

## 2. Two Quick Examples

This chapter provides a two quick examples of how JAAS can be used to provide authentication and authorization. The examples are very simple, using the flat-file based Policy implementation provided by Sun Microsystems. Because both examples are simple you can get your feet wet enough to understand the basic concepts and prepare for the more in-depth discussion that follows.

### 2.1. A Simple, Cheesy Example

This example illustrates using of a JAAS policy file to grant permissions to the executing code. Our application will check to see if it's been granted permission to write to a file called `cheese.txt`. The first time we run the application, permission will be denied because the permission has been commented out in the policy file. Then, we'll uncomment the permission grant in the policy file, giving the code permission to write to the file. Finally, with the correct permission granted, the application will be able to write to `cheese.txt`.

#### 2.1.1. The “Application”

Here is the application code:

```
package chp02;

import java.io.File;
import java.io.IOException;

public class Chp02aMain {

    public static void main(String[] args) throws IOException {
        File file = new File("build/conf/cheese.txt");
        try {
            file.canWrite();
            System.out.println("We can write to cheese.txt");
        } catch (SecurityException e) {
            System.out.println("We can NOT write to cheese.txt");
        }
    }
}
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

The above code simply checks to see if we have been granted permission to write to the file `build/conf/cheese.txt`. When we run the application for the first time, we'll turn on the Java security manager by specifying in the system property `java.security.manager`. By default, the security manager is very restrictive in what permissions are granted: only the bare minimum needed to execute the program and check some basic system properties are granted. The default set of permissions does not include access to just any file, such as `cheese.txt`.

### 2.1.2 Running Without Permission

To run the program for the first time, execute the command `ant run-chp02a`. This Ant command will do the following:

1. Compile the code.
2. "Build" the required configuration files, such as the policy.
3. Execute the command to run the application.

The command that runs the application, which the Ant task executes on your behalf, is:

```
java -cp build/java
-Djava.security.manager
-Djava.security.policy=build/conf/chp02a.policy
chp02.Chp02aMain
```

This command turns on the Java security manager, and specified the policy file to use. The security manager performs permission checks as needed, while the policy file describes the permissions that are granted to executing code and users. We'll learn much more about the security manager and the policy file in upcoming sections and chapters.

When this command is run for the first time, you'll see the following output:

```
run-chp02a:
[java] We can NOT write to cheese.txt
```

This output indicates that the application has not been granted permission to write to the `cheese` file. A quick look at the policy file will confirm this:

```
grant
{
//  permission java.io.FilePermission "build/conf/cheese.txt", "write";
};
```

We'll go over the policy file format later, but for now all you need to notice is that permission to write to the `cheese` file has been commented out with the leading `//`.

### 2.1.3 Running with Permission

Before we execute the application again, uncomment the `grant` by opening the file `src/conf/chp02a.policy`, and deleting the leading double slashes. After doing this, when we run the ant command `ant run-chp02a` again, we'll see the below output:



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
run-chp02a:  
[java] We can write to cheese.txt
```

With the permission grant uncommented in the policy file, our code now has been granted permission to write to the file `cheese.txt`.

## 2.2 *User Based Authentication and Authorization*

In the second example, we're interested in protecting JAAS itself from being hacked by users of the application it's protecting. For simplicities sake, the example won't protect JAAS from all possible attempts to hack it. Rather, the example will focus on simply protecting access to the `Policy` file. The `Policy` file specifies which permissions logged in users, and the application in general, are granted. Any user that can modify the `Policy` file can potentially grant themselves all permissions, compromising the security of the system. Thus, restricting access to the `Policy` file is very important.

The example system will have two types of users: normal users and systems administrators:

- Normal users cannot access the `Policy` file.
- Only system administrators will be allowed to modify the `Policy` file. Normal users will not.

A `Principal` will represent each of these users. The `Policy` file will declare which permissions each `Principal`, and thus user, is given. The “application” will be represented by a small class with a `main` method. The application will log each type of user in, and then attempt to access the `Policy` file to demonstrate how access is both checked and restricted with JAAS.

