9 Capability-based security and macaroons

This chapter covers

- Sharing individual resources via capability URLs
- Avoiding confused deputy attacks against identity-based access control
- Integrating capabilities with a RESTful API design
- Hardening capabilities with macaroons and contextual caveats

In chapter 8, you implemented identity-based access controls that represent the mainstream approach to access control in modern API design. Sometimes identity-based access controls can come into conflict with other principles of secure API design. For example, if a Natter user wishes to share a message that they wrote with a wider audience, they would like to just copy a link to it. But this won't work unless the users they are sharing the link with are also members of the Natter social space it was posted to, because they won't be granted access. The only way to grant those users access to that message is to either make them members of the space, which violates the principle of least authority (because they now have access to all the messages in that space), or else to copy and paste the whole message into a different system.

People naturally share resources and delegate access to others to achieve their goals, so an API security solution should make this simple and secure; otherwise, your users will find insecure ways to do it anyway. In this chapter, you'll implement capability-based access control techniques that enable secure sharing by taking the principle of least authority (POLA) to its logical conclusion and allowing fine-grained control over access to individual resources. Along the way, you'll see how capabilities prevent a general category of attacks against APIs known as confused deputy attacks.

DEFINITION A confused deputy attack occurs when a component of a system with elevated privileges can be tricked by an attacker into carrying out actions that the attacker themselves would not be allowed to perform.

The CSRF attacks of chapter 4 are classic examples of confused deputy attacks, where the web browser is tricked into carrying out the attacker's requests using the victim's session cookie.

9.1 Capability-based security

A capability is an unforgeable reference to an object or resource together with a set of permissions to access that resource. To illustrate how capability-based security differs from identity-based security, consider the following two ways to copy a file on UNIX[1] systems:

- `cp a.txt b.txt`
- `cat <a.txt >b.txt`

The first, using the `cp` command, takes as input the name of the file to copy and the name of the file to copy it to. The second, using the `cat` command, instead takes as input two file descriptors: one opened for reading and the other opened for writing. It then simply reads the data from the first file descriptor and writes it to the second.

DEFINITION A file descriptor is an abstract handle that represents an open file along with a set of permissions on that file. File descriptors are a type of capability.

If you think about the permissions that each of these commands needs, the `cp` command needs to be able to open any file that you can name for both reading and writing. To allow this, UNIX runs the `cp` command with the same permissions as your own user account, so it can do anything you can do, including deleting all your files and emailing your private photos to a stranger. This violates POLA because the command is given far more permissions than it needs. The `cat` command, on the other hand, just needs to read from its input and write to its output. It doesn't need any permissions at all (but of course UNIX gives it all your permissions anyway). A file descriptor is an example of a capability, because it combines a reference to some resource along with a set of permissions to act on that resource.

Compared with the more dominant identity-based access control techniques discussed in chapter 8, capabilities have several differences:

- Access to resources is via unforgeable references to those objects that also grant authority to access that resource. In an identity-based system, anybody can attempt to access a resource, but they might be denied access depending on who they are. In a capability-based system, it is impossible to send a request to a resource if you do not have a capability to access it. For example, it is impossible to write to a file descriptor that your process doesn't have. You'll see in section 9.2 how this is implemented for REST APIs.
- Capabilities provide fine-grained access to individual resources, and often support POLA more naturally than identity-based systems. It is much easier to delegate a small part of your authority to somebody else by giving them some capabilities without giving them access to your whole account.
- The ability to easily share capabilities can make it harder to determine who has access to which resources via your API. In practice this is often true for identity-based systems too, as people share access in other ways (such as by sharing passwords).
- Some capability-based systems do not support revoking capabilities after they have been granted. When revocation is supported, revoking a widely shared capability may deny access to more people than was intended.

One of the reasons why capability-based security is less widely used than identity-based security is due to the widespread belief that capabilities are hard to control due to easy sharing and the apparent difficulty of revocation. In fact, these problems are solved by real-world capability systems as discussed in the paper Capability Myths Demolished by Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro (**http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf**). To take one example, it is often assumed that capabilities can be used only for discretionary access control, because the creator of an object (such as a file) can share capabilities to access that file with anyone. But in a pure capability system, communications between people are also controlled by capabilities (as is the ability to create files in the first place), so if Alice creates a new file, she can share a capability to access this file with Bob only if she has a capability allowing her to communicate with Bob. Of course, there's nothing to stop Bob asking Alice in person to perform actions on the file, but that is a problem that no access control system can prevent.

A brief history of capabilities

Capability-based security was first developed in the context of operating systems such as KeyKOS in the 1970s and has been applied to programming languages and network protocols since then. The IBM System/38, which was the predecessor of the successful AS/400 (now IBM i), used capabilities for managing access to objects. In the 1990s, the E programming language (**http://erights.org**) combined capability-based security with object-oriented (OO) programming to create object-capability-based security (or ocaps), where capabilities are just normal object references in a memory-safe OO programming language. Object-capability-based security fits well with conventional wisdom regarding good OO design and design patterns, because both emphasize eliminating global variables and avoiding static methods that perform side effects.

E also included a secure protocol for making method calls across a network using capabilities. This protocol has been adopted and updated by the Cap'n Proto (**https:// capnproto.org/rpc.html#security**) framework, which provides a very efficient binary protocol for implementing APIs based on remote procedure calls. Capabilities are also now making an appearance on popular websites and REST APIs.

9.2 Capabilities and REST

The examples so far have been based on operating system security, but capability-based security can also be applied to REST APIs available over HTTP. For example, suppose you've developed a Natter iOS app that allows the user to select a profile picture, and you want to allow users to upload a photo from their Dropbox account. Dropbox supports OAuth2 for third-party apps, but the access allowed by OAuth2 scopes is relatively broad; typically, a user can grant access only to all their files or else create an app-specific folder separate from the rest of their files. This can work well when the application needs regular access to lots of your files, but in this case your app needs only temporary access to download a single file chosen by the user. It violates POLA to grant permanent read-only access to your entire Dropbox just to upload one photo. Although OAuth scopes are great for restricting permissions granted to third-party apps, they tend to be static and applicable to all users. Even if you had a scope for each individual file, the app would have to already know which file it needed access to at the point of making the authorization request.**2**

To support this use case, Dropbox developed the Chooser and Saver APIs (see **https:// www.dropbox.com/developers/chooser** and **https://www.dropbox.com/developers/ saver**), which allow an app developer to ask the user for one-off access to specific files in their Dropbox.

Rather than starting an OAuth flow, the app developer instead calls an SDK function that will display a Dropbox-provided file selection UI as shown in figure 9.1. Because this UI is implemented as a separate browser window running on **dropbox.com** and not as part of the third-party app, it can show all the user's files. When the user selects a file, Dropbox returns a capability to the application that allows it to access just the file that the user selected for a short period of time (4 hours currently for the Chooser API).

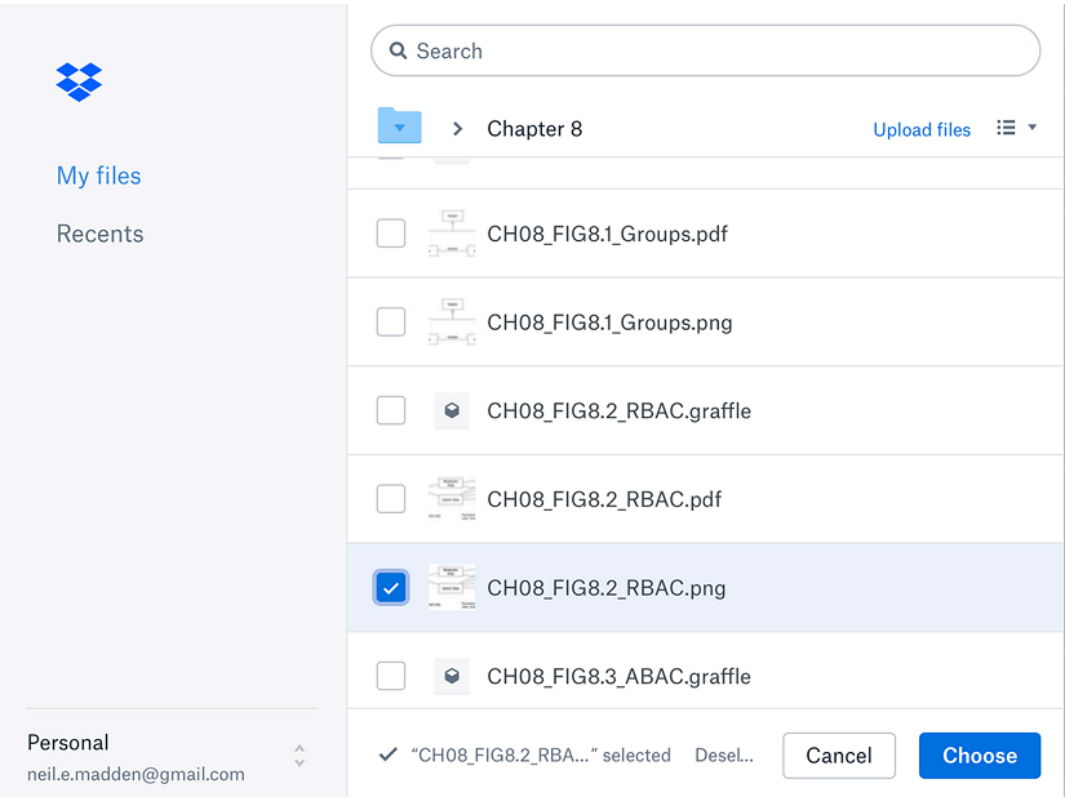

Figure 9.1 The Dropbox Chooser UI allows a user to select individual files to share with an application. The app is given time-limited read-only access to just the files the user selects.

The Chooser and Saver APIs provide a number of advantages over a normal OAuth2 flow for this simple file sharing use case:

- The app author doesn't have to decide ahead of time what resource it needs to access. Instead, they just tell Dropbox that they need a file to open or to save data to and Dropbox lets the user decide which file to use. The app never gets to see a list of the user's other files at all.
- Because the app is not requesting long-term access to the user's account, there is no need for a consent page to ensure the user knows what access they are granted. Selecting a file in the UI implicitly indicates consent and because the scope is so fine-grained, the risks of abuse are much lower.

- The UI is implemented by Dropbox and so is consistent for every app and web page that uses the API. Little details like the “Recent” menu item work consistently across all apps.

For these use cases, capabilities provide a very intuitive and natural user experience that is also significantly more secure than the alternatives. It’s often assumed that there is a natural trade-off between security and usability: the more secure a system is, the harder it must be to use. Capabilities seem to defy this conventional wisdom, because moving to a more fine-grained management of permissions allows more convenient patterns of interaction. The user chooses the files they want to work with, and the system grants the app access to just those files, without needing a complicated consent process.

