PKCS#11 Wrapper for Java

from IAIK
http://jce.iaik.tugraz.at

by

Karl Scheibelhofer
Karl.Scheibelhofer@iaik.tugraz.at

Version 1.2.16
16 February 2006

Programmer's Manual

Introduction

The IAIK PKCS#11 Wrapper for Java [1] is a programming library that allows Java programs to access PKCS#11 [2] modules. Such PKCS#11 modules provide access to cryptographic hardware like smart cards or hardware security modules. Normally, the manufacturers of cryptographic hardware provide PKCS#11 modules for their products. There is no standard API for Java to access such modules.

The library consists of two major parts: the Java part and the native part. The application does not need to access the native part itself; it only uses the Java classes and interfaces of the library. Internally, the Java part of the library uses the native part to connect to the PKCS#11 module of the cryptographic hardware. This is necessary, because a PKCS#11 module is a native DLL or shared library.

The following paragraphs show how to use the Wrapper by stepping through a simple example. You can find more demo programs in the `demo` subdirectory. The remaining part of this document informs in more detail on how to use the library.

The documentation assumes that the reader is familiar with the basic principles of PKCS#11. There is a general overview chapter in the PKCS#11 specification from RSA Laboratories [2]. It gives a brief introduction into the basics of PKCS#11.

Write a simple Program

First, we write a simple program that looks like the class below. You can find the source of this example in the file `demo/src/demo/pkcs/pkcs11/ModuleInfo.java`.

```
package demo.pkcs.pkcs11;

import iaik.pkcs.pkcs11.Module;
import iaik.pkcs.pkcs11.Info;

public class ModuleInfo {

  public static void main(String[] args) {
    if (args.length == 1){
      try {
        Module pkcs11Module = Module.getInstance(args[0]);
        pkcs11Module.initialize(null);

        Info info = pkcs11Module.getInfo();
        System.out.println(info);

        pkcs11Module.finalize(null);
      } catch (Throwable ex) {
        ex.printStackTrace();
      }
    } else {
      printUsage();
      System.exit(1);
    }
  }

  protected static void printUsage() {
    System.out.println("ModuleInfo <PKCS#11 module name>");
    System.out.println("e.g.: ModuleInfo pk2priv.dll");
  }
}
```

This program will load a PKCS#11 module and print information about it to the console.

Compile the program

Use your Java compiler to compile the program like any other Java program. You must just include the `iaikPkcs11Wrapper.jar` file. The compile command will look like this, if you are in the `demo` directory.

```
javac -classpath ../java/lib/iaikPkcs11Wrapper.jar -sourcepath src -d
classes src/demo/pkcs/pkcs11/ModuleInfo.java
```

Keep in mind that this is a single command line. Write it in one line without a line break.

Run the Program

To run the program, you have to find out the name of your PKCS#11 module. For instance, if you use GemSAFE smart cards, this will be `pk2priv.dll`. For other products, the module will have a different name. Refer to the documentation of your hardware. If you have absolutely no glue, try one of these:

```
iButton - dspkcs.dll
Schlumberger Cryptoflex or Cyberflex Access - slbck.dll
GemSAFE – pk2priv.dll or gclib.dll
SeTec - SetTokI.dll
ActiveCard - acpkcs.dll
ID2 – id2cbox.dll
Eracom – cryptoki.dll
G&D StarCos SPK – aetpkss1.dll
Rainbow iKey 3000 – aetpkss1.dll
Rainbow iKey 1000 or 2000 and DataKey – dkck201.dll
Rainbow CryptoSwift HSM – iveacryptoki.dll
Oberthur AuthentIC - AuCryptoki2-0.dll
Orga Micardo - MicardoPKCS11.dll
IBM MFC - CccSigIT.dll
Utimaco SafeGuard - pkcs201n.dll
SmartTrust - smartp11.dll
Aladdin eToken – eTpkcs11.dll
Eutron CryptoIdentity or Algorithmic Research MiniKey – sadaptor.dll
TeleSec – pkcs11.dll
nCipher nFast or nShield – cknfast.dll
Chrysalis – cryst201.dll
IBM 4758 – cryptoki.dll
Siemens (HiPath SIcurity Card API) – siecap11.dll
A-Sign Premium – psepkcs11.dll
Netscape or Mozilla – softokn3.dll
ASE Card – asepkcs.dll
Apollo OS card from SC$^2$ Technology – Apollo_Cryptoki.dll
IBM Client Security Software for TPM (TCP) – ibmpkcst.dll
SUN Crypto Accelerator SCA1000 – libpkcs11.so
SUN Crypto Accelerator SCA4000 – libvpkcs11.so
```

