Oracle9*i* Forms:
JAR File Signing for JInitiator 1.3

Technical White Paper
September 2002

ORACLE

Oracle9*i* Forms: JAR File Signing for JInitiator 1.3

PURPOSE

This document provides a technical overview of JAR file signing for use with Oracle9*i* Forms[1] and JInitiator 1.3

INTRODUCTION

As developers begin to take advantage of some of the client side integration features of Oracle9*i* Forms, one of the operations that is often required is the process of signing JAR files so that they are allowed to operate outside of the Java Applet "sandbox". This sandbox is a term for the inbuilt security restrictions placed on Java code running as an applet (usually within a browser). The code has very limited access to client machine resources and is not, for instance, allowed to read or write to the local hard-disks, or print to the printer. This security mechanism is common to all applets that run in the browser, not just the Forms applet and custom code that you call from it.

It is possible, however, for the Java code that executes within an applet, to be trusted by the local Java Virtual Machine (JInitiator in this case), and to have a wider set of privileges. But in order for this to happen, the code must be signed using a Java code signing certificate.

This paper discusses how certificates are generated, JAR files are signed and certificate distribution is achieved.

WHAT IS SIGNING

Signing is the process of applying a digital signature to your code. This signature is designed to prove two things:

- Who the code comes from
- The fact that the code has not been interfered with since it was signed

The signing of JAR files uses the technique of public key cryptography. The signer holds a pair of keys, one private and known only to the signer and one public that can be known generally to the world at large. The principal underlying public key cryptography is that a value can be encrypted using the public key of a pair, but the

[1] This paper is also relevant to those Forms 6*i* patches which are compatible with JInitiator 1.3, when run with that version of JInitiator.

encrypted value can only be decrypted using the private key and public key in combination. Thus you can encrypt a message with a public key, safe in the knowledge that only the holder of the private key will be able to read it.

This process can also work in reverse, so a signature can be included with your JAR file that has been encrypted with your private key and can be verified using the complementary public key.

SIGNING OPTIONS

When signing a JAR file you are able to use certificate that is backed up by a certificate authority or you can simply create your own certificate for "Self Signing"

Certificate Authorities

Certificate authorities such as VeriSign[2], RSA[3] or Thawte[4] are organizations which issue certificates for code signing (as well as for other purposes such as securing web sites through SSL).

These certificate authorities issue certificates and carry out various background checks on companies that apply for them.. The principle being that the certificate authority trusts the supplier of the code on your behalf. All browsers, and JInitiator contain the so called root certificates from these certificate authorities which can be used to verify that the stated certificate authority has indeed issued a particular certificate.

If you are creating pluggable components for Forms which need to be signed and will be distributed to an unknown or third party customer base, then it is recommended that you purchase a code signing certificate from one of these certificate authorities so that any consumers of your code can have full confidence in it.

Your chosen certificate authority will supply information on how to obtain and use a code signing certificate.

Self Signing

The rest of this paper covers the second option – self signed code. The certificate used in this case (unlike a certificate from a certificate authority) does not cost anything to obtain, but of course, given that anyone can create one without control, it is really only suitable for use in a regulated environment – for instance on an intranet. Most Forms applications, however, fit this profile and so this technique can be used.

The rest of this document will take you through the process of signing and deploying your own JAR files with a self-signed certificate.

---

[2] http://www.verisign.com/
[3] http://www.rsasecurity.com/
[4] http://www.thawte.com

PREREQUISITS AND PREPARATION

For this purposes of this exercise we will refer to three different machines:

1. The signing machine – this is the machine that has the public/private key pair created on it and which will be used to actually sign the JAR files
2. The client machines – these are the machines which will run the browser hosting the Forms applications. We will assume that these are using JInitiator 1.3.1.9 installed into the *C:\Program Files\Oracle\JInitiator 1.3.1.9* directory. However, this paper applies to any version of JInitiator 1.3.
3. The application server – the machine, which will host the JAR file(s) for download and which runs the Forms application.

These machines could all be the same machine in practice, but keep the logical distinction between them clear. When signing JARs and testing the results you should always try to deploy to a separate client machine if possible.