Confused deputies and ambient authority

Many common vulnerabilities in APIs and other software are variations on what is known as a confused deputy attack, such as the CSRF attacks discussed in chapter 4, but many kinds of injection attack and XSS are also caused by the same issue. The problem occurs when a process is authorized to act with your authority (as your “deputy”), but an attacker can trick that process to carry out malicious actions. The original confused deputy (**http://cap-lore.com/CapTheory/ConfusedDeputy.html**) was a compiler running on a shared computer. Users could submit jobs to the compiler and provide the name of an output file to store the result to. The compiler would also keep a record of each job for billing purposes. Somebody realized that they could provide the name of the billing file as the output file and the compiler would happily overwrite it, losing all records of who had done what. The compiler had permissions to write to any file and this could be abused to overwrite a file that the user themselves could not access.

In CSRF, the deputy is your browser that has been given a session cookie after you logged in. When you make requests to the API from JavaScript, the browser automatically adds the cookie to authenticate the requests. The problem is that if a malicious website makes requests to your API, then the browser will also attach the cookie to those requests, unless you take additional steps to prevent that (such as the anti-CSRF measures in chapter 4). Session cookies are an example of ambient authority: the cookie forms part of the environment in which a web page runs and is transparently added to requests. Capability-based security aims to re-

move all sources of ambient authority and instead require that each request is specifically authorized according to POLA.

DEFINITION When the permission to perform an action is automatically granted to all requests that originate from a given environment this is known as ambient authority. Examples of ambient authority include session cookies and allowing access based on the IP address a request comes from. Ambient authority increases the risks of confused deputy attacks and should be avoided whenever possible.

9.2.1 Capabilities as URIs

File descriptors rely on special regions of memory that can be altered only by privileged code in the operating system kernel to ensure that processes can't tamper or create fake file descriptors. Capability-secure programming languages are also able to prevent tampering by controlling the runtime in which code runs. For a REST API, this isn't an option because you can't control the execution of remote clients, so another technique needs to be used to ensure that capabilities cannot be forged or tampered with. You have already seen several techniques for creating unforgeable tokens in chapters 4, 5, and 6, using unguessable large random strings or using cryptographic techniques to authenticate the tokens. You can reuse these token formats to create capability tokens, but there are several important differences:

- Token-based authentication conveys the identity of a user, from which their permissions can be looked up. A capability instead directly conveys some permissions and does not identify a user at all.
- Authentication tokens are designed to be used to access many resources under one API, so are not tied to any one resource. Capabilities are instead directly coupled to a resource and can be used to access only that resource. You use different capabilities to access different resources.
- A token will typically be short-lived because it conveys wide-ranging access to a user's account. A capability, on the other hand, can live longer because it has a much narrower scope for abuse.

REST already has a standard format for identifying resources, the URI, so this is the natural representation of a capability for a REST API. A capability represented as a URI is known as a capability URI. Capability URIs are widespread on the web, in the form of links sent in password reset emails, GitHub Gists, and document sharing as in the Dropbox example.

DEFINITION A capability URI (or capability URL) is a URI that both identifies a resource and conveys a set of permissions to access that resource. Typically, a capability URI encodes an unguessable token into some part of the URI structure.

To create a capability URI, you can combine a normal URI with a security token. There are several ways that you can do this, as shown in figure 9.2.

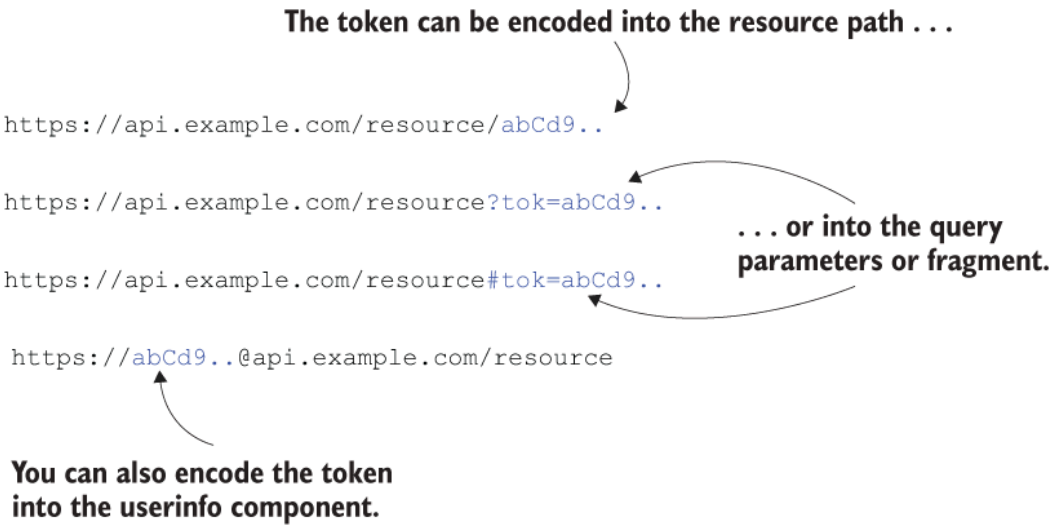

Figure 9.2 There are many ways to encode a security token into a URI. You can encode it into the resource path, or you can provide it using a query parameter. More sophisticated representations encode the token into the fragment or userinfo elements of the URI, but these require some client-side parsing.

A commonly used approach is to encode a random token into the path component of the URI, which is what the Dropbox Chooser API does, returning URIs like the following:

```
https://dl.dropboxusercontent.com/1/view/8ygmwuqzf1l6x7c/
➥ book/graphics/CH08_FIG8.2_RBAC.png
```

In the Dropbox case, the random token is encoded into a prefix of the actual file path. Although this is a natural representation, it means that the same resource may be represented by URIs with completely different paths depending on the token, so a client that receives access to the same resource through different capability URIs may not be able to tell that they actually refer to the same resource. An alternative is to pass the token as a query parameter, in which case the Dropbox URI would look like the following:

```
https://dl.dropboxusercontent.com/1/view/
➥ book/graphics/CH08_FIG8.2_RBAC.png?token=8ygmwuqzf1l6x7c
```

There is a standard form for such URIs when the token is an OAuth2 token defined by RFC 6750 (**https://tools.ietf.org/html/rfc6750#section-2.3**) using the parameter name `access_token`. This is often the simplest approach to implement because it requires no changes to existing resources, but it shares some security weaknesses with the path-based approach:

- Both URI paths and query parameters are frequently logged by web servers and proxies, which can make the capability available to anybody who has access to the logs. Using TLS will prevent proxies from seeing the URI, but a request may still pass through several servers unencrypted in a typical deployment.
- The full URI may be visible to third parties through the HTTP `Referer` header or the `window.referrer` variable exposed to content running in an HTML iframe. You can use the `Referrer-Policy` header and `rel="noreferrer"` attribute on links in your UI to prevent this leakage. See **http://mng.bz/1g0g** for details.
- URIs used in web browsers may be accessible to other users by looking at your browser history.

To harden capability URIs against these threats, you can encode the token into the fragment component or the URI or even the userinfo part that was originally designed for storing HTTP Basic credentials in a URI. Neither the fragment nor the userinfo component of a URI are sent to a web server by default, and they are both stripped from URIs communicated in `Referer` headers.

Credentials in URIs: A lesson from history

The desire to share access to private resources simply by sharing a URI is not new. For a long time, browsers supported encoding a username and password into a HTTP URL in the form http://alice:secret@example.com/resource. When such a link was clicked, the browser would send the username and password using HTTP Basic authentication (see chapter 3). Though convenient, this is widely considered to be a security disaster. For a start, sharing a username and password provides full access to your account to anybody who sees the URI. Secondly, attackers soon realized that this could be used to create con-

vincing phishing links such as http://www.google.com:80@evil.example.com/login.html. An unsuspecting user would see the google .com domain at the start of the link and assume it was genuine, when in fact this is just a username and they will really be sent to a fake login page on the attacker's site. To prevent these attacks, browser vendors have stopped supporting this URI syntax and most now aggressively remove login information when displaying or following such links. Although capability URIs are significantly more secure than directly sharing a password, you should still be aware of any potential for misuse if you display URIs to users.

CAPABILITY URIS FOR REST APIS

The drawbacks of capability URIs just mentioned apply when they are used as a means of navigating a website. When capability URIs are used in a REST API many of these issues don't apply:

- The `Referer` header and `window.referrer` variables are populated by browsers when a user directly navigates from one web page to another, or when one page is embedded into another in an iframe. Neither of these apply to the typical JSON responses from an API because these are not directly rendered as pages.
- Similarly, because users don't typically navigate directly to API endpoints, these URIs will not end up in the browser history.
- API URIs are also unlikely to be bookmarked or otherwise saved for a long period of time. Typically, a client knows a few permanent URIs as entry points to an API and then navigates to other URIs as it accesses resources. These resource URIs can use short-lived tokens to mitigate against tokens being leaked in access logs. This idea is explored further in section 9.2.3.

In the remainder of the chapter, you'll use capability URIs with the token encoded into the query parameter because this is simple to implement. To mitigate any threat from tokens leaking in log files, you'll use short-lived tokens and apply further protections in section 9.2.4.

Pop quiz

1. Which of the following are good places to encode a token into a capability URI?
    1. The fragment
    2. The hostname
    3. The scheme name

4. The port number
5. The path component
6. The query parameters
7. The userinfo component

2. Which of the following are differences between capabilities and token-based authentication?
    1. Capabilities are bulkier than authentication tokens.
    2. Capabilities can't be revoked, but authentication tokens can.
    3. Capabilities are tied to a single resource, while authentication tokens are applicable to all resources in an API.
    4. Authentication tokens are tied to an individual user identity, while capability tokens can be shared between users
    5. Authentication tokens are short-lived, while capabilities often have a longer lifetime.

The answers are at the end of the chapter.

9.2.2 Using capability URIs in the Natter API

To add capability URIs to Natter, you first need to implement the code to create a capability URI. To do this, you can reuse an existing `TokenStore` implementation to create the token component, encoding the resource path and permissions into the token attributes as shown in listing 9.1. Because capabilities are not tied to an individual user account, you should leave the username field of the token blank. The token can then be encoded into the URI as a query parameter, using the standard `access_token` field from RFC 6750. You can use the `java.net.URI` class to construct the capability URI, passing in the path and query parameters. Some of the capability URIs you'll create will be long-lived, but others will be short-lived to mitigate against tokens being stolen. To support this, allow the caller to specify how long the capability should live for by adding an expiry `Duration` argument that is used to set the expiry time of the token.

Open the Natter API project[3] and navigate to src/main/java/com/manning/ apisecurityinaction/controller and create a new file named CapabilityController.java with the content of listing 9.1 and save the file.

