6 Self-contained tokens and JWTs

This chapter covers

- Scaling token-based authentication with encrypted client-side storage
- Protecting tokens with MACs and authenticated encryption
- Generating standard JSON Web Tokens
- Handling token revocation when all the state is on the client

You've shifted the Natter API over to using the database token store with tokens stored in Web Storage. The good news is that Natter is really taking off. Your user base has grown to millions of regular users. The bad news is that the token database is struggling to cope with this level of traffic. You've evaluated different database backends, but you've heard about stateless tokens that would allow you to get rid of the database entirely. Without a database slowing you down, Natter will be able to scale up as the user base continues to grow. In this chapter, you'll implement self-contained tokens securely, and examine some of the security trade-offs compared to database-backed tokens. You'll also learn about the JSON Web Token (JWT) standard that is the most widely used token format today.

DEFINITION JSON Web Tokens (JWTs, pronounced "jots") are a standard format for self-contained security tokens. A JWT consists of a set of claims about a user represented as a JSON object, together with a header describing the format of the token. JWTs are cryptographically protected against tampering and can also be encrypted.

6.1 Storing token state on the client

The idea behind stateless tokens is simple. Rather than store the token state in the database, you can instead encode that state directly into the token ID and send it to the client. For example, you could serialize the token fields into a JSON object, which you then Base64url-encode to create a string that you can use as the token ID. When the token is presented back to the API, you then simply decode the token and parse the JSON to recover the attributes of the session.

Listing 6.1 shows a JSON token store that does exactly that. It uses short keys for attributes, such as `sub` for the subject (username), and `exp` for the expiry time, to save space. These are standard JWT attributes, as you'll learn in section 6.2.1. Leave the `revoke` method blank for now, you will come back to that shortly in section 6.5. Navigate to the src/main/java/com/manning/apisecurityinaction/token folder and create a new file JsonTokenStore.java in your editor. Type in the contents of listing 6.1 and save the new file.

WARNING This code is not secure on its own because pure JSON tokens can be altered and forged. You'll add support for token authentication in section 6.1.1.

Listing 6.1 The JSON token store

```
package com.manning.apisecurityinaction.token;

import org.json.*;
import spark.Request;
import java.time.Instant;
import java.util.*;
import static java.nio.charset.StandardCharsets.UTF_8;

public class JsonTokenStore implements TokenStore {
    @Override
    public String create(Request request, Token token) {
        var json = new JSONObject();
        json.put("sub", token.username);                          (1)
        json.put("exp", token.expiry.getEpochSecond());           (1)
        json.put("attrs", token.attributes);                      (1)

        var jsonBytes = json.toString().getBytes(UTF_8);          (2)
        return Base64url.encode(jsonBytes);                       (2)
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        try {
            var decoded = Base64url.decode(tokenId);              (3)
            var json = new JSONObject(new String(decoded, UTF_8)); (3)
            var expiry = Instant.ofEpochSecond(json.getInt("exp")); (3)
            var username = json.getString("sub");                 (3)
            var attrs = json.getJSONObject("attrs");              (3)

            var token = new Token(expiry, username);              (3)
            for (var key : attrs.keySet()) {                      (3)
```

```
                token.attributes.put(key, attrs.getString(key));      ❸
            }

            return Optional.of(token);
        } catch (JSONException e) {
            return Optional.empty();
        }
    }

    @Override
    public void revoke(Request request, String tokenId) {
        // TODO                                                       ❹
    }
}
```

❶ Convert the token attributes into a JSON object.

❷ Encode the JSON object with URL-safe Base64-encoding.

❸ To read the token, decode it and parse the JSON to recover the attributes.

❹ Leave the revoke method blank for now.

6.1.1 Protecting JSON tokens with HMAC

Of course, as it stands, this code is completely insecure. Anybody can log in to the API and then edit the encoded token in their browser to change their username or other security attributes! In fact, they can just create a brand-new token themselves without ever logging in. You can fix that by reusing the `HmacTokenStore` that you created in chapter 5, as shown in figure 6.1. By appending an authentication tag computed with a secret key known only to the API server, an attacker is prevented from either creating a fake token or altering an existing one.

To enable HMAC-protected tokens, open Main.java in your editor and change the code that constructs the `DatabaseTokenStore` to instead create a `JsonTokenStore`:

```
    TokenStore tokenStore = new JsonTokenStore();                     ❶
    tokenStore = new HmacTokenStore(tokenStore, macKey);              ❷
    var tokenController = new TokenController(tokenStore);
```

❶ Construct the JsonTokenStore.

❷ Wrap it in a HmacTokenStore to ensure authenticity.

You can try it out to see your first stateless token in action:

```
$ curl -H 'Content-Type: application/json' -u test:password \
  -X POST https://localhost:4567/sessions
{"token":"eyJzdWIiOiJ0ZXN0IiwiZXhwIjoxNTU5NTgyMTI5LCJhdHRycyI6e319.
➥ INFgLC3cAhJ8DjzPgQfHBHvU_uItnFjt568mQ43V7YI"}
```

Pop quiz

1. Which of the STRIDE threats does the `HmacTokenStore` protect against? (There may be more than one correct answer.)
    1. Spoofing
    2. Tampering
    3. Repudiation
    4. Information disclosure
    5. Denial of service
    6. Elevation of privilege

The answer is at the end of the chapter.

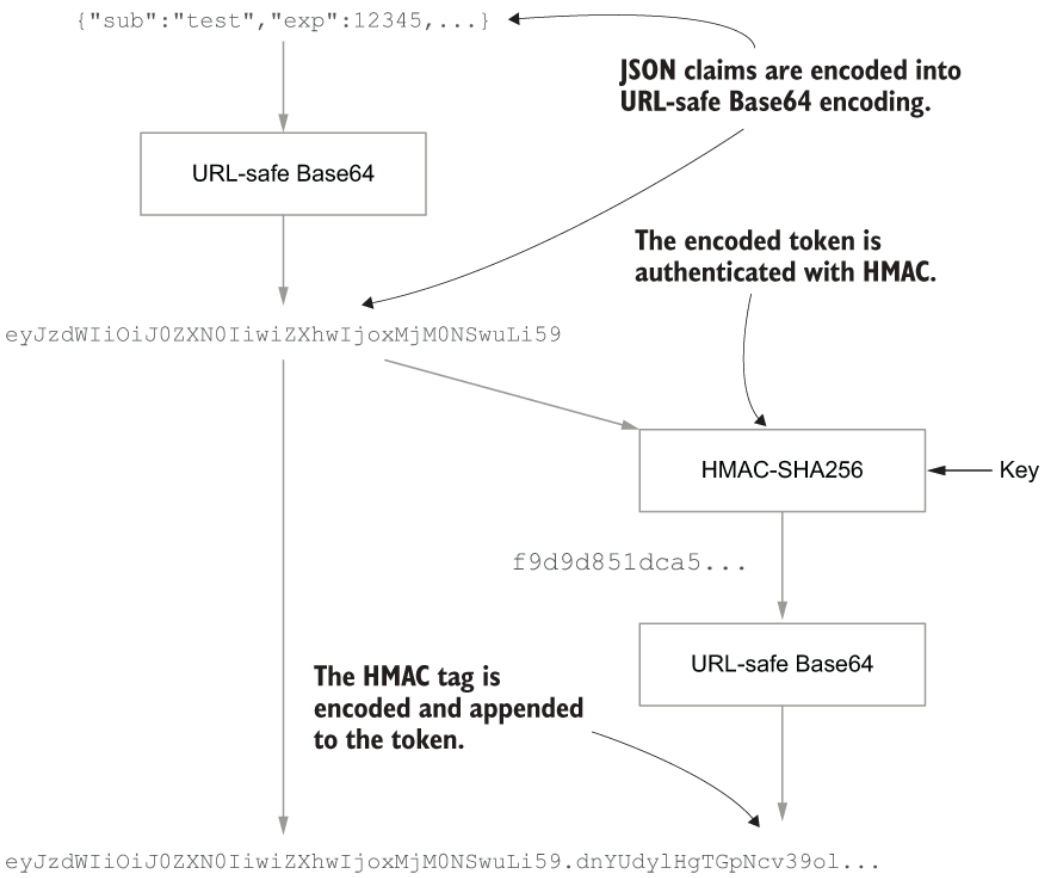

Figure 6.1 An HMAC tag is computed over the encoded JSON claims using a secret key. The HMAC tag is then itself encoded into URL-safe Base64 format and appended to the token, using a period as a separator. As a period is not a valid character in Base64 encoding, you can use this to find the tag later.

6.2 JSON Web Tokens

Authenticated client-side tokens have become very popular in recent years, thanks in part to the standardization of JSON Web Tokens in 2015. JWTs are very similar to the JSON tokens you have just produced, but have many more features:

- A standard header format that contains metadata about the JWT, such as which MAC or encryption algorithm was used.
- A set of standard claims that can be used in the JSON content of the JWT, with defined meanings, such as `exp` to indicate the expiry time and `sub` for the subject, just as you have been using.
- A wide range of algorithms for authentication and encryption, as well as digital signatures and public key encryption that are covered later in this book.

Because JWTs are standardized, they can be used with lots of existing tools, libraries, and services. JWT libraries exist for most programming languages now, and many API frameworks include built-in support for

JWTs, making them an attractive format to use. The OpenID Connect (OIDC) authentication protocol that's discussed in chapter 7 uses JWTs as a standard format to convey identity claims about users between systems.

