeySpec(Arrays.copyOfRange(psk, 0, 16),              ❶
        "AES");                                                             ❶
var encKey = new SecretKeySpec(Arrays.copyOfRange(psk, 16, 32),             ❶
        "AES");                                                             ❶
var randomIv = new byte[16];                                                ❷
new SecureRandom().nextBytes(randomIv);                                     ❷
var header = "Test header".getBytes();
var body = CBORObject.NewMap()
        .Add("sensor", "F5671434")
        .Add("reading", 1234).EncodeToBytes();

var siv = new SivMode();
var ciphertext = siv.encrypt(encKey, macKey, body,                          ❸
        header, randomIv);                                                  ❸
var plaintext = siv.decrypt(encKey, macKey, ciphertext,                     ❹
        header, randomIv);                                                  ❹
```

❶ Load the key and split into separate MAC and encryption keys.

❷ Generate a random IV with the best entropy you have available.

❸ Encrypt the body passing the header and random IV as associated data.

❹ Decrypt by passing the same associated data blocks.

Pop quiz

5. Misuse-resistant authenticated encryption (MRAE) modes of operation protect against which one of the following security failures?
    1. Overheating
    2. Nonce reuse
    3. Weak passwords
    4. Side-channel attacks
    5. Losing your secret keys
6. True or False: SIV-AES is just as secure even if you repeat a nonce.

The answers are at the end of the chapter.

12.4 Key distribution and management

In a normal API architecture, the problem of how keys are distributed to clients and servers is solved using a public key infrastructure (PKI), as you learned in chapter 10. To recap:

- In this architecture, each device has its own private key and associated public key.
- The public key is packaged into a certificate that is signed by a certificate authority (CA) and each device has a permanent copy of the public key of the CA.
- When a device connects to another device (or receives a connection), it presents its certificate to identify itself. The device authenticates with the associated private key to prove that it is the rightful holder of this certificate.
- The recipient can verify the identity of the other device by checking that its certificate is signed by the trusted CA and has not expired, been revoked, or in any other way become invalid.

This architecture can also be used in IoT environments and is often used for more capable devices. But constrained devices that lack the capacity for public key cryptography are unable to make use of a PKI and so other alternatives must be used, based on symmetric cryptography. Symmetric cryptography is efficient but requires the API client and server to have access to the same key, which can be a challenge if there are a large number of devices involved. The key distribution techniques described in the next few sections aim to solve this problem.

12.4.1 One-off key provisioning

The simplest approach is to provide each device with a key at the time of device manufacture or at a later stage when a batch of devices is initially acquired by an organization. One or more keys are generated securely and then permanently stored in read-only memory (ROM) or EEPROM (electrically erasable programmable ROM) on the device. The same keys are then encrypted and packaged along with device identity information and stored in a central directory such as LDAP, where they can be accessed by API servers to authenticate and decrypt requests from clients or to encrypt responses to be sent to those devices. The architecture is shown in figure 12.9. A hardware security module (HSM) can be used to securely store the master encryption keys inside the factory to prevent compromise.