Listing 9.1 Generating capability URIs

```
package com.manning.apisecurityinaction.controller;
```

```
import com.manning.apisecurityinaction.token.SecureTokenStore;
import com.manning.apisecurityinaction.token.TokenStore.Token;
import spark.*;
import java.net.*;
import java.time.*;
import java.util.*;
import static java.time.Instant.now;
public class CapabilityController {
    private final SecureTokenStore tokenStore;                          (1)

    public CapabilityController(SecureTokenStore tokenStore) {          (1)
        this.tokenStore = tokenStore;                                   (1)
    }

    public URI createUri(Request request, String path, String perms,
            Duration expiryDuration) {

        var token = new Token(now().plus(expiryDuration), null);        (2)
        token.attributes.put("path", path);                             (3)
        token.attributes.put("perms", perms);                           (3)
        var tokenId = tokenStore.create(request, token);
        var uri = URI.create(request.uri());
        return uri.resolve(path + "?access_token=" + tokenId);          (4)
    }
}
```

(1) Use an existing SecureTokenStore to generate tokens.

(2) Leave the username null when creating the token.

(3) Encode the resource path and permissions into the token.

(4) Add the token to the URI as a query parameter.

You can now wire up code to create the `CapabilityController` inside your main method, so open Main.java in your editor and create a new instance of the object along with a token store for it to use. You can use any secure token store implementation, but for this chapter you'll use the `DatabaseTokenStore` because it creates short tokens and therefore short URIs.

NOTE If you worked through chapter 6 and chose to mark the `DatabaseTokenStore` as a `ConfidentialTokenStore` only, then you'll need to wrap it in a `HmacTokenStore` in the following snippet. Refer to chapter 6 (section 6.4) if you get stuck.

You should also pass the new controller as an additional argument to the `SpaceController` constructor, because you will shortly use it to create capability URIs:

```
var database = Database.forDataSource(datasource);
var capController = new CapabilityController(
        new DatabaseTokenStore(database));
var spaceController = new SpaceController(database, capController);
var userController = new UserController(database);
```

Before you can start generating capability URIs, though, you need to make one tweak to the database token store. The current store requires that every token has an associated user and will raise an error if you try to save a token with a `null` username. Because capabilities are not identity-based, you need to remove this restriction. Open `schema.sql` in your editor and remove the not-null constraint from the `tokens` table by deleting the words `NOT NULL` from the end of the `user_id` column definition. The new table definition should look like the following:

```
CREATE TABLE tokens(
    token_id VARCHAR(30) PRIMARY KEY,
    user_id VARCHAR(30) REFERENCES users(user_id),      (1)
    expiry TIMESTAMP NOT NULL,
    attributes VARCHAR(4096) NOT NULL
);
```

(1) Remove the NOT NULL constraint here.

RETURNING CAPABILITY URIS

You can now adjust the API to return capability URIs that can be used to access social spaces and messages. Where the API currently returns a simple path to a social space or message such as `/spaces/1`, you'll instead return a full capability URI that can be used to access it. To do this, you need to add the `CapabilityController` as a new argument to the `SpaceController` constructor, as shown in listing 9.2. Open SpaceController.java in your editor and add the new field and constructor argument.

Listing 9.2 Adding the CapabilityController

```
public class SpaceController {
  private static final Set<String> DEFINED_ROLES =
          Set.of("owner", "moderator", "member", "observer");

  private final Database database;
  private final CapabilityController capabilityController;                    (1)

  public SpaceController(Database database,
                         CapabilityController capabilityController) {          (1)
    this.database = database;
    this.capabilityController = capabilityController;                          (1)
  }
```

(1) Add the CapabilityController as a new field and constructor argument.

The next step is to adjust the `createSpace` method to use the `CapabilityController` to create a capability URI to return, as shown in listing 9.3. The code changes are very minimal: simply call the `createUri` method to create the capability URI. As the user that creates a space is given full permissions over it, you can pass in all permissions when creating the URI. Once a space has been created, the only way to access it will be through the capability URI, so ensure that this link doesn't expiry by passing a large expiry time. Then use the `uri.toASCIIString()` method to convert the URI into a properly encoded string. Because you're going to use capabilities for access you can remove the lines that insert into the `user_roles` table; these are no longer needed. Open SpaceController.java in your editor and adjust the implementation of the `createSpace` method to match listing 9.3. New code is highlighted in bold.

Listing 9.3 Returning a capability URI

```
  public JSONObject createSpace(Request request, Response response) {
    var json = new JSONObject(request.body());
    var spaceName = json.getString("name");
    if (spaceName.length() > 255) {
      throw new IllegalArgumentException("space name too long");
    }
    var owner = json.getString("owner");
    if (!owner.matches("[a-zA-Z][a-zA-Z0-9]{1,29}")) {
      throw new IllegalArgumentException("invalid username");
    }
    var subject = request.attribute("subject");
    if (!owner.equals(subject)) {
```

```
        throw new IllegalArgumentException(
                "owner must match authenticated user");
      }

      return database.withTransaction(tx -> {
        var spaceId = database.findUniqueLong(
            "SELECT NEXT VALUE FOR space_id_seq;");

        database.updateUnique(
            "INSERT INTO spaces(space_id, name, owner) " +
                "VALUES(?, ?, ?);", spaceId, spaceName, owner);

        var expiry = Duration.ofDays(100000);                    (1)
        var uri = capabilityController.createUri(request,        (2)
                "/spaces/" + spaceId, "rwd", expiry);            (2)

        response.status(201);
        response.header("Location", uri.toASCIIString());        (3)

        return new JSONObject()
                .put("name", spaceName)
                .put("uri", uri);                                (2)
      });
    }
```

(1) Ensure the link doesn't expire.

(2) Create a capability URI with full permissions.

(3) Return the URI as a string in the Location header and JSON response.

VALIDATING CAPABILITIES

Although you are returning a capability URL, the Natter API is still using RBAC to grant access to operations. To convert the API to use capabilities instead, you can replace the current `UserController.lookupPermissions` method, which determines permissions by looking up the authenticated user's roles, with an alternative that reads the permissions directly from the capability token. Listing 9.4 shows the implementation of a `lookupPermissions` filter for the `CapabilityController`.

The filter first checks for a capability token in the `access_token` query parameter. If no token is present, then it returns without setting any permissions. This will result in no access being granted. After that, you need

to check that the resource being accessed exactly matches the resource that the capability is for. In this case, you can check that the path being accessed matches the path stored in the token attributes, by looking at the `request.pathInfo()` method. If all these conditions are satisfied, then you can set the permissions on the request based on the permissions stored in the capability token. This is the same `perms` request attribute that you set in chapter 8 when implementing RBAC, so the existing permission checks on individual API calls will work as before, picking up the permissions from the capability URI rather than from a role lookup. Open CapabilityController.java in your editor and add the new method from listing 9.4.

Listing 9.4 Validating a capability token

```
public void lookupPermissions(Request request, Response response) {
    var tokenId = request.queryParams("access_token");                    ❶
    if (tokenId == null) { return; }

    tokenStore.read(request, tokenId).ifPresent(token -> {                ❷
        var tokenPath = token.attributes.get("path");                     ❷
        if (Objects.equals(tokenPath, request.pathInfo())) {              ❷
            request.attribute("perms",                                    ❸
                    token.attributes.get("perms"));                       ❸
        }
    });
}
```

❶ Look up the token from the query parameters.

❷ Check that the token is valid and matches the resource path.

❸ Copy the permissions from the token to the request.

To complete the switch-over to capabilities you then need to change the filters used to lookup the current user's permissions to instead use the new capability filter. Open Main.java in your editor and locate the three `before ()` filters that currently call `userController::lookupPermissions` and change them to call the capability controller filter. I've highlighted the change of controller in bold:

```
before("/spaces/:spaceId/messages",
        capController::lookupPermissions);
before("/spaces/:spaceId/messages/*",
```

```
        capController::lookupPermissions);
before("/spaces/:spaceId/members",
        capController::lookupPermissions);
```

You can now restart the API server, create a user, and then create a new social space. This works exactly like before, but now you get back a capability URI in the response to creating the space:

```
$ curl -X POST -H 'Content-Type: application/json' \
    -d '{"name":"test","owner":"demo"}' \
    -u demo:password https://localhost:4567/spaces
{"name":"test",
➥ "uri":"https://localhost:4567/spaces/1?access_token=
➥ jKbRWGFDuaY5yKFyiiF3Lhfbz-U"}
```

TIP You may be wondering why you had to create a user and authenticate before you could create a space in the last example. After all, didn't we just move away from identity-based security? The answer is that the identity is not being used to authorize the action in this case, because no permissions are required to create a new social space. Instead, authentication is required purely for accountability, so that there is a record in the audit log of who created the space.

9.2.3 HATEOAS

You now have a capability URI returned from creating a social space, but you can't do much with it. The problem is that this URI allows access to only the resource representing the space itself, but to read or post messages to the space the client needs to access the sub-resource `/spaces/1/messages` instead. Previously, this wouldn't be a problem because the client could just construct the path to get to the messages and use the same token to also access that resource. But a capability token gives access to only a single specific resource, following POLA. To access the messages, you'll need a different capability, but capabilities are unforgeable so you can't just create one! It seems like this capability-based security model is a real pain to use.

If you are a RESTful design aficionado, you may know that having the client just know that it needs to add `/messages` to the end of a URI to access the messages is a violation of a central REST principle, which is that client interactions should be driven by hypertext (links). Rather than a client needing to have specific knowledge about how to access resources

in your API, the server should instead tell the client where resources are and how to access them. This principle is given the snappy title Hypertext as the Engine of Application State, or HATEOAS for short. Roy Fielding, the originator of the REST design principles, has stated that this is a crucial aspect of REST API design (**http://mng.bz/Jx6v**).

PRINCIPLE HATEOAS, or hypertext as the engine of application state, is a central principle of REST API design that states that a client should not need to have specific knowledge of how to construct URIs to access your API. Instead, the server should provide this information in the form of hyperlinks and form templates.

The aim of HATEOAS is to reduce coupling between the client and server that would otherwise prevent the server from evolving its API over time because it might break assumptions made by clients. But HATEOAS is also a perfect fit for capability URIs because we can return new capability URIs as links in response to using another capability URI, allowing a client to securely navigate from resource to resource without needing to manufacture any URIs by themselves.[4]

You can allow a client to access and post new messages to the social space by returning a second URI from the `createSpace` operation that allows access to the messages resource for this space, as shown in listing 9.5. You simply create a second capability URI for that path and return it as another link in the JSON response. Open SpaceController.java in your editor again and update the end of the createSpace method to create the second link. The new lines of code are highlighted in bold.