The JWT standards zoo

While JWT itself is just one specification (**https://tools.ietf.org/html/rfc7519**), it builds on a collection of standards collectively known as JSON Object Signing and Encryption (JOSE). JOSE itself consists of several related standards:

- JSON Web Signing (JWS, **https://tools.ietf.org/html/rfc7515**) defines how JSON objects can be authenticated with HMAC and digital signatures.
- JSON Web Encryption (JWE, **https://tools.ietf.org/html/rfc7516**) defines how to encrypt JSON objects.
- JSON Web Key (JWK, **https://tools.ietf.org/html/rfc7517**) describes a standard format for cryptographic keys and related metadata in JSON.
- JSON Web Algorithms (JWA, **https://tools.ietf.org/html/rfc7518**) then specifies signing and encryption algorithms to be used.

JOSE has been extended over the years by new specifications to add new algorithms and options. It is common to use JWT to refer to the whole collection of specifications, although there are uses of JOSE beyond JWTs.

A basic authenticated JWT is almost exactly like the HMAC-authenticated JSON tokens that you produced in section 6.1.1, but with an additional JSON header that indicates the algorithm and other details of how the JWT was produced, as shown in figure 6.2. The Base64url-encoded format used for JWTs is known as the JWS Compact Serialization. JWS also defines another format, but the compact serialization is the most widely used for API tokens.

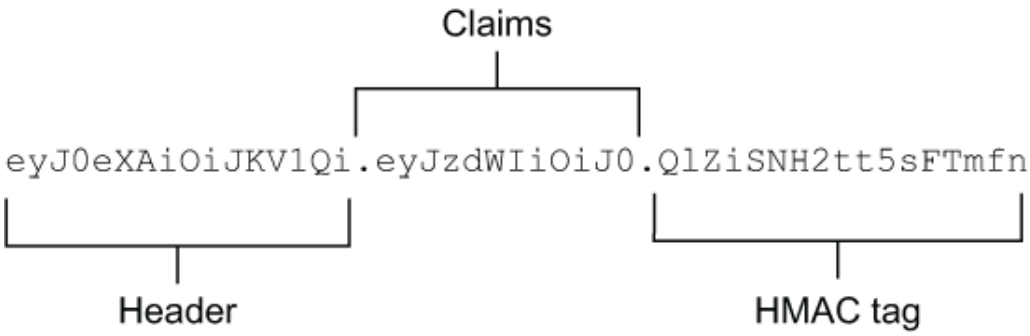

Figure 6.2 The JWS Compact Serialization consists of three URL-safe Base64-encoded parts, separated by periods. First comes the header, then

the payload or claims, and finally the authentication tag or signature. The values in this diagram have been shortened for display purposes.

The flexibility of JWT is also its biggest weakness, as several attacks have been found in the past that exploit this flexibility. JOSE is a kit-of-parts design, allowing developers to pick and choose from a wide variety of algorithms, and not all combinations of features are secure. For example, in 2015 the security researcher Tim McClean discovered vulnerabilities in many JWT libraries (**http://mng.bz/awKz**) in which an attacker could change the algorithm header in a JWT to influence how the recipient validated the token. It was even possible to change it to the value `none`, which instructed the JWT library to not validate the signature at all! These kinds of security flaws have led some people to argue that JWTs are inherently insecure due to the ease with which they can be misused, and the poor security of some of the standard algorithms.

PASETO: An alternative to JOSE

The error-prone nature of the standards has led to the development of alternative formats intended to be used for many of the same purposes as JOSE but with fewer tricky implementation details and opportunities for misuse. One example is PASETO (**https://paseto.io**), which provides either symmetric authenticated encryption or public key signed JSON objects, covering many of the same use-cases as the JOSE and JWT standards. The main difference from JOSE is that PASETO only allows a developer to specify a format version. Each version uses a fixed set of cryptographic algorithms rather than allowing a wide choice of algorithms: version 1 requires widely implemented algorithms such as AES and RSA, while version 2 requires more modern but less widely implemented algorithms such as Ed25519. This gives an attacker much less scope to confuse the implementation and the chosen algorithms have few known weaknesses.

I'll let you come to your own conclusions about whether to use JWTs. In this chapter you'll see how to implement some of the features of JWTs from scratch, so you can decide if the extra complexity is worth it. There are many cases in which JWTs cannot be avoided, so I'll point out security best practices and gotchas so that you can use them safely.

6.2.1 The standard JWT claims

One of the most useful parts of the JWT specification is the standard set of JSON object properties defined to hold claims about a subject, known as a

claims set. You've already seen two standard JWT claims, because you used them in the implementation of the `JsonTokenStore`:

- The `exp` claim indicates the expiry time of a JWT in UNIX time, which is the number of seconds since midnight on January 1, 1970 in UTC.
- The `sub` claim identifies the subject of the token: the user. Other claims in the token are generally making claims about this subject.

JWT defines a handful of other claims too, which are listed in table 6.1. To save space, each claim is represented with a three-letter JSON object property.

Table 6.1 Standard JWT claims

Claim	Name	Purpose
`iss`	Issuer	Indicates who created the JWT. This is a single string and often the URI of the authentication service.
`aud`	Audience	Indicates who the JWT is for. An array of strings identifying the intended recipients of the JWT. If there is only a single value, then it can be a simple string value rather than an array. The recipient of a JWT must check that its identifier appears in the audience; otherwise, it should reject the JWT. Typically, this is a set of URIs for APIs where the token can be used.
`iat`	Issued-At	The UNIX time at which the JWT was created.
`nbf`	Not-Before	The JWT should be rejected if used before this time.
`exp`	Expiry	The UNIX time at which the JWT expires and should be rejected by recipients.
`sub`	Subject	The identity of the subject of the JWT. A string. Usually a username or other unique identifier.
`jti`	JWT ID	A unique ID for the JWT, which can be used to detect replay.

Of these claims, only the issuer, issued-at, and subject claims express a positive statement. The remaining fields all describe constraints on how the token can be used rather than making a claim. These constraints are intended to prevent certain kinds of attacks against security tokens, such as replay attacks in which a token sent by a genuine party to a service to gain access is captured by an attacker and later replayed so that the attacker can gain access. Setting a short expiry time can reduce the window of opportunity for such attacks, but not eliminate them. The JWT ID can

be used to add a unique value to a JWT, which the recipient can then remember until the token expires to prevent the same token being replayed. Replay attacks are largely prevented by the use of TLS but can be important if you have to send a token over an insecure channel or as part of an authentication protocol.

DEFINITION A replay attack occurs when an attacker captures a token sent by a legitimate party and later replays it on their own request.

The issuer and audience claims can be used to prevent a different form of replay attack, in which the captured token is replayed against a different API than the originally intended recipient. If the attacker replays the token back to the original issuer, this is known as a reflection attack, and can be used to defeat some kinds of authentication protocols if the recipient can be tricked into accepting their own authentication messages. By verifying that your API server is in the audience list, and that the token was issued by a trusted party, these attacks can be defeated.

6.2.2 The JOSE header

Most of the flexibility of the JOSE and JWT standards is concentrated in the header, which is an additional JSON object that is included in the authentication tag and contains metadata about the JWT. For example, the following header indicates that the token is signed with HMAC-SHA-256 using a key with the given key ID:

```
{
  "alg": "HS256",          ❶
  "kid": "hmac-key-1"      ❷
}
```

❶ The algorithm

❷ The key identifier

Although seemingly innocuous, the JOSE header is one of the more error-prone aspects of the specifications, which is why the code you have written so far does not generate a header, and I often recommend that they are stripped when possible to create (nonstandard) headless JWTs. This can be done by removing the header section produced by a standard JWT library before sending it and then recreating it again before validating a

received JWT. Many of the standard headers defined by JOSE can open your API to attacks if you are not careful, as described in this section.

DEFINITION A headless JWT is a JWT with the header removed. The recipient recreates the header from expected values. For simple use cases where you control the sender and recipient this can reduce the size and attack surface of using JWTs but the resulting JWTs are nonstandard. Where headless JWTs can't be used, you should strictly validate all header values.

The tokens you produced in section 6.1.1 are effectively headless JWTs and adding a JOSE header to them (and including it in the HMAC calculation) would make them standards-compliant. From now on you'll use a real JWT library, though, rather than writing your own.

THE ALGORITHM HEADER

The `alg` header identifies the JWS or JWE cryptographic algorithm that was used to authenticate or encrypt the contents. This is also the only mandatory header value. The purpose of this header is to enable cryptographic agility, allowing an API to change the algorithm that it uses while still processing tokens issued using the old algorithm.

DEFINITION Cryptographic agility is the ability to change the algorithm used for securing messages or tokens in case weaknesses are discovered in one algorithm or a more performant alternative is required.

Although this is a good idea, the design in JOSE is less than ideal because the recipient must rely on the sender to tell them which algorithm to use to authenticate the message. This violates the principle that you should never trust a claim that you have not authenticated, and yet you cannot authenticate the JWT until you have processed this claim! This weakness was what allowed Tim McClean to confuse JWT libraries by changing the `alg` header.

A better solution is to store the algorithm as metadata associated with a key on the server. You can then change the algorithm when you change the key, a methodology I refer to as key-driven cryptographic agility. This is much safer than recording the algorithm in the message, because an attacker has no ability to change the keys stored on your server. The JSON Web Key (JWK) specification allows an algorithm to be associated with a key, as shown in listing 6.2, using the `alg` attribute. JOSE defines standard names for many authentication and encryption algorithms and the

standard name for HMAC-SHA256 that you'll use in this example is `HS256`. A secret key used for HMAC or AES is known as an octet key in JWK, as the key is just a sequence of random bytes and octet is an alternative word for byte. The key type is indicated by the `kty` attribute in a JWK, with the value `oct` used for octet keys.

DEFINITION In key-driven cryptographic agility, the algorithm used to authenticate a token is stored as metadata with the key on the server rather than as a header on the token. To change the algorithm, you install a new key. This prevents an attacker from tricking the server into using an incompatible algorithm.