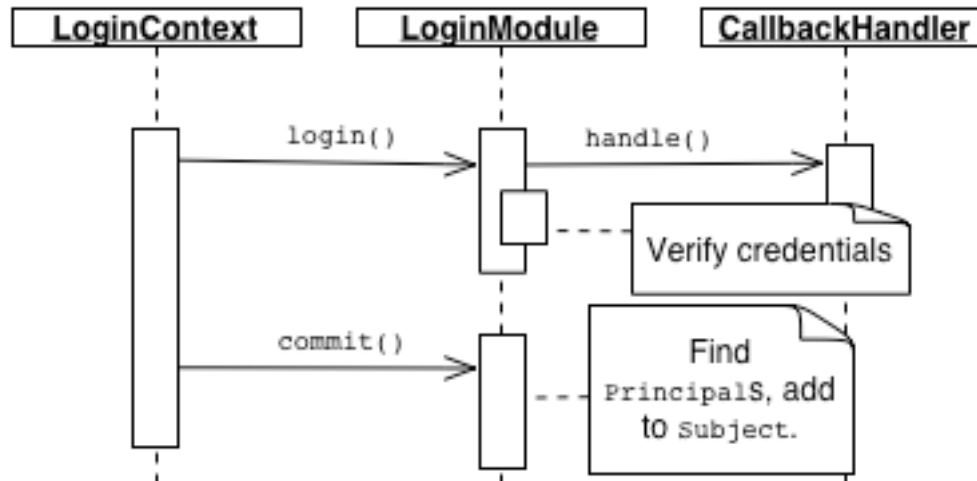
### 2.2.1. *Logging in the User*

Before JAAS can be used, the user must be logged in. As noted in the previous chapter, a user is represented by a `Subject`, which holds on to the identities of that user, represented by `Principals`. In the example, then, the concepts of a “normal user” and a “system administrator” are each represented by a `Principal`.

```
UserPrincipal(String username)  
SysAdminPrincipal(String username)
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5  
License: <http://creativecommons.org/licenses/by-nc/2.5/>



The diagram below illustrates the high-level process of logging in a user:

1. Collect credentials for the user, done by the `handle()` method on `CallbackHandler`.
2. Verify the credentials, performed by the `LoginModule` implementation.
3. Associated Principals accordingly with a Subject, also done by the `LoginModule` implementation.

JAAS coordinates all this via the `LoginContext`, which has pluginable items called `LoginModules` that do steps 2 and 3. Multiple `LoginModules` may be configured, allowing multiple sources to contribute Principals.

### 2.2.2. The “Application”

Below is the code that runs the tests for the simple example.

```

package chp02;

import java.io.File;
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import chp02.auth.SimpleCallbackHandler;

public class Chp02Main {

    public static void main(String[] args) throws Exception {

        File policyFile = new File("build/conf/chp02.policy");

        testAccess(policyFile, "user", "password");
        testAccess(policyFile, "sysadmin", "password");
    }
}

```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
}

static void testAccess(final File policyFile,
    final String username, final String password)
    throws LoginException {
    // Login a user
    SimpleCallbackHandler cb = new SimpleCallbackHandler(username,
        password);
    LoginContext ctx = new LoginContext("chp02", cb);
    ctx.login();
    Subject subject = ctx.getSubject();
    System.out.println("Logged in " + subject);

    // Create privileged action block which limits permissions
    // to only the Subject's permissions.
    try {
        Subject.doAsPrivileged(subject, new PrivilegedAction() {

            public Object run() {
                policyFile.canRead();
                System.out.println(username + " can access Policy file.");
                return null;
            }
        }, null);
    } catch (SecurityException e) {
        System.out.println(username + " can NOT access Policy file.");
    }
}
}
```

The method `testAccess()` is used to test a specific user's ability to read the Policy file.

First, a custom `CallbackHandler`, `SimpleCallbackHandler` is instantiated and passed to the `LoginModule`. `CallbackHandlers` are the part of JAAS that are responsible for collecting the credentials for users. A custom callback handler works in concert with a custom `LoginModule` to authenticate a user, adding `Principals` to the `Subject` being authenticated as all goes well.

The `LoginContext` is a final class in JAAS that coordinates running all the `LoginModules`, and determines what to do if there are any problems along the way. The `LoginContext` is configured through a properties file where each grouping of `LoginModules` are given a name. Thus, when the `LoginContext` is instantiated, the name of the `LoginModule` group is passed to it to tell the `LoginModule` which group to use.

Once the `LoginContext` has authenticated all the users (delegating to the `LoginModules` configured), we can get the authenticated `Subject`, which will contain the appropriate `Principals`. When the user "user" is authenticated, their `Subject` will have a `UserPrincipal`. When the user "sysadmin" is authenticated, their `Subject` will have a `SysAdminPrincipal`.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

Next, with the Subject authenticated, we attempt to read the Policy file. Creating a `java.io.File` instance for the policy does this. A security check is done within the `canRead()` method, and will throw an exception if it fails.