Listing 9.5 Adding a messages link

```
    var uri = capabilityController.createUri(request,
            "/spaces/" + spaceId, "rwd", expiry);
    var messagesUri = capabilityController.createUri(request,          (1)
            "/spaces/" + spaceId + "/messages", "rwd", expiry);        (1)

    response.status(201);
    response.header("Location", uri.toASCIIString());

    return new JSONObject()
            .put("name", spaceName)
            .put("uri", uri)
            .put("messages", messagesUri);                             (2)
```

❶ Create a new capability URI for the messages.

❷ Return the messages URI as a new field in the response.

If you restart the API server again and create a new space, you'll see both URIs are now returned. A GET request to the `messages` URI will return a list of messages in the space, and this can now be accessed by anybody with that capability URI. For example, you can open that link directly in a web browser. You can also POST a new message to the same URI. Again, this operation requires authentication in addition to the capability URI because the message explicitly claims to be from a particular user and so the API should authenticate that claim. Permission to post the message comes from the capability, while proof of identity comes from authentication:

```
$ curl -X POST -H 'Content-Type: application/json' \
    -u demo:password \                                    ❶
    -d '{"author":"demo","message":"Hello!"}' \
  'https://localhost:4567/spaces/1/messages?access_token=
➥ u9wu69dl5L8AT9FNe03TM-s4H8M'                           ❷
```

❶ Proof of identity is supplied by authenticating.

❷ Permission to post is granted by the capability URI alone.

SUPPORTING DIFFERENT LEVELS OF ACCESS

The capability URIs returned so far provide full access to the resources that they identify, as indicated by the `rwd` permissions (read-write-delete, if you remember from chapter 3). This means that it's impossible to give somebody else access to the space without giving them full access to delete other user's messages. So much for POLA!

One solution to this is to return multiple capability URIs with different levels of access, as shown in listing 9.6. The space owner can then give out the more restricted URIs while keeping the URI that grants full privileges for trusted moderators only. Open SpaceController.java again and add the additional capabilities from the listing. Restart the API and try performing different actions with different capabilities.

Listing 9.6 Restricted capabilities

```
        var uri = capabilityController.createUri(request,
                "/spaces/" + spaceId, "rwd", expiry);
        var messagesUri = capabilityController.createUri(request,
                "/spaces/" + spaceId + "/messages", "rwd", expiry);
        var messagesReadWriteUri = capabilityController.createUri(    (1)
                request, "/spaces/" + spaceId + "/messages", "rw",     (1)
                expiry);                                               (1)
        var messagesReadOnlyUri = capabilityController.createUri(     (1)
                request, "/spaces/" + spaceId + "/messages", "r",      (1)
                expiry);                                               (1)

        response.status(201);
        response.header("Location", uri.toASCIIString());

        return new JSONObject()
                .put("name", spaceName)
                .put("uri", uri)
                .put("messages-rwd", messagesUri)                      (2)
                .put("messages-rw", messagesReadWriteUri)              (2)
                .put("messages-r", messagesReadOnlyUri);               (2)
```

(1) Create additional capability URIs with restricted permissions.

(2) Return the additional capabilities.

To complete the conversion of the API to capability-based security, you need to go through the other API actions and convert each to return appropriate capability URIs. This is largely a straightforward task, so we won't cover it here. One aspect to be aware of is that you should ensure that the capabilities you return do not grant more permissions than the capability that was used to access a resource. For example, if the capability used to list messages in a space granted only read permissions, then the links to individual messages within a space should also be read-only. You can enforce this by always basing the permissions for a new link on the permissions set for the current request, as shown in listing 9.7 for the `findMessages` method. Rather than providing read and delete permissions for all messages, you instead use the permissions from the existing request. This ensures that users in possession of a moderator capability will see links that allow both reading and deleting messages, while ordinary access through a read-write or read-only capability will only see read-only message links.

Listing 9.7 Enforcing consistent permissions

```
    var perms = request.<String>attribute("perms")                    ❶
            .replace("w", "");                                        ❷
    response.status(200);
    return new JSONArray(messages.stream()
        .map(msgId -> "/spaces/" + spaceId + "/messages/" + msgId)
        .map(path ->
            capabilityController.createUri(request, path, perms))     ❸
        .collect(Collectors.toList()));
```

❶ Look up the permissions from the current request.

❷ Remove any permissions that are not applicable.

❸ Create new capabilities using the revised permissions.

Update the remaining methods in the SpaceController.java file to return appropriate capability URIs, remembering to follow POLA. The GitHub repository accompanying the book (**https://github.com/NeilMadden/apisecurityinaction**) has completed source code if you get stuck, but I'd recommend trying this yourself first.

TIP You can use the ability to specify different expiry times for links to implement useful functionality. For example, when a user posts a new message, you can return a link that lets them edit it for a few minutes only. A separate link can provide permanent read-only access. This allows users to correct mistakes but not change historical messages.

Pop quiz

3. The capability URIs for each space use never-expiring database tokens. Over time, this will fill the database with tokens. Which of the following are ways you could prevent this?
    1. Hashing tokens in the database
    2. Using a self-contained token format such as JWTs
    3. Using a cloud-native database that can scale up to hold all the tokens
    4. Using the `HmacTokenStore` in addition to the `DatabaseTokenStore`
    5. Reusing an existing token when the same capability has already been issued
4. Which is the main reason why HATEOAS is an important design principle when using capability URIs? Pick one answer.
    1. HATEOAS is a core part of REST.

2. Capability URIs are hard to remember.
3. Clients can't be trusted to make their own URIs.
4. Roy Fielding, the inventor of REST, says that it's important.
5. A client can't make their own capability URIs and so can only access other resources through links.

The answers are at the end of the chapter.

9.2.4 Capability URIs for browser-based clients

In section 9.2.1, I mentioned that putting the token in the URI path or query parameters is less than ideal because these can leak in audit logs, `Referer` headers, and through your browser history. These risks are limited when capability URIs are used in an API but can be a real problem when these URIs are directly exposed to users in a web browser client. If you use capability URIs in your API, browser-based clients will need to somehow translate the URIs used in the API into URIs used for navigating the UI. A natural approach would be to use capability URIs for this too, reusing the tokens from the API URIs. In this section, you'll see how to do this securely.

One approach to this problem is to put the token in a part of the URI that is not usually sent to the server or included in `Referer` headers. The original solution was developed for the Waterken server that used capability URIs extensively, under the name web-keys (**http://waterken.sourceforge.net/web-key/**). In a web-key, the unguessable token is stored in the fragment component of the URI; that is, the bit after a # character at the end of the URI. The fragment is normally used to jump to a particular location within a larger document, and has the advantage that it is never sent to the server by clients and never included in a `Referer` header or `window.referrer` field in JavaScript, and so is less susceptible to leaking. The downside is that because the server doesn't see the token, the client must extract it from the URI and send it to the server by other means.

In Waterken, which was designed for web applications, when a user clicked a web-key link in the browser, it loaded a simple template JavaScript page. The JavaScript then extracted the token from the query fragment (using the `window.location.hash` variable) and made a second call to the web server, passing the token in a query parameter. The flow is shown in figure 9.3.

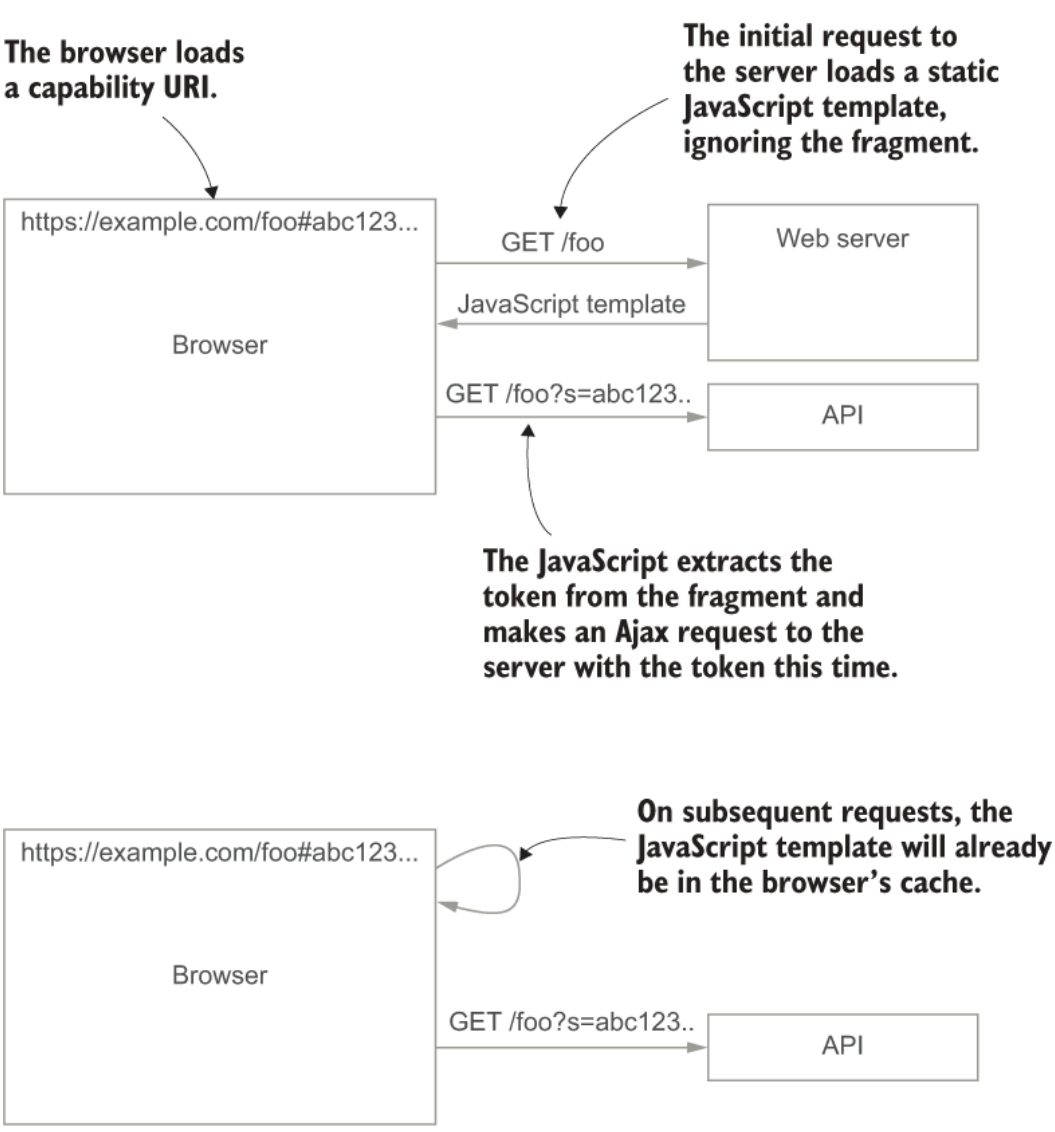

Figure 9.3 In the Waterken web-key design for capability URIs, the token is stored in the fragment of the URI, which is never sent to the server. When a browser loads such a URI, it will initially load a static JavaScript page that then extracts the token from the fragment and uses it to make Ajax requests to the API. The JavaScript template can be cached by the browser, avoiding the extra roundtrip for subsequent requests.