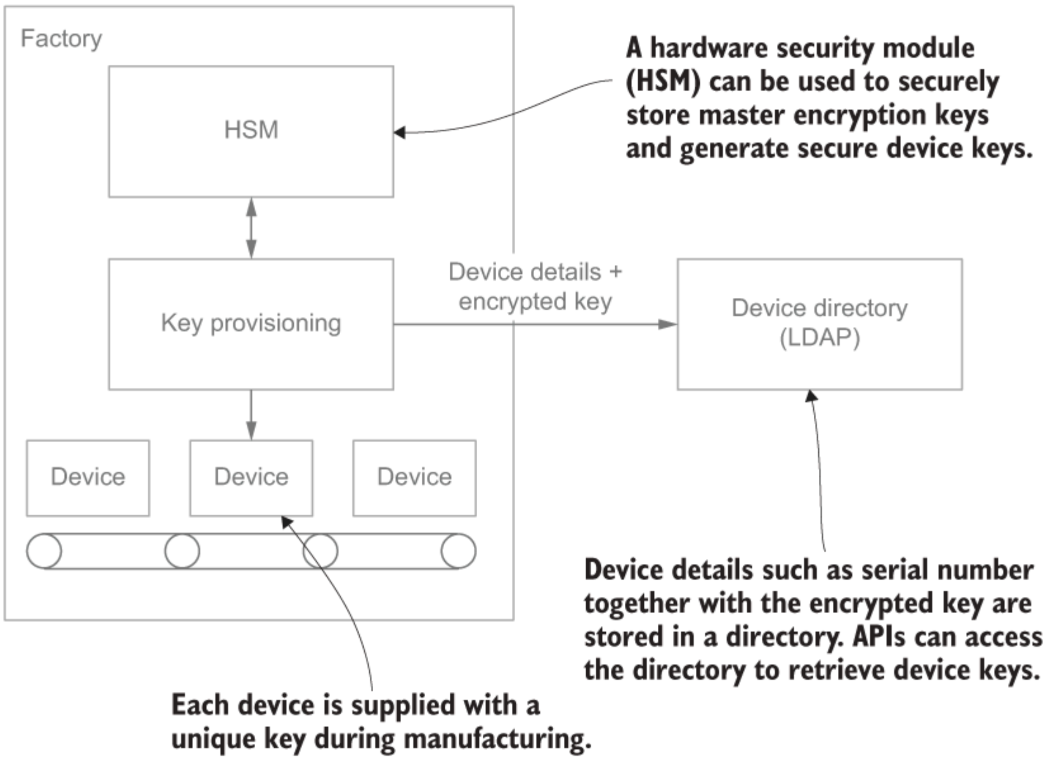

Factory
HSM
Key provisioning
Device
Device
Device
Device details + encrypted key
Device directory (LDAP)
A hardware security module (HSM) can be used to securely store master encryption keys and generate secure device keys.
Device details such as serial number together with the encrypted key are stored in a directory. APIs can access the directory to retrieve device keys.
Each device is supplied with a unique key during manufacturing.

Figure 12.9 Unique device keys can be generated and installed on a device during manufacturing. The device keys are then encrypted and stored along with device details in an LDAP directory or database. APIs can later retrieve the encrypted device keys and decrypt them to secure communications with that device.

An alternative to generating completely random keys during manufacturing is to derive device-specific keys from a master key and some device-specific information. For example, you can use HKDF from chapter 11 to derive a unique device-specific key based on a unique serial number or ethernet hardware address assigned to each device. The derived key is stored on the device as before, but the API server can derive the key for each device without needing to store them all in a database. When the device connects to the server, it authenticates by sending the unique information (along with a timestamp or a random challenge to prevent replay), using its device key to create a MAC. The server can then derive the same device key from the master key and use this to verify the MAC. For example, Microsoft's Azure IoT Hub Device Provisioning Service uses a scheme similar to this for group enrollment of devices using a symmetric key; for more information, see **http://mng.bz/gg4l**.

12.4.2 Key distribution servers

Rather than installing a single key once when a device is first acquired, you can instead periodically distribute keys to devices using a key distribution server. In this model, the device uses its initial key to enroll with the key distribution server and then is supplied with a new key that it can use for future communications. The key distribution server can also make this key available to API servers when they need to communicate with that device.

LEARN MORE The E4 product from Teserakt (**https://teserakt.io/e4/**) includes a key distribution server that can distribute encrypted keys to devices over the MQTT messaging protocol. Teserakt has published a series of articles on the design of its secure IoT architecture, designed by respected cryptographers, at **http://mng.bz/5pKz**.

Once the initial enrollment process has completed, the key distribution server can periodically supply a fresh key to the device, encrypted using the old key. This allows the device to frequently change its keys without needing to generate them locally, which is important because constrained devices are often severely limited in access to sources of entropy.

Remote attestation and trusted execution

Some devices may be equipped with secure hardware that can be used to establish trust in a device when it is first connected to an organization's network. For example, the device might have a Trusted Platform Module (TPM), which is a type of hardware security module (HSM) made popular by Microsoft. A TPM can prove to a remote server that it is a particular model of device from a known manufacturer with a particular serial number, in a process known as remote attestation. Remote attestation is achieved using a challenge-response protocol based on a private key, known as an Endorsement Key (EK), that is burned into the device at manufacturing time. The TPM uses the EK to sign an attestation statement indicating the make and model of the device and can also provide details on the current state of the device and attached hardware. Because these measurements of the device state are taken by firmware running within the secure TPM, they provide strong evidence that the device hasn't been tampered with.