The library needs its native part to work. Thus, you have to inform the Java VM where to find the native part (the `pkcs11wrapper.dll` or `libpkcs11wrapper.so`). You can do this at the command line when starting the program. The command will look like this, if you are in the `demo` directory. Replace the `pk2priv.dll` with the name of your PKCS#11 module.

```
java -classpath classes;../java/lib/iaikPkcs11Wrapper.jar
-Djava.library.path=../native/platforms/win32/release
demo.pkcs.pkcs11.ModuleInfo pk2priv.dll
```

Once again, keep in mind that this is a single command line. Write it in one line without a line break. The output of the program will look like this.

```
Cryptoki Version: 2.00
ManufacturerID: Gemplus
Library Description: PKCS#11 Private Cryptoki
Library Version: 3.00
```

If you get an exception that looks like this

```
java.io.IOException: The specified module could not be found.
p11module.dll
        at iaik.pkcs.pkcs11.wrapper.PKCS11Implementation.connect(Native
Method)
        at
iaik.pkcs.pkcs11.wrapper.PKCS11Implementation.<init>(PKCS11Implementation.j
ava:118)
        at
iaik.pkcs.pkcs11.wrapper.PKCS11Connector.connectToPKCS11Module(PKCS11Connec
tor.java:53)
        at iaik.pkcs.pkcs11.Module.getInstance(Module.java:139)
        at demo.pkcs.pkcs11.ModuleInfo.main(ModuleInfo.java:37)
```

the VM and the Operating System did not find the specified PKCS#11 module. You can try to specify the module with its full file-path. If this does not help either, ensure you have the right file.

You have just written a Java program that uses PKCS#11. To find more sophisticated examples, have a look in the `demo\src\demo\pkcs\pkcs11` directory. If you are dealing with already personalized smart cards, cards that already have key-pairs and certificates on them, `demo.pkcs.pkcs11.GetInfo` is a good example to start with. It shows you a lot of information about your card: version information, serial number, supported algorithms, keys, certificates and much more. The Sample Code and Demos section below provides some more information about the demos.

Basic usage of PKCS#11

The basic usage of PKCS#11 is roughly always the same. First, you connect to a concrete PKCS#11 module.

```
Module module = Module.getInstance("slbck.dll");
```

You have to replace `slbck.dll` with the name of the PKCS#11 module for your hardware (see Run the Program). Before the application can start using the module, it has to initialize the module.

```
module.initialize (new DefaultInitializeArgs());
```

Then you can select a slot.

```
// list all slots (readers) in which there is currenlty a token present
Slot[] slotsWithToken =
    module.getSlotList(Module.SlotRequirement.TOKEN_PRESENT);
```

Now, you can take one of these slots (for example the first one) and get the token.

```
Token token = slotsWithToken[0].getToken();
```

Then you open a session on this token. In this sample, a read-only session, what means that you cannot write data to the token or manipulate data on it, but you can do cryptographic operations like signing.

```
Session session =
    token.openSession(Token.SessionType.SERIAL_SESSION,
                      Token.SessionReadWriteBehavior.RO_SESSION,
                      null,
                      null);
```

If you want to sign some data, you would try to find a key on the token.

```
// we search for a RSA private key which we can use for signing
RSAPrivateKey searchTemplate = new RSAPrivateKey();
searchTemplate.getSign().setBooleanValue(Boolean.TRUE);

// search for a key
session.findObjectsInit(searchTemplate);
Object[] matchingKeys;
RSAPrivateKey signatureKey;
if ((matchingKeys = session.findObjects(1)).length > 0) {
  signatureKey = (RSAPrivateKey) matchingKeys[0]);
} else {
  // we have not found a suitable key, we cannot contiue
}
// do not forget to finish the find operation
session.findObjectsFinal();
```

You can also specify more attributes in the search template, to get one specific key; for instance, a key ID, a label or the key's modulus.

Now, you can sign some data.

```
byte[] data = ...;
// select the signature mechanism, ensure your token supports it
Mechanism signatureMechanism = Mechanism.SHA1_RSA_PKCS;
// initialize for signing
session.signInit(signatureMechanism, signatureKey);
byte[] signatureValue = session.sign(data);
```

The resulting signature value is a RSA signature according to PKCS#1 (v 1.5). For more advanced code samples, please have a look at the included demos.