Before doing anything, you will need a copy of the Java Standard Edition[5] 1.3.1 Software Development Kit (SDK). If your "signing machine" has Oracle9*i*DS installed, then you will already have the software in the %ORACLE_HOME%/jdk directory. Otherwise you can download and install the SDK directly from Sun: http://java.sun.com/j2se/1.3/download.html

If downloading the SDK from Sun, install it into a directory such as *c:\java131* (the sample location used throughout this paper)

[5] Or the Enterprise Edition "J2EE"

CREATING THE CERTIFICATE

The first step to signing is to create a private / public key pair for the signing . The Java SDK contains a tool to do this: **keytool**, which can be found, along with the other tools in the /bin directory of the SDK (e.g. *c:\java131\bin*).

Let's examine the various bits of information that will be needed to create the certificate:

- Your Identity also called your "Distinguished Name".
- A Name and Password for your certificate.
- The file to act as a certificate store.
- How long you want the certificate to be valid for.

To deal with each of those in turn:

Your Identity – The Distinguished Name

The certificates that we are dealing with here are of the X.509 (v 1) standard. In order to generate the certificate you will need some information to identify who you are. This information is in a standard format, a X.500 Distinguished Name (DN), which is used to populate various fields within the X.509 certificate. The following table lists the values that can be supplied:

Property	Abbreviation	Example
CommonName	CN	Duncan Mills
OrganizationUnit	OU	Tools Product Management
OrganizationName	O	Oracle Corporation
LocalityName	L	Redwood Shores
StateName	S	California
Country	C	US

Figure 1 – X.500 Distinguished Name fields

When creating the certificate, the abbreviations are used in a comma separated list. You do not have to supply values for every field in the DN, but should provide enough information to uniquely identify the signer.

An example DN string for use in creating a certificate might be:

```
CN=Duncan Mills, OU=Tools Product Management, O=Oracle
Corporation, C=US
```

Note: The field elements of the DN must appear in the correct order in the string list

The Certificate Alias (Name) And Password

The Alias is the name that you'll be giving to this signing identity, which is how it will be subsequently referred to when signing with the certificate or importing it. It has a corresponding password, which protects it in the key store, and has to be supplied when using the identity to sign a JAR file.

In the example here, and through out the rest of the document, we'll use the alias PJC_IDENTITY and the password PJC_IDENTITY_PWD[6].

The KeyStore (Certificate Store)

The KeyStore is the file that stores your certificates. All the operations you carry out with KeyTool, and JAR file signing itself, need to know about this certificate store. You can use an existing KeyStore if you have one, or, as we will in this case, create one specifically for holding our signing identities.

A KeyStore can be created implicitly when you run KeyTool to create a new identity. In this case we'll use a brand new KeyStore "*keystore*" in a directory called *certificates* under the Java SDK root (*c:\java131*).

The directory to hold the KeyStore must exist, so we need to explicitly create a *c:\java131\certificates* directory before running KeyTool.

Just like identities, the KeyStore is password protected. We'll use the password KEYSTORE_PWD in this example

Validity

The final bit of information we will need to supply is the validity for the certificate, how long it will be effective. In this example we'll use a year. Validity is measured in days so we'll supply 365 as the figure.

Finally - Creating The Certificate

So we have now gathered all the information that we'll need to create a new self signed code certificate. The KeyTool utility provides several functions which can be listed by typing "keytool" or "keytool –help" at the command prompt. To generate our identity we'll use the following arguments:

[6] Of course this password is only as an illustration. For real certificates and key stores choose your passwords with care, and keep them secure.

