Pheox

Java CryptoAPI

JCAPI User's Guide

THIS PAGE INTENTIONALLY LEFT BLANK

PHEOX JAVA CRYPTOAPI

JCAPI User's Guide

Pheox AB
Ställverksvägen 3
SE-37439 Karlshamn
Sweden
www.pheox.com

THE LICENSE IS A NON-TRANSFERABLE NON-EXCLUSIVE COMPANY LICENSE.

THE LICENSED COMPANY MUST ADHERE TO AND RESPECT THE TERMS GIVEN BY THIS AGREEMENT, OTHERWISE WILL THE LICENSE EXPIRE WITHOUT ANY PRIOR NOTICE FOR THE LICENSED COMPANY. THE LICENSE WILL ALSO EXPIRE IF THE LICENSED COMPANY IS CLOSED DOWN.

A COMPANY IS WITHIN THIS AGREEMENT DEFINED TO BE A SUBJECT THAT IS REGISTERED AND GRANTED BUSINESS ACTIVITIES BY THE AUTHORITIES IN THE COUNTRY IN WHICH THE SUBJECT OPERATES.

THE LICENSED COMPANY IS GRANTED THE RIGHTS TO:

- INSTALL AN UNLIMITED NUMBER OF JCAPI INSTANCES WITHIN THE LICENSED COMPANY.
- USE JCAPI FOR ANY NUMBER OF DEVELOPERS DURING DEVELOPMENT OF A PRODUCT THAT IS DEVELOPED BY THE LICENSED COMPANY.
- USE AND LINK JCAPI INTO AN UNLIMITED NUMBER OF PRODUCTS, AND COPIES OF THE PRODUCTS, THAT IS DEVELOPED BY THE LICENSED COMPANY AS LONG AS THE JCAPI INTERFACE IS NOT DIRECTLY OR INDIRECTLY EXPOSED BY THE PRODUCTS.
- USE AND LINK JCAPI INTO AN UNLIMITED NUMBER OF PRODUCTS, AND COPIES OF THE PRODUCTS, THAT IS OWNED BY THE LICENSED COMPANY AS LONG AS THE JCAPI INTERFACE IS NOT DIRECTLY OR INDIRECTLY EXPOSED BY THE PRODUCTS.

THE LICENSED COMPANY MAY NOT IN ANY WAY SELL OR TRANSFER THE JCAPI LICENSE OR JCAPI PRODUCT ITSELF TO ANOTHER PARTY. THE LICENSED COMPANY MAY NOT USE REVERSE ENGINEERING OR ANY OTHER METHODS/TOOLS/WAYS TO EXAMINE/DISCOVER THE JCAPI CODE. THE LICENSED COMPANY MAY NOT LINK JCAPI INTO A PRODUCT THAT IS SOLD OR TRANSFERRED TO ANOTHER PARTY AS A SOFTWARE DEVELOPMENT KIT (SDK).

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL PHEOX AB BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

THE JCAPI PRODUCT IS SUBJECT TO EXPORT CONTROL IF EXPORTED OUT OF THE EUROPEAN UNION.
THE JCAPI PRODUCT IS OWNED BY PHEOX AB, SWEDEN

Table of Content

Chapter

1

Introduction to JCAPI

Explains the basis of Pheox Java CryptoAPI and how it relates to Microsoft's CryptoAPI and Java Cryptography Extension.

The Pheox product *Java CryptoAPI* (JCAPI) is a *Java Cryptography Extension* (JCE) provider that provides access to key- and certificate stores on Microsoft operating systems.

The JCAPI software consists of a JAR file which in turn holds the following components:

- A set of Java classes.
- A *Dynamic Linked Library* (DLL) file named `JCAPI32.DLL` (on 32-bit systems), or `JCAPI64.DLL` (on 64-bit systems).

Microsoft CryptoAPI

All cryptographic operations are performed by the native *Microsoft CryptoAPI* (MS CAPI) layer. The MS CAPI layer supports pluggable *Cryptographic Service Providers* (CSP) i.e. native code written by different vendors that implements crypto interfaces provided by MS CAPI to support cryptographic algorithms and functions.

Pheox Java CryptoAPI

The JCAPI Java classes responsible for cryptographic operations will use the *Java Native Interface* (JNI) technology to delegate an operation to the JCAPI DLL, which in turn will use a specific MS CAPI enabled CSP to complete the actual operation. JCAPI is thus a JCE compliant light-weight mediator library that delegates all crypto related operations to a specific CSP available in the native MS CAPI layer.

Some CSPs do not provide means to access private keys stored on hardware tokens through certain MS CAPI functions required by JCAPI. In these occasions, JCAPI will use PKCS#11 (if available by the CSP) internally to access these keys.

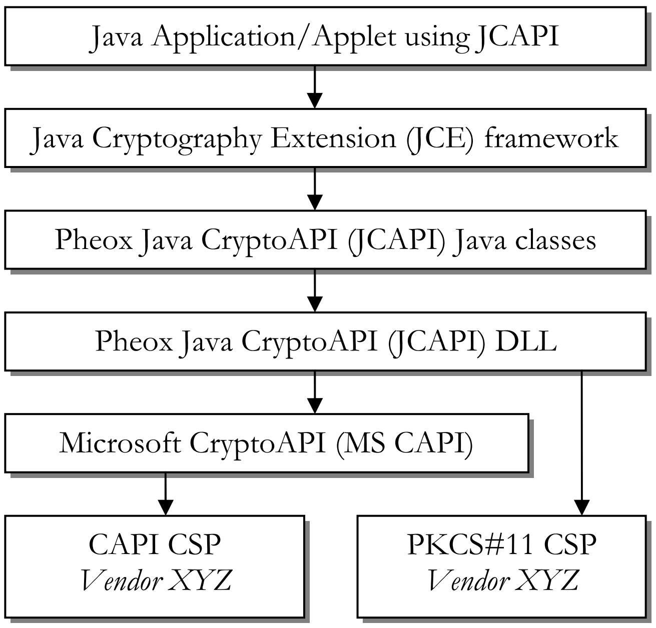

Java Application/Applet using JCAPI
Java Cryptography Extension (JCE) framework
Pheox Java CryptoAPI (JCAPI) Java classes
Pheox Java CryptoAPI (JCAPI) DLL
Microsoft CryptoAPI (MS CAPI)
CAPI CSP
Vendor XYZ
PKCS#11 CSP
Vendor XYZ

FIG 1.1 Application layers

JCAPI supply *Service Provider Interface* (SPI) implementations for the following Java classes:

- `java.security.KeyStore` – is implemented by `com.pheox.jcapi.JCAPIKeyStore`
- `java.security.Signature` – is implemented by `com.pheox.jcapi.JCAPISignature`
- `javax.crypto.Cipher` – is implemented by `com.pheox.jcapi.JCAPIAsymmetricCipherDynamic` and `com.pheox.jcapi.JCAPISymmetricCipherDynamic`
- `java.security.SecureRandom` – is implemented by `com.pheox.jcapi.JCAPISecureRandom`
- `java.security.MessageDigestSpi` – is implemented by `com.pheox.jcapi.JCAPIMessageDigestDynamic`

This means that you can use JCAPI as an ordinary JCE provider to:

- Add, remove and access certificates and key entries stored in a MS CAPI system store.
- Create and verify signatures based on RSA or DSA key pairs accessed through MS CAPI and PKCS#11.

- Encrypt and decrypt data based on asymmetric RSA key pairs accessed through MS CAPI and PKCS#11.

- Encrypt and decrypt data based on symmetric key algorithms such as DES, TripleDES, AES, RC2, and RC4 accessed through MS CAPI.

- Get secure random numbers through MS CAPI to be used for creating secure cryptographic keys and padding data. By default, MS CAPI uses a software based *Pseudo Random Number Generator* (PRNG), but true *Random Number Generators* (RNG) can be used as well through MS CAPI and JCAPI if supported hardware and MS CAPI drivers are installed.

- Create and verify message digests (hashed data) through MS CAPI using the algorithms MD2, MD4, MD5, SHA-1, SHA-256, SHA-384, and SHA-512.

Supported Features

These are the basic features supported by JCAPI:

- Add, remove, list and access X.509 certificates to/from a Microsoft system (certificate) store.
- Add, remove, access, import and export asymmetric RSA & DSA keys. *Note: private keys cannot be exported if they are marked as non-exportable by MS CAPI e.g. if the key resides on a hardware token.*
- Create signatures with RSA private keys that are either plain Java private key objects, or stored on disk or hardware tokens through MS CAPI or PKCS#11. The following algorithms are supported:
    - SHA512withRSA
    - SHA384withRSA
    - SHA256withRSA
    - SHA1withRSA
    - MD5withRSA
    - MD2withRSA
    - SHAMD5withRSA
    - NONEwithRSA
- Verify signatures with RSA public keys that are either plain Java public key objects, or resides in a X.509 certificate stored in a Microsoft system store.
- Create signatures with DSA private keys that are either plain Java private key objects, or stored on disk or hardware tokens through MS CAPI or PKCS#11. The following algorithms are supported:
    - SHA1withDSA
- Verify signatures with DSA public keys that are either plain Java public key objects, or resides in a X.509 certificate stored in a Microsoft system store.
- Encrypt data with RSA public keys that are either plain Java public key objects, or resides in a X.509 certificate stored in a Microsoft system store. The following algorithms, modes and paddings are supported:
    - RSA/ECB/PKCS1Padding
    - RSA/ECB/OAEPPadding

- Decrypt encrypted data with RSA private keys that are either plain Java private key objects, or stored on disk or hardware tokens through MS CAPI or PKCS#11.

- Wrap and unwrap symmetric- and asymmetric keys with RSA key pairs through MS CAPI and PKCS#11.

- Encrypt and decrypt data using symmetric keys through MS CAPI. The following algorithms, modes, paddings, and key lengths are supported:

Algorithm	**Modes**	**Padding**	**Key lengths**
AES	ECB, CBC	PKCS#5	128, 192, 256
3DES	ECB, CBC	PKCS#5	112, 168
DES	ECB, CBC	PKCS#5	56
RC2	ECB, CBC	PKCS#5	40 - 128
RC4	None	None	40 - 128

- Create and verify message digests (hashed data) through your preferred MS CAPI CSP. The following algorithms are supported by default:
    - MD2
    - MD4
    - MD5
    - SHA-1
    - SHA-256
    - SHA-384
    - SHA-512

- Get secure random numbers through MS CAPI, either generated through collected low level events in the operating system, or generated through hardware by third party CSP vendors.

