CSCI 355 LAB #11 Spring 2007

Java Applet Security

Objectives:
1. Learn how to write a Java Applet to run in a web browser.

2. Learn how to write Java Applets with Signed Certificates.

Laboratory Assignment:

Exercise 1: Java Applets

Copy the file **TempConvert.java** from `~cs355/Lab11` on the linux machines.

Compile and run the program to see what it does. Note that the user can enter a temperature in either text field.

For this exercise you are to convert this Java application to a Java Applet. Here is a cookbook procedure to convert a Java application with a GUI to a Java Applet.

1. Add the import statement:

   ```
   import javax.swing.JApplet;
   ```

2. Replace the **extends JFrame** with **extends JApplet**.

   The original program creates a window with the **JFrame** class. The **JFrame** window is not needed since the Applet program will display on an area of the web browser's window.

3. Replace the line of the constructor's header with **public void init()**

   In an Applet you need to override the **init()** method of the **JApplet** superclass. This is where the Applet will start.

4. Remove all code for initializing the JFrame window including the **Container** line.

5. Remove all occurrences of the object prefix and period "contentPane." from the lines of code. Don't remove the lines just the prefix.

   The container is now the Applet in question (this).

6. Remove the **main** method.

   Since an Applet starts at the **init()** method, the **main** method is not needed.

7. Don't call **System.exit()** method in an Applet.

Compile the Java code.

To test your new Applet use Java's **appletviewer**. But first, create an html file, say **TempConver.html**, as follows:

```
<!-- Assumes class files are in same directory as the .html file -->
<html>
   <head>
      <title>Temperature Conversion Applet</title>
   </head>
   <body>
       <h2> Temperatures can be entered in either field.</h2>
       <applet code = "TempConvert.class" width = "200" height = "80">
       </applet>
   </body>
</html>
```

Now run the **appletviewer** with the **.html** file.

```
     appletviewer TempConver.html
```

It is best to always debug your Applet with **appletviewer** before trying the Applet in a web browser.

To run your Applet in a web browser, copy *all* three **.class** files and the **.html** file to your `public_html` directory. Make your home directory, the `public_html` directory and the four files readable by the world.

The URL will be:

```
     http://www.linux.bucknell.edu/˜loginName/TempConver.html
```

You should see a coffee cup as the Applet is dynamically loaded across the Internet. Note that Java Applets must be turned on in the web browser and some browsers may be missing the required plugin.

Exercise 2: Java Applet Accessing Resource Outside Sandbox

To do the previous exercise you did not use any resources outside of Java's Sandbox (protection domain), therefore, you did not need to worry about Java's Security Manager. In this exercise you will write an Applet that requires a resource outside of Java's Sandbox. Therefore, you must deal with Java's Security Manager.

Copy your Applet from Exercise 1 and give it a new name, e.g., Ex2.java. You are to read a simple web page `www.eg.bucknell.edu/˜cs355/lab11.html` and extract the third line (Great!) in the file. See Lab 6 on how to open a socket and read a web page. Once you have read the proper line on the web page you are to display the text in the Applet.

Change the grid layout in the Applet to allow a third row and add a new JLabel initialized to six blanks. Add the new JLabel to the third row. When the user enters a Centigrade temperature, read the text from the web page and display it as the new JLabel.

Try your Applet first using **appletviewer**. Since you are accessing a resource outside of Java's Sandbox, you will receive a "access denied" SecurityException when the Applet tries to open the socket.

You will need to create a **policy** file. Enter the following text in a file called **lab11.policy**.

```
// Java security policy file
```

```
grant {
   // allow all socket operations on port 80 of www.eg.bucknell.edu
   permission java.net.SocketPermission "www.eg.bucknell.edu:80",
     "accept,listen,connect,resolve";
};
```

Now run Appletviewer using the policy file as follows:

```
appletviewer -J-Djava.security.policy=lab11.policy Ex2.html
```

It should run.

If you are having problems, try, as an intermediate step, copying the **lab11.html** file to your account on **linuxcomp3** (the web server's host). When this version of your program works with **appletviewer**, then try reading the file on the remote host **www.eg.bucknell.edu**.