Argument	Purpose
-genkey	Tells KeyTool to generate a public / private key pair
-dname	Defines the X.500 distinguished name for this certificate
-alias	The name of the signing identity to create
-keypass	The password to protect that identity
-keystore	The location and name of the keystore (certificate database)
-storepass	The password for the keystore

Figure 2 – Arguments to KeyTool to create a new identity

Putting all that together we can issue the full command to create the identity (This assumes that the c:\jdk131\certificates directory has already been created). The path to keytool can be supplied in the command, or as in this case the command is issued from the Java SDK \bin directory:

```
keytool -genkey
        -dname "cn=Duncan Mills,
                ou=Tools Product Management,
                 o=Oracle Corporation,
                 c=US"
        -alias pjc_identity
        -keypass pjc_identity_pwd
        -keystore c:\jdk131\certificates\keystore
        -storepass keystore_pwd
        -validity 365
```

Note that this command (and following commands) should all be entered on a single line from the command prompt. They have been reformatted here onto multiple lines for clarity.

Figure 3 – The KeyTool command

The above command will generate a 1024 bit key using the SHA1withDSA algorithm[7].

You should then verify that the certificate exists in the KeyStore. The *keytool –list* command can be used to do this, again supplying the name of the KeyStore and the KeyStore password.

```
keytool -list
        -keystore c:\jdk131\certificates\keystore
        -storepass keystore_pwd
```

Figure 4 – Listing the contents of the KeyStore

This should return a result to the console something like:

[7] For more information on KeyTool see the documentation provided with the Java SDK.

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

pjc_identity, Thu Sep 05 15:12:02 BST 2002, keyEntry,
Certificate fingerprint (MD5):
FF:78:38:8E:62:A7:61:6C:BB:DA:4E:B7:FF:76:29:E2
```

Figure 5 – Results of the "keytool –list" operation

SIGNING YOUR JAR FILES

Now that the signing identity has been created you can go ahead and sign your JAR files. This assumes that you have packaged up your PJC code using the **jar** tool, or using a "Simple JAR File" deployment profile in Oracle9*i* JDeveloper.

The process of signing will change the JAR file so it is advisable to take a backup copy before starting the signing process. If the Java code has been developed on another machine, you will need to copy the unsigned JAR file to the "Signing Machine".

Signing is done using the **jarsigner** tool in the Java SDK /bin directory. Jarsigner requires much of the same information as was used to create the certificate – the KeyStore, the KeyStore password, the alias for the identity and its password.

The only thing to watch out here is that the identity alias is always specified as the last argument and is not prefixed with a "-alias" keyword.

So again we'll use the PJC_IDENTITY as the alias, and the JAR file we'll sign in this example is *signedPJC.jar*. The other information is the same as was used with keytool.

```
jarsigner -keystore c:\jdk131\certificates\keystore
          -storepass keystore_pwd
          -keypass pjc_identity_pwd
          signedPJC.jar
          pjc_identity
```

Figure 6 – Signing a JAR file with the PJC_IDENTITY

Once the JAR file has been signed, you can confirm that it has worked using the "jarsigner –verify" command.