Because the JavaScript template itself contains no sensitive data and is the same for all URIs, it can be served with long-lived cache-control headers and so after the browser has loaded it once, it can be reused for all subsequent capability URIs without an extra call to the server, as shown in the lower half of figure 9.3. This approach works well with single-page apps (SPAs) because they often already use the fragment in this way to permit navigation in the app without causing the page to reload while still populating the browser history.

WARNING Although the fragment component is not sent to the server, it will be included if a redirect occurs. If your app needs to redirect to another site, you should always explicitly include a fragment component in the redirect URI to avoid accidentally leaking tokens in this way.

Listing 9.8 shows how to parse and load a capability URI in this format from a JavaScript API client. It first parses the URI using the `URL` class and extracts the token from the `hash` field, which contains the fragment component. This field include the literal "#" character at the start, so use `hash.substring(1)` to remove this. You should then remove this component from the URI to send to the API and instead add the token back as a query parameter. This ensures that the `CapabilityController` will see the token in the expected place. Navigate to src/main/resources/public and create a new file named capability.js with the contents of the listing.

NOTE This code assumes that UI pages correspond directly to URIs in your API. For an SPA this won't be true, and there is (by definition) a single UI page that handles all requests. In this case, you'll need to encode the API path and the token into the fragment together in a form such as `#/spaces/1/messages&tok=abc123`. Modern frameworks such as Vue or React can use the HTML 5 history API to make SPA URIs look like normal URIs (without the fragment). When using these frameworks, you should ensure the token is in the real fragment component; otherwise, the security benefits are lost.

Listing 9.8 Loading a capability URI from JavaScript

```
function getCap(url, callback) {
    let capUrl = new URL(url);                    (1)
    let token = capUrl.hash.substring(1);         (1)
    capUrl.hash = '';                             (2)
    capUrl.search = '?access_token=' + token;     (3)

    return fetch(capUrl.href)                     (4)
    .then(response => response.json())
    .then(callback)
    .catch(err => console.error('Error: ', err));
}
```

(1) Parse the URL and extract the token from the fragment (hash) component.

(2) Blank out the fragment.

(3) Add the token to the URI query parameters.

(4) Now fetch the URI to call the API with the token.

9.2.5 Combining capabilities with identity

All calls to the Natter API are now authorized purely using capability tokens, which are scoped to an individual resource and not tied to any user. As you saw with the simple message browser example in the last section, you can even hard-code read-only capability URIs into a web page to allow completely anonymous browsing of messages. Some API calls still require user authentication though, such as creating a new space or posting a message. The reason is that those API actions involve claims about who the user is, so you still need to authenticate those claims to ensure they are genuine, for accountability reasons rather than for authorization. Otherwise, anybody with a capability URI to post messages to a space could use it to impersonate any other user.

Pop quiz

5. Which of the following is the main security risk when including a capability token in the fragment component of a URI?
    1. URI fragments aren't RESTful.
    2. The random token makes the URI look ugly.
    3. The fragment may be leaked in server logs and the HTTP `Referer` header.
    4. If the server performs a redirect, the fragment will be copied to the new URI.
    5. The fragment may already be used for other data, causing it to be overwritten.

The answer is at the end of the chapter.

You may also want to positively identify users for other reasons, such as to ensure you have an accurate audit log of who did what. Because a capability URI may be shared by lots of users, it is useful to identify those users independently from how their requests are authorized. Finally, you may want to apply some identity-based access controls on top of the capability-based access. For example, in Google Docs (**https://docs.google.com**) you can share documents using capability URIs, but you can also restrict this sharing to only users who have an account in your company's domain. To access the document, a user needs to both have the link and be signed into a Google account linked to the same company.

There are a few ways to communicate identity in a capability-based system:

- You can associate a username and other identity claims with each capability token. The permissions in the token are still what grants access, but the token additionally authenticates identity claims about the user that can be used for audit logging or additional access checks. The major downside of this approach is that sharing a capability URI lets the recipient impersonate you whenever they make calls to the API using that capability. Nevertheless, this approach can be useful when generating short-lived capabilities that are only intended for a single user. The link sent in a password reset email can be seen as this kind of capability URI because it provides a limited-time capability to reset the password tied to one user's account.
- You could use a traditional authentication mechanism, such as a session cookie, to identify the user in addition to requiring a capability token, as shown in figure 9.4. The cookie would no longer be used to authorize API calls but would instead be used to identify the user for audit logging or for additional checks. Because the cookie is no longer used for access control, it is less sensitive and so can be a long-lived persistent cookie, reducing the need for the user to frequently log in.

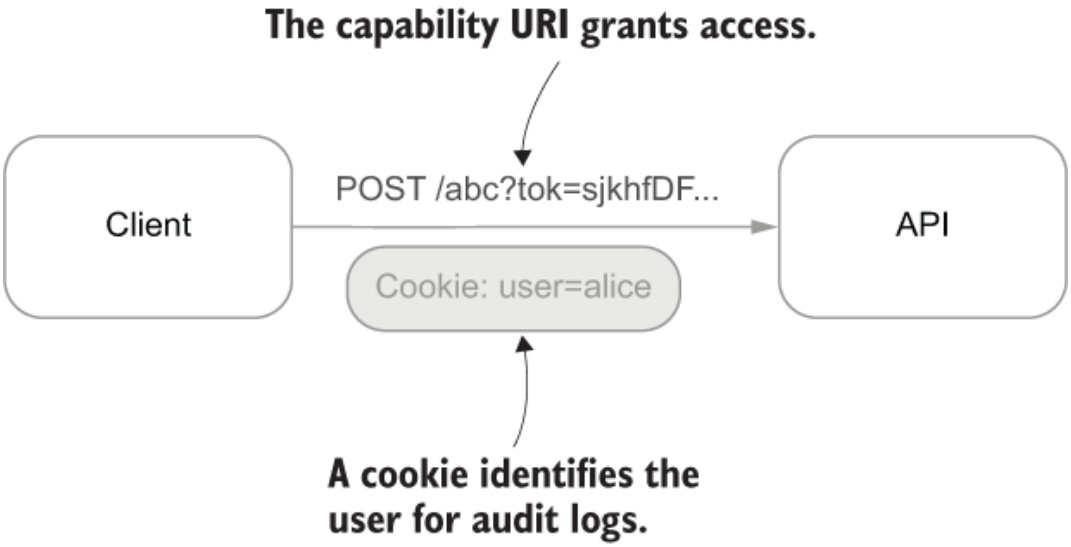

Figure 9.4 By combining capability URIs with a traditional authentication mechanism such as cookies, the API can enforce access using capabilities while authenticating identity claims using the cookie. The same capability URI can be shared between users, but the API is still able to positively identify each of them.

When developing a REST API, the second option is often attractive because you can reuse traditional cookie-based authentication technologies

such as a centralized OpenID Connect identity provider (chapter 7). This is the approach taken in the Natter API, where the permissions for an API call come from a capability URI, but some API calls need additional user authentication using a traditional mechanism such as HTTP Basic authentication or an authentication token or cookie.

To switch back to using cookies for authentication, open the Main.java file in your editor and find the lines that create the `TokenController` object. Change the `tokenStore` variable to use the `CookieTokenStore` that you developed back in chapter 4:

```
SecureTokenStore tokenStore = new CookieTokenStore();
var tokenController = new TokenController(tokenStore);
```

9.2.6 Hardening capability URIs

You may wonder if you can do away with the anti-CSRF token now that you're using capabilities for access control, which are immune to CSRF. This would be a mistake, because an attacker that has a genuine capability to access the API can still use a CSRF attack to make their requests appear to come from a different user. The authority to access the API comes from the attacker's capability URI, but the identity of the user comes from the cookie. If you keep the existing anti-CSRF token though, clients are required to send three credentials on every request:

- The cookie identifying the user
- The anti-CSRF token
- The capability token authorizing the specific request

This is a bit excessive. At the same time, the capability tokens are vulnerable to being stolen. For example, if a capability URI meant for a moderator is stolen, then it can be used by anybody to delete messages. You can solve both problems by tying the capability tokens to an authenticated user and preventing them being used by anybody else. This removes one of the benefits of capability URIs--that they are easy to share--but improves the overall security:

- If a capability token is stolen, it can't be used without a valid login cookie for the user. If the cookie is set with the HttpOnly and Secure flags, then it becomes much harder to steal.

- You can now remove the separate anti-CSRF token because each capability URI effectively acts as an anti-CSRF token. The cookie can't be used without the capability and the capability can't be used without the cookie.

Listing 9.9 shows how to associate a capability token with an authenticated user by populating the `username` attribute of the token that you previously left blank. Open the CapabilityController.java file in your editor and add the highlighted lines of code.

Listing 9.9 Linking a capability with a user

```
public URI createUri(Request request, String path, String perms,
                     Duration expiryDuration) {
    var subject = (String) request.attribute("subject");                ❶
    var token = new Token(now().plus(expiryDuration), subject);         ❷
    token.attributes.put("path", path);
    token.attributes.put("perms", perms);

    var tokenId = tokenStore.create(request, token);

    var uri = URI.create(request.uri());
    return uri.resolve(path + "?access_token=" + tokenId);
}
```

❶ Look up the authenticated user.

❷ Associate the capability with the user.

You can then adjust the `lookupPermissions` method in the same file to return no permissions if the username associated with the capability token doesn't match the authenticated user, as shown in listing 9.10. This ensures that the capability can't be used without an associated session for the user and that the session cookie can only be used when it matches the capability token, effectively preventing CSRF attacks too.

Listing 9.10 Verifying the user

```
public void lookupPermissions(Request request, Response response) {
    var tokenId = request.queryParams("access_token");
    if (tokenId == null) { return; }

    tokenStore.read(request, tokenId).ifPresent(token -> {
```

```
        if (!Objects.equals(token.username,                    (1)
                request.attribute("subject"))) {               (1)
            return;                                            (1)
        }                                                      (1)

        var tokenPath = token.attributes.get("path");
        if (Objects.equals(tokenPath, request.pathInfo())) {
            request.attribute("perms",
                    token.attributes.get("perms"));
        }
    });
}
```

(1) If the authenticated user doesn't match the capability, it returns no permissions.

You can now delete the code that checks the anti-CSRF token in the `CookieTokenStore` if you wish and rely on the capability code to protect against CSRF. Refer to chapter 4 to see how the original version looked before CSRF protection was added. You'll also need to adjust the `TokenController.validateToken` method to not reject a request that doesn't have an anti-CSRF token. If you get stuck, check out chapter09-end of the GitHub repository accompanying the book, which has all the required changes.