Although TPM attestation is strong, a TPM is not a cheap component to add to your IoT devices. Some CPUs include support for a Trusted Execution Environment (TEE), such as ARM TrustZone, which allows signed software to be run in a special secure mode of execution, isolated from the normal operating system and other code. Although less resistant to physical attacks than a TPM, a TEE can be used to implement security critical functions such as remote attestation. A TEE can also be used as a poor man's HSM, providing an additional layer of security over pure software solutions.
Rather than writing a dedicated key distribution server, it is also possible to distribute keys using an existing protocol such as OAuth2. A draft standard for OAuth2 (currently expired, but periodically revived by the OAuth working group) describes how to distribute encrypted symmetric keys alongside an OAuth2 access token (**http:// mng.bz/6AZy**), and RFC 7800 describes how such a key can be encoded into a JSON Web Token (**https://tools.ietf.org/html/rfc7800#section-3.3**). The same technique can be used with CBOR Web Tokens (**http://mng.bz/oRaM**). These techniques allow a device to be given a fresh key every time it gets an access token, and any API servers it communicates with can retrieve the key in a standard way from the access token itself or through token introspection. Use of OAuth2 in an IoT environment is discussed further in chapter 13.

12.4.3 Ratcheting for forward secrecy

If your IoT devices are sending confidential data in API requests, using the same encryption key for the entire lifetime of the device can present a risk. If the device key is compromised, then an attacker can not only decrypt any future communications but also all previous messages sent by that device. To prevent this, you need to use cryptographic mechanisms that provide forward secrecy as discussed in section 12.2. In that section, we looked at public key mechanisms for achieving forward secrecy, but you can also achieve this security goal using purely symmetric cryptography through a technique known as ratcheting.

DEFINITION Ratcheting in cryptography is a technique for replacing a symmetric key periodically to ensure forward secrecy. The new key is derived from the old key using a one-way function, known as a ratchet, because it only moves in one direction. It's impossible to derive an old key from the new key so previous conversations are secure even if the new key is compromised.

There are several ways to derive the new key from the old one. For example, you can derive the new key using HKDF with a fixed context string as in the following example:

```
var newKey = HKDF.expand(oldKey, "iot-key-ratchet", 32, "HMAC");
```

TIP It is best practice to use HKDF to derive two (or more) keys: one is used for HKDF only, to derive the next ratchet key, while the other is used for encryption or authentication. The ratchet key is sometimes called a chain key or chaining key.

If the key is not used for HMAC, but instead used for encryption using AES or another algorithm, then you can reserve a particular nonce or IV value to be used for the ratchet and derive the new key as the encryption of an all-zero message using that reserved IV, as shown in listing 12.17 using AES in Counter mode. In this example, a 128-bit IV of all 1-bits is reserved for the ratchet operation because it is highly unlikely that this value would be generated by either a counter or a randomly generated IV.

WARNING You should ensure that the special IV used for the ratchet is never used to encrypt a message.

Listing 12.17 Ratcheting with AES-CTR

```
private static byte[] ratchet(byte[] oldKey) throws Exception {
    var cipher = Cipher.getInstance("AES/CTR/NoPadding");
    var iv = new byte[16];                                ❶
    Arrays.fill(iv, (byte) 0xFF);                         ❶
    cipher.init(Cipher.ENCRYPT_MODE,
            new SecretKeySpec(oldKey, "AES"),             ❷
            new IvParameterSpec(iv));                     ❷
    return cipher.doFinal(new byte[32]);                  ❸
}
```

❶ Reserve a fixed IV that is used only for ratcheting.

❷ Initialize the cipher using the old key and the fixed ratchet IV.

❸ Encrypt 32 zero bytes and use the output as the new key.

After performing a ratchet, you should ensure the old key is scrubbed from memory so that it can't be recovered, as shown in the following example:

```
var newKey = ratchet(key);
Arrays.fill(key, (byte) 0);       ❶
key = newKey;                     ❷
```

❶ Overwrite the old key with zero bytes.

❷ Replace the old key with the new key.

TIP In Java and similar languages, the garbage collector may duplicate the contents of variables in memory, so copies may remain even if you attempt to wipe the data. You can use `ByteBuffer.allocateDirect()` to create off-heap memory that is not managed by the garbage collector.