```
jarsigner –verify signedPJC.jar
```

Figure 7 – Verifying a JAR file

DEPLOYMENT OF THE JARS AND CERTIFICATES

To deploy your signed JAR file you would copy it to the application server machine in the normal way. This generally involves:

1. Deploying the JAR file to a mapped virtual directory on the application server machine – Usually the *forms90/java* directory so that the JAR is stored with the base Forms JARs.
2. Adding the name of the signed JAR file to the **ARCHIVE_JINI=** entry in your **formsweb.cfg** file (globally or for a specific profile).

Then when the user accesses the application from a browser on the client machine, the extra signed JAR file will be downloaded and cached.

Automatic Import of Certificates

One of the desirable features of JInitiator 1.3 (and the Sun 1.3 & 1.4 Plug-ins) is that they do not require that self-signing certificates be manually imported onto each client. Instead, when the Java runtime encounters a JAR file that is signed with an unknown certificate, it will pop up a dialog asking the user if they want to trust this code.

Figure 8 – JInitiator prompting the user to trust the new identity

Notice how this dialog contains the warning "*The security certificate was issued by a company that is not trusted*". This warning appears because this is self signed code and the certificate is not backed by a certificate authority. As you can see for Internet usage this message would worry the consumers of your application (and rightly so). So you should seriously consider obtaining a certificate from a certificate authority if that is your deployment profile.

The User has the choice to "Deny" the installation of the certificate. If they do this then your custom code, and possibly your application will not work.

If they choose to "Grant this session", then the code will run for the lifetime of this browser window. The user could exit the Forms application and refresh the page, and the code will still be trusted, but if they close the browser window, they will be prompted again the next time that they run the application.

If they choose "Grant Always" then the certificate will be stored into JInitiator's own KeyStore file[8] and they will never be prompted again until they upgrade to a new version of JInitiator.

Finally the user can choose to view the certificate. This will allow them to examine the certificate, including the information defined as the Distinguished Name of the signer and the certificate signature.

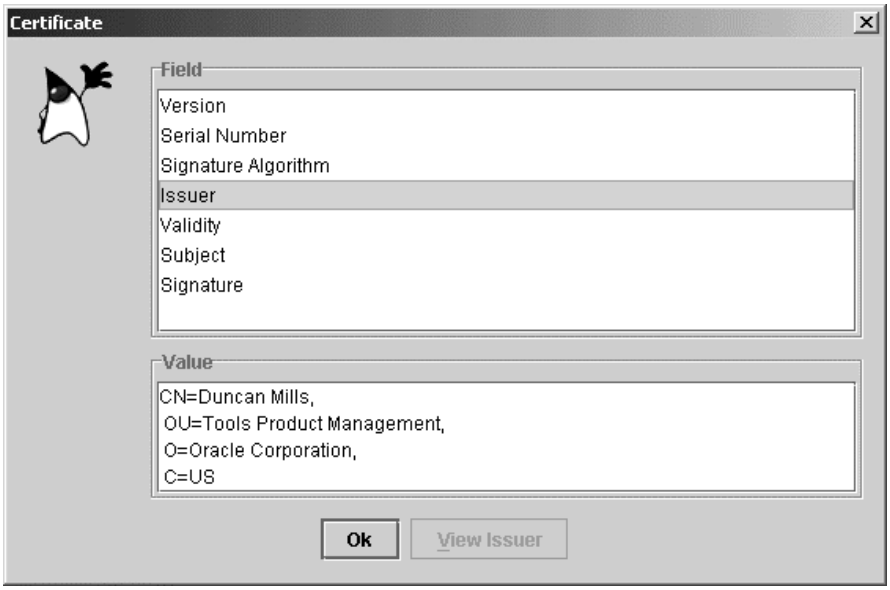

Figure 9 – Viewing the DN of the signer

[8] Note the certificate within the JAR file only contains the Public key of the key pair used to sign the code, so this import doesn't mean that the client machine can now be used to sign JARs with this identity. It will lack the private key to do so.

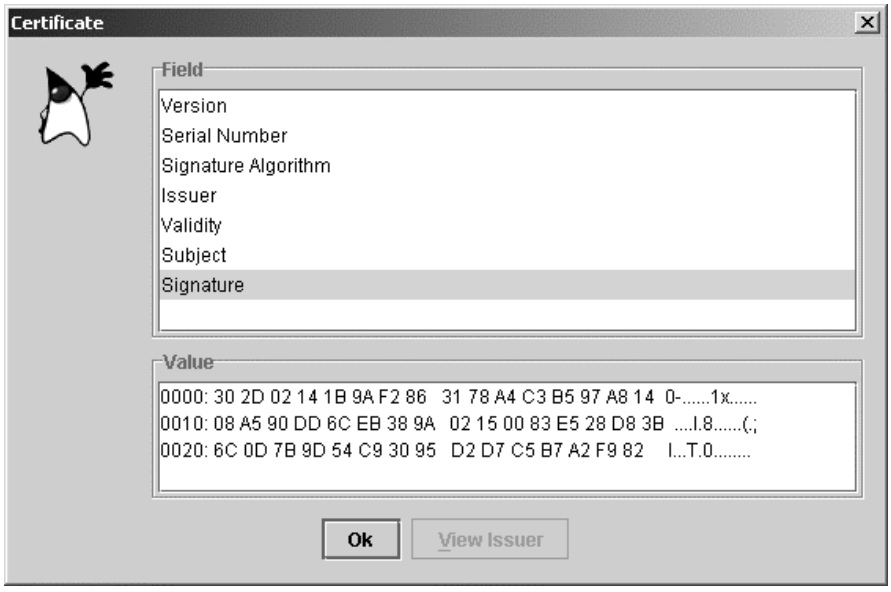

Figure 10 – Viewing the signature of the certificate

For additional security, the signature of the certificate could be published on a web-page accessible from your application to allow users to verify that the JAR has indeed been signed with a valid signature.

What if the JAR file has been altered?

If the JAR file that is downloaded has been changed since it was signed then the trust dialog will not appear at all. Instead an error will appear on the java console[9].

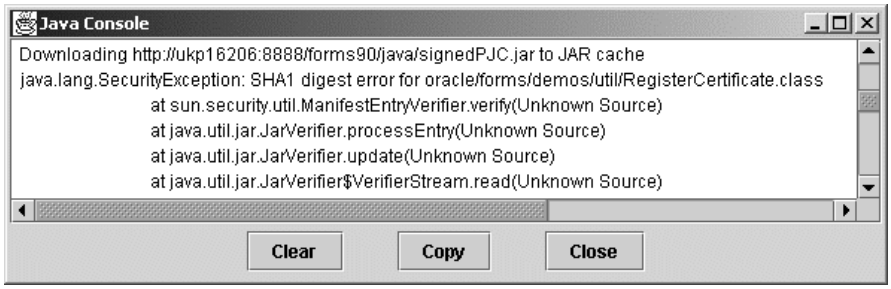

Figure 11 – Security exception caused by an changed class in a signed JAR file

The security exception will detail the class that has been changed since it was signed.

[9] The Java console is the target for most printed messages such as exceptions raised by Java code. To view the console in JInitiator 1.3, double click on the Duke icon  in the Windows System Tray. The console will appear and Duke will wave. 

You can also check your version of the JAR file using the "jarsigner –verify" command. If the class had been changed there you would see the output:

```
jarsigner: java.lang.SecurityException: SHA1 digest
     error for
     oracle/forms/demos/util/RegisterCertificate.class
```

Figure 12 – Jarsigner –verify output for an altered JAR file

If a JAR file has been altered this way you will need to re-sign it, ensuring that the correct class files are included. The new version of the JAR file can then be deployed to the application server machine. The next time a client machine runs the application, the new JAR file will automatically be downloaded to replace the "corrupted" JAR file in the JInitiator cache.

PROBLEMS WITH MULTIPLE SIGNING IDENTITIES

There is currently a restriction within Oracle9*i* Forms. If a Forms application is started within JInitiator or the Sun Java Plug-in, which uses code signed with multiple certificates (as would be the case with a Forms application in combination with your own PJC), the plug-in will not recognize the second certificate the first time that it is presented with it. This can cause a Forms application to apparently fail to initialize. This restriction will be lifted in a later release of Forms, but meanwhile there are three workarounds to it:

1. Use the same certificate for all JAR files

The current Forms restriction is associated with an application that has multiple certificates. So one of the simplest workarounds is to re-sign the forms JAR file (e.g. f90all.jar) with your own certificate, which is the same one you use to sign any custom code. The disadvantage with this method is that you will need to carry out this operation each time you consume a new patch of Forms.

In the circumstance where your custom JAR contains Oracle classes such as **oracle.ewt.*** you will have to use this technique to sign both the Forms JAR and your own JARs with the same certificate. This is due to a restriction within 1.3(+) of the JVM where all classes within the same Java package must be signed using the same certificate. (see the Oracle9*i*DS Forms Release notes for more information on this issue)

The process of resigning a Forms JAR file is as follows (We'll use f90all.jar as an example). Remember to take a backup copy of the original f90all.jar before re-signing:

i) Create a temporary directory, for example *c:\temp\jar*. Open a command prompt session at that directory

ii) Unpack the f90all.jar (assuming this is located in *c:\ids\forms90\java*) into this directory using the following jar command[10]:

```
c:\jdk131\bin\jar –xvf c:\ids\forms90\java\f90all.jar
```

Figure 13 – Using the jar utility to unpack a JAR file

iii) Delete the *c:\temp\jar\meta-inf* directory created by the unpacking of the JAR.

iv) Repackage the JAR file in the *c:\temp\jar* directory using the jar command again. For clarity we'll use the new name f90all_resigned.jar:

```
c:\jdk131\bin\jar –cf f90all_resigned.jar *.*
```

Figure 14 – Using the jar utility to re-create a JAR file

v) Sign the new version of the Forms JAR file with your custom signing identity:

[10] For help on the various command line switches to the jar command, type "jar" at the command prompt without any arguments.