### 2.2.3. Authentication Code

There are three custom authentication parts to needed for our example:

1. A custom `LoginModule` for logging in Subjects, adding the appropriate Principals.
2. A custom `CallbackHandler` to collect a Subject's credentials for our custom `LoginModule`,
3. A configuration properties file to configure JAAS to use the custom `LoginModule`.

In this section, we'll go over each.

#### *Custom LoginModule and CallbackHandler*

`LoginModules` are given the responsibility of authenticating a Subject based on the credentials provided. Credentials can be anything that helps confirm the identity of a Subject. The most common credentials are username and password. Once a `LoginModule` has verified the identity of a Subject, the `LoginModule` will add Principals, as appropriate the Subject.

JAAS can be configured to use any number of `LoginModules`, allowing disparate authentication sources to contribute Principals to a Subject. Because multiple `LoginModules` can be used to authenticate a user, a multi-phase process is used to log users in. This is covered in more detail in THE CHAPTER ON LOGINMODULES. For now, you just need to know that the login module is used the authorize a Subject, while the `commit()` method used to add Principals to a fully authenticated Subject.

The custom `LoginModule` used for the above is below:

```
package chp02.auth;

import java.io.IOException;
import java.security.Principal;
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
import chp02.SysAdminPrincipal;
import chp02.UserPrincipal;

public class SimpleLoginModule implements LoginModule {

    private Subject subject;
    private CallbackHandler callbackHandler;
    private String name;
    private String password;

    public void initialize(Subject subject,
        CallbackHandler callbackHandler, Map sharedState, Map options)
    {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
    }

    public boolean login() throws LoginException {
        // Each callback is responsible for collecting a credential
        // needed to authenticate the user.
        NameCallback nameCB = new NameCallback("Username");
        PasswordCallback passwordCB = new PasswordCallback("Password",
            false);
        Callback[] callbacks = new Callback[] { nameCB, passwordCB };
        // Delegate to the provided CallbackHandler to gather the
        // username and password.
        try {
            callbackHandler.handle(callbacks);
        } catch (IOException e) {
            e.printStackTrace();
            LoginException ex = new LoginException(
                "IOException logging in.");
            ex.initCause(e);
            throw ex;
        } catch (UnsupportedCallbackException e) {
            String className = e.getCallback().getClass().getName();
            LoginException ex = new LoginException(className
                + " is not a supported Callback.");
            ex.initCause(e);
            throw ex;
        }

        // Now that the CallbackHandler has gathered the
```



```
// username and password, use them to
// authenticate the user against the expected passwords.
name = nameCB.getName();
password = String.valueOf(passwordCB.getPassword());

if ("sysadmin".equals(name) && "password".equals(password)) {
    // login in sysadmin
    return true;
} else if ("user".equals(name) && "password".equals(password)) {
    // login user
    return true;
} else {
    return false;
}
}

public boolean commit() {
    // If this method is called, the user successfully
    // authenticated, we can add the appropriate
    // Principles to the Subject.
    if ("sysadmin".equals(name)) {
        Principal p = new SysAdminPrincipal(name);

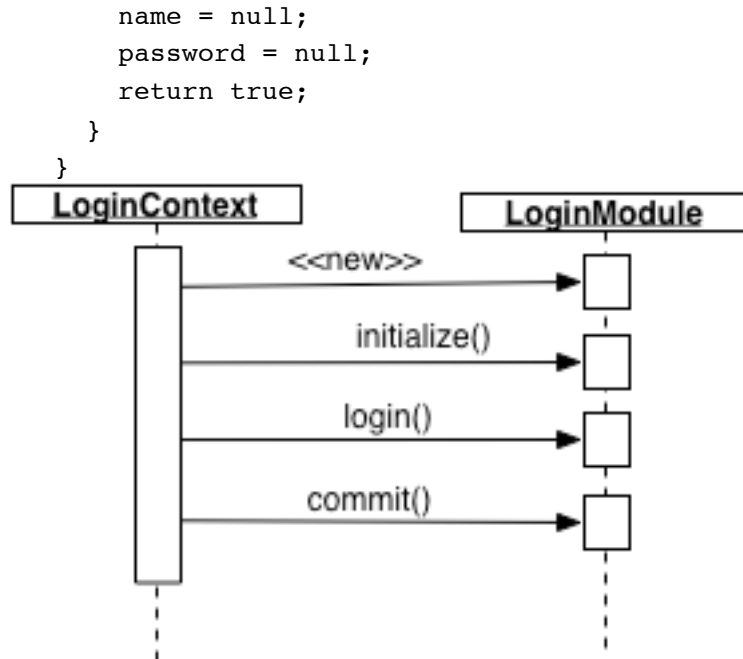
        subject.getPrincipals().add(p);
        password = null;
        return true;
    } else if ("user".equals(name)) {
        Principal p = new UserPrincipal(name);
        subject.getPrincipals().add(p);
        password = null;
        return true;
    } else {
        return false;
    }
}

public boolean abort() {
    name = null;
    password = null;
    return true;
}

public boolean logout() {
```