Listing 6.2 A JWK with algorithm claim

```
{
    "kty": "oct",
    "alg": "HS256",                                    ❶
    "k": "9ITYj4mt-TLYT2b_vnAyCVurks1r2uzCLw7sOxg-75g"  ❷
}
```

❶ The algorithm the key is to be used for

❷ The Base64-encoded bytes of the key itself

The JWE specification also includes an `enc` header that specifies the cipher used to encrypt the JSON body. This header is less error-prone than the `alg` header, but you should still validate that it contains a sensible value. Encrypted JWTs are discussed in section 6.3.3.

SPECIFYING THE KEY IN THE HEADER

To allow implementations to periodically change the key that they use to authenticate JWTs, in a process known as key rotation, the JOSE specifications include several ways to indicate which key was used. This allows the recipient to quickly find the right key to verify the token, without having to try each key in turn. The JOSE specs include one safe way to do this (the `kid` header) and two potentially dangerous alternatives listed in table 6.2.

DEFINITION Key rotation is the process of periodically changing the keys used to protect messages and tokens. Changing the key regularly ensures that the usage limits for a key are never reached and if any one key is

compromised then it is soon replaced, limiting the time in which damage can be done.

Table 6.2 Indicating the key in a JOSE header

Header	Contents	Safe?	Comments
`kid`	A key ID	Yes	As the key ID is just a string identifier, it can be safely looked up in a server-side set of keys.
`jwk`	The full key	No	Trusting the sender to give you the key to verify a message loses all security properties.
`jku`	An URL to retrieve the full key	No	The intention of this header is that the recipient can retrieve the key from a HTTPS endpoint, rather than including it directly in the message, to save space. Unfortunately, this has all the issues of the `jwk` header, but additionally opens the recipient up to SSRF attacks.

DEFINITION A server-side request forgery (SSRF) attack occurs when an attacker can cause a server to make outgoing network requests under the attacker's control. Because the server is on a trusted network behind a firewall, this allows the attacker to probe and potentially attack machines on the internal network that they could not otherwise access. You'll learn more about SSRF attacks and how to prevent them in chapter 10.

There are also headers for specifying the key as an X.509 certificate (used in TLS). Parsing and validating X.509 certificates is very complex so you should avoid these headers.

6.2.3 Generating standard JWTs

Now that you've seen the basic idea of how a JWT is constructed, you'll switch to using a real JWT library for generating JWTs for the rest of the chapter. It's always better to use a well-tested library for security when one is available. There are many JWT and JOSE libraries for most programming languages, and the **https://jwt.io** website maintains a list. You should check that the library is actively maintained and that the developers are aware of historical JWT vulnerabilities such as the ones mentioned in this chapter. For this chapter, you can use Nimbus JOSE + JWT from **https://connect2id .com/products/nimbus-jose-jwt**, which is a well-maintained open source (Apache 2.0 licensed) Java JOSE library.

Open the pom.xml file in the Natter project root folder and add the following dependency to the dependencies section to load the Nimbus library:

```
    <dependency>
      <groupId>com.nimbusds</groupId>
      <artifactId>nimbus-jose-jwt</artifactId>
      <version>8.19</version>
    </dependency>
```

Listing 6.3 shows how to use the library to generate a signed JWT. The code is generic and can be used with any JWS algorithm, but for now you'll use the `HS256` algorithm, which uses HMAC-SHA-256, just like the existing `HmacTokenStore`. The Nimbus library requires a `JWSSigner` object for generating signatures, and a `JWSVerifier` for verifying them. These objects can often be used with several algorithms, so you should also pass in the specific algorithm to use as a separate `JWSAlgorithm` object. Finally, you should also pass in a value to use as the audience for the generated JWTs. This should usually be the base URI of the API server, such as https:// /localhost:4567. By setting and verifying the audience claim, you ensure that a JWT can't be used to access a different API, even if they happen to use the same cryptographic key. To produce the JWT you first build the claims set, set the `sub` claim to the username, the `exp` claim to the token expiry time, and the `aud` claim to the audience value you got from the constructor. You can then set any other attributes of the token as a custom claim, which will become a nested JSON object in the claims set. To sign the JWT you then set the correct algorithm in the header and use the `JWSSigner` object to calculate the signature. The `serialize()` method will then produce the JWS Compact Serialization of the JWT to return as the token identifier. Create a new file named SignedJwtTokenStore.java under src/main/resources/com/manning/apisecurityinaction/token and copy the contents of the listing.

Listing 6.3 Generating a signed JWT

```
package com.manning.apisecurityinaction.token;

import javax.crypto.SecretKey;
import java.text.ParseException;
import java.util.*;
import com.nimbusds.jose.*;
import com.nimbusds.jwt.*;
```

```
import spark.Request;

public class SignedJwtTokenStore implements TokenStore {
    private final JWSSigner signer;                                    (1)
    private final JWSVerifier verifier;                                (1)
    private final JWSAlgorithm algorithm;                              (1)
    private final String audience;                                     (1)

    public SignedJwtTokenStore(JWSSigner signer,                       (1)
                JWSVerifier verifier, JWSAlgorithm algorithm,          (1)
                String audience) {                                     (1)
        this.signer = signer;                                          (1)
        this.verifier = verifier;                                      (1)
        this.algorithm = algorithm;                                    (1)
        this.audience = audience;                                      (1)
    }

    @Override
    public String create(Request request, Token token) {
        var claimsSet = new JWTClaimsSet.Builder()                     (2)
                .subject(token.username)                               (2)
                .audience(audience)                                    (2)
                .expirationTime(Date.from(token.expiry))               (2)
                .claim("attrs", token.attributes)                      (2)
                .build();                                              (2)
        var header = new JWSHeader(JWSAlgorithm.HS256);                (3)
        var jwt = new SignedJWT(header, claimsSet);                    (3)
        try {
            jwt.sign(signer);                                          (4)
            return jwt.serialize();                                    (5)
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        // TODO
        return Optional.empty();
    }

    @Override
    public void revoke(Request request, String tokenId) {
        // TODO
    }
}
```

❶ Pass in the algorithm, audience, and signer and verifier objects.

❷ Create the JWT claims set with details about the token.

❸ Specify the algorithm in the header and build the JWT.

❹ Sign the JWT using the JWSSigner object.

❺ Convert the signed JWT into the JWS compact serialization.

To use the new token store, open the Main.java file in your editor and change the code that constructs the `JsonTokenStore` and `HmacTokenStore` to instead construct a `SignedJwtTokenStore`. You can reuse the same `macKey` that you loaded for the `HmacTokenStore`, as you're using the same algorithm for signing the JWTs. The code should look like the following, using the `MACSigner` and `MACVerifier` classes for signing and verification using HMAC:

```
var algorithm = JWSAlgorithm.HS256;                                    ❶
var signer = new MACSigner((SecretKey) macKey);                        ❶
var verifier = new MACVerifier((SecretKey) macKey);                    ❶
TokenStore tokenStore = new SignedJwtTokenStore(                       ❷
        signer, verifier, algorithm, "https://localhost:4567");        ❷
var tokenController = new TokenController(tokenStore);
```

❶ Construct the MACSigner and MACVerifier objects with the macKey.

❷ Pass the signer, verifier, algorithm, and audience to the SignedJwtTokenStore.

You can now restart the API server, create a test user, and log in to see the created JWT:

```
$ curl -H 'Content-Type: application/json' \
  -d '{"username":"test","password":"password"}' \
  https://localhost:4567/users
{"username":"test"}
$ curl -H 'Content-Type: application/json' -u test:password \
  -d '' https://localhost:4567/sessions
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0IiwiYXVkIjoiaHR0cH
➥ M6XC9cL2xvY2FsaG9zdDo0NTY3IiwiZXhwIjoxNTc3MDA3ODcyLCJhdHRycyI
➥ 6e319.nMxLeSG6pmrPOhRSNKF4v31eQZ3uxaPVyj-Ztf-vZQw"}
```

You can take this JWT and paste it into the debugger at **https://jwt.io** to validate it and see the contents of the header and claims, as shown in figure 6.3.

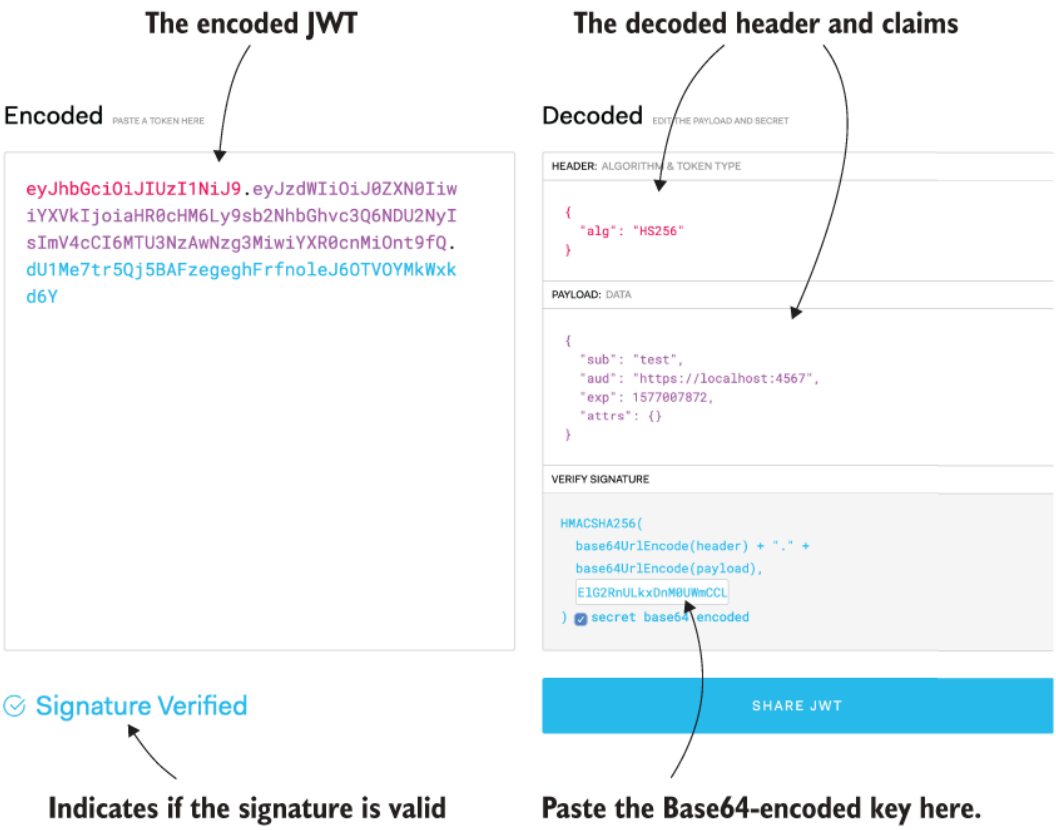

Figure 6.3 The JWT in the jwt.io debugger. The panels on the right show the decoded header and payload and let you paste in your key to validate the JWT. Never paste a JWT or key from a production environment into a website.