```
c:\jdk131\bin\jarsigner
          -keystore c:\jdk131\certificates\keystore
          -storepass keystore_pwd
          -keypass pjc_identity_pwd
          f90all_resigned.jar
          pjc_identity
```

Figure 15 – Re-Signing the new version of the Forms f90all.jar file with the PJC_IDENTITY

vi) Copy the new JAR file to the */forms90/java* directory on the application server machine for deployment, and either rename it to f90all.jar or alter your formsweb.cfg configurations to refer to f90all_resigned.jar rather than f90all.jar

2. Register Your Custom Signing Certificate Separately

The next technique for overcoming the multiple certificate problem involves adding a small code fragment to the JAR file and some changes to the deployment templates (This is discussed in detail later in document).

3. Install the Certificate on Each Client

The third, and probably least attractive, workaround is to install the specific X.509 certificate that you use for signing you custom code would be to install the certificate onto each client that will run the application. This could be a manual process of physically importing the certificate onto each machine in turn, or could be accomplished by re-packaging JInitiator with the certificate pre-installed.

This approach is not recommended given the simplicity of Methods 1 and 2.

WORKING AROUND THE MULTIPLE CERTIFICATE IMPORT PROBLEM

In thisl section of the paper we'll go through the alterations that you will have to make on your application server if you are taking the second approach to this problem.(2. *Register Your Custom Signing Certificate Separately*).

If you have chosen to manually import certificates onto the client, or to re-sign the Forms JAR files with your own signing identity you can disregard this section.

Changes to your Java code

For this workaround you will need to create a small applet within the JAR file (add this before you sign it!). The job of this applet is to be run standalone, separately from the Forms applet, to trigger the certificate registration process in JInitiator 1.3 (or the Sun Java Plug-In). The applet code is generic, and for safety, you can include it into all of your signed JAR files. The Java package name and class name can be those shown here or your own.

```
package oracle.forms.demos.util;
import java.applet.Applet;

public class RegisterCertificate extends Applet
{
  public RegisterCertificate()
  {
  }

  public void init()
  {
    //Basically do nothing - just output a
    //message to the Java console
    System.out.println("RegisterCertificate complete");
  }

}
```

Figure 16 – Source code for the RegisterCertificate Applet

Changes to the Template HTML file

The principle behind the workaround is to access the signing identity in a standalone fashion, before it is used in combination with the Forms JAR file(s). The simplest way to ensure that this always happen, and that it will continue to happen if the certificate is changed, is to add an extra applet tag to the template HTML files used by your Forms applications.

For an un-customized Forms install this will be the file basejini.htm in the *forms90/server* directory.

Take a copy of this file, for instance calling the new version signedjini.htm. add the following html fragment to this file above the line that reads "**<!-- Forms applet definition (start) -->**"