- Built-in support for tested PKCS#11 CSP manufacturers that is compliant with the functions required by JCAPI. This means that a compliant CSP is guaranteed to work within the scope of JCAPI functionality. It also means that whenever access to a private key stored on a hardware token is required (i.e. during signing and decryption), JCAPI will automatically use PKCS#11 instead of the MS CAPI layer. Some CSP vendors do not implement proper support for some MS CAPI functions that is used for accessing a private key. In these cases, MS CAPI and JCAPI will fail to accomplish the operation. JCAPI supports the following list of PKCS#11 CSPs:

  - SafeSign CSP Version 1.0
  - SI_CSP
  - AR Base Cryptographic Provider
  - FTSafe ePass2000 RSA Cryptographic Service Provider
  - eToken Base Cryptographic Provider
  - SmartTrust Cryptographic Service Provider
  - Athena ASECard Crypto CSP
  - Datakey RSA CSP
  - Advanced Card Systems CSP v1.5
  - SafeNet RSA CSP

- Dynamically adding PKCS#11 CSPs into JCAPI. Even though a PKCS#11 CSP is not listed as a JCAPI supported CSP, it will in nearly all cases work with JCAPI. However, a dynamically added CSP will not be supported by us if problems arise, simply because we will not be able to reproduce the error condition without required hardware and software.

- Private key call-back interface for PKCS#11 providers. You can provide your own preferred Java call-back implementation to be called whenever a private key is accessed through PKCS#11 i.e. during signing and decryption through a PKCS#11 CSP that is supported by JCAPI or added dynamically by yourself. JCAPI provides a default implementation for a Swing-based call-back dialog where the user can enter the required PIN code.

- Use the PKCS#7 framework to encode and decode signed or enveloped data messages through MS CAPI.

- List, configure, and query MS CAPI system (certificate) stores. You can list all available system stores and configure JCAPI to use a certain system store for a specific type of certificate.

- Create and delete MS CAPI system (certificate) stores with arbitrary names.

- Configure where (system store registry location) certificates are stored and accessed from. You can choose between Current Service, Current User, Current User Group Policy, Local Machine, Local Machine Enterprise, Local Machine Group Policy, Services, and Users. Default location used is Current User.

- Create `java.security.KeyStore` instances which maps to one specific MS CAPI system store only. This is very handy when SSL/TLS is to be used for handling private keys and trusted certificates. The following additional key store types are supported:

- msks-MY
- msks-ROOT
- msks-KEYSTORE
- msks-TRUSTSTORE

- Use a MS CAPI system (certificate) store as an un-trusted store i.e. all certificates located in that store will be considered un-trusted by JCAPI and thus be rejected whenever they appear in another system store (or in a certificate chain).

- List all available MS CAPI CSPs and configure what CSP that shall be used by JCAPI for a specific cryptographic operation.

- Configure what RSA supported CSP that JCAPI shall use. JCAPI uses by default the following CSPs in preferred order:

    1. Microsoft Enhanced RSA and AES Cryptographic Provider
    2. Microsoft Enhanced Cryptographic Provider v1.0
    3. Microsoft Strong Cryptographic Provider
    4. Microsoft Base Cryptographic Provider v1.0

- Configure what DSA supported CSP that JCAPI shall use. JCAPI uses by default the following CSPs in preferred order:

    - Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider
    - Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
    - Microsoft Base DSS Cryptographic Provider
    - Microsoft DH SChannel Cryptographic Provider

- Set and get MS CAPI friendly names for certificates.

- Get MS CAPI friendly names for system (certificate) stores.

- Get detailed information about your PKCS#11 hardware token through the JCAPI PKCS#11 information class.

- List and use all supported algorithms supported by all MS CAPI CSPs.

- List and use all supported key lengths that are supported for a MS CAPI CSP for each algorithm.

- Get the key usage information about each DSA/RSA private key stored in a MS CAPI system store.

- Create a dynamic JCAPI cryptographic instance wrapped into one of the following Java Cryptography Extension (JCE) interfaces:
    - `javax.crypto.Cipher`
    - `java.security.KeyStore`
    - `java.security.MessageDigest`
    - `java.security.SecureRandom`
    - `java.security.Signature`

  Being able to dynamically create and wrap an arbitrary cryptographic algorithm in MS CAPI into a standardized JCE interface, is an extremely powerful feature.
  It gives the programmer the possibility to query MS CAPI in runtime about what algorithms that can be used for encryption and decryption (symmetric/asymmetric), and for creating signatures and message digests. When a desired algorithm has been found, it can then be wrapped into a standard JCE class.

- Full SSL/TLS support. Use JCAPI seamlessly with other SSL/TLS frameworks (JSSE etc.) with just a few lines of extra code. Using unprotected (exportable) private keys, and protected private keys stored in MS CAPI is fully supported.

- Base64 encode & decode data.

- JCAPI is supported on a wide range of Microsoft operating systems. Please see the *Compatibility* chapter for more information.

- JCAPI is supported on many Java versions. Please see the *Compatibility* chapter for more information.

- JCAPI is signed with a qualified code signing certificate issued by DigiCert that is trusted by all modern web browsers which makes it suitable in trusted applets.

Chapter 2

Compatibility

This chapter will describe for you what Microsoft operating systems and which Java versions that JCAPI can be successfully used with.

Supported Microsoft Operating Systems

JCAPI has been successfully tested and verified on the following versions of Microsoft Windows:

32-bit Operating Systems

Operating System	Comments
Windows 2000	No encryption/decryption is supported on a clean installation, and only RSA keys of a maximum length of 512 bits can be used when creating and verifying signatures[1]. In order to use all functions provided by JCAPI, and RSA keys of greater lengths, you have to either install the *High Encryption Pack* for Internet Explorer, or install *Service Pack 2* or higher. *Recommendation: Install Service Pack 2 or higher to avoid problems. Note: Java 1.5 will also require Service Pack 2 or higher.*
Windows Server 2003	All operations are supported by default.
Windows Server 2008	All operations are supported by default.
Windows Server 2008 RC2	All operations are supported by default.
Windows XP	All operations are supported by default.
Windows Vista	All operations are supported by default.
Windows 7	All operations are supported by default.

Windows 8	All operations are supported by default.

1. To be able to utilise greater key lengths and support for direct access to RSA keys for encryption/decryption, the *Microsoft Enhanced Cryptographic Provider* must be available. This is not the case on most Windows versions, and will thus require the *High Encryption Pack* or an upgrade of Internet Explorer.

64-bit Operating Systems

Operating System	Comments
Windows Server 2003	All operations are supported by default.
Windows Server 2008	All operations are supported by default.
Windows Server 2008 RC2	All operations are supported by default.
Windows XP	All operations are supported by default.
Windows Vista	All operations are supported by default.
Windows 7	All operations are supported by default.
Windows 8	All operations are supported by default.

Supported Java versions

JCAPI has been successfully tested and verified with the following versions of Java:

32-bit Java Editions

- Java 1.4
- Java 1.5
- Java 6
- Java 7

64-bit Java Editions

- Java 1.5
- Java 6
- Java 7

Only *Java Virtual Machines* (JVM) from Oracle are supported.

Supported PKCS#11 CSPs

JCAPI provides built-in support for many PKCS#11 CSPs. This means that whenever a private key is to be accessed through one of these CSPs, JCAPI will by-pass MS CAPI and automatically use the CPS's PKCS#11 driver instead. This will remove many problems related to accessing private keys stored on hardware tokens through MS CAPI, and give you the opportunity to use your own Java private key call-back implementation if needed.

These are the JCAPI supported PKCS#11 CSPs:

CSP Name: SafeSign CSP Version 1.0
Tested Hardware Token: Eutron Cryptoidentity ITSEC-P USB Token
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: SI_CSP
Tested Hardware Token: Eutron Cryptoidentity ITSEC-I USB Token
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: AR Base Cryptographic Provider
Tested Hardware Token: Eutron Cryptoidentity 5 USB Token
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: FTSafe ePass2000 RSA Cryptographic Service Provider
Tested Hardware Token: Feitian ePass2000 ROCKEY USB Token
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: eToken Base Cryptographic Provider
Tested Hardware Token: Aladdin eToken PRO 32k USB Token
Comments: Supported on all JCAPI supported 32-bit and 64-bit Windows

operating systems.

CSP Name: SmartTrust Cryptographic Service Provider
Tested Hardware Token: Telia eID SmartCard
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: Athena ASECard Crypto CSP
Tested Hardware Token: ASECard SmartCard
Comments: Supported on all JCAPI supported 32-bit Windows operating systems.

CSP Name: Datakey RSA CSP
Tested Hardware Token: Rainbow iKey 2032 USB
Comments:

CSP Name: Advanced Card Systems CSP v1.5
Tested Hardware Token: ACR38 USB
Comments:

CSP Name: SafeNet RSA CSP
Tested Hardware Token: Rainbow iKey 2032 USB
Comments:

Note: JCAPI do not include MS CAPI or PKCS#11 drivers (runtime libraries) for these CSPs. These drivers must be installed by the user.

Chapter

3

Installation & Configuration

Explains how to install and configure JCAPI for your system.

Before the installation takes place, make sure that your system fulfils the requirements by JCAPI as stated in the previous chapter.

Next, download the JCAPI executable file from your user account on the Pheox web site (https://pheox.com/customer/download/products).

Installation of JCAPI

To avoid eventual problems with the installation, please make sure that you first uninstall any previous versions of JCAPI if available.

Double click on the executable file `jcapi.exe` and the installation wizard will guide you through the simple installation process.

When the installation has been completed, the JCAPI JAR file and documentation (optional) will be stored by default in the JCAPI program directory `C:\Program Files\JCAPI` for users with administrator or power user privileges. For all other users, the default installation directory will be `C:\Documents and Settings\<user name>\Local Settings\Application Data\JCAPI`.

The JCAPI example programs are optional and will, if chosen, be installed by default in the directory `C:\ProgramData` for users with administrator or power user privileges. For all other users, the default installation directory will be `C:\Documents and Settings\All Users\Application Data\JCAPI`.

You can also access the JCAPI documents and example files through your program menu by clicking *Start -> All Programs -> JCAPI*

Using JCAPI

To use JCAPI in your Java application, you have to:

1. Include the JAR file `JCAPI.jar` into your Java application's class path. This JAR file is located in the JCAPI program directory.