SHARING ACCESS

Because capability URIs are now tied to individual users, you need a new mechanism to share access to social spaces and individual messages. Listing 9.11 shows a new operation to allow a user to exchange one of their own capability URIs for one for a different user, with an option to specify a reduced set of permissions. The method reads a capability URI from the input and looks up the associated token. If the URI matches the token and the requested permissions are a subset of the permissions granted by the original capability URI, then the method creates a new capability token with the new permissions and user and returns the requested URI. This new URI can then be safely shared with the intended user. Open the CapabilityController.java file and add the new method.

Listing 9.11 Sharing capability URIs

```
public JSONObject share(Request request, Response response) {
    var json = new JSONObject(request.body());
```

```
    var capUri = URI.create(json.getString("uri"));                    ❶
    var path = capUri.getPath();                                       ❶
    var query = capUri.getQuery();                                     ❶
    var tokenId = query.substring(query.indexOf('=') + 1);             ❶

    var token = tokenStore.read(request, tokenId).orElseThrow();       ❷
    if (!Objects.equals(token.attributes.get("path"), path)) {         ❷
        throw new IllegalArgumentException("incorrect path");          ❷
    }                                                                  ❷

    var tokenPerms = token.attributes.get("perms");                    ❸
    var perms = json.optString("perms", tokenPerms);                   ❸
    if (!tokenPerms.contains(perms)) {                                 ❸
        Spark.halt(403);                                               ❸
    }                                                                  ❸
    var user = json.getString("user");
    var newToken = new Token(token.expiry, user);                      ❹
    newToken.attributes.put("path", path);                             ❹
    newToken.attributes.put("perms", perms);                           ❹
    var newTokenId = tokenStore.create(request, newToken);             ❹

    var uri = URI.create(request.uri());                               ❺
    var newCapUri = uri.resolve(path + "?access_token="                ❺
            + newTokenId);                                             ❺
    return new JSONObject()                                            ❺
            .put("uri", newCapUri);                                    ❺
}
```

❶ Parse the original capability URI and extract the token.

❷ Look up the token and check that it matches the URI.

❸ Check that the requested permissions are a subset of the token permissions.

❹ Create and store the new capability token.

❺ Return the requested capability URI.

You can now add a new route to the `Main` class to expose this new operation. Open the Main.java file and add the following line to the `main` method:

```
post("/capabilities", capController::share);
```

You can now call this endpoint to exchange a privileged capability URI, such as the messages-rwd URI returned from creating a space, as in the following example:

```
curl -H 'Content-Type: application/json' \
  -d '{"uri":"/spaces/1/messages?access_token=
➥ 0ed8-IohfPQUX486d0kr03W8Ec8", "user":"demo2", "perms":"r"}' \
  https://localhost:4567/share
{"uri":"/spaces/1/messages?access_token=
➥ 1YQqZdNAIce5AB_Z8J7ClMrnx68"}
```

The new capability URI in the response can only be used by the demo2 user and provides only read permission on the space. You can use this facility to build resource sharing for your APIs. For example, if a user directly shares a capability URI of their own with another user, rather than denying access completely you could allow them to request access. This is what happens in Google Docs if you follow a link to a document that you don't have access to. The owner of the document can then approve access. In Google Docs this is done by adding an entry to an access control list (chapter 3) associated with each document, but with capabilities, the owner could generate a capability URI instead that is then emailed to the recipient.

9.3 Macaroons: Tokens with caveats

Capabilities allow users to easily share fine-grained access to their resources with other users. If a Natter user wants to share one of their messages with somebody who doesn't have a Natter account, they can easily do this by creating a read-only capability URI for that specific message. The other user will be able to read only that one message and won't get access to any other messages or the ability to post messages themselves.

Sometimes the granularity of capability URIs doesn't match up with how users want to share resources. For example, suppose that you want to share read-only access to a snapshot of the conversations since yesterday in a social space. It's unlikely that the API will always supply a capability URI that exactly matches the user's wishes; the `createSpace` action already returns four URIs, and none of them quite fit the bill.

Macaroons provide a solution to this problem by allowing anybody to append caveats to a capability that restrict how it can be used. Macaroons were invented by a team of academic and Google researchers in a paper published in 2014 (**https://ai.google/ research/pubs/pub41892**).

DEFINITION A macaroon is a type of cryptographic token that can be used to represent capabilities and other authorization grants. Anybody can append new caveats to a macaroon that restrict how it can be used.

To address our example, the user could append the following caveats to their capability to create a new capability that allows only read access to messages since lunchtime yesterday:

```
method = GET
since >= 2019-10-12T12:00:00Z
```

Unlike the share method that you added in section 9.2.6, macaroon caveats can express general conditions like these. The other benefit of macaroons is that anyone can append a caveat to a macaroon using a macaroon library, without needing to call an API endpoint or have access to any secret keys. Once the caveat has been added it can't be removed.

Macaroons use HMAC-SHA256 tags to protect the integrity of the token and any caveats just like the `HmacTokenStore` you developed in chapter 5. To allow anybody to append caveats to a macaroon, even if they don't have the key, macaroons use an interesting property of HMAC: the authentication tag output from HMAC can itself be used as a key to sign a new message with HMAC. To append a caveat to a macaroon, you use the old authentication tag as the key to compute a new HMAC-SHA256 tag over the caveat, as shown in figure 9.5. You then throw away the old authentication tag and append the caveat and the new tag to the macaroon. Because it's infeasible to reverse HMAC to recover the old tag, nobody can remove caveats that have been added unless they have the original key.

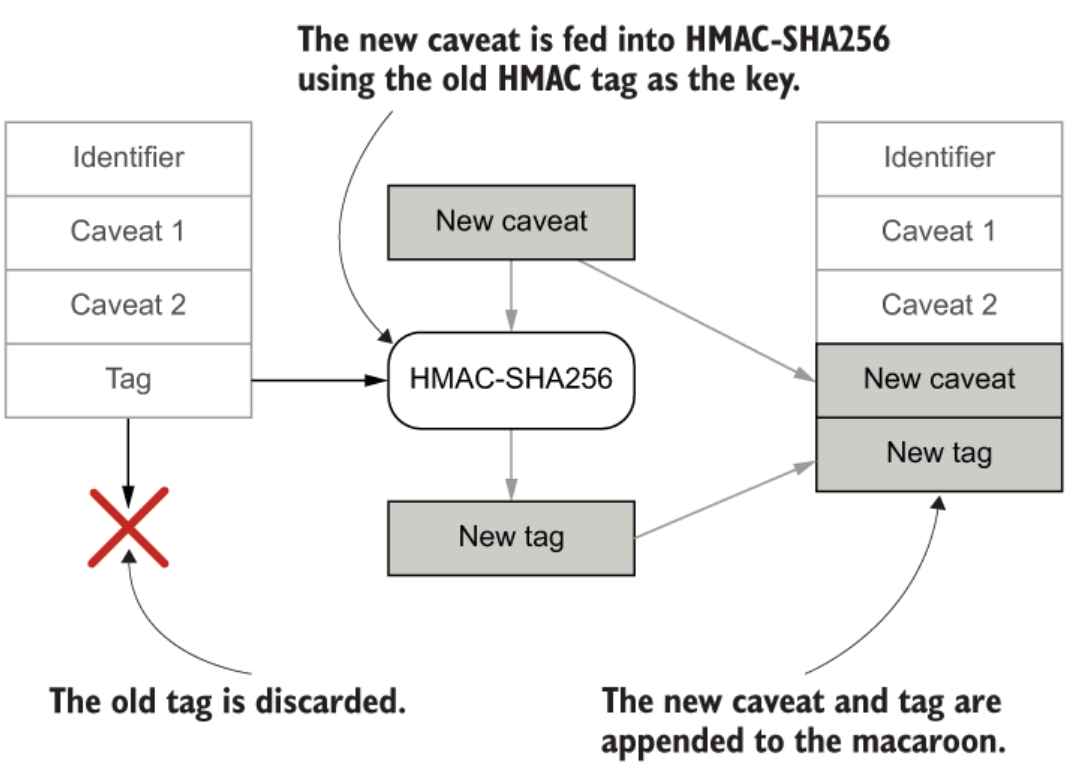

Figure 9.5 To append a new caveat to a macaroon, you use the old HMAC tag as the key to authenticate the new caveat. You then throw away the old tag and append the new caveat and tag. Because nobody can reverse HMAC to calculate the old tag, they cannot remove the caveat.

WARNING Because anybody can add a caveat to a macaroon, it is important that they are used only to restrict how a token is used. You should never trust any claims in a caveat or grant additional access based on their contents.

When the macaroon is presented back to the API, it can use the original HMAC key to reconstruct the original tag and all the caveat tags and check if it comes up with the same signature value at the end of the chain of caveats. Listing 9.12 shows an example of how to verify an HMAC chain just like that used by macaroons.

First initialize a `javax.crypto.Mac` object with the API's authentication key (see chapter 5 for how to generate this) and then compute an initial tag over the macaroon unique identifier. You then loop through each caveat in the chain and compute a new HMAC tag over the caveat, using the old tag as the key.5 Finally, you compare the computed tag with the tag that was supplied with the macaroon using a constant-time equality function. Listing 9.14 is just to demonstrate how it works; you'll use a real macaroon library in the Natter API so you don't need to implement this method.

Listing 9.12 Verifying the HMAC chain

```
private boolean verify(String id, List<String> caveats, byte[] tag)
        throws Exception {
    var hmac = Mac.getInstance("HmacSHA256");                    ❶
    hmac.init(macKey);                                           ❶
    var computed = hmac.doFinal(id.getBytes(UTF_8));             ❷
    for (var caveat : caveats) {                                 ❸
        hmac.init(new SecretKeySpec(computed, "HmacSHA256"));    ❸
        computed = hmac.doFinal(caveat.getBytes(UTF_8));         ❸
    }
    return MessageDigest.isEqual(tag, computed);                 ❹
}
```

❶ Initialize HMAC-SHA256 with the authentication key.

❷ Compute an initial tag over the macaroon identifier.

❸ Compute a new tag for each caveat using the old tag as the key.

❹ Compare the tags with a constant-time equality function.

After the HMAC tag has been verified, the API then needs to check that the caveats are satisfied. There's no standard set of caveats that APIs support, so like OAuth2 scopes it's up to the API designer to decide what to support. There are two broad categories of caveats supported by macaroon libraries:

- First-party caveats are restrictions that can be easily verified by the API at the point of use, such as restricting the times of day at which the token can be used. First-party caveats are discussed in more detail in section 9.3.3.
- Third-party caveats are restrictions which require the client to obtain a proof from a third-party service, such as proof that the user is an employee of a particular company or that they are over 18. Third-party caveats are discussed in section 9.3.4.

