Distributed Systems – Labs

Securing Java

Learning Outcomes:

Upon completion of this lab, the student should be able to

1. Demonstrate a clear understanding of the requirement for security in distributed Java applications.
2. Implement applications that use the Java security manager to protect access to vital resources.
3. Use Java's digital signing tools to provide authentication in networked Java applications.

Tasks

Applet Restrictions

1. Extract and compile T1\WriteFile.java. This is an applet and must be viewed through a web page. Create a file named WriteFile.html, and save the following code in it:

```
<html>
    <body>
        <applet code="WriteFile.class" width="500" height="150">
</applet>
    </body>
</html>
```

2. From the command prompt, you can use Java's cut down browser to try to view the web page with the applet in it:

```
:/> appletviewer WriteFile.html
```

You will notice that you get a `java.security.AccessControlException`. This is because the applet must run in a sandbox, where it is not permitted to write to the local disk.

3. In order to give the applet the permissions it requires we must create a policy file. This can be done manually, or the Java `policytool` can be used, as follows:

```
:/> policytool
```

You'll be presented with the following screen.

Click "Add Policy Entry" to get the following screen.

`codebase` refers to the original location of the applet code i.e. where it was downloaded from.

`Signed By` is used when we digitally sign code so we can authenticate that it was written by a certain person.

If you have both a codebase and a Signed By entry, the permission(s) will be granted only to code that is both from the specified location and signed by the named alias

Leave both fields blank.

Click "Add Permission" to get the following screen.

Do the following to grant code from the specified CodeBase permission to write (and thus also to create) the file named writetest.

Choose File Permission from the Permission drop-down list. The complete permission type name (java.io.FilePermission) now appears in the text box to the right of the drop-down list.

Type the following in the text box to the right of the list labeled Target Name to specify the file named writetest:

writetest

Specify write access by choosing the write option from the Actions drop-down list.

Select OK -> Done -> File -> Save As -> "writetest.policy"

Open the policy file with TextPad and view its contents.

Now, view the web page using appletviewer while providing the policy file:

```
:/> appletviewer -J-Djava.security.policy=writetest.policy WriteFile.html
```

You'll notice that now the applet is successful in writing to the file.

4. Extract the file T4\GetProps.java. This file accesses a number of system properties and prints out their values. Compile it and run it to see how it works.

5. Now run it with the security manager invoked - you can do this from the command line as follows:

```
:/> java -cp . -Djava.security.manager GetProps
```

The system policy file, loaded by default, grants all code permission to access some commonly useful properties such as "os.name" and "java.version". These properties are not security-sensitive, so granting such permissions does not pose a problem.

The other properties GetProps tries to access, "user.home" and "java.home", are not among the properties for which the system policy file grants read permission. Thus as soon as GetProps attempts to access the first of these properties ("user.home"), the security manager prevents the access and reports an AccessControlException. This exception indicates that the policy currently in effect, which consists of entries in one or more policy files, doesn't allow permission to read the "user.home" property.

6. Using policytool, or simply using TextPad, create a policy file named getprops.policy that contains the following code:

```
grant {

permission java.util.PropertyPermission "java.home", "read";

permission java.util.PropertyPermission "user.home", "read";

};
```

7. Run the application again with the policy file.

```
:/> java –cp . -Djava.security.manager   -
Djava.security.policy=getprops.policy GetProps
```

Notice that this time the application is given the required freedom.

Signing Code and Granting It Permissions

8. In this scenario, Susan wishes to send code to Ray. Ray wants to ensure that when the code is received, it has not been tampered with along the way - for instance someone could have intercepted the code exchange e-mail and replaced the code with a virus.

Create two directories named susan and ray. Extract the program T8\Count.java and save it in susan. Create a file named data.txt and save it in the susan folder. Run the Count program, which will count the number of characters in the file:

```
:/> java Count data.txt
```

9. Because Susan wants to send her data with authentication, she must create a public/private key pair. The java keytool allows users to do this:

:/> keytool -genkey -alias signFiles -keypass kpi135 -keystore susanstore -storepass ab987c

Enter all the required information for Susan. The `keytool` will create a certificate that contains Susan's public key and all her details.

This certificate will be valid for 90 days, the default validity period if you don't specify a -validity option.

The certificate is associated with the private key in a keystore entry referred to by the alias signFiles. The private key is assigned the password kpi135.

10. Susan now wants to digitally sign the code to send it to Ray. The first step is to put the code into a JAR file:

```
:/> jar cvf Count.jar Count.class
```

Then the `jarsigner` tool can be used to sign the JAR:

```
:/> jarsigner -keystore susanstore -signedjar sCount.jar
Count.jar signFiles
```

You will be prompted for the store password (ab987c) and the private key password (kpi135).

11. Susan can now export a copy of her certificate and send this with the signed JAR to Ray:

```
:/> keytool -export -keystore susanstore -alias signFiles –file SusanJones.cer
```

The certificate will be in the file SusanJones.cer - this contains her public key which Ray can use to authenticate the origin of the file he received.

12. Copy (simulated e-mail) the file SusanJones.cer and the signed JAR sCount.jar to the ray folder. Create or copy a file named data.txt to put in Ray's folder. Try to execute the code with the security manager in place:

```
:/> java -Djava.security.manager -classpath sCount.jar Count
```

Note the AccessControlException - you are not permitted access the disk with the Security Manager in operation.

13. Ray will now create his own keystore (use any password you want), into which he will import Susan's details:

```
:/> keytool -import -alias susan -file SusanJones.cer -keystore raystore
```

14. Ray can now verify that the code, sCount.jar was signed by Susan:

```
:/> jarsigner -verify -verbose -keystore raystore sCount.jar
```

15. Ray can also give any code signed by Susan permission to access certain files or perform operations it would not otherwise be permitted to do, by creating the following policy file (raypolicy.policy) (using policytool or TextPad):

```
keystore "raystore";

grant signedBy "susan" {

permission java.io.FilePermission "data.txt", "read";

};
```

and using this when running the code:

```
   :/> java -Djava.security.manager -Djava.security.policy=raypolicy.policy -
classpath sCount.jar Count data.txt
```

Further Reading

Java tutorial section on

* Security
* Quick Tour of Controlling Applets
* Signing Code and Granting It Permissions