2. Add JCAPI as a security provider in your Java application/applet with: `Security.addProvider(new JCAPIProvider());`

When JCAPI is added to the JCE framework as a security provider, the following actions are taken by JCAPI:

1. Check if the file `JCAPI32.dll` or `JCAPI64.dll` exists in your temporary directory (usually `C:\Documents and Settings\<`*`user`*`>\Local Settings\Temp`). Note: the exact name for this directory can be determined by the Java system property `java.io.tmpdir`

2. If the DLL file does not exist, then JCAPI will extract this file from inside the JAR file and store it in your temporary directory. If the DLL file already exist in your temporary directory, then JCAPI will compare the existing file with the one stored in its JAR file and overwrite the existing DLL file if they are not identical.

3. Load the DLL file into your application's/applet's JVM process.

Hint: the name of the DLL file and the target directory can be reconfigured through the class *`com.pheox.jcapi.JCAPIDLL`*. *See JCAPI Javadocs for more information.*

Our suggestion for you to familiarize yourself with the capabilities of JCAPI, is by examining and running the example programs included in the installation.

To compile all example programs at once, execute the following command:

```
Compile_all.bat
```

To compile all example programs in a specific directory, execute the following command:

```
Compile.bat
```

To run a compiled example program, use the command `run.bat <`*`program name`*`>`. For example:

```
run.bat ListAllCerts
```

Chapter

4

Basic Concepts

Get to know the basic programming- & design concepts within JCAPI and how it maps to Microsoft CAPI. Exemptions from the JCE standard are also discussed. This chapter assumes prior experiences with the JCE framework.

JCAPI is basically a key store JCE provider with added cryptography abilities to enable encryption/decryption of data, and creation/verification of signatures with key pairs managed through the key store.

You can use JCAPI out of the box as an ordinary JCE provider and thus skip this chapter if you intend to use it for ordinary key management towards MS CAPI. It is, however, recommended that you learn the differences between MS CAPI and the JCE framework to get a better understanding of JCAPI.

System Stores in Microsoft CAPI

MS CAPI uses system stores to hold different categories of certificates. A system store is a collection of certificates that is usually stored on disk. Each system store is intended for a special purpose and can be accessed through its unique name. The number of system stores is dynamic since they can be created and deleted at will.

The following common system stores are usually available through MS CAPI:

- *MY* - contains key entries i.e. your personal certificates with corresponding private keys.
- *CA* - contains your trusted intermediate CA certificates.
- *ROOT* - contains your trusted CA root certificates.
- *TRUSTED* - contains certificates from trusted publishers that are used for verifying code (e.g. scripts) signatures.
- *DISALLOWED* - contains certificates from publishers that are considered un-trusted.

- *ADDRESSBOOK* - usually contains certificates from other parties that you trust e.g. friends and colleagues. This system store is also used by the *Outlook* mail client.

A certificate stored in a system store can be addressed in a number of ways by MS CAPI, for example by its issuer and subject distinguished name, the hash value of the certificate, or by its public key etc.

The MS CAPI interface is designed to not publish (export) the content of a private key when one is needed for signing and decryption. Instead, so called *key handles* are used by the crypto functions when a private key needs to be addressed. However, the content of a private key can be published outside CAPI through an exportation mechanism, but this will only work if the private key is marked as exportable by the source.

Java KeyStore

The Java `KeyStore` (see class `java.security.KeyStore`) is, unlike MS CAPI, designed to be a single key store that holds both key- and certificate entries in a single set. The entries inside a key store are, unlike MS CAPI, addressed by the use of *aliases*. An alias is an arbitrary but unique name that represents a single key- or certificate entry within the key store. You can however, unlike MS CAPI, use many key store instances at the same time where each instance is a separate set of entries.

A side effect of this design is that intermediate- and root certificates in a chain cannot be accessed by any other way than through the key entry alias (i.e. by getting the certificate chain) since they do not have an alias themselves. This also means that the key store will contain copies of these certificates if any other key entry contains the same certificate issuer(s).

Private keys are always published outside the key store, meaning that the Java `KeyStore` is designed for returning the real content of a private key when asked for. This will introduce several problems and risks since, for example, private keys might be stored on a hardware token where the key will never be allowed to be exported from, or another party might try to read the memory of your program or intercept your code to get hold of the content of the private key. A private key should never leave its source.

Pheox JCAPI

JCAPI implements the *Service Provider Interface* (SPI) for the Java classes `KeyStore`, `Cipher`, `Signature`, `MessageDigest`, and `SecureRandom`. Due to this, the JCAPI key store will function as an ordinary Java `KeyStore` in respect to its interface, but some discrepancies exists regarding the mapping towards MS CAPI.

Since MS CAPI uses different system stores to hold specific types of certificates, JCAPI will naturally follow this work procedure. The following MS CAPI system stores are used by default by JCAPI when inserting/importing keys and certificates (*note: you can programmatically replace these stores in runtime, see chapter 4: Advanced Topics*):

- *MY* – used for key entries i.e. your personal certificates with corresponding private keys.
- *CA* – used for your trusted intermediate CA certificates.
- *ROOT* – used for your trusted CA root certificates.
- *ADDRESSBOOK* – used for your certificate entries from other parties that you trust e.g. friends and colleagues. *Note: This system store might not exist by default on a Windows installation. If it does not exist, then JCAPI will create it automatically for you.*

JCAPI will ensure that all certificates can be addressed independent of their system store location, and that no copies of the same certificate can exist in the same system store

All system stores available through MS CAPI will be used by default by JCAPI when certificates are to be searched for, meaning that when JCAPI is asked to return all aliases available, then all available system stores will be searched through. You can list all available MS CAPI system stores by using the method:

```
JCAPIUtil.getCertStoreNames();
```

An alias in JCAPI is made by JCAPI itself since there exist no similar addressing of certificates in MS CAPI. An alias to an entry in a MS CAPI system store is defined by JCAPI to be: *`<system store name>|<base 64 encoded hash value of certificate>`*

For example:

```
MY|9zkeijN8xMHa69GHnqFA+mPRwDc=
```

This also means that you will not be able to define your own aliases for certificates that are imported into MS CAPI (e.g. by calling method `setCertificateEntry(String, Certificate)` in the `KeyStore` class). JCAPI will silently ignore these given aliases.

The content of an RSA or DSA private key is by default not published by JCAPI when the key is fetched from MS CAPI (i.e. by calling the method `getKey(String, char[])` in the `KeyStore` class) due to:

- Minimizing the risk of exposing the content to a monitoring party.
- Avoiding pre-assumptions of published key content in other parts of the code when the key content cannot be published. For example, getting a private key from the key store and then put into the JCAPI cipher class for decryption would generate an error if the cipher class could only accept a “real” published private key when the given private key could not be exported from the source.

To avoid the problems involved with published private keys, JCAPI will by the default use an internal key handle to address a private key in its source. You can however use both "real" private keys and the JCAPI internal key handles simultaneously in the JCAPI interface since JCAPI will adjust for it internally when needed.

You can force JCAPI to export private keys for all `KeyStore` instances by using:

```
JCAPIProperties.getInstance().setPrivateKeyExportable(true);
```

An exported private key is always returned by JCAPI either as a `java.security.interfaces.RSAPrivateCrtKey` key, or a `java.security.interfaces.DSAPrivateKey` key, while an internal private key handle object is represented as either a `com.pheox.jcapi.JCAPIRSAPrivateKey`, or a `com.pheox.jcapi.JCAPIDSAPrivateKey` key. You can use the following check to determine if a private key object is a key handle object or a "real" exported key:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
Key k = ks.getKey(your_alias, null);
if(k instanceof JCAPIRSAPrivateKey ||
   k instanceof JCAPIDSAPrivateKey)
  System.err.println("Key is not exportable");
else
  System.out.println("Key is exported.");
```

Chapter

5

Advanced Topics

Learn how to configure JCAPI to meet your needs. You'll be able to list and change the CSPs and system stores used. Learn how use an un-trusted store for filtering of certificates.

There are several utility classes included in JCAPI that can be used for tuning your applications in numerous ways to fit your specific requirements.

Most of the supported options are available through the following JCAPI classes:

- `com.pheox.jcapi.JCAPIProperties`
- `com.pheox.jcapi.JCAPIUtil`
- `com.pheox.jcapi.JCAPIDLL`
- `com.pheox.jcapi.JCAPIPKCS11Util`
- `com.pheox.jcapi.JCAPICryptoFactory`

Detailed information about these classes is to be found in the JCAPI Javadocs.

Managing CAPI Cryptographic Service Providers

JCAPI will by default use the most advanced RSA/DSA supported MS CAPI CSP available. When JCAPI is added to the JCE framework as a new provider, it will query MS CAPI for a list of all available CSPs. JCAPI will then choose a RSA CSP in the following preferred order:

1. Microsoft Enhanced RSA and AES Cryptographic Provider
2. Microsoft Enhanced Cryptographic Provider v1.0
3. Microsoft Strong Cryptographic Provider
4. Microsoft Base Cryptographic Provider v1.0

For DSA, JCAPI will choose a CSP in the following preferred order:

1. Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider
2. Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
3. Microsoft Base DSS Cryptographic Provider
4. Microsoft DH SChannel Cryptographic Provider

Show currently used MS CAPI CSP

Use the following to get the current RSA supported MS CAPI CSP used by JCAPI:

```
JCAPIProperties.getInstance().getRSACSP();
```

Use the following to get the current DSA supported MS CAPI CSP used by JCAPI:

```
JCAPIProperties.getInstance().getDSACSP();
```

List all available MS CAPI CSPs

Use the following to get a list of all available MS CAPI CSPs:

```
JCAPIUtil.getCSPs();
```

Re-configure JCAPI to use another MS CAPI CSP

Execute the following to tell JCAPI to use another RSA supported CSP:

```
JCAPIProperties.getInstance().setRSACSP(String);
```

Execute the following to tell JCAPI to use another DSA supported CSP:

```
JCAPIProperties.getInstance().setDSACSP(String);
```

Usually you should not change the default CSP used by JCAPI since it might cause undesired side effects and problems that are hard to track. Use it only if you are **absolutely sure** about what you are doing.

Reset JCAPI used MS CAPI CSP

If you have re-configured JCAPI to use another RSA supported CSP, and want JCAPI to reset itself to use the default RSA CSP, then execute the following:

```
JCAPIProperties.getInstance().resetRSACSP();
```

If you have re-configured JCAPI to use another DSA supported CSP, and want JCAPI to reset itself to use the default DSA CSP, then execute the following:

```
JCAPIProperties.getInstance().resetDSACSP();
```

Managing MS CAPI system stores

JCAPI can be configured to use other MS CAPI system stores (a k a certificate stores) than the default set of system stores described in the previous chapter. This is a very powerful property since you can, in runtime, change what system store(s) that should be available, and where certain types of certificates shall be stored into, and accessed from.

Configuring the set of system stores to use is done through an instance of class `com.pheox.jcapi.JCAPIKeyStoreProperties`. This class acts as a wrapper over an instance of the standard JCE key store class `java.security.KeyStore`. This means that each configuration is instance based and will thus only affect that particular key store instance. This will permit multiple key store instances with different configurations running at the same time without any global side effects.

Managing the full set of available system stores

Use the following to list all MS CAPI system stores available in JCAPI:

```
String[] certStores = JCAPIUtil.getCertStoreNames();
```

Use the following to list all MS CAPI system stores available through a JCAPI based `KeyStore` instance:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String[] certStores = ksprop.getCertStoreNames();
```

Use the following to set the list of MS CAPI system stores that will be available through a JCAPI based `KeyStore` instance:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String[] certStores = new String[]{"MY", “ADDRESSBOOK”, “CA”,
“ROOT”};
ksprop.setCertStoreNames(certStores);
```

Managing the key entry system store

A key entry is here defined to be a chain of X.509 certificates where the first certificate in the chain has an associated private key. The chain will contain one (self-signed) or more certificates. The first certificate in the chain will be inserted into, or fetched from, the MS CAPI key entry store. By default, JCAPI will use the `MY` system store for this.

Use the following to get the name of the key entry system store used (default: `MY`):

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String keyEntryStore = ksprop.getKeyEntryStoreName();
```

Use the following to change the name of the key entry system store to use:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setKeyEntryStoreName(“MY”);
```

Managing the intermediate certificates system store

An intermediate certificate is one that is usually issued by a subordinate *Certificate Authority* (CA). Simplified, this type of certificate is not self-signed and constitutes the rest of a certificate chain apart from the key entry certificate and the root CA certificate.

Intermediate certificates are stored into, and fetched from, a specific MS CAPI system store. By default, JCAPI will use the `CA` system store for this.

Use the following to get the name of the intermediate certificates system store used (default: `CA`):

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String certStore = ksprop.getIntermediateCertStoreName();
```

Use the following to change the name of the intermediate certificates system store to use:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setIntermediateCertStoreName (“CA”);
```

Managing the root certificates system store

A root certificate is a self-signed certificate that is issued by a *Certificate Authority* (CA). Root certificates are stored into, and fetched from, a specific MS CAPI system store. By default, JCAPI will use the `ROOT` system store for this.

Use the following to get the name of the root certificates system store used (default: `ROOT`):

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String certStore = ksprop.getRootCertStoreName();
```

Use the following to change the name of the root certificates system store to use:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setRootCertStoreName (“ROOT”);
```

Managing the certificate entry system store

A certificate entry is here defined to be an X.509 certificate that is not a key entry and resides in any of the available system stores. JCAPI will use a specific system store when a certificate entry is to be stored/imported i.e. when method `setCertificateEntry(String, Certificate)` in class `KeyStore` is executed. By default, JCAPI will use the `ADDRESSBOOK` system store for this.

The JCE framework defines a certificate entry to be a certificate belonging to another party that is trusted by you. The JCAPI design assumes all certificates found in all system stores (except the key entry system store since it contains only key entries) to be trusted. If this is not what you want, then you can either use an un-trusted system store to filter out a set of certificates, or you can re-configure JCAPI in runtime to only access a restricted set of system stores. Here is an example of how to re-configure a JCAPI based `KeyStore` instance to only accept certificates stored in the `ROOT` system store to be trusted certificate entries:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setCertStoreNames(new String[]{"ROOT"});
ks.isCertificateEntry(<your alias>);
```

Use the following to get the name of the certificate entry system store used (default: `ADDRESSBOOK`):

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String certStore = ksprop.getCertEntryStoreName();
```

Use the following to change the name of the certificate entry system store to use when certificate entries are to be inserted:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setCertEntryStoreName (“ADDRESSBOOK”);
```

Managing the un-trusted certificates system store

An un-trusted certificates store is a system store containing certificates that are not trusted by JCAPI. This kind of store will act as a filter to discard certificates that are not considered trusted by you. Some Windows operating systems use the system store `DISALLOWED` to hold these certificates. By default, JCAPI do not use an un-trusted certificates system store.

Use the following to get the name of the un-trusted certificates system store used:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
String certStore = ksprop.getUntrustedCertStoreName();
```

Use the following to set the name of the un-trusted certificates system store to use:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setUntrustedCertStoreName (“DISALLOWED”);
```

Using an exclusive system store

You can configure a JCAPI `KeyStore` instance to use an exclusive system store. This means that the `KeyStore` instance will set and get all key entries, certificate entries, and trusted certificates from the single defined exclusive system store only. Here is an example of how to create and configure a JCAPI based `KeyStore` instance that will use the MS CAPI system store `ROOT` as its exclusive store:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setExclusiveCertStoreName(“ROOT”);
```

Importing private keys using different MS CAPI protection mechanisms

When you import private keys into MS CAPI using the method `KeyStore.setKeyEntry (String alias, Key key, char[] password, Certificate[] chain)` and supplying no password, the default behaviour from JCAPI is to import these keys as exportable with no access protection. If the `password` parameter is not `null`, then the private key will be marked as non-exportable and the native CSP's protection dialog will be displayed for the client to choose the desired protection mechanism, for example to password protect the private key.

If you would like to override the default behaviour when importing your private keys, then you can use the following methods available in the class `com.pheox.jcapi.JCAPIKeyStoreProperties`:

- `void setCreateExportablePrivateKeysInMsCapi(boolean flag)`
  Configure the `JCAPIKeyStore` instance to mark private keys as exportable or not when they are imported into MS CAPI.
  If `flag` is `true`, then the clients will be able to export the native private key's data out from MS CAPI when they have been imported through the `JCAPIKeyStore` instance. If `flag` is `false`, then the private keys will not be able to be exported out from MS CAPI when they are imported through the `JCAPIKeyStore` instance.

- `void setCreateProtectedPrivateKeysInMsCapi(boolean flag)`
  Configure the `JCAPIKeyStore` instance to import private keys into MS CAPI with the protected flag set or not when they are imported into MS CAPI.
  If `flag` is `true`, then the CSP's native protection dialog will be displayed to the client (to set password etc) when the private key is imported into MS CAPI through JCAPI. If `flag` is `false`, then no such dialog will be displayed to the client when a private key is to be imported into MS CAPI through JCAPI.

Here is an example of how to configure a `JCAPIKeyStore` instance to import private keys into MS CAPI as non-exportable but without any access protection:

```
KeyStore ks = KeyStore.getInstance("msks", "JCAPI");
ks.load(null, null);
JCAPIKeyStoreProperties ksprop = new
JCAPIKeyStoreProperties(ks);
ksprop.setCreateExportablePrivateKeysInMsCapi(false);
ksprop.setCreateProtectedPrivateKeysInMsCapi(false)
```

Managing the location of system stores used

By default, JCAPI uses the system stores that are available for the current user. This means that certificates and keys that are inserted into MS CAPI, through JCAPI, will be available to the current user only. The user can, however, access keys and certificates in system stores for private access (Current User) or shared/remote access (e.g. Local Machine).

If you would like to create, delete and access certificates from other locations than *Current User*, then you can configure your preferred location through the JCAPI class `com.pheox.jcapi.JCAPISystemStoreRegistryLocation`.

JCAPI supports the following list of registry locations for system stores:

- `CERT_SYSTEM_STORE_CURRENT_SERVICE`

- CERT_SYSTEM_STORE_CURRENT_USER *(default)*
- CERT_SYSTEM_STORE_CURRENT_USER_GROUP_POLICY
- CERT_SYSTEM_STORE_LOCAL_MACHINE
- CERT_SYSTEM_STORE_LOCAL_MACHINE_ENTERPRISE
- CERT_SYSTEM_STORE_LOCAL_MACHINE_GROUP_POLICY
- CERT_SYSTEM_STORE_SERVICES
- CERT_SYSTEM_STORE_USERS

Use the following to get the current registry location used:

```
JCAPISystemStoreRegistryLocation location =
JCAPIProperties.getInstance().getSystemStoreRegistryLocation();
```

Use the following example to use the *Local Machine* location:

```
JCAPISystemStoreRegistryLocation location = new
JCAPISystemStoreRegistryLocation(JCAPISystemStoreRegistryLocatio
n.CERT_SYSTEM_STORE_LOCAL_MACHINE);
JCAPIProperties.getInstance().setSystemStoreRegistryLocation(loc
ation);
```

Please note that by changing the location used, the internal list of accessible system stores will be reset since a new registry location might add/change/remove system stores that can be accessed. You can call method ***`JCAPIUtil.getCertStoreNames()`*** *to see what stores are available through the current registry location used.*

Use the following to tell JCAPI to use its default location (*Current User*) again:

```
JCAPIProperties.getInstance().resetSystemStoreRegistryLocation();
```

Getting the friendly name of a system store

MS CAPI uses internal names for identifying system stores e.g. *MY* or *CA*, while a friendly name is used as a more user-friendly and localizable alternative.

Examples of system store friendly names (when using the English language) are *Personal*, *Other people*, *Intermediate Certification Authorities* and so on.

Use the following example to get the friendly name of the *MY* system store:

```
String friendlyName = JCAPIUtil.getCertStoreFriendlyName(”MY”);
```

Getting and setting the friendly name of a certificate

MS CAPI supports friendly names for certificates to make it easier for the human eye to identify a specific certificate.

JCAPI does not support access to certificates using a friendly name, but provide means for getting and setting the friendly name as a way of presenting a certificate entry to a human reader, for example, through a graphical user-interface.

Use the following example to get and set the friendly name of a certificate with alias `MY|zdTfrmAAgH9AA4AsFx4dF=`:

```
String alias = ”MY|zdTfrmAAgH9AA4AsFx4dF=”;
String fname = JCAPIUtil. getCertificateFriendlyName(alias);
JCAPIUtil.setCertificateFriendlyName(alias, ”My certificate”);
```

MS CAPI Algorithms & Key Lengths

You can through JCAPI list all algorithms and key lengths that are supported by any registered MS CAPI CSP.

Algorithms

To get a list of algorithms supported by a specific CSP, use the static method `getAlgorithmsByCSP(String)` in class `JCAPIUtil`. This method will return a list of `JCAPICSPAlgorithm` instances.