Ratcheting only works if both the client and the server can determine when a ratchet occurs; otherwise, they will end up using different keys. You should therefore perform ratchet operations at well-defined moments. For example, each device might ratchet its key at midnight every day, or every hour, or perhaps even after every 10 messages.[11] The rate at which ratchets should be performed depends on the number of requests that the device sends, and the sensitivity of the data being transmitted.

Ratcheting after a fixed number of messages can help to detect compromise: if an attacker is using a device's stolen secret key, then the API server will receive extra messages in addition to any the device sent and so will perform the ratchet earlier than the legitimate device. If the device discovers that the server is performing ratcheting earlier than expected, then this is evidence that another party has compromised the device secret key.

12.4.4 Post-compromise security

Although forward secrecy protects old communications if a device is later compromised, it says nothing about the security of future communications. There have been many stories in the press in recent years of IoT devices being compromised, so being able to recover security after a compromise is a useful security goal, known as post-compromise security.

DEFINITION Post-compromise security (or future secrecy) is achieved if a device can ensure security of future communications after a device has been compromised. It should not be confused with forward secrecy which protects confidentiality of past communications.

Post-compromise security assumes that the compromise is not permanent, and in most cases it's not possible to retain security in the presence of a persistent compromise. However, in some cases it may be possible to re-establish security once the compromise has ended. For example, a path traversal vulnerability might allow a remote attacker to view the contents of files on a device, but not modify them. Once the vulnerability is found and patched, the attacker's access is removed.

DEFINITION A path traversal vulnerability occurs when a web server allows an attacker to access files that were not intended to be made available by manipulating the URL path in requests. For example, if the web server publishes data under a /data folder, an attacker might send a request for /data/../../../etc/shadow.[12] If the webserver doesn't carefully check paths, then it may serve up the local password file.

If the attacker manages to steal the long-term secret key used by the device, then it can be impossible to regain security without human involvement. In the worst case, the device may need to be replaced or restored to factory settings and reconfigured. The ratcheting mechanisms discussed in section 12.4.3 do not protect against compromise, because if the attacker ever gains access to the current ratchet key, they can easily calculate all future keys.

Hardware security measures, such as a secure element, TPM, or TEE (see section 12.4.1) can provide post-compromise security by ensuring that an attacker never directly gains access to the secret key. An attacker that has active control of the device can use the hardware to compromise communications while they have access, but once that access is removed, they will no longer be able to decrypt or interfere with future communications.

A weaker form of post-compromise security can be achieved if an external source of key material is mixed into a ratcheting process periodically. If the client and server can agree on such key material without the attacker learning it, then any new derived keys will be unpredictable to the attacker and security will be restored. This is weaker than using secure hardware, because if the attacker has stolen the device's key, then, in principle, they can eavesdrop or interfere with all future communications and intercept or control this key material. However, if even a single communication exchange can occur without the attacker interfering, then security can be restored.

There are two main methods to exchange key material between the server and the client:

- They can directly exchange new random values encrypted using the old key. For example, a key distribution server might periodically send the client a new key encrypted with the old one, as described in section 12.4.2, or both parties might send random nonces that are mixed into the key derivation process used in ratcheting (section 12.4.3). This is the weakest approach because a passive attacker who is able to eavesdrop can use the random values directly to derive the new keys.
- They can use Diffie-Hellman key agreement with fresh random (ephemeral) keys to derive new key material. Diffie-Hellman is a public key algorithm in which the client and server only exchange public keys but use local private keys to derive a shared secret. Diffie-Hellman is secure against passive eavesdroppers, but an attacker who is able to impersonate the device with a stolen secret key may still be able to perform an active man-in-the-middle attack to compromise security. IoT devices deployed in accessible locations may be particularly vulnerable to man-in-the-middle attacks because an attacker could have physical access to network connections.

DEFINITION A man-in-the-middle (MitM) attack occurs when an attacker actively interferes with communications and impersonates one or both parties. Protocols such as TLS contain protections against MitM attacks, but they can still occur if long-term secret keys used for authentication are compromised.

Post-compromise security is a difficult goal to achieve and most solutions come with costs in terms of hardware requirements or more complex cryptography. In many IoT applications, the budget would be better spent trying to avoid compromise in the first place, but for particularly sensitive devices or data, you may want to consider adding a secure element or other hardware security mechanism to your devices.