9.3.1 Contextual caveats

A significant advantage of macaroons over other token forms is that they allow the client to attach contextual caveats just before the macaroon is used. For example, a client that is about to send a macaroon to an API over an untrustworthy communication channel can attach a first-party caveat limiting it to only be valid for HTTP PUT requests to that specific URI for the next 5 seconds. That way, if the macaroon is stolen, then the

damage is limited because the attacker can only use the token in very restricted circumstances. Because the client can keep a copy of the original unrestricted macaroon, their own ability to use the token is not limited in the same way.

DEFINITION A contextual caveat is a caveat that is added by a client just before use. Contextual caveats allow the authority of a token to be restricted before sending it over an insecure channel or to an untrusted API, limiting the damage that might occur if the token is stolen.

The ability to add contextual caveats makes macaroons one of the most important recent developments in API security. Macaroons can be used with any token-based authentication and even OAuth2 access tokens if your authorization server supports them.[6] On the other hand, there is no formal specification of macaroons and awareness and adoption of the format is still quite limited, so they are not as widely supported as JWTs (chapter 6).

9.3.2 A macaroon token store

To use macaroons in the Natter API, you can use the open source jmacaroons library (**https://github.com/nitram509/jmacaroons**). Open the pom.xml file in your editor and add the following lines to the dependencies section:

```
<dependency>
  <groupId>com.github.nitram509</groupId>
  <artifactId>jmacaroons</artifactId>
  <version>0.4.1</version>
</dependency>
```

You can now build a new token store implementation using macaroons as shown in listing 9.13. To create a macaroon, you'll first use another `TokenStore` implementation to generate the macaroon identifier. You can use any of the existing stores, but to keep the tokens compact you'll use the `DatabaseTokenStore` in these examples. You could also use the `JsonTokenStore`, in which case the macaroon HMAC tag also protects it against tampering.

You then create the macaroon using the `MacaroonsBuilder.create()` method, passing in the identifier and the HMAC key. An odd quirk of the macaroon API means you have to pass the raw bytes of the key using

`macKey.getEncoded()`. You can also give an optional hint for where the macaroon is intended to be used. Because you'll be using these with capability URIs that already include the full location, you can leave that field blank to save space. You can then use the `macaroon.serialize()` method to convert the macaroon into a URL-safe base64 string format. In the same Natter API project you've been using so far, navigate to src/main/java/com/manning/ apisecurityinaction/token and create a new file called MacaroonTokenStore.java. Copy the contents of listing 9.13 into the file and save it.

WARNING The location hint is not included in the authentication tag and is intended only as a hint to the client. Its value shouldn't be trusted because it can be tampered with.

Listing 9.13 The MacaroonTokenStore

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

import java.security.Key;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Optional;

import com.github.nitram509.jmacaroons.*;
import com.github.nitram509.jmacaroons.verifier.*;
import spark.Request;

public class MacaroonTokenStore implements SecureTokenStore {
    private final TokenStore delegate;
    private final Key macKey;
    private MacaroonTokenStore(TokenStore delegate, Key macKey) {
        this.delegate = delegate;
        this.macKey = macKey;
    }

    @Override
    public String create(Request request, Token token) {
        var identifier = delegate.create(request, token);     (1)
        var macaroon = MacaroonsBuilder.create("",            (2)
                macKey.getEncoded(), identifier);             (2)
        return macaroon.serialize();                          (3)
    }
}
```

❶ Use another token store to create a unique identifier for this macaroon.

❷ Create the macaroon with a location hint, the identifier, and the authentication key.

❸ Return the serialized URL-safe string form of the macaroon.

Like the `HmacTokenStore` from chapter 4, the macaroon token store only provides authentication of tokens and not confidentiality unless the underlying store already provides that. Just as you did in chapter 5, you can create two static factory methods that return a correctly typed store depending on the underlying token store:

- If the underlying token store is a `ConfidentialTokenStore`, then it returns a `SecureTokenStore` because the resulting store provides both confidentiality and authenticity of tokens.
- Otherwise, it returns an `AuthenticatedTokenStore` to make clear that confidentiality is not guaranteed.

These factory methods are shown in listing 9.14 and are very similar to the ones you created in chapter 5, so open the MacaroonTokenStore.java file again and add these new methods.

Listing 9.14 Factory methods

```
public static SecureTokenStore wrap(                          ❶
        ConfidentialTokenStore tokenStore, Key macKey) {      ❶
    return new MacaroonTokenStore(tokenStore, macKey);        ❶
}                                                             ❶

public static AuthenticatedTokenStore wrap(                   ❷
        TokenStore tokenStore, Key macKey) {                  ❷
    return new MacaroonTokenStore(tokenStore, macKey);        ❷
}                                                             ❷
```

❶ If the underlying store provides confidentiality of token data, then return a SecureTokenStore.

❷ Otherwise, return an AuthenticatedTokenStore.

To verify a macaroon, you deserialize and validate the macaroon using a `MacaroonsVerifier`, which will verify the HMAC tag and check any caveats. If the macaroon is valid, then you can look up the identifier in the delegate token store. To revoke a macaroon, you simply deserialize and revoke the identifier. In most cases, you shouldn't check the caveats on the token when it is being revoked, because if somebody has gained access to your token, the least malicious thing they can do with it is revoke it! However, in some cases, malicious revocation might be a real threat, in which case you could verify the caveats to reduce the risk of this occurring. Listing 9.15 shows the operations to read and revoke a macaroon token. Open the MacaroonTokenStore .java file again and add the new methods.

Listing 9.15 Reading a macaroon token

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    var macaroon = MacaroonsBuilder.deserialize(tokenId);          (1)
    var verifier = new MacaroonsVerifier(macaroon);                (1)
    if (verifier.isValid(macKey.getEncoded())) {                   (1)
        return delegate.read(request, macaroon.identifier);        (2)
    }
    return Optional.empty();
}

@Override
public void revoke(Request request, String tokenId) {
    var macaroon = MacaroonsBuilder.deserialize(tokenId);
    delegate.revoke(request, macaroon.identifier);                 (3)
}
```

(1) Deserialize and validate the macaroon signature and caveats.

(2) If the macaroon is valid, then look up the identifier in the delegate token store.

(3) To revoke a macaroon, revoke the identifier in the delegate store.

WIRING IT UP

You can now wire up the `CapabilityController` to use the new token store for capability tokens. Open the Main.java file in your editor and find the lines that construct the `CapabilityController`. Update the file to use

the `MacaroonTokenStore` instead. You may need to first move the code that reads the `macKey` from the keystore (see chapter 6) from later in the file. The code should look as follows, with the new part highlighted in bold:

```
var keyPassword = System.getProperty("keystore.password",
        "changeit").toCharArray();
var keyStore = KeyStore.getInstance("PKCS12");
keyStore.load(new FileInputStream("keystore.p12"),
        keyPassword);
var macKey = keyStore.getKey("hmac-key", keyPassword);
var encKey = keyStore.getKey("aes-key", keyPassword);

var capController = new CapabilityController(
        MacaroonTokenStore.wrap(
             new DatabaseTokenStore(database), macKey));
```

If you now use the API to create a new space, you'll see the macaroon tokens being used in the capability URIs returned from the API call. You can copy and paste those tokens into the debugger at **http://macaroons.io** to see the component parts.

CAUTION You should not paste tokens from a production system into any website. At the time of writing, macaroons.io doesn't even support SSL.

As currently written, the macaroon token store works very much like the existing HMAC token store. In the next sections, you'll implement support for caveats to take full advantage of the new token format.

9.3.3 First-party caveats

The simplest caveats are first-party caveats, which can be verified by the API purely based on the API request and the current environment. These caveats are represented as strings and there is no standard format. The only commonly implemented first-party caveat is to set an expiry time for the macaroon using the syntax:

```
time < 2019-10-12T12:00:00Z
```

You can think of this caveat as being like the expiry (exp) claim in a JWT (chapter 6). The tokens issued by the Natter API already have an expiry

time, but a client might want to create a copy of their token with a more restricted expiry time as discussed in section 9.3.1 on contextual caveats.

To verify any expiry time caveats, you can use a `TimestampCaveatVerifier` that comes with the jmacaroons library as shown in listing 9.16. The macaroons library will try to match each caveat to a verifier that is able to satisfy it. In this case, the verifier checks that the current time is before the expiry time specified in the caveat. If the verification fails, or if the library is not able to find a verifier that matches a caveat, then the macaroon is rejected. This means that the API must explicitly register verifiers for all types of caveats that it supports. Trying to add a caveat that the API doesn't support will prevent the macaroon from being used. Open the MacaroonTokenStore.java file in your editor again and update the `read` method to verify expiry caveats as shown in the listing.

Listing 9.16 Verifying the expiry timestamp

```
@Override
public Optional<Token> read(Request request, String tokenId) {
    var macaroon = MacaroonsBuilder.deserialize(tokenId);
    var verifier = new MacaroonsVerifier(macaroon);
    verifier.satisfyGeneral(new TimestampCaveatVerifier());          (1)
    if (verifier.isValid(macKey.getEncoded())) {
        return delegate.read(request, macaroon.identifier);
    }
    return Optional.empty();
}
```

(1) Add a TimestampCaveatVerifier to satisfy the expiry caveat.

You can also add your own caveat verifiers using two methods. The simplest is the `satisfyExact` method, which will satisfy caveats that exactly match the given string. For example, you can allow a client to restrict a macaroon to a single type of HTTP method by adding the line:

```
verifier.satisfyExact("method = " + request.requestMethod());
```

to the `read` method. This ensures that a macaroon with the caveat `method = GET` can only be used on HTTP GET requests, effectively making it read-only. Add that line to the `read` method now.

A more general approach is to implement the `GeneralCaveatVerifier` interface, which allows you to implement arbitrary conditions to satisfy a caveat. Listing 9.17 shows an example verifier to check that the `since` query parameter to the `findMessages` method is after a certain time, allowing you to restrict a client to only view messages since yesterday. The class parses the caveat and the parameter as `Instant` objects and then checks that the request is not trying to read messages older than the caveat using the `isAfter` method. Open the MacaroonTokenStore.java file again and add the contents of listing 9.17 as an inner class.

Listing 9.17 A custom caveat verifier

```
private static class SinceVerifier implements GeneralCaveatVerifier {
    private final Request request;

    private SinceVerifier(Request request) {
        this.request = request;
    }

    @Override
    public boolean verifyCaveat(String caveat) {
        if (caveat.startsWith("since > ")) {
            var minSince = Instant.parse(caveat.substring(8));

            var reqSince = Instant.now().minus(1, ChronoUnit.DAYS);
            if (request.queryParams("since") != null) {
                reqSince = Instant.parse(request.queryParams("since"));
            }
            return reqSince.isAfter(minSince);
        }

        return false;
    }
}
```

❶ Check the caveat matches and parse the restriction.

