Signed Applet Tutorial

by Larry Siden

Introduction

While volunteering on a pro-bono project at Menlo Innovations my colleagues and I encountered a problem. A team had developed a Java applet in which a user types in a URL and clicks a button. The applet then downloads the web page specified by the URL and extracts any links (i.e. <a href="...">) and displays them. While we tested this in our development environment's applet viewer everything seemed to work fine. But when we tried to use the applet in a browser and connect to Google, it seemed to hang. Opening the Java console window revealed

> java.security.AccessControlException: access denied (java.net.SocketPermission www.google.com connect,resolve)

Background

Java virtual machines run applets under a different security regime than applications. By default, applications are implicitly trusted. The designers of the JVM specification assumed that users start applications at their own initiative and can therefore take responsibility for the application's behavior on their machine. Such code is considered to be *trusted*. Applets, on the other hand, are started automatically by the browser after it downloads and displays a page. Users cannot be expected to know what applets a page might contain before they download it, and therefore cannot take responsibility for the applet's behavior on their machine. Applets, therefore, are considered by default to be *untrusted*. Among other restrictions, an applet cannot, by default, open a socket referred to by a URL who's domain different from the domain of the page that contains the applet. This is part of the security architecture that browsers employ to protect users' computing resources from malicious or faulty applets.

The JVM's security policy is set by a the file `$JAVA_HOME/jre/lib/security/java.policy`. Here is an excerpt from the one found on my computer:

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/*" {
        permission java.security.AllPermission;
};

// default permissions granted to all domains

grant {
        // allows anyone to listen on un-privileged ports
        permission java.net.SocketPermission "localhost:1024-", "listen";

        ...
};
```

It's syntax is described by Default Policy Implementation and Policy File Syntax . The first **grant** construct permits any code that lives in the directory `$JAVA_HOME/lib/ext/*` to do anything. Such code is considered to be trusted. This makes sense only if this directory and its children are not writable to ordinary users.

When I create an applet and test it with Eclipse's VM, Eclipse creates a file named **java.policy.applet** in

the project root directory. Here are the contents:

```
/* AUTOMATICALLY GENERATED ON Tue Apr 16 17:20:59 EDT 2002*/
/* DO NOT EDIT */

grant {
  permission java.security.AllPermission;
};
```

As you can see, there is no **codebase** parameter here. This grant construct allows code from any codebase permission to do anything. Eclipse's authors assume that Eclipse users can take responsibility for the behavior of applets that they write and test on their own machines. So this explains why my team did not encounter any security exception when we ran the applet from Eclipse. Eclipse ran the JVM in the same working directory where it created the file **java.policy.applet** seen above. In this environment, the applet could do anything it asked to.

In order to recreate this scenario at home, I wrote a simple applet at home that produces the same behavior our team encountered. To download the entire project, type:

```
cvs -d :pserver:guest@lsiden.homeip.net:/cvsroot login
cvs -d :pserver:guest@lsiden.homeip.net:/cvsroot checkout Signed-Applet
```

This will create a directory called **Signed-Applet** under the current working directory. The CVS archive is not writable for user **guest**.

This small applet produces the same behavior by requesting `SocketPermission` as described above to connect to any port in the domain `www.google.com`. Because of the security policy, the actual domain is irrelevant, since it is different from `localhost`. This is exactly what our team's applet was doing under the hood when it called `URL.getContent()` which is shorthand for `URL.openConnection().getContent()`. `URLConnection.openConnection()` makes exactly this call:

```
System.getSecurityManager().checkPermission(new
SocketPermission("www.google.com", "connect"));
```

which throws the `SocketException` that we were seeing in the Java Console window. `SocketException` is a subclass of `IOException`.

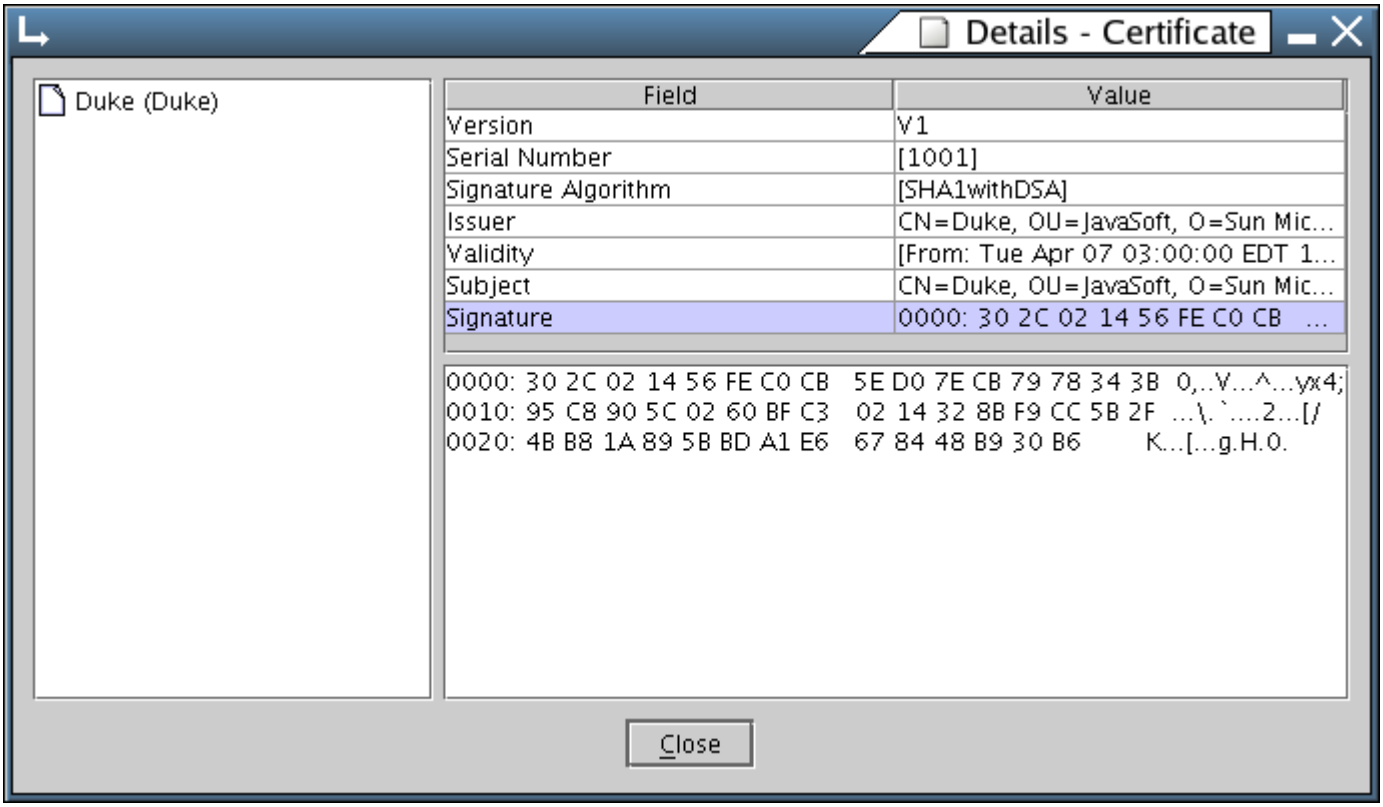
Digital Certificates

The solution to this conundrum is to create obtain a digital certificate and use it to sign the applet. When a well-behaved browser downloads a page that contains a signed applet, before running the applet it displays a certificate in a message box.

This certificate claims that the applet comes from the party named within and contains the digital signature of a *certificate authority*. In the above example, the certificate holder is a party named "Duke" and the issuer is Netscape. The certificate contains identifying information about the certificate holder and the certificate issuer, or trusted authority. A certificate authority is a third party that is trusted to verify a certificate applicant's credentials. When the authority is satisfied with its applicant's credentials, it issues it a digitally signed certificate.

(In this instance the certificate makes no claims as to the trustworthiness of either the certificate holder or the certificate issuer. It is for demonstration purposes only. If your computer contains sensitive data, you should not trust this certificate.)

The certificate authority may also have its own certificate that was generated by an even more trusted authority that verfied *its* credentials. Each certificate will refer to the certificate of its issuing authority. This may continue for several levels and is called a *chain of trust*. The chain of trust ends with a top-level authority that issues its own certificate based on its own reputation. The chain of trust in this example extend up only one level.



The image on the right is what appears on the user's screen when they click the button labelled 'More Details'. Every certificate contains a numeric hash or *digest* of the certificate contents, which can be seen in the bottom right pane. A user can use the digest if he/she choses to contact the authority to verbally confirm the validity of the certificate. The user could ask the authority for the digest of the certificate it issued and compare the response with the digest displayed on his/her screen.

Despite the robustness of public/private key encryption, and the thoroughness of the specification, digital certificates have yet to be universally adopted as a means of establishing trust when conducting business transactions. For a critique on digital certificates, see The Emperor's New Clothes: The Shocking Truth About Digital Signatures and Internet Commerce by Jane K. Winn.

Certificate authorities typically charge a fee for the service of validating their clients' credentials. However, for testing and demo purposes, we may create a self-signed certificate. The information given in a self-signed certificate has not been validated by a trusted third party.

The following section will cover the basic steps to creating a signed applet.

How To Create a Signed Applet