Now try to run the Applet in the web browser. Since web browsers have a much stricter security policy, the Applet should load but a small red "X" should appear to indicate that it has blocked the illegal access to the socket. We fix this in the next exercise.

Exercise 3: Java Applet with Signed Certificates

We saw in the last exercise that untrusted remote code in a web browser can not access a socket on a different host. Doing this is a dangerous security problem. As users browse the web, they don't want web sites secretly loading an Applet, then reading or writing files on their hard drive or opening sockets.

In this exercise, you will learn how to digitally sign the Applet with a certificate. To make the process secure, there is a series of steps you must do.

Reference: "Java 2 Platform Security" by Ramesh Nagappan, Ray Lai, Christopher Steel. Date: Jan. 6, 2006, 30 pages. About one third of way through are good examples of signing an Applet.

`http://www.informit.com/articles/printerfriendly.asp?p=433382&rl=1`

The Java 2 platform has three *key management* tools to facilitate creating signed applets:

1. The **keytool** is used to create pairs of public and private keys, to import and display certificate chains, to export certificates, and to generate X.509 v1 self-signed certificates.
2. The **jarsigner** tool is used to sign JAR files and also to verify the authenticity of the signature(s) of signed JAR files.
3. The **policytool** is used to create and modify the security policy configuration files. We could have used the **policytool** to create the above policy file but it was easier just to type it.

Step 1: Make a JAR file of the compiled class files.

Reference: `http://www.eg.bucknell.edu/~CS475/F06-S07/jar-file.html`

SUN's Java distribution has an archive tool called **jar**. The **jar** tool is a Java application that combines multiple files into a single JAR archive file. The **jar** tool also compresses files. In addition, it allows individual entries in a file to be signed with a digital signature so that their origin can be authenticated.

This is important for authors of Java applets that will be sent across a network. See Unix tool **jarsigner** for signing jar files.

The syntax for the **jar** tool is almost identical to the syntax for the Unix **tar** command. See the **jar** manual page.

```
      % man jar
```

Use **jar** to archive all your Applet's .class files into a **.jar** file.

```
      % jar cvf myJar.jar *.class
```

It will tell you how much the files were deflated (compressed).

Step 2: Generate key pairs.

Using the **keytool** utility, create the key pair and self-signed certificate. The JAR file will be signed with the creator's private key and the signature is verified by the communicating peer of the JAR file with the public key in the pair.

```
keytool -genkey -alias sign1 -keystore myStore -keypass myKpass -storepass mySpass
```

This **keytool -genkey** command generates a key pair that is identified by the alias (name) **sign1**. Subsequent keytool commands are required to use this alias and the key password (**-keypass myKpass**) to access the private key in the generated pair. The generated key pair is stored in a keystore database called **myStore** (**-keystore myStore**) in the current directory and is accessed with the **mySpass** password (**-storepass mySpass**). The command also prompts the signer to input information about the certificate, such as name, organization, location, and so forth.

If you run **keytool -genkey** move than once, change the **-alias** field, say to **sign2** and so on.

Step 3: Sign the JAR file.

Using the **jarsigner** utility, sign the JAR file and verify the signature on the JAR files.

```
jarsigner -keystore myStore -storepass mySpass -keypass myKpass -signedjar SignedEx3.jar myJar.jar sign1
```

The **-storepass mySpass** and **-keystore myStore** options specify the keystore database and password where the private key for signing the JAR file is stored. The **-keypass myKpass** option is the password to the private key, **SignedEx3.jar** is the name of the signed JAR file, and **sign1** is the alias (name) to the private key. **jarsigner** extracts the certificate from the keystore and attaches it to the generated signature of the signed JAR file.

Step 4: Modify the .html file.

Add to the .html file the **archive** attribute with the name of the signed JAR file.

```
<applet code="Ex3.class" archive="SignedEx3.jar" width=200 height=80>
</applet>
```

After the Applet loads, you should see a certificate asking whether you trust it or not. Select trust and the Applet should run in the web browser now.

Hand In:

Please demonstrate the solutions of Exercises 1, 2 and 3 to your instructor.