```
<!-- Registration applet definition (start) -->
<OBJECT classid="%jinit_classid%"
        codebase="/forms90/jinitiator/%jinit_exename%"
        WIDTH="0"
        HEIGHT="0"
        HSPACE="0"
        VSPACE="0">
<PARAM NAME="TYPE"       VALUE="%jinit_mimetype%">
<PARAM NAME="CODEBASE"   VALUE="%codebase%">
<PARAM NAME="CODE"       VALUE="%pjcRegisterApplet%" >
<PARAM NAME="ARCHIVE"    VALUE="%pjcArchive%" >
<COMMENT>
<EMBED SRC="" PLUGINSPAGE="%jinit_download_page%"
        TYPE="%jinit_mimetype%"
        java_codebase="%codebase%"
        java_code="%pjcRegisterApplet%"
        java_archive="%pjcArchive%"
        WIDTH="0"
        HEIGHT="0"
        HSPACE="0"
        VSPACE="0"
>
<NOEMBED>
</COMMENT>
</NOEMBED></EMBED>
</OBJECT>
<!-- Registration applet definition (end) -->
```

Figure 17 – Registration Applet Tag

This tag will create a second applet on the page. As it appears before the Forms applet tag, JInitiator will load this applet first and in doing so will encounter the signing certificate on its own. At this point the expected trust dialog will appear, and the user will be able to "Grant this session" or "Grant always", trusting the certificate. Then, when the Forms Applet is loaded, the code will already be trusted and the restriction bypassed.

Note that this new applet has a size of 0x0 pixels, so nothing shows up on the page. The user just sees the trust dialog, and indeed, only sees it the very first time they run the page.

Changes to the Formsweb.cfg file.

As well as changing the template file we have to tell the Forms Servlet to use this template, and what values to use for the new substitution variables **%pjcRegisterApplet%** and **%pjcArchive%** that are used in the new applet definition:

- *pjcRegisterApplet* is the name of the new applet class that we added to the PJC JAR file for this purpose. So if you used the example RegisterCertificate applet, as defined above, the value for this will be "`oracle.forms.demos.util.RegisterCertificate`"
- *pjcArchive* is the name of your JAR file that has been signed and contains the pjcRegisterApplet class.

Along with these two variables you will have to change the value of the **baseHTMLjinitiator** value in the formsweb.cfg file for this profile to point to your custom template.

So your config entry in the formsweb.cfg file might look like this:

```
[signed_app]
pageTitle=Oracle9iAS Forms – Application with Client Access
form=filewriter.fmx
archive_jini=f90all_jinit.jar,signedPJC.jar
pjcRegisterApplet=oracle.forms.demos.util.RegisterCertificate
pjcArchive=signedPJC.jar
baseHTMLjinitiator=signedjini.htm
width=743
height=600
```

Figure 18 – Config entry adapted for "Pre-registration"

COMMON QUESTIONS AND ANSWERS

Some of the more common questions about JAR file signing for Forms are covered in this section.

Can I use the Forms certificate to sign my own JAR files?

No, the private key of the Forms certificate is not distributed.

Can I re-sign the Forms JAR files with my own Certificate?

You can re-sign all of the Forms JAR Files except f90all_jinit.jar, which is a specially compressed version of f90all.jar. You will have to re-apply your signing whenever you upgrade or patch Forms.

Do I need to create a different signing identity for each of my JAR files?

No, you can use a single identity to sign multiple JAR files.

If my clients upgrade their JInitiator version, will they have to re-import my custom certificates?

Yes. Each install of JInitiator has its own keystore. After an upgrade, the new version of JInitiator will prompt the user again to trust the self signing identity.

Is there a way that I can embed my certificates into the installable version of JInitiator, before making it available to my clients for download?

There is no in-built mechanism within JInitiator or the Java Plug-in to do this. However, it is possible to "wrap" the JInitiator installation executable inside another installer script that first installs JInitiator and then imports your certificates using the *keytool –import* command.

Do I have to use a certificate from a Certificate Authority to sign my files?

No. Using a certificate from a Certificate Authority will establish a chain of trust for your signed code, but it is not a requirement.

Can a JAR file be signed with more than one certificate?

No, a JAR can only be signed with a single certificate.

Does the information in this document apply if I am using JInitiator 1.3 with Forms 6i?

Yes. The information in this document is specific to the JInitiator version not the Forms version.

SUMMARY

The process of signing custom code in JAR files to allow access outside of the Java Applet "sandbox" is simple to carry out, with tools that are readily available.

With the advent of JInitiator 1.3 the deployment of certificates for self signed JAR files becomes essentially automatic, requiring only a minimum of user interaction to achieve, opening the door to the deployment of systems with a much wider range of functionality at no additional deployment cost.

Oracle9*i* Forms: JAR File Signing for JInitiator 1.3
Version 1.2
September 2002
Author: Duncan Mills

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com