Usage Details

Include the JAR-Files, DLL and Shared Library

To simply use the IAIK PKCS#11 wrapper, you have to include the `iaikPkcs11Wrapper.jar` file in your `CLASSPATH` or in some other directory where your Java VM can find it; for example, you can put it in the `jre/lib/ext` directory. You must also put the native part of the wrapper into a directory where the system or the Java VM can find it. For instance, the search path of the operating system can be set via the PATH environment variable on Windows systems or via the LD_LIBRARY_PATH environment variable on UNIX systems. Alternatively, you can tell your VM directly where to search for libraries. The `java.library.path` property holds such search paths for the VM. For example, you can set it via the java command line like `-Djava.library.path=../../native/platforms/win32/Release`. The native part is the DLL called `pkcs11wrapper.dll` for Windows, or the shared library `libpkcs11wrapper.so` for Unix systems. To find the appropriate version of this DLL or shared library go to the `platforms/<platform name>/release` directory, where `<platform name>` is the name of your platform; for instance Win32 for Windows NT or Windows 2000. You should not take the version that is in the `debug` subdirectory. This is the debug version of the native part, which is compiled with DEBUG defined, and it generates a lot of debug output to standard out, which is only useful for debugging.

Other Platforms and JDKs

If you want to use the wrapper with JDK 1.1.8, you need to recompile the native part. To recompile it for JDK 1.1.8, you have to replace the `jvm.lib` file in the `platforms/<platform name>/lib` with the corresponding `.lib` file of your target Java VM; for example, it is called `javai.lib` in SUN's JDK 1.1.8 and resides in the `lib` subdirectory of the JDK's home. Moreover, you need to modify the projects files or the `Makefile` to link to this library. In the `Makefile`, this is an ordinary search and replace. In the MS Visual C++ projects have a look into Project/Settings…/(Select "All Configurations" at the left top)/Linker/Object-/Library-Modules. Change `jvm.lib` to `javai.lib`. This should be sufficient to rebuild the DLL. If you decide that you do not need to support callbacks in your PKCS#11 based application, or if the driver of your PKCS#11 device does not support callbacks, you can compile the DLL with `NO_CALLBACKS` defined. In this case, you do not need the `jvm.lib` or `javai.lib` at all. Callbacks are rarely used in practice. Most applications do not need them.

Porting the Native Part to another Platform

If you want to port the native part of this library to another platform, I suggest doing it like this. Choose one of the existing platforms that is most similar to the new platform. Make a copy of the complete platform directory and give it an appropriate name; for example, make a copy of the `platforms/linux` directory and call it `platforms/solaris` for instance. Then adapt the `platform.h` and `platform.c` files of the copy to fit to your new platform. That should be everything. You can also use the Makefile of one of the existing platforms as template for building the new platform target.

Lower Level Access and Small Footprint

If you want to access PKCS#11 on a more low level, or if you need a minimum footprint system, you can directly build upon the `iaik.pkcs.pkcs11.wrapper` package, which is a straightforward mapping of the PKCS#11 standard to Java. One might even throw out classes not used by the application or library in a special use-case.

Sample Code and Demos

Have a look into the `demo/src/demo/pkcs/pkcs11` directory to see some example code. It should be relatively easy to use for one who is familiar with PKCS#11. Some of the demo and test programs need the IAIK JCE library to compile and run. You can download an evaluation version from http://jce.iaik.tugraz.at. You just need to register (for free). The wrapper itself does not need the JCE library.

All demos are precompiled. For each demo, there is a batch file to run it from the command line. These batch files assume that there is an appropriate `java.exe` in you search path. If not, you can modify the `setEnvironment` batch file.

You may start with the `GetInfo` demo. It displays information about PKCS#11 tokens; information about the PKCS#11 module, information about the slots, information about the tokens and the objects on the tokens. You may also use the `DumpObjects` demo to dump the contents of a token into a directory.

If you have a blank token, you may want to import keys. You can download the key and certificate of a PKCS#12 file to the token using the `DownloadPrivateKey` demo. After downloading a key, you may use it for signing or encryption.

Alternatively, you can generate a new key-pair on the token. You may use the `GenerateKeyPair` demo to generate a new key-pair. Thereafter, you can start the `SignCertificateRequest` demo to create a PKCS#10 certificate request that you can send to a CA. After receiving the certificate, you can import it with the `ImportCertificate` demo.

To sign some data, you can use the `SignAndVerify` or the SignPKCS7 demo. `SignAndVerify` crates a raw signature value, and SignPKCS7 creates a signature in PKCS#7 (version 1.5) format. For verifying raw signatures, you can use `VerifySignature`; for PKCS#7 signatures, you can use `VerifyPKCS7SignedData`. For `VerifySignature`, you need to provide the certificate, which you can get from the card using the `DumpObjects` demo.

References

[1] IAIK Java Cryptography Toolkits,
http://jce.iaik.tugraz.at/

[2] PKCS#11, Version 2.11, by RSA Laboratories,
http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html

[3] Java 2 Platform, by Sun Microsystems,
http://java.sun.com/j2se/

[4] Java Native Interface 1.1, by Sun Microsystems,
http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html