1. **Package the applet into a JAR file.** The applet must be in a JAR file before a certificate can be attached to it. Use the **jar** JDK utility. If the applet was previously referenced with the help of a **codebase** attribute in `<applet>` tag, replace the **codebase** attribute with the **archive** attribute. The value of the **archive** attribute is a URL of a JAR file.
2. **Create a public/private key pair.** The command for this is

   ```
   keytool -genkey
   ```

   keytool is another SDK utility. It will prompt you for a password to your *keystore* and for the remaining parameters, one of which is **alias**, whose value is the name of the key. The keystore is a file that contains your public/private key-pairs, and the public-keys of others with whom you exchange information. See the documentation in the above link.

3. **Create a certificate for the key you created in the previous step.**

   ```
   keytool -selfcert
   ```

   Again, **keytool** will prompt you for a keystore password and remaining parameters. This certificate is now self-signed by you, meaning that it has not been validated by any third party. This is suitable for demo purposes, and may be acceptable to yourself and those who know you because if there is any doubt that the certificate is really yours they can always call you up and ask you for the digest to verify that it is really you and not some impostor that created the certificate. However, if this applet were to be widely distributed, and you wanted it to be accepted by those who do not know you personally, you would certainly want to pay a modest fee to obtain a certificate that is validated by a trusted certificate authority. The procedure for this is straightforward, but beyond the scope of this simple tutorial.
4. Run jarsigner associate this certificate with the JAR file that contains your applet. You will need to give the name of the public key of the certificate you just created. This creates a digest for each file in your JAR and signs them with your private key. These digests or hashes, the public key, and the certificate will all be included in the "WEB-INF" directory of the JAR.

Your applet is now signed. The next time you or someone else downloads it in it's page the browser will present a dialog box displaying the credentials you just created for it and asking the user permission to run it. If he/she chooses not to, the applet will throw the same `AccessControlException` that we saw in the Java Console window the first time we tried to run it in our browser. The difference is that now the user gets to make an informed decision as to whether or not they trust your applet to not harm his/her system.

You will only need to generate the public/private key-pair once, but you will definitely want to automate the steps that create and sign the JAR file, because you will need to repeat those every time you modify anything in your code. You will most likely do this in your ant build-file, which is beyond the scope of this tutorial.

Links

1. Security and the Java Platform
2. JDK(TM) 1.1.x - Signed Applet Example
3. Signed Applets, Browsers, and File Access
4. Digital Certificates Guide
5. RFC 2459
   The Emperor's New Clothes: The Shocking Truth About Digital Signatures and Internet Commerce