The diagram above illustrates the interaction between the `LoginContext` and `LoginModule` when a Subject is being authenticated, the `LoginContext`:

1. Creates an instance of the above `LoginModule`.
2. Calls the `initialize()` method, which gives the `LoginModule` the Subject it will authenticate and the `CallbackHandler` to retrieve credentials with.
3. Calls the `login()` method on the `LoginModule`, telling the `LoginModule` to attempt to authenticate the user.
4. If the `login()` method succeeds, calls the `commit()` method, signaling the `LoginModule` to add Principals to the Subject.
5. If the `login()` method fails or other errors occur, calls the `abort` method, signaling the `LoginModule` to do any cleanup needed (this is not shown in the diagram above).

### *A Closer Look at login() and commit()*

In our example, the most interesting methods are the `login()` and `commit()` methods. The `login()` method uses the `CallbackHandler` passed in to the `initialize` method to collect the credentials required. The `SimpleLoginModule` is only interested in the username and a password. A `NameCallback` and `PasswordCallback` instance are created, and passed to the `CallbackHandler`. The `SimpleCallbackHandler` method `handle` (shown below) simply fills in the passed-in Callbacks:

```

package chp02.auth;

import javax.security.auth.callback.Callback;

```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;

public class SimpleCallbackHandler implements CallbackHandler {

    private String name;
    private String password;

    public SimpleCallbackHandler(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public void handle(Callback[] callbacks) {
        for (int i = 0; i < callbacks.length; i++) {
            Callback callback = callbacks[i];
            if (callback instanceof NameCallback) {
                NameCallback nameCB = (NameCallback) callback;
                nameCB.setName(name);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCB = (PasswordCallback) callback;
                passwordCB.setPassword(password.toCharArray());
            }
        }
    }
}
```

Once the `CallbackHandler` has collected the username and password, they're stored in the `SimpleLoginModule` instance. Then, the stored credentials are compared against hard-coded values<sup>1</sup>: if each type of user has the correct password, the login method returns true, indicating that the `SimpleLoginModule` has verified the identity of the Subject.

The `LoginContext` calls the commit method once the Subject being logged in has been authenticated with all the `LoginModules` required. `SimpleLoginModule`'s commit method is repeated below:

```
public boolean commit() {
    if ("sysadmin".equals(name)) {
        // sysadmin Principal
        Principal p = new SysAdminPrincipal(name);
```

---

<sup>1</sup> In a real system, of course, the credentials wouldn't be hard-coded; they would be looked up in a database or otherwise retrieved.



```
        subject.getPrincipals().add(p);
        password = null;
        return true;
    } else if ("user".equals(name)) {
        // login user Principal
        Principal p = new UserPrincipal(name);
        subject.getPrincipals().add(p);
        password = null;
        return true;
    } else {
        return false;
    }
}
```

Since the `Subject` has been authenticated by the `login` method, the `commit()` method need only check which user has logged in, and add the appropriate `Principal` to the `Subject`. The `commit()` method returns `true` if everything went OK, or `false` if something went wrong.

The `Principals` `SysAdminPrincipal` and `UserPrincipal` are simple implementations of the `Principal` class. We won't go over them here, but THE CHAPTER/SECTION ON PRINCIPALS covers them in more detail.

### *LoginContext Configuration*

JAAS is configured to use the custom `LoginModule` by specifying it's use in a login module properties file. The file specifies groupings of `LoginModules` by "application." Applications are really just ordered groupings of `LoginModules`, each of which may be required or optional for a `Subject` to be successfully authenticated in the context of that application. These groupings may map to the traditional idea of a software application, or they can just be different groupings.

Our example configuration file contains the below:

```
chp01
{
    chp01.auth.SimpleLoginModule REQUIRED;
};
```

This configuration creates an application/group named "chp01." Any `Subject` wishing to be authenticated for that application is required to be successfully authenticated by the `SimpleLoginModule`.

A system property is used to specify the location of the configuration file. In our example, when executing the VM, the following system property is specified

```
-Djava.security.auth.login.config=src/conf/chp01-
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

`loginmodules.properties`

Many applications will need to set the `LoginContext` configuration in a more dynamic way, programmatically at runtime. Chapter 4 covers this. Using a flat-file works fine for our example.