WARNING While jwt.io is a great debugging tool, remember that JWTs are credentials so you should never post JWTs from a production environment into any website.

6.2.4 Validating a signed JWT

To validate a JWT, you first parse the JWS Compact Serialization format and then use the `JWSVerifier` object to verify the signature. The Nimbus `MACVerifier` will calculate the correct HMAC tag and then compare it to the tag attached to the JWT using a constant-time equality comparison, just like you did in the `HmacTokenStore`. The Nimbus library also takes care of basic security checks, such as making sure that the algorithm header is compatible with the verifier (preventing the algorithm mix up attacks discussed in section 6.2), and that there are no unrecognized critical headers. After the signature has been verified, you can extract the JWT claims set and verify any constraints. In this case, you just need to check that the expected audience value appears in the audience claim,

and then set the token expiry from the JWT expiry time claim. The `TokenController` will ensure that the token hasn't expired. Listing 6.4 shows the full JWT validation logic. Open the SignedJwtTokenStore.java file and replace the `read()` method with the contents of the listing.

Listing 6.4 Validating a signed JWT

```
    @Override
    public Optional<Token> read(Request request, String tokenId) {
        try {
            var jwt = SignedJWT.parse(tokenId);                          (1)

            if (!jwt.verify(verifier)) {                                 (1)
                throw new JOSEException("Invalid signature");            (1)
            }                                                            (1)

            var claims = jwt.getJWTClaimsSet();
            if (!claims.getAudience().contains(audience)) {              (2)
                throw new JOSEException("Incorrect audience");           (2)
            }                                                            (2)

            var expiry = claims.getExpirationTime().toInstant();         (3)
            var subject = claims.getSubject();                           (3)
            var token = new Token(expiry, subject);                      (3)
            var attrs = claims.getJSONObjectClaim("attrs");              (3)
            attrs.forEach((key, value) ->                                (3)
                    token.attributes.put(key, (String) value));          (3)

            return Optional.of(token);
        } catch (ParseException | JOSEException e) {
            return Optional.empty();                                     (4)
        }
    }
```

(1) Parse the JWT and verify the HMAC signature using the JWSVerifier.

(2) Reject the token if the audience doesn't contain your API's base URI.

(3) Extract token attributes from the remaining JWT claims.

(4) If the token is invalid, then return a generic failure response.

You can now restart the API and use the JWT to create a new social space:

```
$ curl -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN
➥ 0IiwiYXVkIjoiaHR0cHM6XC9cL2xvY2FsaG9zdDo0NTY3IiwiZXhwIjoxNTc
➥ 3MDEyMzA3LCJhdHRycyI6e319.JKJnoNdHEBzc8igkzV7CAYfDRJvE7oB2md
➥ 6qcNgc_yM' -d '{"owner":"test","name":"test space"}' \
  https://localhost:4567/spaces

{"name":"test space","uri":"/spaces/1"}
```

Pop quiz

2. Which JWT claim is used to indicate the API server a JWT is intended for?
    1. iss
    2. sub
    3. iat
    4. exp
    5. aud
    6. jti
3. True or False: The JWT `alg` (algorithm) header can be safely used to determine which algorithm to use when validating the signature.

The answers are at the end of the chapter.

6.3 Encrypting sensitive attributes

A database in your datacenter, protected by firewalls and physical access controls, is a relatively safe place to store token data, especially if you follow the hardening advice in the last chapter. Once you move away from a database and start storing data on the client, that data is much more vulnerable to snooping. Any personal information about the user included in the token, such as name, date of birth, job role, work location, and so on, may be at risk if the token is accidentally leaked by the client or stolen though a phishing attack or XSS exfiltration. Some attributes may also need to be kept confidential from the user themselves, such as any attributes that reveal details of the API implementation. In chapter 7, you'll also consider third-party client applications that may not be trusted to know details about who the user is.

Encryption is a complex topic with many potential pitfalls, but it can be used successfully if you stick to well-studied algorithms and follow some basic rules. The goal of encryption is to ensure the confidentiality of a message by converting it into an obscured form, known as the ciphertext,

using a secret key. The algorithm is known as a cipher. The recipient can then use the same secret key to recover the original plaintext message. When the sender and recipient both use the same key, this is known as secret key cryptography. There are also public key encryption algorithms in which the sender and recipient have different keys, but we won't cover those in much detail in this book.

An important principle of cryptography, known as Kerckhoff's Principle, says that an encryption scheme should be secure even if every aspect of the algorithm is known, so long as the key remains secret.

NOTE You should use only algorithms that have been designed through an open process with public review by experts, such as the algorithms you'll use in this chapter.

There are several secure encryption algorithms in current use, but the most important is the Advanced Encryption Standard (AES), which was standardized in 2001 after an international competition, and is widely considered to be very secure. AES is an example of a block cipher, which takes a fixed size input of 16 bytes and produces a 16-byte encrypted output. AES keys are either 128 bits, 192 bits, or 256 bits in size. To encrypt more (or less) than 16 bytes with AES, you use a block cipher mode of operation. The choice of mode of operation is crucial to the security as demonstrated in figure 6.4, which shows an image of a penguin encrypted with the same AES key but with two different modes of operation.[1] The Electronic Code Book (ECB) mode is completely insecure and leaks a lot of details about the image, while the more secure Counter Mode (CTR) eliminates any details and looks like random noise.

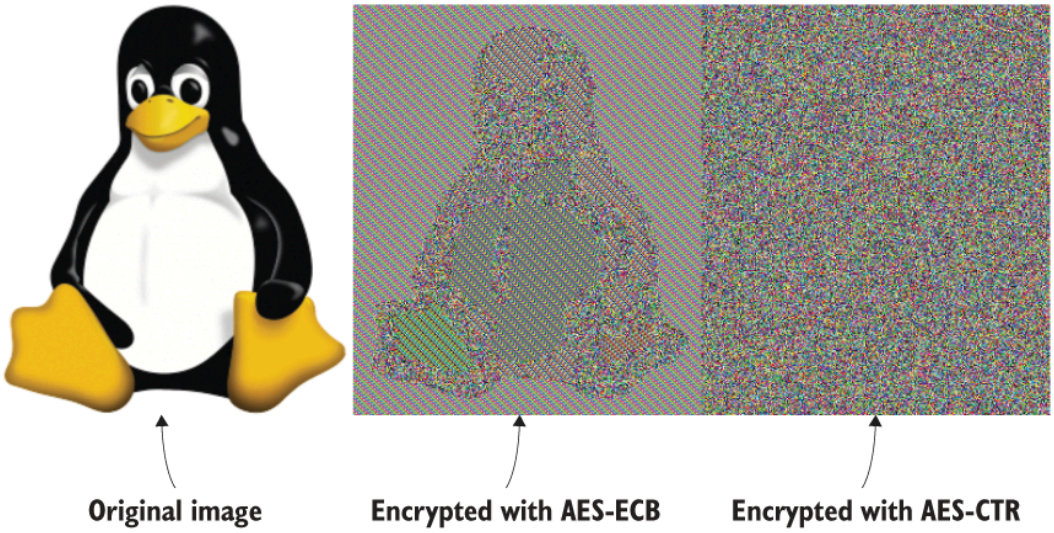

Figure 6.4 An image of the Linux mascot, Tux, that has been encrypted by AES in ECB mode. The shape of the penguin and many features are still

visible despite the encryption. By contrast, the same image encrypted with AES in CTR mode is indistinguishable from random noise. (Original image by Larry Ewing and The GIMP, https://commons.wikimedia.org/wiki/File:Tux.svg.)

DEFINITION A block cipher encrypts a fixed-sized block of input to produce a block of output. The AES block cipher operates on 16-byte blocks. A block cipher mode of operation allows a fixed-sized block cipher to be used to encrypt messages of any length. The mode of operation is critical to the security of the encryption process.

6.3.1 Authenticated encryption

Many encryption algorithms only ensure the confidentiality of data that has been encrypted and don't claim to protect the integrity of that data. This means that an attacker won't be able to read any sensitive attributes in an encrypted token, but they may be able to alter them. For example, if you know that a token is encrypted with CTR mode and (when decrypted) starts with the string `user=brian`, you can change this to read `user=admin` by simple manipulation of the ciphertext even though you can't decrypt the token. Although there isn't room to go into the details here, this kind of attack is often covered in cryptography tutorials under the name chosen ciphertext attack.

DEFINITION A chosen ciphertext attack is an attack against an encryption scheme in which an attacker manipulates the encrypted ciphertext.

In terms of threat models from chapter 1, encryption protects against information disclosure threats, but not against spoofing or tampering. In some cases, confidentiality can also be lost if there is no guarantee of integrity because an attacker can alter a message and then see what error message is generated when the API tries to decrypt it. This often leaks information about what the message decrypted to.

