Verifying the integrity of the SignOn Library

Revision History

Version	Date	By	Comments
1	07/01/10	Brian Graversen	Initial release of document

Introduction

When a client application makes use of the SignOn Library, it is important that it verifies the integrity of the library before using it. The end-users credentials (username, password) is supplied to the SignOn Library, so to prevent a malicious 3rd party from injecting a modified plugin or library dll, all components in the SignOn Library framework has been signed.

The SignOn Library verifies the integrity of all plugins, so the client application is only responsible for verifying the SignOn Library itself. Below we will outline how this can be done from C#, Visual C and Java. Other languages will have similar functionality, but these three examples covers the three different ways of accessing the SignOn Library (directly through .NET, through COM and through a Java wrapper).

Verifying the SignOn Library from C# (.NET)

The C# Client application found in the "Sample Code" folder bundled with the SignOn Library contains a full implementation of the needed verification. We will outline the steps taken, and show snippets of the code to highlight the points.

1. Start by locating the DLL that contains the SignOn Library code

```
Type type = typeof(SignOnFacade);
string location = type.Assembly.Location;
```

2. Start by verifying the cryptographic signature on the SignOn Library DLL. One way to do this, is to use the WinVerifyTrust function found in the Windows API. Unfortunatly no native .NET implementation exists, so we will have to use the interoperability functionality of .NET to call the native code in Windows. The WinVerifyWrapper.cs file in the SignOn Library source code contains everything needed for this purpose. We will highlight the steps taken

   a) Construct a WINTRUST_FILE_INFO structure, mapped identical to the specs

      http://msdn.microsoft.com/en-us/library/aa388206(VS.85).aspx

      This structure should then point to the file location found above.

   b) Construct a WINTRUST_DATA structure, mapped identical to the specs

      http://msdn.microsoft.com/en-us/library/aa388205(VS.85).aspx

      The file-pointer structure created above is then inserted into this structure

   c) Finally call WinVerifyTrust using the WINTRUST_DATA structure created

      http://msdn.microsoft.com/en-us/library/aa388208(VS.85).aspx

```
[DllImport("wintrust.dll", ExactSpelling = true,
                  SetLastError = false, CharSet = CharSet.Unicode)]
static extern UInt32 WinVerifyTrust(
     [In] IntPtr hwnd,
     [In] [MarshelAs(UnmanagedType.LPStruct)] Guid pgActionID,
     [In] WinTrustData pWVTData
);

UInt32 result = WinVerifyTrust(INVALID_HANDLE_VALUE,
                               WinVerifyWrapper.WINTRUST_ACTION_GENERIC_VERIFY_V2,
                               new WinTrustData(location));
```

3. If the cryptographic signature is okay, we continue by extracting the certificate used to sign the DLL, and validate it against the certificate that we trust (the trusted certificate is part of

the SignOn Library bundle).

```
X509Certificate trustedCertificate = ...
X509Certificate certificate = X509Certificate.CreateFromSignedFile(location);
if (certificate.Equals(trustedCertificate)) { ... }
```

Verifying the SignOn Library from Visual C (COM)

Unlike C#, where we know which DLL we access, COM calls are weakly linked, so identifying exactly where the operation is performed can be somewhat difficult. In fact COM server components can be located on a different machine altogether. Since the SignOn Library is bundled with the client application, we can safely assume that the COM operation will be performed locally, and we should be able to find the SignOn Library DLL, and verify the file like in the C# sample code.

The basic flow is outlined below, intermixed by snippets of sample code. To see a full working implementation, look at the Visual C COM Client implementation in the "Sample Code" folder bundled with the SignOn Library.

1. Start by extracting the GUID of the COM component to call. We can use the SignOnFacade as the entry point for this lookup

   ```
   GUID guid = __uuidof(SignOnLibrary::SignOnFacade);
   ```

2. A GUID is on the form "{06210E88-01F5-11D1-B512-0080C781C384}", and we can use this information to locate the COM registration information in the Windows Registry. Given a GUID in the form above, the information we need can be extracted from

   ```
   HKEY_CLASSES_ROOT\CLSID\{06210E88-01F5-11D1-B512-0080C781C384}\InprocServer32\CodeBase
   ```

3. The information in registry will look like this (assuming that library is in the c:\libs). Note that the "CodeBase" key is only present if the assembly was registered using the /codebase argument. If the Global Assembly Cache (GAC) is used, the location of the DLL should be extracted from the GAC.

   ```
   file:///C:/libs/SignOnLibrary.DLL
   ```

4. Strip the file:/// prefix, so we just have the path+filename of the SignOnLibrary.dll file
5. Verifying the integrity of the file is a 2-step operation. First we will validate the cryptographic signature, and next we will verify that the signing certificate is one we trust.
6. WinVerifyTrust() is a Windows API function used for verifying a cryptographic signature. The documentation for this function can be found at

   http://msdn.microsoft.com/en-us/library/aa388208(VS.85).aspx

   ```
   WINTRUST_DATA winTrustData;
   [ ... snip setup of winTrustData ... ]
   GUID policy = WINTRUST_ACTION_GENERIC_VERIFY_V2;
   LONG result = WinVerifyTrust(NULL, &policy, &winTrustData);
   ```

   In the "Visual C COM Client" implementation found in the "Sample Code" a full implementation of a call to WinVerifyTrust can be found.

7. Finally we will have to extract the certificate used to sign the DLL, and compare it to the certificate that we trust (the signing certificate is part of the SignOn Library bundle).
   a) CryptQueryObject is used to extract information from the DLL
   b) CryptMsgGetParam is used to extract the certificate from the above data

c) The extracted certificate is compared to the trusted certificate.

Again a full implementation of this step can be found in the “Visual C COM Client”.

http://msdn.microsoft.com/en-us/library/aa380264(VS.85).aspx
http://msdn.microsoft.com/en-us/library/aa380227(VS.85).aspx

Verifying the SignOn Library from Java

The Java wrapper verifies the integrity of the dll file containing the JNI code that uses COM to talk to the SignOn library. The JNI in turn, verifies the integrity of the SignOnLibrary dll that it uses. It is therefore only necessary to verify the integrity of the SignOnWrapper.jar file.

The jar file has been digitally signed, using the same certificate and secret key used for signing the dll files. Verifying the signature is done by first loading the jar file. Javas classloader refuses to load a signed jar file with an invalid signature. It is therefore only neccesary to verify that the JAR is signed, and that the correct certifcate has been used. That can be done in the following way.

```
CodeSource source = SignOnFacade.class.getProtectionDomain().getCodeSource();

Certificate[] certs = source.getCertificates();
if (certs == null || certs.length != 1) {
        System.out.println("Unexpected number of certificates on SignOnJni jar, was " +
                        (certs == null ? 0 : certs.length) + " but expected 1.");
        return;
}

if (!trustedCert.equals(convertToHex(certs[0].getEncoded()))) {
        System.out.println("The certificate used for signing signOnwrapper.jar " +
                        "is incorrect.");
        return;
}
```

The first if statement verifies that there is one and only one signature on the jar file. The second statement verifies that the expected certificate has been used. In the example application the certificate to look for is hardcoded, but any means of getting it can be used.