Pop quiz

7. True or False: Ratcheting can provide post-compromise security.

The answer is at the end of the chapter.

Answers to pop quiz questions

1. b. NEED_WRAP indicates that the SSLEngine needs to send data to the other party during the handshake.
2. b. AES-GCM fails catastrophically if a nonce is reused, and this is more likely in IoT applications.
3. False. Fresh keys are derived for each session by exchanging random values during the handshake.
4. d. Diffie-Hellman key agreement with fresh ephemeral key pairs is used to ensure forward secrecy.
5. b. MRAE modes are more robust in the case of nonce reuse.
6. False. SIV-AES is less secure if a nonce is reused but loses a relatively small amount of security compared to other modes. You should still aim to use unique nonces for every message.
7. False. Ratcheting achieves forward secrecy but not post-compromise security. Once an attacker has compromised the ratchet key, they can derive all future keys.

Summary

- IoT devices may be constrained in CPU power, memory, storage or network capacity, or battery life. Standard API security practices, based on web protocols and technologies, are poorly suited to such environments and more efficient alternatives should be used.
- UDP-based network protocols can be protected using Datagram TLS. Alternative cipher suites can be used that are better suited to constrained devices, such as those using AES-CCM or ChaCha20-Poly1305.
- X.509 certificates are complex to verify and require additional signature validation and parsing code, increasing the cost of supporting secure communications. Pre-shared keys can eliminate this overhead and use more efficient symmetric cryptography. More capable devices can combine PSK cipher suites with ephemeral Diffie-Hellman to achieve forward secrecy.
- IoT communications often need to traverse multiple network hops employing different transport protocols. End-to-end encryption and authentication can be used to ensure that confidentiality and integrity of API requests and responses are not compromised if an intermediate host is attacked. The COSE standards provide similar capabilities to JOSE with better suitability for IoT devices, but alternatives such as NaCl can be simpler and more secure.

- Constrained devices often lack access to good sources of entropy to generate random nonces, increasing the risk of nonce reuse vulnerabilities. Misuse-resistant authentication encryption modes, such as SIV-AES, are a much safer choice for such devices and offer similar benefits to AES-CCM for code size.
- Key distribution is a complex problem for IoT environments, which can be solved through simple key management techniques such as the use of key distribution servers. Large numbers of device keys can be managed through key derivation, and ratcheting can be used to ensure forward secrecy. Hardware security features provide additional protection against compromised devices.

---

1. DTLS is limited to securing unicast UDP connections and can’t secure multicast broadcasts currently.

2. Refer to chapter 3 if you haven't installed mkcert yet.

3. See **https://wiki.mozilla.org/Security/Server_Side_TLS**.

4. Thomas Pornin, the author of the BearSSL library, has detailed notes on the cost of different TLS cryptographic algorithms at **https://bearssl.org/support.html**.

5. ChaCha20-Poly1305 also suffers from nonce reuse problems similar to GCM, but to a lesser extent. GCM loses all authenticity guarantees after a single nonce reuse, while ChaCha20-Poly1305 only loses these guarantees for messages encrypted with the duplicate nonce.

6. Support for X25519 has also been added to TLS 1.2 and earlier in a subsequent update; see **https://tools .ietf.org/html/rfc8422**.

7. The author of the reference implementation, Jim Schaad, also runs a winery named August Cellars in Oregon if you are wondering about the domain name.

8. It’s unfortunate that COSE tries to handle both cases in a single class of algorithms. Requiring the expand function for HKDF with a hash function is inefficient when the input is already uniformly random. On the other hand, skipping it for AES is potentially insecure if the input is not uniformly random.

9. The sharp-eyed among you may notice that this is a variation of the MAC-then-Encrypt scheme that we said in chapter 6 is not guaranteed to be secure. Although this is generally true, SIV mode has a proof of security so it is an exception to the rule.

10. At 4.5MB, Bouncy Castle doesn't qualify as a compact implementation, but it shows how SIV-AES can be easily implemented on the server.

11. The Signal secure messaging service is famous for its “double ratchet” algorithm (https://signal.org/docs/ specifications/doubleratchet/), which ensures that a fresh key is derived after every single message.

12. Real path-traversal exploits are usually more complex than this, relying on subtle bugs in URL parsing routines.