LEARN MORE You can learn more about how modern encryption algorithms work, and attacks against them, from an up-to-date introduction to cryptography book such as Serious Cryptography by Jean-Philippe Aumasson (No Starch Press, 2018).

To protect against spoofing and tampering threats, you should always use algorithms that provide authenticated encryption. Authenticated encryption algorithms combine an encryption algorithm for hiding sensitive data with a MAC algorithm, such as HMAC, to ensure that the data can't be altered or faked.

DEFINITION Authenticated encryption combines an encryption algorithm with a MAC. Authenticated encryption ensures confidentiality and integrity of messages.

One way to do this would be to combine a secure encryption scheme like AES in CTR mode with HMAC. For example, you might make an `EncryptedTokenStore` that encrypts data using AES and then combine that with the existing `HmacTokenStore` for authentication. But there are two ways you could combine these two stores: first encrypting and then applying HMAC, or, first applying HMAC and then encrypting the token and the tag together. It turns out that only the former is generally secure and is known as Encrypt-then-MAC (EtM). Because it is easy to get this wrong, cryptographers have developed several dedicated authenticated encryption modes, such as Galois/Counter Mode (GCM) for AES. JOSE supports both GCM and EtM encryption modes, which you'll examine in section 6.3.3, but we'll begin by looking at a simpler alternative.

6.3.2 Authenticated encryption with NaCl

Because cryptography is complex with many subtle details to get right, a recent trend has been for cryptography libraries to provide higher-level APIs that hide many of these details from developers. The most well-known of these is the Networking and Cryptography Library (NaCl; **https://nacl.cr.yp.to**) designed by Daniel Bernstein. NaCl (pronounced "salt," as in sodium chloride) provides high-level operations for authenticated encryption, digital signatures, and other cryptographic primitives but hides many of the details of the algorithms being used. Using a high-level library designed by experts such as NaCl is the safest option when implementing cryptographic protections for your APIs and can be significantly easier to use securely than alternatives.

TIP Other cryptographic libraries designed to be hard to misuse include Google's Tink (**https://github.com/google/tink**) and Themis from Cossack Labs (**https://github.com/cossacklabs/themis**). The Sodium library (**https:// libsodium.org**) is a widely used clone of NaCl in C that provides many additional extensions and a simplified API with bindings for Java and other languages.

In this section, you'll use a pure Java implementation of NaCl called Salty Coffee (**https://github.com/NeilMadden/salty-coffee**), which provides a very simple and Java-friendly API with acceptable performance.[2] To add the library to the Natter API project, open the pom.xml file in the root folder of the Natter API project and add the following lines to the dependencies section:

```
    <dependency>
      <groupId>software.pando.crypto</groupId>
      <artifactId>salty-coffee</artifactId>
      <version>1.0.2</version>
    </dependency>
```

Listing 6.5 shows an `EncryptedTokenStore` implemented using the Salty Coffee library's `SecretBox` class, which provides authenticated encryption. Like the `HmacTokenStore`, you can delegate creating the token to another store, allowing this to be wrapped around the `JsonTokenStore` or another format. Encryption is then performed with the `SecretBox.encrypt()` method. This method returns a `SecretBox` object, which has methods for getting the encrypted ciphertext and the authentication tag. The `toString()` method encodes these components into a URL-safe string that you can use directly as the token ID. To decrypt the token, you can use the `SecretBox.fromString()` method to recover the `SecretBox` from the encoded string, and then use the `decryptToString()` method to decrypt it and get back the original token ID. Navigate to the src/main/java/com/manning/apisecurityinaction/token folder again and create a new file named EncryptedTokenStore.java with the contents of listing 6.5.

Listing 6.5 An EncryptedTokenStore

```
package com.manning.apisecurityinaction.token;

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

import software.pando.crypto.nacl.SecretBox;
import spark.Request;

public class EncryptedTokenStore implements TokenStore {

    private final TokenStore delegate;
    private final Key encryptionKey;
```

```
    public EncryptedTokenStore(TokenStore delegate, Key encryptionKey) {
        this.delegate = delegate;
        this.encryptionKey = encryptionKey;
    }

    @Override
    public String create(Request request, Token token) {
        var tokenId = delegate.create(request, token);                    ❶
        return SecretBox.encrypt(encryptionKey, tokenId).toString();      ❷
    }

    @Override
    public Optional<Token> read(Request request, String tokenId) {
        var box = SecretBox.fromString(tokenId);                          ❸
        var originalTokenId = box.decryptToString(encryptionKey);         ❸
        return delegate.read(request, originalTokenId);                   ❸
    }

    @Override
    public void revoke(Request request, String tokenId) {
        var box = SecretBox.fromString(tokenId);                          ❸
        var originalTokenId = box.decryptToString(encryptionKey);         ❸
        delegate.revoke(request, originalTokenId);                        ❸
    }
}
```

❶ Call the delegate TokenStore to generate the token ID.

❷ Use the SecretBox.encrypt() method to encrypt the token.

❸ Decode and decrypt the box and then use the original token ID.

As you can see, the `EncryptedTokenStore` using `SecretBox` is very short because the library takes care of almost all details for you. To use the new store, you'll need to generate a new key to use for encryption rather than reusing the existing HMAC key.

PRINCIPLE A cryptographic key should only be used for a single purpose. Use separate keys for different functionality or algorithms.

Because Java's `keytool` command doesn't support generating keys for the encryption algorithm that `SecretBox` uses, you can instead generate a standard AES key and then convert it as the two key formats are identical. `SecretBox` only supports 256-bit keys, so run the following command

in the root folder of the Natter API project to add a new AES key to the existing keystore:

```
keytool -genseckey -keyalg AES -keysize 256 \
    -alias aes-key -keystore keystore.p12 -storepass changeit
```

You can then load the new key in the `Main` class just as you did for the HMAC key in chapter 5. Open Main.java in your editor and locate the lines that load the HMAC key from the keystore and add a new line to load the AES key:

```
var macKey = keyStore.getKey("hmac-key", keyPassword);      ❶
var encKey = keyStore.getKey("aes-key", keyPassword);       ❷
```

❶ The existing HMAC key

❷ The new AES key

You can convert the key into the correct format with the `SecretBox.key()` method, passing in the raw key bytes, which you can get by calling `encKey.getEncoded()`. Open the Main.java file again and update the code that constructs the `TokenController` to convert the key and use it to create an `EncryptedTokenStore`, wrapping a `JsonTokenStore`, instead of the previous JWT-based implementation:

```
    var naclKey = SecretBox.key(encKey.getEncoded());       ❶
    var tokenStore = new EncryptedTokenStore(               ❷
                new JsonTokenStore(), naclKey);             ❷
    var tokenController = new TokenController(tokenStore);  ❷
```

❶ Convert the key to the correct format.

❷ Construct the EncryptedTokenStore wrapping a JsonTokenStore.

You can now restart the API and login again to get a new encrypted token.

6.3.3 Encrypted JWTs

NaCl's `SecretBox` is hard to beat for simplicity and security, but there is no standard for how encrypted tokens are formatted into strings and different libraries may use different formats or leave this up to the applica-

tion. This is not a problem when tokens are only consumed by the same API that generated them but can become an issue if tokens are shared between many APIs, developed by separate teams in different programming languages. A standard format such as JOSE becomes more compelling in these cases. JOSE supports several authenticated encryption algorithms in the JSON Web Encryption (JWE) standard.

An encrypted JWT using the JWE Compact Serialization looks superficially like the HMAC JWTs from section 6.2, but there are more components reflecting the more complex structure of an encrypted token, shown in figure 6.5. The five components of a JWE are:

1. The JWE header, which is very like the JWS header, but with two additional fields: `enc`, which specifies the encryption algorithm, and `zip`, which specifies an optional compression algorithm to be applied before encryption.
2. An optional encrypted key. This is used in some of the more complex encryption algorithms. It is empty for the direct symmetric encryption algorithm that is covered in this chapter.
3. The initialization vector or nonce used when encrypting the payload. Depending on the encryption method being used, this will be either a 12- or 16-byte random binary value that has been Base64url-encoded.
4. The encrypted ciphertext.
5. The MAC authentication tag.

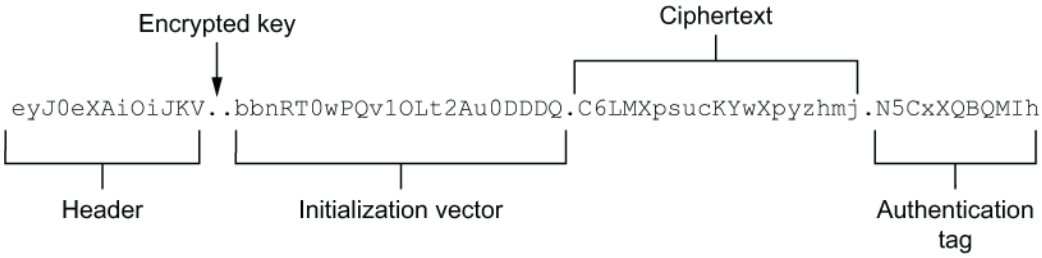

Figure 6.5 A JWE in Compact Serialization consists of 5 components: a header, an encrypted key (blank in this case), an initialization vector or nonce, the encrypted ciphertext, and then the authentication tag. Each component is URL-safe Base64-encoded. Values have been truncated for display.

DEFINITION An initialization vector (IV) or nonce (number-used-once) is a unique value that is provided to the cipher to ensure that ciphertext is always different even if the same message is encrypted more than once. The IV should be generated using a `java.security.SecureRandom` or other cryptographically-secure pseudorandom number generator (CSPRNG).[3] An IV doesn't need to be kept secret.