#### 2.2.4. Authorization Code

Once the `Subject` has been authenticated, we're ready to attempt to access the `Policy` file, showing off how JAAS performs authorization, or permission, checks. The process is as following:

1. The `Subject` is acquired from the `LoginContext`.
2. The static method `Subject.doAsPrivileged` is used to execute a protected block of code on behalf of the `Subject`.
3. The block of code is implemented by a `PrivilegedAction` implementation.

In addition to the code, you must pass in another VM argument that points to the `Policy` configuration file to configure JAAS's default `Policy`.

##### *Privileged Block of Code: doAsPrivileged*

The method `Subject.doAsPrivileged` is used to demark that a sensitive block of code be executing on behalf of a given `Subject`. By passing in `null` as the last argument to the `doAsPrivileged` method, we're telling JAAS to execute the `PrivilegedAction` code with *only* the `Permissions` granted to the `Subject`. This means that the `Subject` must contain at least one `Principal` that has been granted the permission to read the `Policy` file.

The inline implementation of `PrivilegedAction` acts as a closure to pass to JAAS. It wraps the code to be executed with the permissions granted to the `Subject`. The method `File.canRead()` contains an authorization check that eventually results in code like the following being called:

```
FilePermission filePerm = new FilePermission("some.policy", "read");
AccessController.checkPermission(filePerm);
```

In the above code, we:

1. Create a `FilePermission` instance that represents the permission to read the file `some.policy`.
2. Use the `AccessController` to see if the `Principal` currently logged in has been granted to required permission.

If the user has been granted permission to read the file, the `checkPermission()` method silently succeeds. Otherwise, if the `Subject` does not have `Permission`, an `AccessControlException` is thrown. Thus, if you want to avoid thrown exceptions from disrupting your application, to check a `Permission` you have to wrap a try/catch block around the sensitive code, and catch any `AccessControlException` that's thrown. If the exception is thrown, the access check has failed.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

The Permissions granted to each Principal are specified in a Policy configuration file. This file is used by the default, file based, Policy that ships with the SDK. The location of this file is specified by a VM argument.

The two grant entries below are of interest to us<sup>2</sup>:

```
grant Principal chp02.UserPrincipal "user"
{
    // not granted anything
};

grant Principal chp02.SysAdminPrincipal "sysadmin"
{
    permission java.io.FilePermission "/Users/cote/dev/jaas-
book/build/conf/chp02.policy", "read";
};
```

Each of the grant sections above is used to grant (or not grant) Permissions to specific permissions. The syntax used is to specify the class of the Principal, the name the class will have, and then to list the Permissions granted to that Principal.

The permission to read the Policy file is configured by specifying the class of the Permission to grant, the path to the file the Permission covers (the target), and the action the Permission is for. We've purposefully included the commented- out grant for the UserPrincipal to emphasis that the Principal doesn't have that grant.

When this policy is applied, only Subjects that have a SysAdminPrincipal with the name "sysadmin" will be able to read the policy file chp02.policy.

### 2.2.5. *Running the Example*

To make running the example easier, we've provided an Ant target:

1. Go to the root directory of the source code for this book.
2. Type `ant run-chp02`.

The output will include the following output:

```
run-chp02:
[java] Logged in Subject:
[java]     Principal: (UserPrincipal: name=user)
[java] user can NOT access Policy file.
[java] Logged in Subject:
[java]     Principal: (SysAdminPrincipal: name=sysadmin)
[java] sysadmin can access Policy file.
```

---

<sup>2</sup> After running the ant target `ant run-chp02`, the policy file will be available at `build/conf/chp02.policy`.



## 2.3. *Summary*

With the astronaut's- and bird's-eye views of security and Java security, we further brought the discussion down to the worm's-eye view of JAAS in this chapter. By using two simple examples, we discussed the core classes in the JAAS API: `Policy`, `Permission`, `Subject`, and `Principal`. We discussed the roles of each class and spent time decomposing them into their parts. Without too much detailed discussion, which we've saved for the upcoming chapters, we went over on short example of using JAAS to give you a basic sense of both how JAAS works and what JAAS-enabled code looks like.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5  
License: <http://creativecommons.org/licenses/by-nc/2.5/>