❷ Determine the “since” parameter value on the request.

❸ Satisfy the caveat if the request is after the earliest message restriction.

❹ Reject all other caveats.

You can then add the new verifier to the `read` method by adding the following line

```
    verifier.satisfyGeneral(new SinceVerifier(request));
```

next to the lines adding the other caveat verifiers. The finished code to construct the verifier should look as follows:

```
var verifier = new MacaroonsVerifier(macaroon);
verifier.satisfyGeneral(new TimestampCaveatVerifier());
verifier.satisfyExact("method = " + request.requestMethod());
verifier.satisfyGeneral(new SinceVerifier(request));
```

ADDING CAVEATS

To add a caveat to a macaroon, you can parse it using the `MacaroonsBuilder` class and then use the `add_first_party_caveat` method to append caveats, as shown in listing 9.18. The listing is a standalone command-line program for adding caveats to a macaroon. It first parses the macaroon, which is passed as the first argument to the program, and then loops through any remaining arguments treating them as caveats. Finally, it prints out the resulting macaroon as a string again. Navigate to the src/main/ java/com/manning/apisecurityinaction folder and create a new file named CaveatAppender.java and type in the contents of the listing.

Listing 9.18 Appending caveats

```
package com.manning.apisecurityinaction;

import com.github.nitram509.jmacaroons.MacaroonsBuilder;
import static com.github.nitram509.jmacaroons.MacaroonsBuilder.deserializ

public class CaveatAppender {
    public static void main(String... args) {
        var builder = new MacaroonsBuilder(deserialize(args[0]));     1
        for (int i = 1; i < args.length; ++i) {                       2
            var caveat = args[i];                                     2
            builder.add_first_party_caveat(caveat);                   2
        }                                                             2
        System.out.println(builder.getMacaroon().serialize());        3
```

```
    }
}
```

❶ Parse the macaroon and create a MacaroonsBuilder.

❷ Add each caveat to the macaroon.

❸ Serialize the macaroon back into a string.

IMPORTANT Compared to the server, the client needs only a few lines of code to append caveats and doesn't need to store any secret keys.

To test out the program, use the Natter API to create a new social space and receive a capability URI with a macaroon token. In this example, I've used the `jq` and `cut` utilities to extract the macaroon token, but you can manually copy and paste if you prefer:

```
MAC=$(curl -u demo:changeit -H 'Content-Type: application/json' \
  -d '{"owner":"demo","name":"test"}' \
  https://localhost:4567/spaces | jq -r '["messages-rw"]' \
  | cut -d= -f2)
```

You can then append a caveat, for example setting the expiry time a minute or so into the future:

```
NEWMAC=$(mvn -q exec:java \
  -Dexec.mainClass= com.manning.apisecurityinaction.CaveatAppender \
  -Dexec.args="$MAC 'time < 2020-08-03T12:05:00Z'")
```

You can then use this new macaroon to read any messages in the space until it expires:

```
curl -u demo:changeit -i \
  "https://localhost:4567/spaces/1/messages?access_token=$NEWMAC"
```

After the new time limit expires, the request will return a 403 Forbidden error, but the original token will still work (just change `$NEWMAC` to `$MAC` in the query to test this). This demonstrates the core advantage of macaroons: once you've configured the server it's very easy (and fast) for a client to append contextual caveats that restrict the use of a token, pro-

tecting those tokens in case of compromise. A JavaScript client running in a web browser can use a JavaScript macaroon library to easily append caveats every time it uses a token with just a few lines of code.

9.3.4 Third-party caveats

First-party caveats provide considerable flexibility and security improvements over traditional tokens on their own, but macaroons also allow third-party caveats that are verified by an external service. Rather than the API verifying a third-party caveat directly, the client instead must contact the third-party service itself and obtain a discharge macaroon that proves that the condition is satisfied. The two macaroons are cryptographically tied together so that the API can verify that the condition is satisfied without talking directly to the third-party service.

DEFINITION A discharge macaroon is obtained by a client from a third-party service to prove that a third-party caveat is satisfied. A third-party service is any service that isn't the client or the server it is trying to access. The discharge macaroon is cryptographically bound to the original macaroon such that the API can ensure that the condition has been satisfied without talking directly to the third-party service.

Third-party caveats provide the basis for loosely coupled decentralized authorization and provide some interesting properties:

- The API doesn't need to directly communicate with the third-party service.
- No details about the query being answered by the third-party service are disclosed to the client. This can be important if the query contains personal information about a user.
- The discharge macaroon proves that the caveat is satisfied without revealing any details to the client or the API.
- Because the discharge macaroon is itself a macaroon, the third-party service can attach additional caveats to it that the client must satisfy before it is granted access, including further third-party caveats.

For example, a client might be issued with a long-term macaroon token to performing banking activities on behalf of a user, such as initiating payments from their account. As well as first-party caveats restricting how much the client can transfer in a single transaction, the bank might attach a third-party caveat that requires the client to obtain authorization for each payment from a transaction authorization service. The transaction

authorization service checks the details of the transaction and potentially confirms the transaction directly with the user before issuing a discharge macaroon tied to that one transaction. This pattern of having a single long-lived token providing general access, but then requiring short-lived discharge macaroons to authorize specific transactions is a perfect use case for third-party caveats.

CREATING THIRD-PARTY CAVEATS

Unlike a first-party caveat, which is a simple string, a third-party caveat has three components:

- A location hint telling the client where to locate the third-party service.
- A unique unguessable secret string, which will be used to derive a new HMAC key that the third-party service will use to sign the discharge macaroon.
- An identifier for the caveat that the third-party can use to identify the query. This identifier is public and so shouldn't reveal the secret.

To add a third-party caveat to a macaroon, you use the `add_third_party_caveat` method on the `MacaroonsBuilder` object:

```
    macaroon = MacaroonsBuilder.modify(macaroon)              ❶
        .add_third_party_caveat("https://auth.example.com",   ❷
            secret, caveatId)                                 ❷
        .getMacaroon();
```

❶ Modify an existing macaroon to add a caveat.

❷ Add the third-party caveat.

The unguessable secret should be generated with high entropy, such as a 256-bit value from a `SecureRandom`:

```
var key = new byte[32];
new SecureRandom().nextBytes(key);
var secret = Base64.getEncoder().encodeToString(key);
```

When you add a third-party caveat to a macaroon, this secret is encrypted so that only the API that verifies the macaroon will be able to de-

crypt it. The party appending the caveat also needs to communicate the secret and the query to be verified to the third-party service. There are two ways to accomplish this, with different trade-offs:

- The caveat appender can encode the query and the secret into a message and encrypt it using a public key from the third-party service. The encrypted value is then used as the identifier for the third-party caveat. The third-party can then decrypt the identifier to discover the query and secret. The advantage of this approach is that the API doesn't need to directly talk to the third-party service, but the encrypted identifier may be quite large.
- Alternatively, the caveat appender can contact the third-party service directly (via a REST API, for example) to register the caveat and secret. The third-party service would then store these and return a random value (known as a ticket) that can be used as the caveat identifier. When the client presents the identifier to the third-party it can look up the query and secret in its local storage based on the ticket. This solution is likely to produce smaller identifiers, but at the cost of additional network requests and storage at the third-party service.

There's currently no standard for either of these two options describing what the API for registering a caveat would look like for the second option, or which public key encryption algorithm and message format would be used for the first. There is also no standard describing how a client presents the caveat identifier to the third-party service. In practice, this limits the use of third-party caveats because client developers need to know how to integrate with each service individually, so they are typically only used within a closed ecosystem.

Pop quiz

6. Which of the following apply to a first-party caveat? Select all that apply.
    1. It's a simple string.
    2. It's satisfied using a discharge macaroon.
    3. It requires the client to contact another service.
    4. It can be checked at the point of use by the API.
    5. It has an identifier, a secret string, and a location hint.
7. Which of the following apply to a third-party caveat? Select all that apply.
    1. It's a simple string.
    2. It's satisfied using a discharge macaroon.

3. It requires the client to contact another service.
4. It can be checked at the point of use by the API.
5. It has an identifier, a secret string, and a location hint.

The answers are at the end of the chapter.

Answers to pop quiz questions

1. a, e, f, or g are all acceptable places to encode the token. The others are likely to interfere with the functioning of the URI.
2. c, d, and e.
3. b and e would prevent tokens filling up the database. Using a more scalable database is likely to just delay this (and increase your costs).
4. e. Without returning links, a client has no way to create URIs to other resources.
5. d. If the server redirects, the browser will copy the fragment to the new URL unless a new one is specified. This can leak the token to other servers. For example, if you redirect the user to an external login service, the fragment component is not sent to the server and is not included in Referer headers.
6. a and d.
7. b, c, and e.

Summary

- Capability URIs can be used to provide fine-grained access to individual resources via your API. A capability URI combines an identifier for a resource along with a set of permissions to access that resource.
- As an alternative to identity-based access control, capabilities avoid ambient authority that can lead to confused deputy attacks and embrace POLA.
- There are many ways to form capability URIs that have different trade-offs. The simplest forms encode a random token into the URI path or query parameters. More secure variants encode the token into the fragment or userinfo components but come at a cost of increased complexity for clients.
- Tying a capability URI to a user session increases the security of both, because it reduces the risk of capability tokens being stolen and can be used to prevent CSRF attacks. This makes it harder to share capability URIs.

- Macaroons allow anybody to restrict a capability by appending caveats that can be cryptographically verified and enforced by an API. Contextual caveats can be appended just before a macaroon is used to secure a token against misuse.
- First-party caveats encode simple conditions that can be checked locally by an API, such as restricted the time of day at which a token can be used. Third-party caveats require the client to obtain a discharge macaroon from an external service proving that it satisfies a condition, such that the user is an employee of a certain company or is over 18 years old.

---

1. This example is taken from "Paradigm Regained: Abstraction Mechanisms for Access Control." See **http:// mng.bz/Mog7**.

2. There are proposals to make OAuth work better for these kinds of transactional one-off operations, such as https:/ /oauth.xyz, but these largely still require the app to know what resource it wants to access before it begins the flow.

3. You can get the project from **https://github.com/NeilMadden/apisecurityinaction** if you haven't worked through chapter 8. Check out branch chapter09.

4. In this chapter, you'll return links as URIs within normal JSON fields. There are standard ways of representing links in JSON, such as JSON-LD (**https://json-ld.org**), but I won't cover those in this book.

5. If you are a functional programming enthusiast, then this can be elegantly written as a left-fold or reduce operation.

6. My employer, ForgeRock, has added experimental support for macaroons to their authorization server software.