JWE divides specification of the encryption algorithm into two parts:

- The `enc` header describes the authenticated encryption algorithm used to encrypt the payload of the JWE.
- The `alg` header describes how the sender and recipient agree on the key used to encrypt the content.

There are a wide variety of key management algorithms for JWE, but for this chapter you will stick to direct encryption with a secret key. For direct encryption, the algorithm header is set to `dir` (direct). There are currently two available families of encryption methods in JOSE, both of which provide authenticated encryption:

- `A128GCM`, `A192GCM`, and `A256GCM` use AES in Galois Counter Mode (GCM).
- `A128CBC-HS256`, `A192CBC-HS384`, and `A256CBC-HS512` use AES in Cipher Block Chaining (CBC) mode together with either HMAC in an EtM configuration as described in section 6.3.1.

DEFINITION All the encryption algorithms allow the JWE header and IV to be included in the authentication tag without being encrypted. These are known as authenticated encryption with associated data (AEAD) algorithms.

GCM was designed for use in protocols like TLS where a unique session key is negotiated for each session and a simple counter can be used for the nonce. If you reuse a nonce with GCM then almost all security is lost: an attacker can recover the MAC key and use it to forge tokens, which is catastrophic for authentication tokens. For this reason, I prefer to use CBC with HMAC for directly encrypted JWTs, but for other JWE algorithms GCM is an excellent choice and very fast.

CBC requires the input to be padded to a multiple of the AES block size (16 bytes), and this historically has led to a devastating vulnerability known as a padding oracle attack, which allows an attacker to recover the full plaintext just by observing the different error messages when an API tries to decrypt a token they have tampered with. The use of HMAC in JOSE prevents this kind of tampering and largely eliminates the possibility of padding oracle attacks, and the padding has some security benefits.

WARNING You should avoid revealing the reason why decryption failed to the callers of your API to prevent oracle attacks like the CBC padding oracle attack.

What key size should you use?

AES allows keys to be in one of three different sizes: 128-bit, 192-bit, or 256-bit. In principle, correctly guessing a 128-bit key is well beyond the capability of even an attacker with enormous amounts of computing power. Trying every possible value of a key is known as a brute-force attack and should be impossible for a key of that size. There are three exceptions in which that assumption might prove to be wrong:

- A weakness in the encryption algorithm might be discovered that reduces the amount of effort required to crack the key. Increasing the size of the key provides a security margin against such a possibility.
- New types of computers might be developed that can perform brute-force searches much quicker than existing computers. This is believed to be true of quantum computers, but it's not known whether it will ever be possible to build a large enough quantum computer for this to be a real threat. Doubling the size of the key protects against known quantum attacks for symmetric algorithms like AES.
- Theoretically, if each user has their own encryption key and you have millions of users, it may be possible to attack every key simultaneously for less effort than you would expect from naively trying to break them one at a time. This is known as a batch attack and is described further in **https://blog.cr.yp.to/ 20151120-batchattacks.html**.
  At the time of writing, none of these attacks are practical for AES, and for short-lived authentication tokens the risk is significantly less, so 128-bit keys are perfectly safe. On the other hand, modern CPUs have special instructions for AES encryption so there's very little extra cost for 256-bit keys if you want to eliminate any doubt.

Remember that the JWE CBC with HMAC methods take a key that is twice the size as normal. For example, the `A128CBC-HS256` method requires a 256-bit key, but this is really two 128-bit keys joined together rather than a true 256-bit key.

6.3.4 Using a JWT library

Due to the relative complexity of producing and consuming encrypted JWTs compared to HMAC, you'll continue using the Nimbus JWT library in this section. Encrypting a JWT with Nimbus requires a few steps, as shown in listing 6.6.

- First you build a JWT claims set using the convenient `JWTClaimsSet.Builder` class.

- You can then create a `JWEHeader` object to specify the algorithm and encryption method.
- Finally, you encrypt the JWT using a `DirectEncrypter` object initialized with the AES key.

The `serialize()` method on the `EncryptedJWT` object will then return the JWE Compact Serialization. Navigate to src/main/java/com/manning/apisecurityinaction/token and create a new file name EncryptedJwtTokenStore.java. Type in the contents of listing 6.6 to create the new token store and save the file. As for the `JsonTokenStore`, leave the `revoke` method blank for now. You'll fix that in section 6.6.

Listing 6.6 The EncryptedJwtTokenStore

```
package com.manning.apisecurityinaction.token;

import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.*;
import com.nimbusds.jwt.*;
import spark.Request;

import javax.crypto.SecretKey;
import java.text.ParseException;
import java.util.*;

public class EncryptedJwtTokenStore implements TokenStore {

    private final SecretKey encKey;

    public EncryptedJwtTokenStore(SecretKey encKey) {
        this.encKey = encKey;
    }

    @Override
    public String create(Request request, Token token) {
        var claimsBuilder = new JWTClaimsSet.Builder()                    (1)
                .subject(token.username)                                  (1)
                .audience("https://localhost:4567")                       (1)
                .expirationTime(Date.from(token.expiry));                 (1)
        token.attributes.forEach(claimsBuilder::claim);                   (1)
        var header = new JWEHeader(JWEAlgorithm.DIR,                      (2)
                EncryptionMethod.A128CBC_HS256);                          (2)
        var jwt = new EncryptedJWT(header, claimsBuilder.build());        (2)

        try {
```

```
            var encrypter = new DirectEncrypter(encKey);          ❸
            jwt.encrypt(encrypter);                               ❸
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }

        return jwt.serialize();                                   ❹
    }

    @Override
    public void revoke(Request request, String tokenId) {
    }
}
```

❶ Build the JWT claims set.

❷ Create the JWE header and assemble the header and claims.

❸ Encrypt the JWT using the AES key in direct encryption mode.

❹ Return the Compact Serialization of the encrypted JWT.

Processing an encrypted JWT using the library is just as simple as creating one. First, you parse the encrypted JWT and then decrypt it using a `DirectDecrypter` initialized with the AES key, as shown in listing 6.7. If the authentication tag validation fails during decryption, then the library will throw an exception. To further reduce the possibility of padding oracle attacks in CBC mode, you should never return any details about why decryption failed to the user, so just return an empty `Optional` here as if no token had been supplied. You can log the exception details to a debug log that is only accessible to system administrators if you wish. Once the JWT has been decrypted, you can extract and validate the claims from the JWT. Open EncryptedJwtTokenStore.java in your editor again and implement the read method as in listing 6.7.

Listing 6.7 The JWT read method

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    try {
        var jwt = EncryptedJWT.parse(tokenId);                    ❶

        var decryptor = new DirectDecrypter(encKey);              ❷
        jwt.decrypt(decryptor);                                   ❷
```

```
            var claims = jwt.getJWTClaimsSet();
            if (!claims.getAudience().contains("https://localhost:4567")) {
                return Optional.empty();
            }
            var expiry = claims.getExpirationTime().toInstant();          ❸
            var subject = claims.getSubject();                            ❸
            var token = new Token(expiry, subject);                       ❸
            var ignore = Set.of("exp", "sub", "aud");                     ❸
            for (var attr : claims.getClaims().keySet()) {                ❸
                if (ignore.contains(attr)) continue;                      ❸
                token.attributes.put(attr, claims.getStringClaim(attr));  ❸
            }
            return Optional.of(token);
        } catch (ParseException | JOSEException e) {
            return Optional.empty();                                      ❹
        }
    }
```

❶ Parse the encrypted JWT.

❷ Decrypt and authenticate the JWT using the DirectDecrypter.

❸ Extract any claims from the JWT.

❹ Never reveal the cause of a decryption failure to the user.

You can now update the main method to switch to using the `EncryptedJwtTokenStore`, replacing the previous `EncryptedTokenStore`. You can reuse the AES key that you generated in section 6.3.2, but you'll need to cast it to the more specific `javax.crypto.SecretKey` class that the Nimbus library expects. Open Main.java and update the code to create the token controller again:

```
TokenStore tokenStore = new EncryptedJwtTokenStore(
    (SecretKey) encKey);                                                  ❶
var tokenController = new TokenController(tokenStore);
```

❶ Cast the key to the more specific SecretKey class.

Restart the API and try it out:

```
$ curl -H 'Content-Type: application/json' \
  -u test:password -X POST https://localhost:4567/sessions
```

```
{"token":"eyJlbmMiOiJBMjU2R0NNIiwiYWxnIjoiZGlyIn0..hAOoOsgfGb8yuhJD
➥ .kzhuXMMGunteKXz12aBSnqVfqtlnvvzqInLqp83zBwUW_rqWoQp5wM_q2D7vQxpK
➥ TaQR4Nuc-D3cPcYt7MXAJQ.ZigZZclJPDNMlP5GM1oXwQ"}
```

Compressed tokens

The encrypted JWT is a bit larger than either a simple HMAC token or the NaCl tokens from section 6.3.2. JWE supports optional compression of the JWT Claims Set before encryption, which can significantly reduce the size for complex tokens. But combining encryption and compression can lead to security weaknesses. Most encryption algorithms do not hide the length of the plaintext message that was encrypted, and compression reduces the size of a message based on its content. For example, if two parts of a message are identical, then it may combine them to remove the duplication. If an attacker can influence part of a message, they may be able to guess the rest of the contents by seeing how much it compresses. The CRIME and BREACH attacks (**http://breachattack.com**) against TLS were able to exploit this leak of information from compression to steal session cookies from compressed HTTP pages. These kinds of attacks are not always a risk, but you should carefully consider the possibility before enabling compression. Unless you really need to save space, you should leave compression disabled.

Pop quiz

4. Which STRIDE threats does authenticated encryption protect against? (There are multiple correct answers.)
    1. Spoofing
    2. Tampering
    3. Repudiation
    4. Information disclosure
    5. Denial of service
    6. Elevation of privilege
5. What is the purpose of the initialization vector (IV) in an encryption algorithm?
    1. It's a place to add your name to messages.
    2. It slows down decryption to prevent brute force attacks.
    3. It increases the size of the message to ensure compatibility with different algorithms.

4. It ensures that the ciphertext is always different even if a duplicate message is encrypted.
6. True or False: An IV should always be generated using a secure random number generator.

The answers are at the end of the chapter.

6.4 Using types for secure API design

Imagine that you have implemented token storage using the kit of parts that you developed in this chapter, creating a `JsonTokenStore` and wrapping it in an `EncryptedTokenStore` to add authenticated encryption, providing both confidentiality and authenticity of tokens. But it would be easy for somebody to accidentally remove the encryption if they simply commented out the `EncryptedTokenStore` wrapper in the main method, losing both security properties. If you'd developed the `EncryptedTokenStore` using an unauthenticated encryption scheme such as CTR mode and then manually combined it with the `HmacTokenStore`, the risk would be even greater because not every way of combining those two stores is secure, as you learned in section 6.3.1.

The kit-of-parts approach to software design is often appealing to software engineers, because it results in a neat design with proper separation of concerns and maximum reusability. This was useful when you could reuse the `HmacTokenStore`, originally designed to protect database-backed tokens, to also protect JSON tokens stored on the client. But a kit-of-parts design is opposed to security if there are many insecure ways to combine the parts and only a few that are secure.

PRINCIPLE Secure API design should make it very hard to write insecure code. It is not enough to merely make it possible to write secure code, because developers will make mistakes.

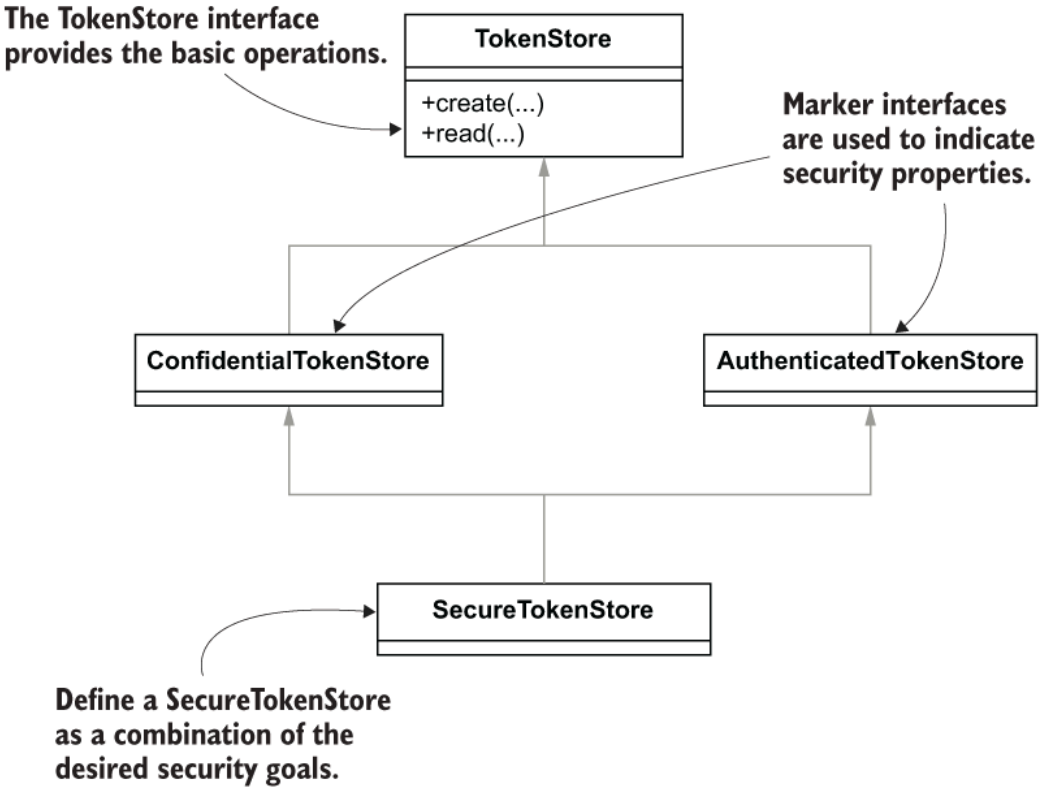

Figure 6.6 You can use marker interfaces to indicate the security properties of your individual token stores. If a store provides only confidentiality, it should implement the ConfidentialTokenStore interface. You can then define a SecureTokenStore by subtyping the desired combination of security properties. In this case, it ensures both confidentiality and authentication.

You can make a kit-of-parts design harder to misuse by using types to enforce the security properties you need, as shown in figure 6.6. Rather than all the individual token stores implementing a generic `TokenStore` interface, you can define marker interfaces that describe the security properties of the implementation. A `ConfidentialTokenStore` ensures that token state is kept secret, while an `AuthenticatedTokenStore` ensures that the token cannot be tampered with or faked. We can then define a `SecureTokenStore` that is a sub-type of each of the security properties that we want to enforce. In this case, you want the token controller to use a token store that is both confidential and authenticated. You can then update the `TokenController` to require a `SecureTokenStore`, enforcing that an insecure implementation is not used by mistake.

DEFINITION A marker interface is an interface that defines no new methods. It is used purely to indicate that the implementation has certain desirable properties.

Navigate to src/main/java/com/manning/apisecurityinaction/token and add the three new marker interfaces, as shown in listing 6.8. Create three

separate files, ConfidentialTokenStore.java, AuthenticatedTokenStore.java, and SecureTokenStore.java to hold the three new interfaces.

Listing 6.8 The secure marker interfaces

```
package com.manning.apisecurityinaction.token;                    (1)

public interface ConfidentialTokenStore extends TokenStore {      (1)
}                                                                  (1)

package com.manning.apisecurityinaction.token;                    (2)

public interface AuthenticatedTokenStore extends TokenStore {     (2)
}                                                                  (2)

package com.manning.apisecurityinaction.token;                    (3)

public interface SecureTokenStore extends ConfidentialTokenStore,  (3)
    AuthenticatedTokenStore {                                      (3)
}                                                                  (3)
```

(1) The ConfidentialTokenStore marker interface should go in ConfidentialTokenStore.java.

(2) The AuthenticatedTokenStore should go in AuthenticatedTokenStore.java.

(3) The SecureTokenStore combines them and goes in SecureTokenStore.java.

You can now change each of the token stores to implement an appropriate interface:

- If you assume that the backend cookie storage is secure against injection and other attacks, then the `CookieTokenStore` can be updated to implement the `SecureTokenStore` interface.
- If you've followed the hardening advice from chapter 5, the `DatabaseTokenStore` can also be marked as a `SecureTokenStore`. If you want to ensure that it is always used with HMAC for extra protection against tampering, then mark it as only confidential.
- The `JsonTokenStore` is completely insecure on its own, so leave it implementing the base `TokenStore` interface.

- The `SignedJwtTokenStore` provides no confidentiality for claims in the JWT, so it should only implement the `AuthenticatedTokenStore` interface.
- The `HmacTokenStore` turns any `TokenStore` into an `AuthenticatedTokenStore`. But if the underlying store is already confidential, then the result is a `SecureTokenStore`. You can reflect this difference in code by making the `HmacTokenStore` constructor private and providing two static factory methods instead, as shown in listing 6.9. If the underlying store is confidential, then the first method will return a `SecureTokenStore`. For anything else, the second method will be called and return only an `AuthenticatedTokenStore`.
- The `EncryptedTokenStore` and `EncryptedJwtTokenStore` can both be changed to implement `SecureTokenStore` because they both provide authenticated encryption that achieves the combined security goals no matter what underlying store is passed in.

Listing 6.9 Updating the HmacTokenStore

```
public class HmacTokenStore implements SecureTokenStore {                    ❶

    private final TokenStore delegate;
    private final Key macKey;

    private HmacTokenStore(TokenStore delegate, Key macKey) {                 ❷
        this.delegate = delegate;
        this.macKey = macKey;
    }
    public static SecureTokenStore wrap(ConfidentialTokenStore store,         ❸
                                        Key macKey) {                         ❸
        return new HmacTokenStore(store, macKey);                             ❸
    }                                                                         ❸
    public static AuthenticatedTokenStore wrap(TokenStore store,              ❹
                                           Key macKey) {                      ❹
        return new HmacTokenStore(store, macKey);                             ❹
    }                                                                         ❹
```

❶ Mark the HmacTokenStore as secure.

❷ Make the constructor private.

❸ When passed a ConfidentialTokenStore, returns a SecureTokenStore.

❹ When passed any other TokenStore, returns an AuthenticatedTokenStore.

You can now update the `TokenController` class to require a `SecureTokenStore` to be passed to it. Open TokenController.java in your editor and update the constructor to take a `SecureTokenStore`:

```
    public TokenController(SecureTokenStore tokenStore) {
        this.tokenStore = tokenStore;
    }
```

This change makes it much harder for a developer to accidentally pass in an implementation that doesn't meet your security goals, because the code will fail to type-check. For example, if you try to pass in a plain `JsonTokenStore`, then the code will fail to compile with a type error. These marker interfaces also provide valuable documentation of the expected security properties of each implementation, and a guide for code reviewers and security audits to check that they achieve them.

6.5 Handling token revocation

Stateless self-contained tokens such as JWTs are great for moving state out of the database. On the face of it, this increases the ability to scale up the API without needing additional database hardware or more complex deployment topologies. It's also much easier to set up a new API with just an encryption key rather than needing to deploy a new database or adding a dependency on an existing one. After all, a shared token database is a single point of failure. But the Achilles' heel of stateless tokens is how to handle token revocation. If all the state is on the client, it becomes much harder to invalidate that state to revoke a token. There is no database to delete the token from.

There are a few ways to handle this. First, you could just ignore the problem and not allow tokens to be revoked. If your tokens are short-lived and your API does not handle sensitive data or perform privileged operations, then you might be comfortable with the risk of not letting users explicitly log out. But few APIs fit this description; almost all data is sensitive to somebody. This leaves several options, almost all of which involve storing some state on the server after all:

- You can add some minimal state to the database that lists a unique ID associated with the token. To revoke a JWT, you delete the corresponding record from the database. To validate the JWT, you must now perform a database lookup to check if the unique ID is still in the database. If it is not, then the token has been revoked. This is known as an allowlist.[4]
- A twist on the above scheme is to only store the unique ID in the database when the token is revoked, creating a blocklist of revoked tokens. To validate, make sure that there isn't a matching record in the database. The unique ID only needs to be blocked until the token expires, at which point it will be invalid anyway. Using short expiry times helps keep the blocklist small.
- Rather than blocking individual tokens, you can block certain attributes of a set of tokens. For example, it is a common security practice to invalidate all of a user's existing sessions when they change their password. Users often change their password when they believe somebody else may have accessed their account, so invalidating any existing sessions will kick the attacker out. Because there is no record of the existing sessions on the server, you could instead record an entry in the database saying that all tokens issued to user Mary before lunchtime on Friday should be considered invalid. This saves space in the database at the cost of increased query complexity.
- Finally, you can issue short-lived tokens and force the user to reauthenticate regularly. This limits the damage that can be done with a compromised token without needing any additional state on the server but provides a poor user experience. In chapter 7, you'll use OAuth2 refresh tokens to provide a more transparent version of this pattern.

6.5.1 Implementing hybrid tokens

The existing `DatabaseTokenStore` can be used to implement a list of valid JWTs, and this is the simplest and most secure default for most APIs. While this involves giving up on the pure stateless nature of a JWT architecture, and may initially appear to offer the worst of both worlds--reliance on a centralized database along with the risky nature of client-side state--in fact, it offers many advantages over each storage strategy on its own:

- Database tokens can be easily and immediately revoked. In September 2018, Facebook was hit by an attack that exploited a vulnerability in some token-handling code to quickly gain access to the accounts of many users (**https://newsroom .fb.com/news/2018/09/security-update/**). In the wake of the attack, Facebook revoked 90 million tokens, forcing those users to reauthenticate. In a disaster situation, you don't want to be waiting hours for tokens to expire or suddenly finding scalability issues with your blocklist when you add 90 million new entries.
- On the other hand, plain database tokens may be vulnerable to token theft and forgery if the database is compromised, as described in section 5.3 of chapter 5. In that chapter, you hardened database tokens by using the `HmacTokenStore` to prevent forgeries. Wrapping database tokens in a JWT or other authenticated token format achieves the same protections.
- Less security-critical operations can be performed based on data in the JWT alone, avoiding a database lookup. For example, you might decide to let a user see which Natter social spaces they are a member of and how many unread messages they have in each of them without checking the revocation status of the token, but require a database check when they actually try to read one of those or post a new message.
- Token attributes can be moved between the JWT and the database depending on how sensitive they are or how likely they are to change. You might want to store some basic information about the user in the JWT but store a last activity time for implementing idle timeouts in the database because it will change frequently.

DEFINITION An idle timeout (or inactivity logout) automatically revokes an authentication token if it hasn't been used for a certain amount of time. This can be used to automatically log out a user if they have stopped using your API but have forgotten to log out manually.

Listing 6.10 shows the `EncryptedJwtTokenStore` updated to list valid tokens in the database. It does this by taking an instance of the `DatabaseTokenStore` as a constructor argument and uses that to create a dummy token with no attributes. If you wanted to move attributes from the JWT to the database, you can do that here by populating the attributes in the database token and removing them from the JWT token. The token ID returned from the database is then stored inside the JWT as the standard JWT ID (`jti`) claim. Open JwtTokenStore.java in your editor and update it to allowlist tokens in the database as in the listing.

Listing 6.10 Allowlisting JWTs in the database

```
public class EncryptedJwtTokenStore implements SecureTokenStore {

    private final SecretKey encKey;
    private final DatabaseTokenStore tokenAllowlist;

    public EncryptedJwtTokenStore(SecretKey encKey,
                          DatabaseTokenStore tokenAllowlist) {
        this.encKey = encKey;
        this.tokenAllowlist = tokenAllowlist;
    }
    @Override
    public String create(Request request, Token token) {
        var allowlistToken = new Token(token.expiry, token.username);
        var jwtId = tokenAllowlist.create(request, allowlistToken);

        var claimsBuilder = new JWTClaimsSet.Builder()
                .jwtID(jwtId)
                .subject(token.username)
                .audience("https://localhost:4567")
                .expirationTime(Date.from(token.expiry));
        token.attributes.forEach(claimsBuilder::claim);

        var header = new JWEHeader(JWEAlgorithm.DIR,
              EncryptionMethod.A128CBC_HS256);
        var jwt = new EncryptedJWT(header, claimsBuilder.build());

        try {
            var encryptor = new DirectEncrypter(encKey);
            jwt.encrypt(encryptor);
        } catch (JOSEException e) {
            throw new RuntimeException(e);
        }

        return jwt.serialize();
    }
```

❶ Inject a DatabaseTokenStore into the EncryptedJwtTokenStore to use for the allowlist.

❷ Save a copy of the token in the database but remove all the attributes to save space.

❸ Save the database token ID in the JWT as the JWT ID claim.

To revoke a JWT, you then simply delete it from the database token store, as shown in listing 6.11. Parse and decrypt the JWT as before, which will validate the authentication tag, and then extract the JWT ID and revoke it from the database. This will remove the corresponding record from the database. While you still have the JwtTokenStore.java open in your editor, add the implementation of the revoke method from the listing.

Listing 6.11 Revoking a JWT in the database allowlist

```
@Override
public void revoke(Request request, String tokenId) {
    try {
        var jwt = EncryptedJWT.parse(tokenId);                    (1)
        var decryptor = new DirectDecrypter(encKey);              (1)
        jwt.decrypt(decryptor);                                   (1)
        var claims = jwt.getJWTClaimsSet();                       (1)

        tokenAllowlist.revoke(request, claims.getJWTID());        (2)
    } catch (ParseException | JOSEException e) {
        throw new IllegalArgumentException("invalid token", e);
    }
}
```

(1) Parse, decrypt, and validate the JWT using the decryption key.

(2) Extract the JWT ID and revoke it from the DatabaseTokenStore allowlist.

The final part of the solution is to check that the allowlist token hasn't been revoked when reading a JWT token. As before, parse and decrypt the JWT using the decryption key. Then extract the JWT ID and perform a lookup in the `DatabaseTokenStore`. If the entry exists in the database, then the token is still valid, and you can continue validating the other JWT claims as before. But if the database returns an empty result, then the token has been revoked and so it is invalid. Update the `read()` method in JwtTokenStore.java to implement this addition check, as shown in listing 6.12. If you moved some attributes into the database, then you could also copy them to the token result in this case.

Listing 6.12 Checking if a JWT has been revoked

```
    var jwt = EncryptedJWT.parse(tokenId);                        (1)
    var decryptor = new DirectDecrypter(encKey);                  (1)
```

```
    jwt.decrypt(decryptor);                                          ❶

    var claims = jwt.getJWTClaimsSet();
    var jwtId = claims.getJWTID();                                   ❷
    if (tokenAllowlist.read(request, jwtId).isEmpty()) {             ❷
        return Optional.empty();                                     ❸
    }
    // Validate other JWT claims
```

❶ Parse and decrypt the JWT.

❷ Check if the JWT ID still exists in the database allowlist.

❸ If not, then the token is invalid; otherwise, proceed with validating other JWT claims.

Answers to pop quiz questions

1. a and b. HMAC prevents an attacker from creating bogus authentication tokens (spoofing) or tampering with existing ones.
2. e. The aud (audience) claim lists the servers that a JWT is intended to be used by. It is crucial that your API rejects any JWT that isn't intended for that service.
3. False. The algorithm header can't be trusted and should be ignored. You should associate the algorithm with each key instead.
4. a, b, and d. Authenticated encryption includes a MAC so protects against spoofing and tampering threats just like HMAC. In addition, these algorithms protect confidential data from information disclosure threats.
5. d. The IV (or nonce) ensures that every ciphertext is different.
6. True. IVs should be randomly generated. Although some algorithms allow a simple counter, these are very hard to synchronize between API servers and reuse can be catastrophic to security.

Summary

- Token state can be stored on the client by encoding it in JSON and applying HMAC authentication to prevent tampering.
- Sensitive token attributes can be protected with encryption, and efficient authenticated encryption algorithms can remove the need for a separate HMAC step.

- The JWT and JOSE specifications provide a standard format for authenticated and encrypted tokens but have historically been vulnerable to several serious attacks.
- When used carefully, JWT can be an effective part of your API authentication strategy but you should avoid the more error-prone parts of the standard.
- Revocation of stateless JWTs can be achieved by maintaining an allowlist or blocklist of tokens in the database. An allowlisting strategy is a secure default offering advantages over both pure stateless tokens and unauthenticated database tokens.

[1.] This is a very famous example known as the ECB Penguin. You'll find the same example in many introductory cryptography books.

[2.] I wrote Salty Coffee, reusing cryptographic code from Google's Tink library, to provide a simple pure Java solution. Bindings to libsodium are generally faster if you can use a native library.

[3.] A nonce only needs to be unique and could be a simple counter. However, synchronizing a counter across many servers is difficult and error-prone so it's best to always use a random value.

[4.] The terms allowlist and blocklist are now preferred over the older terms whitelist and blacklist due to negative connotations associated with the old terms.