```
String csp = ”Microsoft Enhanced Cryptographic Provider v1.0”;
JCAPICSPAlgorithm[] algs = JCAPIUtil.getAlgorithmsByCSP(csp);
```

The `JCAPICSPAlgorithm` class can then be used for querying the particular algorithm through the following methods:

- `String getCSP()` - Returns the CSP name for this algorithm e.g. `"Microsoft Enhanced Cryptographic Provider v1.0"`.

- `int getProviderType()` - Returns the provider type for this algorithm e.g. `PROV_RSA_FULL`.

- `int getIdentifier()` - Returns the algorithm identifier for this algorithm e.g. `CALG_3DES`.

- `String getName()` - Returns the name of this algorithm e.g. `"AES"`.
- `int getCipherMode()` - Returns the cipher mode e.g. `CRYPT_MODE_CBC`.
- `int getPadding()` - Returns the padding e.g. `PKCS5_PADDING`.

This class also contains constants that are used both in JCAPI and MS CAPI to define the properties of an algorithm. All these constants are originally defined in WinCrypt.h and they are used by JCAPI to give a one-to-one configuration mapping between MS CAPI and JCAPI.

The available constants can be divided into the following categories:

- OID algorithm identifiers. Defined with prefix `OID_`
- MS CAPI provider types. Defined with prefix `PROV_`
- MS CAPI algorithm identifiers. Defined with prefix `CALG_`
- MS CAPI cipher modes. Defined with prefix `CRYPT_`
- MS CAPI padding schemes. Defined with suffix `_PADDING`

Key lengths

We can query MS CAPI about supported key lengths for a specific algorithm and CSP. Each algorithm will have a minimum, maximum, and default key length. These key lengths may be the same for specific algorithms. Note: some CSPs don't return key lengths at all for some algorithms.

To get the key lengths for a specific algorithm and CSP, use the static method `getAlgorithmKeyLengths(JCAPICSPAlgorithm)` in class `JCAPIUtil`. This method will return an immutable `JCAPICSPAlgorithmKeyLengths` instance.

```
JCAPICSPAlgorithm alg = <instance gotten from previous calls>;
JCAPICSPAlgorithmKeyLengths keyLengths =
JCAPIUtil.getAlgorithmKeyLengths(alg);
```

The `JCAPICSPAlgorithmKeyLengths` class can then be used for examining the supported key lengths through the following provided methods:

- `int getDefaultKeyLength()` - Returns the default key length (in bits) for the algorithm.
- `int getMinimumKeyLength()` - Returns the minimum key length (in bits) for the algorithm.
- `int getMaximumKeyLength()` - Returns the maximum key length (in bits) for the algorithm.

Dynamic Cryptographic Factory

The class `JCAPICryptoFactory` is used for creating a dynamic JCAPI cryptographic instance wrapped into one of the following *Java Cryptography Extension* (JCE) interfaces:

- `javax.crypto.Cipher`
- `java.security.KeyStore`
- `java.security.MessageDigest`
- `java.security.SecureRandom`
- `java.security.Signature`

Being able to dynamically create and wrap an arbitrary cryptographic algorithm in MS CAPI into a standardized JCE interface is an extremely powerful feature.

It gives the programmer the possibility to query MS CAPI in runtime about what algorithms that can be used for encryption and decryption (symmetric/asymmetric), and for creating signatures and message digests. When a desired algorithm has been found, it can then be wrapped into a JCE class through one of the methods provided by this class.

The standard JCAPI `Cipher`, `MessageDigest`, `KeyStore`, `SecureRandom`, and `Signature` classes uses a hard coded MS CAPI CSP, MS CAPI provider type, and MS CAPI algorithm identifier. But when you use this class, you can get a total dynamic behavior by providing your own CSP, provider type, algorithm identifier etc. to let JCAPI use your preferred native MS CAPI provider implementation for almost any given algorithm. Just instantiate the `JCAPICSPAlgorithm` class with your preferred data and pass it on to one of the following methods provided by this class to get JCAPI to work with your native MS CAPI provider through a standard JCE instance:

- `Cipher createCipherInstance(JCAPICSPAlgorithm alg)` - Returns a `javax.crypto.Cipher` instance based on the information given in parameter `alg`.

- `KeyStore createKeyStoreInstance(String rsaProvider, String dsaProvider, String[] storeNames, String certEntryStoreName, String keyEntryStoreName, String immediateCertStoreName, String rootCertStoreName, String untrustedCertStoreName, boolean isPrivateKeysExportable)` - Returns a `java.security.KeyStore` instance based on the information given in the input parameters.

- `MessageDigest createMessageDigestInstance(JCAPICSPAlgorithm alg)` - Returns a `java.security.MessageDigest` instance based on the information given in parameter `alg`.

- `SecureRandom createSecureRandomInstance(String cspName,int providerType)` - Returns a `java.security.SecureRandom` instance based on the information given in parameters `cspName` and `providerType`.

- `Signature createSignatureInstance(JCAPICSPAlgorithm hashAlg,JCAPICSPAlgorithm keyAlg)` - Returns a `java.security.Signature` instance based on the information given in parameters `hashAlg` and `keyAlg`.

Dynamic ciphers

This example will show you how to create a `javax.crypto.Cipher` instance which holds a JCAPI dynamic cipher instance which in turn uses a native MS CAPI CSP for encryption and decryption using the AES-128 algorithm:

```
String cspName = "Microsoft Enhanced RSA and AES Cryptographic
Provider";
int providerType = JCAPICSPAlgorithm.PROV_RSA_AES;
int algId = JCAPICSPAlgorithm.CALG_AES_128;
String algName = "AES";
int cipherMode = JCAPICSPAlgorithm.CRYPT_MODE_CBC;
int padding = JCAPICSPAlgorithm.PKCS5_PADDING;
JCAPICSPAlgorithm alg = new JCAPICSPAlgorithm(cspName,
providerType, algId, algName, cipherMode, padding);
Cipher c = JCAPICryptoFactory.createCipherInstance(alg);
```

Dynamic key stores

This example will show you how to create a `java.security.KeyStore` instance which holds a JCAPI dynamic key store instance which in turn will only access the MS CAPI stores *My*, *CA*, and *Root*:

```
String rsaCspName = JCAPIProperties.getInstance().getRSACSP();
String dsaCspName = JCAPIProperties.getInstance().getDSACSP();
String keyEntryCertStore = "My";
String intermediateCertStore = "CA";
String rootCertStore = "Root";
String[] certStores = new String[]{keyEntryCertStore,
intermediateCertStore, rootCertStore};
boolean exportPrivateKeys = false;
KeyStore ks =
JCAPICryptoFactory.createKeyStoreInstance(rsaCspName,
dsaCspName, certStores, keyEntryCertStore, keyEntryCertStore,
intermediateCertStore, rootCertStore, null, exportPrivateKeys);
```

Dynamic message digests

This example will show you how to create a `java.security.MessageDigest` instance which holds a JCAPI dynamic message digest instance which in turn uses a native MS CAPI CSP for hashing data using the SHA-512 algorithm:

```
String cspName = "Microsoft Enhanced RSA and AES Cryptographic
Provider";
int providerType = JCAPICSPAlgorithm.PROV_RSA_AES;
int algId = JCAPICSPAlgorithm.CALG_SHA512;
String algName = "SHA-512";
JCAPICSPAlgorithm alg = new JCAPICSPAlgorithm(cspName,
providerType, algId, algName);
MessageDigest md =
JCAPICryptoFactory.createMessageDigestInstance(alg);
```

Dynamic secure random

This example will show you how to create a `java.security.SecureRandom` instance which holds a JCAPI dynamic secure random instance which in turn uses a native MS CAPI CSP to generate secure random bytes:

```
String cspName = "Microsoft Base Cryptographic Provider v1.0";
int provType = JCAPICSPAlgorithm.PROV_RSA_FULL;
SecureRandom sr =
JCAPICryptoFactory.createSecureRandomInstance(cspName,
provType);
```

Dynamic signatures

This example will show you how to create a `java.security.Signature` instance which holds a JCAPI dynamic signature instance which in turn uses a native MS CAPI CSP for creation and verification of signatures using the RSA key algorithm with the SHA-1 hash algorithm:

```
String cspName = "Microsoft Enhanced Cryptographic Provider
v1.0";
int providerType = JCAPICSPAlgorithm.PROV_RSA_FULL;
int keyAlgId = JCAPICSPAlgorithm.CALG_RSA_SIGN;
String keyAlgName = "RSA";
JCAPICSPAlgorithm keyAlg = new JCAPICSPAlgorithm(cspName,
providerType, keyAlgId, keyAlgName);
int hashAlgId = JCAPICSPAlgorithm.CALG_SHA1;
String hashAlgName = "SHA-1";
JCAPICSPAlgorithm hashAlg = new JCAPICSPAlgorithm(cspName,
providerType, hashAlgId, hashAlgName);
Signature s =
JCAPICryptoFactory.createSignatureInstance(hashAlg, keyAlg);
```

Private key usage information

The key usage will tell you the primary purpose of using that particular private key. It is very common to use one private key for signing, and another private key for encryption/wrapping. In MS CAPI this is also known as `AT_SIGNATURE` and `AT_KEYEXCHANGE`.

Some CSPs don't allow a signing key to be used for encryption, and vice versa.

You can get the key usage information for a private key managed by MS CAPI through the static method `getPrivateKeyUsage(String)` in class `JCAPIUtil`. This method will return either `KEY_USAGE_SIGNATURE` or `KEY_USAGE_ENCRYPTION` for the given alias depending on if the associated private key is used for signing- or encryption purposes:

```
String alias = <alias which has an associated private key>
switch(JCAPIUtil.getPrivateKeyUsage(alias))
{
  case JCAPIUtil.KEY_USAGE_ENCRYPTION :
    System.out.println("Key can be used for encryption.");
    break;
  case JCAPIUtil.KEY_USAGE_SIGNATURE :
    System.out.println("Key can be used for signing.");
    break;
}
```

Managing the JCAPI DLL file

When the JCAPI DLL file is extracted from the JAR file, it will by default be named `JCAPI32.dll` (on 32-bit systems) or `JCAPI64.dll` (on 64-bit systems), and stored in your temporary directory. You can configure JCAPI to use another filename and target directory for the DLL. Please note that JCAPI will by default extract the DLL file from within its JAR file and overwrite any existing file if they have the same file name but different content.

Use the following method to change the name of the DLL when it is extracted:

```
JCAPIDLL.getInstance().setFilename(String);
```

Use the following method to change the target directory for the extracted DLL:

```
JCAPIDLL.getInstance().setDirectory(File);
```

Note: the configuration of the DLL must be performed before JCAPI is added to the JCE framework (i.e. before `Security.addProvider(JCAPIProvider)`), since the DLL is extracted and attached to the JVM during this operation.

To learn more about on how to configure and how to get the current status of the JCAPI DLL:

- See the example program `ConfigureDLL.java`.
- Read the JCAPI Javadoc for class `com.pheox.jcapi.JCAPIDLL`.

Exception handling

All JCAPI SPI classes throw the same exceptions during the same conditions as defined by their respective JCE classes. Since JCAPI also works on a lower level towards the MS CAPI and PKCS#11 layers, there might be occasions where "unexpected" exceptions appears in the JCAPI DLL due to a problem that is bound to one of these layers. An error of this kind will lose much of its meaning if it's encapsulated into a normal Java security exception. Therefore, JCAPI defines two types of basic exception types:

- `JCAPIJNIException` - This class extends the `java.lang.Exception` class and is used for indicating an internal error in the JCAPI DLL e.g. if encrypted data could not be decrypted due to an invalid key. An instance of this class is always created and thrown by the JCAPI DLL. The thrown exception is always caught inside JCAPI and transformed into a "known and accepted" JCE framework exception to be thrown to the user of JCAPI. In other words, a user of JCAPI will never have to catch this exception outside JCAPI.

- `JCAPIJNIRuntimeException` - This class extends the `java.lang.RuntimeException` class and is used for indicating an unexpected internal error in the JCAPI DLL e.g. CSP errors. An instance of this class is always created and thrown by the JCAPI DLL, and is never caught by the JCAPI Java classes. In other words, a user of JCAPI might have to catch this runtime exception outside JCAPI if required. If a method in JCAPI can throw this exception, then it is specified in the Javadoc for each method, even though it is a runtime exception.

Apart from the above mentioned exception classes, JCAPI do also provide some utility exception classes e.g. to detect if the user has cancelled an operation etc. Please read the *Exception Summary* in the JCAPI Javadocs to learn more about the different exceptions provided.

Note: when a JCAPI specific exception is thrown from inside a method, it will always be noted in a throw clause for each method in the JCAPI Javadocs.

Chapter
6

PKCS#7

This chapter will teach you how to work with the PKCS#7 capabilities provided by JCAPI.
You will learn how to encode and decode PKCS#7 messages that have been either signed or enveloped.

JCAPI support encoding and decoding of PKCS#7 messages that are either signed or enveloped (encrypted). To make it easier for the developer, JCAPI does follow the same way of encoding and decoding messages as defined in the PKCS#7 standard, and does also provide the same set of entities as defined in the standard i.e. *SignedData*, *SignerInfo*, *EnvelopedData*, and *RecipientInfo*.

JCAPI has two main classes used for encoding and decoding:

- `JCAPIPKCS7Encoder` - Encodes a signed or enveloped message.
- `JCAPIPKCS7Decoder` - Decodes a signed or enveloped message.

To be able to encode a message, we must first create it through a set of data content types. To be able to decode a message we must have a set of data content types to decode the message into. These data content types are in JCAPI represented by the following classes:

- `JCAPIPKCS7SignedData` - This class represents the type *SignedData* as defined in the PKCS#7 standard.
- `JCAPIPKCS7SignerInfo` - This class represents the type *SignerInfo* as defined in the PKCS#7 standard.
- `JCAPIPKCS7EnvelopedData` - This class represents the type *EnvelopedData* as defined in the PKCS#7 standard.
- `JCAPIPKCS7RecipientInfo` - This class represents the type *RecipientInfo* as defined in the PKCS#7 standard.

Signed Data

The `JCAPIPKCS7SignedData` class represents the type *SignedData* as defined in the PKCS#7 standard. The *SignedData* content type consists of content of any type and encrypted message digests of the content for zero or more signers. The encrypted digest for a signer is a digital signature on the content for that signer.

This class holds the following properties when a PKCS#7 *SignedData* message is to be encoded/decoded:

- One or more `JCAPIPKCS7SignerInfo` instances i.e. one for each signer.
- Zero, one, or more `X509Certificate` certificates. It is intended that the set be sufficient to contain chains from a recognized root or top-level certification authority to all of the signers in the message.
- Zero, one, or more `X509CRL` CRLs. It is intended that the set contain information sufficient to determine whether or not any of the certificates mentioned above are hot listed, but such correspondence is not necessary.

The following constructor and methods are available:

- `JCAPIPKCS7SignedData()` - Default constructor.
- `int getContentType()` - Returns the constant `JCAPIPKCS7ContentInfo.CONTENT_TYPE_SIGNED_DATA`.
- `void addSignerInfo(JCAPIPKCS7SignerInfo signer)` - Add a signer of type `JCAPIPKCS7SignerInfo` to the list of signers. The signer to be added must have a private key associated with its certificate since it will be used for signing the PKCS#7 message.
- `void addSignerInfos(JCAPIPKCS7SignerInfo[] signers)` - Add one or more signers of type `JCAPIPKCS7SignerInfo` to the list of signers. Each signer to be added must have a private key associated with its certificate since it will be used for signing the PKCS#7 message.
- `void addCertificate(X509Certificate cert)` - Add an additional certificate to the list of additional certificates to be included in the PKCS#7 message. It is intended that the set be sufficient to contain chains from a recognized root or top-level certification authority to all of the signers in the message.
- `void addCertificates(X509Certificate[] certs)` - Add additional certificates to the list of additional certificates to be included in the PKCS#7 message. It is intended that the set be sufficient to contain chains from a recognized root or top-level certification authority to all of the signers in the message.

- `void addCRL(X509CRL crl)` - Add a CRL to the list of CRLs to be included in the PKCS#7 message. It is intended that the set contain information sufficient to determine whether or not any of the additional certificates are hot listed, but such correspondence is not necessary.

- `void addCRLs(X509CRL[] crls)` - Add one or more CRLs to the list of CRLs to be included in the PKCS#7 message. It is intended that the set contain information sufficient to determine whether or not any of the additional certificates are hot listed, but such correspondence is not necessary.

- `JCAPIPKCS7SignerInfo[] getSignerInfos()` - Returns the signers of the PKCS#7 message.

- `X509Certificate[] getCertificates()` - Returns the additional certificates stored in the PKCS#7 message.

- `X509CRL[] getCRLs()` - Returns the CRLs stored in the PKCS#7 message.

An example of how to create a *SignedData* instance for encoding:

```
X509Certificate signerCert = <certificate instance>;
JCAPIPKCS7SignerInfo signer = new
JCAPIPKCS7SignerInfo(signerCert);
JCAPIPKCS7SignedData signedData = new JCAPIPKCS7SignedData();
signedData.addSignerInfo(signer);
JCAPIPKCS7Encoder encoder = new JCAPIPKCS7Encoder();
encoder.init(signedData);
```

An example of how to get a *SignedData* instance after decoding:

```
JCAPIPKCS7Decoder decoder = new JCAPIPKCS7Decoder();
decoder.init();
...decode file...
if(decoder.getContentTypeId() ==
JCAPIPKCS7ContentInfo.CONTENT_TYPE_SIGNED_DATA)
  JCAPIPKCS7SignedData data =
(JCAPIPKCS7SignedData)decoder.getContentInfo();
```

Signer Info

The `JCAPIPKCS7SignerInfo` class represents the type *SignerInfo* as defined in the PKCS#7 standard. One instance of this class represent one signer in a PKCS#7 *SignedData* encoded message i.e. such a message can contain one or more instances of this class, one for each signer.

An instance of this class will be generated:

- By the programmer when data is to be signed and encoded through an instance of `JCAPIPKCS7Encoder`.

- By an instance of class `JCAPIPKCS7Decoder` when a signed message has been decoded.

The following constructors and methods are available:

- `JCAPIPKCS7SignerInfo(X509Certificate cert)` - Use this constructor for defining a signer for the encoding of a PKCS#7 *SignedData* message, see JCAPI class `JCAPIPKCS7SignedData`. The default digest (hash) algorithm OID used will be `JCAPICSPAlgorithm.OID_OIWSEC_sha1`.

- `JCAPIPKCS7SignerInfo(X509Certificate cert, String digestAlgorithmOID)` - Use this constructor for defining a signer for the encoding of a PKCS#7 *SignedData* message, see JCAPI class `JCAPIPKCS7SignedData`. *Hint: you can find a lot of OID constants defined in the class* *`JCAPICSPAlgorithm`.*

- `X500Principal getIssuer()` - Returns the issuer distinguished name (X.500 principal) to be used for identifying the signer's certificate.

- `BigInteger getSerialNumber()` - Returns the serial number of the signing certificate.

- `X509Certificate getCertificate()` - Returns the signer's certificate, if available in any MS CAPI certificate store, including the store associated with the PKCS#7 message itself. Note: if this instance was generated during a decoding session i.e. by `JCAPIPKCS7Decoder`, then this method might return `null` if the signer's certificate cannot be found. According to the PKCS#7 standard, an encoded *SignedData* message holds the issuer and serial number for each signer, and not the certificate itself.

- `String getDigestAlgorithmOID()` - Returns the message digest (hash) object identifier. Default OID is *1.3.14.3.2.26* (see, `JCAPICSPAlgorithm.OID_OIWSEC_sha1`).

- `byte[] getEncryptedDigest()` - Returns the signer's encrypted digest i.e. the signature.

- `boolean isValidSignature()` - Returns the result of the signature verification. JCAPI verifies each signer's signature during the decoding session using the public key associated with the certificate information defined for the signer. Note: if signer's certificate cannot be found in any store or in the message itself, then this method will return `false`.

An example of how to create a *SignerInfo* instance for encoding:

```
X509Certificate signerCert = <certificate instance>;
JCAPIPKCS7SignerInfo signer = new
JCAPIPKCS7SignerInfo(signerCert);
JCAPIPKCS7SignedData signedData = new JCAPIPKCS7SignedData();
signedData.addSignerInfo(signer);
JCAPIPKCS7Encoder encoder = new JCAPIPKCS7Encoder();
encoder.init(signedData);
```

An example of how to get a *SignerInfo* instance after decoding:

```
JCAPIPKCS7Decoder decoder = new JCAPIPKCS7Decoder();
decoder.init();
...decode file...
if(decoder.getContentTypeId() ==
JCAPIPKCS7ContentInfo.CONTENT_TYPE_SIGNED_DATA)
{
  JCAPIPKCS7SignedData data =
(JCAPIPKCS7SignedData)decoder.getContentInfo();
  JCAPIPKCS7SignerInfo[] infos = data.getSignerInfos();
}
```

Enveloped Data

The `JCAPIPKCS7EnvelopedData` class represents the type *EnvelopedData* as defined in the PKCS#7 standard. The *EnvelopedData* content type consists of encrypted content of any type and encrypted content-encryption keys for one or more recipients. The combination of encrypted content and encrypted content-encryption key for a recipient is a digital envelope for that recipient.

This class holds the following properties when a PKCS#7 *EnvelopedData* message is to be encoded/decoded:

- One or more `JCAPIPKCS7RecipientInfo` instances i.e. one for each recipient.
- An encryption algorithm. Default is RSA with TripleDES and CBC mode i.e. `JCAPICSPAlgorithm.OID_RSA_DES_EDE3_CBC`.
- An MS CAPI CSP. If none is given, then the default MS CAPI RSA CSP is used, see method `JCAPIProperties.getRSACSP()`.
- An MS CAPI provider type for the chosen CSP. If none is given, then the provider type for the default CSP will be used, see method `JCAPIProperties.getCachedProviderTypeByCSP(String)`.

The following constructor and methods are available:

- `JCAPIPKCS7EnvelopedData()` - Default constructor. Creates an empty instance with no recipients, default CSP, and the provider type for the default CSP will be used.

- `int getContentType()` - Returns the constant `JCAPIPKCS7ContentInfo.CONTENT_TYPE_ENVELOPED_DATA`. This interface method is declared in `JCAPIPKCS7ContentInfo`.

- `void addRecipientInfo(JCAPIPKCS7RecipientInfo recipient)` - Add a recipient to the list of recipients to be included in the PKCS#7 message. The content of the PKCS#7 message will be encrypted using the public key in the recipient's certificate, and decrypted by its associated private key. Default encryption algorithm is `JCAPICSPAlgorithm.OID_RSA_DES_EDE3_CBC`.

- `void addRecipientInfos(JCAPIPKCS7RecipientInfo[] recipients)` - Add one or more recipients to the list of recipients to be included in the PKCS#7 message. The content of the PKCS#7 message will be encrypted using the public key in each recipient's certificate, and decrypted by its associated private key. Default encryption algorithm is `JCAPICSPAlgorithm.OID_RSA_DES_EDE3_CBC`.

- `JCAPIPKCS7RecipientInfo[] getRecipients()` - Returns the list of recipients in the PKCS#7 message.

- `void setAlgorithmOID(String alg)` - Set the encryption algorithm object identifier to use during enveloping. Default encryption algorithm is `JCAPICSPAlgorithm.OID_RSA_DES_EDE3_CBC`.

- `void setCsp(String csp)` - Set the MS CAPI CSP to use during encryption. If parameter `csp` is `null`, then the MS CAPI default provider will be used.

- `void setProviderType(int type)` - Set the MS CAPI provider type to use during encryption.

An example of how to create an *EnvelopedData* instance for encoding:

```
X509Certificate recipientCert = <certificate instance>;
JCAPIPKCS7RecipientInfo recipientInfo = new
JCAPIPKCS7RecipientInfo(recipientCert);
JCAPIPKCS7EnvelopedData envelopedData = new
JCAPIPKCS7EnvelopedData();
envelopedData.addRecipientInfo(recipientInfo);
JCAPIPKCS7Encoder encoder = new JCAPIPKCS7Encoder();
encoder.init(envelopedData);
```

An example of how to get an *EnvelopedData* instance after decoding:

```
JCAPIPKCS7Decoder decoder = new JCAPIPKCS7Decoder();
decoder.init();
...decode file...
if(decoder.getContentTypeId() ==
JCAPIPKCS7ContentInfo.CONTENT_TYPE_ENVELOPED_DATA)
  JCAPIPKCS7EnvelopedData data =
(JCAPIPKCS7EnvelopedData)decoder.getContentInfo();
```

Recipient Info

The `JCAPIPKCS7RecipientInfo` class represents the type *RecipientInfo* as defined in the PKCS#7 standard. One instance of this class represent one recipient in a PKCS#7 *EnvelopedData* encoded message i.e. such a message can contain one or more instances of this class, one for each recipient.

An instance of this class will be generated:

- By the programmer when data is to be enveloped and encoded through an instance of `JCAPIPKCS7Encoder`.

- By an instance of class `JCAPIPKCS7Decoder` when an enveloped message has been decoded.

The following constructor and methods are available:

- `JCAPIPKCS7RecipientInfo(X509Certificate cert)` - Set the recipient's certificate. The public key stored in the recipient's certificate will be used for encrypting the symmetric key that is used for encrypting the data content.

- `String getKeyEncryptionAlgorithmOID()` - Returns the key encryption algorithm object identifier. The key encryption algorithm identifies the key-encryption (asymmetric) algorithm under which the content-encryption key (symmetric) is encrypted with the recipient's public key.

- `X500Principal getIssuer()` - Returns the issuer distinguished name (X.500 principal) of the recipient's certificate.

- `BigInteger getSerialNumber()` - Returns the serial number of the recipient's certificate.

- `byte[] getEncryptedSymmetricKey()` - Returns the encrypted symmetric key that was used for encrypting/decrypting the content in the envelope.

An example of how to create a *RecipientInfo* instance for encoding:

```
X509Certificate recipientCert = <certificate instance>;
JCAPIPKCS7RecipientInfo recipientInfo = new
JCAPIPKCS7RecipientInfo(recipientCert);
JCAPIPKCS7EnvelopedData envelopedData = new
JCAPIPKCS7EnvelopedData();
envelopedData.addRecipientInfo(recipientInfo);
JCAPIPKCS7Encoder encoder = new JCAPIPKCS7Encoder();
encoder.init(envelopedData);
```

An example of how to get *RecipientInfo* instance(s) after decoding:

```
JCAPIPKCS7Decoder decoder = new JCAPIPKCS7Decoder();
decoder.init();
...decode file...
if(decoder.getContentTypeId() ==
JCAPIPKCS7ContentInfo.CONTENT_TYPE_ENVELOPED_DATA)
{
  JCAPIPKCS7EnvelopedData data =
(JCAPIPKCS7EnvelopedData)decoder.getContentInfo();
  JCAPIPKCS7RecipientInfo[] recipients = data.getRecipients();
}
```

The encoder

The `JCAPIPKCS7Encoder` class is used for encoding PKCS#7 signed-data content and enveloped-data content.

The process by which signed data is constructed involves the following steps:

1. For each signer, a message digest (hash) is computed on the content with a signer-specific message-digest algorithm. JCAPI will by default use the algorithm SHA-1. If the signer is authenticating any information other than the content, the message digest of the content and the other information are digested with the signer's message digest algorithm, and the result becomes the *message digest*.

2. For each signer, the message digest and associated information are encrypted with the signer's asymmetric private key.

3. For each signer, the encrypted message digest and other signer-specific information are collected into a *SignerInfo* value. Certificates and certificate-revocation lists for each signer, and those not corresponding to any signer, are collected in this step.

4. The message-digest algorithms for all the signers and the *SignerInfo* values for all the signers are collected together with the content into a *SignedData* value.

The process by which enveloped data is constructed involves the following steps:

1. A symmetric content-encryption key for a particular content-encryption algorithm is generated at random. JCAPI will by default use the symmetric key algorithm Triple DES (3DES) with CBC block mode and PKCS#7 padding.

2. For each recipient, the content-encryption key is encrypted with the recipient's asymmetric public key.

3. For each recipient, the encrypted content-encryption key and other recipient-specific information are collected into a *RecipientInfo* value.

4. The content is encrypted with the content-encryption key.

5. The *RecipientInfo* values for all the recipients are collected together with the encrypted content into a *EnvelopedData* value.

This class mimics much of the normal encryption process used in the Java Cryptography Extension (JCE) framework i.e. an instance of this class will encode a message through this procedure:

1. Initialize the instance through one of its available `init()` methods by passing an instance of one of the following content values:
    - `JCAPIPKCS7SignedData`
    - `JCAPIPKCS7EnvelopedData`
2. Repeatedly feed content data into one of its `update()` methods.
3. Finalize the encoding session by invoking one of its `doFinal()` methods.

All symmetric- and asymmetric cryptographic operations, and ASN.1/BER coding are performed by a native MS CAPI CSP, which makes these operations very fast.

To learn more about how this class can be used, please look at the following example programs that can be found in JCAPI's examples directory, by default located in `C:\<`*`application data directory`*`>\JCAPI\examples`:

- `pkcs7\PKCS7EnvelopeTest.java`
- `pkcs7\PKCS7SignedDataTest.java`

The decoder

The JCAPIPKCS7Decoder class is used for decoding PKCS#7 signed-data content and enveloped-data content.

The process by which signed data is constructed involves the following steps:

1. For each signer, a message digest (hash) is computed on the content with a signer-specific message-digest algorithm. JCAPI will by default use the algorithm SHA-1. If the signer is authenticating any information other than the content, the message digest of the content and the other information are digested with the signer's message digest algorithm, and the result becomes the *message digest*.
2. For each signer, the message digest and associated information are encrypted with the signer's asymmetric private key.
3. For each signer, the encrypted message digest and other signer-specific information are collected into a *SignerInfo* value. Certificates and certificate-revocation lists for each signer, and those not corresponding to any signer, are collected in this step.

4. The message-digest algorithms for all the signers and the *SignerInfo* values for all the signers are collected together with the content into a *SignedData* value.

The process by which enveloped data is constructed involves the following steps:

1. A symmetric content-encryption key for a particular content-encryption algorithm is generated at random. JCAPI will by default use the symmetric key algorithm Triple DES (3DES) with CBC block mode and PKCS#7 padding.

2. For each recipient, the content-encryption key is encrypted with the recipient's asymmetric public key.

3. For each recipient, the encrypted content-encryption key and other recipient-specific information are collected into a *RecipientInfo* value.

4. The content is encrypted with the content-encryption key.

5. The *RecipientInfo* values for all the recipients are collected together with the encrypted content into a *EnvelopedData* value.

This class mimics much of the normal encryption process used in the Java Cryptography Extension (JCE) framework i.e. an instance of this class will decode a message using this procedure:

1. Initialize the instance through its `init()` method.

2. Repeatedly feed bytes of the encoded PKCS#7 message into one of its `update()` methods.

3. Finalize the decoding session by invoking one of its `doFinal()` methods.

All symmetric- and asymmetric cryptographic operations, and ASN.1/BER coding are performed by a native MS CAPI CSP, which makes these operations very fast.

To learn more about how this class can be used, please look at the following example programs that can be found in JCAPI's examples directory, by default located in `C:\<`*`application data directory`*`>\JCAPI\examples`:

- `pkcs7\PKCS7EnvelopeTest.java`

- `pkcs7\PKCS7SignedDataTest.java`

Chapter
7

PKCS#11

This chapter will show you how to use a PKCS#11 hardware token together with JCAPI when you want to access the private key through the PKCS#11 layer instead of through the MS CAPI layer.
You will learn how to list, add, & delete PKCS#11 CSPs in JCAPI, and how to implement your own PKCS#11 private key PIN call-back class.

JCAPI has built-in support for PKCS#11 hardware tokens. The main reason for this is to bypass MS CAPI when access to private keys stored on the token is required. There are situations when bypassing MS CAPI is required, such as when we don't want the MS CAPI CSP to display a PIN dialog to the user to enter his/her PIN code for accessing the private key, or when the MS CAPI CSP has not implemented the required MS CAPI functions used by JCAPI to create signatures and/or decrypt data.

Managing the PKCS#11 CSPs

JCAPI supports a set of PKCS#11 CSPs. A supported PKCS#11 CSP will have the following properties:

- It has been successfully tested by Pheox with JCAPI.
- Pheox will give support for the CSP in respect to JCAPI related problems.
- JCAPI will skip the MS CAPI layer and access the private key(s) directly through the PKCS#11 layer and thus avoiding problems with certain CSPs that do not implement the required MS CAPI functions for accessing a private key.
- The *Graphical User Interface* (GUI) PIN code dialog window implemented by the CSP will be overridden by a Java Swing based dialog window, which in turn can be replaced by your own dialog if required.

Note: JCAPI do not include drivers for the supported CSPs. These drivers (DLL files) must already have been installed by another party.

The following CSPs are supported by JCAPI in 32-bit environments:

CSP	DLL

eToken Base Cryptographic Provider	eTpkcs11.dll
SafeSign CSP Version 1.0	aetpksse.dll
SI_CSP	SI_PKCS11.dll
AR Base Cryptographic Provider	sadaptor.dll
FTSafe ePass2000 RSA Cryptographic Service Provider	ep2pk11.dll
SmartTrust Cryptographic Service Provider	SmartP11.dll
Athena ASECard Crypto CSP	asepkcs.dll
Datakey RSA CSP	dkck201.dll
Advanced Card Systems CSP v1.5	acospkcs11.dll
SafeNet RSA CSP	dkck201.dll

The following CSPs are supported by JCAPI in 64-bit environments:

CSP	DLL
eToken Base Cryptographic Provider	eTpkcs11.dll

Add your own PKCS#11 CSP into JCAPI

Since JCAPI uses a generic approach to access a CSP, you can yourself add your own PKCS#11 provider into JCAPI if it's not already supported by default. A user-added PKCS#11 CSP will be managed as a supported CSP inside JCAPI, but please note that Pheox will not give support for a user-added CSP if problems arise.

You must supply the following information for JCAPI in order to add your own PKCS#11 provider:

1. The MS CAPI name of the CSP.

2. The name of the PKCS#11 CSP's DLL file. *Note: you do not have to include the directory path if the DLL file resides in a directory that exists in the* `PATH` *environment variable.*

Use the following to list all CSPs available in order to get the name of your CSP:

```
JCAPIUtil.getCSPs();
```

Here's an example of how to add a new PKCS#11 provider into JCAPI:

```
String cspName = ”FooBar Cryptographic Service Provider”;
String fileName = ”foobar.dll”;
JCAPIPKCS11Util().addPKCS11CSP(cspName, fileName);
```

List all registered PKCS#11 CSPs in JCAPI

You can list all PKCS#11 CSPs that are natively supported by JCAPI and those that have been registered additionally (user-added).

Use the following to list all natively JCAPI supported PKCS#11 CSPs:

```
JCAPIPKCS11Util.getSupportedPKCS11CSPs();
```

Use the following to list all PKCS#11 CSPs that have been added into JCAPI:

```
JCAPIPKCS11Util.getAddedPKCS11CSPs();
```

Remove a PKCS#11 CSP from JCAPI

You can remove PKCS#11 CSPs from JCAPI (both natively supported and user-added).

There might be a good reason to remove a PKCS#11 CSP when you want to:

1. Re-define the CSP i.e. for example if you want to change the name of the PKCS#11 CSP DLL.

2. Use the PIN call-back implementation provided by the CSP instead of JCAPI's when access to the private key is needed.

3. Skip PKCS#11 access and use only the CSP's MS CAPI driver due to missing PKCS#11 DLL etc.

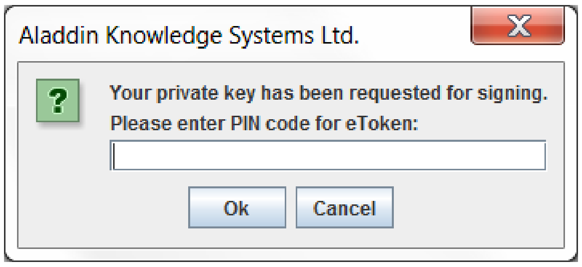
Use the following to remove a PKCS#11 CSP, for example *Datakey RSA CSP*:

```
JCAPIPKCS11Util.removePKCS11CSP(”Datakey RSA CSP”);
```

Implement your own PKCS#11 PIN call-back class

When a protected private key that is stored on a hardware token, is to be accessed, then a GUI dialog window implemented by the CSP will be shown for the user to enter his/her PIN code. JCAPI will use its own Java Swing based dialog window instead of the CSP's own implementation for all PKCS#11 CSPs. JCAPI provides a call-back interface for you to implement if there's a need to customize your own dialog window.

Here is a screenshot of the JCAPI default Java Swing based dialog when a private key, stored on an Aladdin hardware token, is to be accessed:



Your private key has been requested for signing.
Please enter PIN code for eToken:

If you want to provide your own PKCS#11 PIN call-back implementation, then you must implement the JCAPI interface `JCAPIPKCS11PINCallback`. It has only one method specification that must be implemented:

- `String getPINCode(String slotManufacturer, String tokenLabel, String alias, int operation)`

Your implementation of method `getPINCode` will be called by the JCAPI DLL every time access to a private key stored a token is required. JCAPI will, through the input parameters, provide you with the necessary information for you to decide what private key that is accessed and for what operation the private key is accessed. The input parameters provided are:

- `slotManufacturer` - The name of the manufacturer for the chosen PKCS#11 slot.
- `tokenLabel` - The label of the token for the chosen PKCS#11 slot.
- `alias` - The alias of the certificate whose associated private key is about to be accessed.
- `operation` - The purpose of accessing the private key. It can be either `OPERATION_SIGNING` or `OPERATION_DECRYPTION`.

But before JCAPI can call your own implementation, you have to register your class through the static method `setPINCallback(JCAPIPKCS11PINCallback)`.

Here is an example of how to create and register a simple “silent” call-back implementation which will override the Swing based PIN dialog provided by JCAPI.

```
final static class MyPINCallback implements
JCAPIPKCS11PINCallback
{
  public String getPINCode(String slotManufacturer, String
tokenLabel, String alias, int operation)
  {
    System.out.println("Gave PIN for " + tokenLabel + " on " +
slotManufacturer + ".");
    String s;
    switch(operation)
    {
      case JCAPIPKCS11PINCallback.OPERATION_SIGNING :
        s = "signing";
        break;
      case JCAPIPKCS11PINCallback.OPERATION_DECRYPTION :
        s = "decryption";
        break;
      default :
        s = "unknown";
    }
    System.out.println("The PIN was given for " + s + "
purpose.");
    System.out.println("JCAPI alias for accessed certificate on
PKCS#11 token is: " + alias);
    return "1234"; //The clever PIN code.
  }
}
...
JCAPIPKCS11Util.setPINCallback(new
RegisterPINCallback.MyPINCallback());
```

Getting PKCS#11 token information

JCAPI can provide you with detailed information about a PKCS#11 hardware token that is plugged into your system. This information is encapsulated into an instance of class `JCAPIPKCS11TokenInfo` which can be retrieved from the static method `getPKCS11TokenInfo(String)` in class `JCAPIPKCS11Util`. This method will take a JCAPI certificate alias as argument and return a `JCAPIPKCS11TokenInfo` instance if the associated certificate can be accessed through a PKCS#11 connected token, else `null` is returned.

The `JCAPIPKCS11TokenInfo` class contains the following getters:

- `String getJCAPIAlias()` - Returns the JCAPI alias of this PKCS#11 entry.
- `String getCSP()` - Returns the name of the CSP used for managing this PKCS#11 entry.

- `String getDLLName()` - Returns the name of the PKCS#11 DLL file, including absolute path, used by the PKCS#11 CSP for this entry.
- `String getTokenLabel()` - Returns the token label for this PKCS#11 entry.
- `String getTokenManufacturer()` - Returns the token manufacturer name for this PKCS#11 entry.
- `String getTokenSerialNumber()` - Returns the token serial number for this PKCS#11 entry.
- `String getSlotDescription()` - Returns the slot description for this PKCS#11 entry.
- `String getSlotManufacturer()` - Returns the slot manufacturer name for this PKCS#11 entry.
- `int getSlotId()` - Returns the slot id number for this PKCS#11 entry.
- `int getSlotIndex()` - Returns the slot index for this PKCS#11 entry.

This example will try to get the token information for each available alias:

```
Security.addProvider(new JCAPIProvider());
KeyStore ks = KeyStore.getInstance("msks-MY", "JCAPI");
ks.load(null, null);

JCAPIPKCS11TokenInfo info = null;
for(java.util.Enumeration e = ks.aliases(); e.hasMoreElements(); )
{
  String alias = (String)e.nextElement();
  info = JCAPIPKCS11Util.getPKCS11TokenInfo(alias);
  if(info != null)
  {
    System.out.println("JCAPI alias: " + info.getJCAPIAlias());
    System.out.println("MS CAPI CSP: " + info.getCSP());
    System.out.println("PKCS#11 DLL file: " + info.getDLLName());
    System.out.println("Token label: " + info.getTokenLabel());
    System.out.println("Token manufacturer: " +
info.getTokenManufacturer());
    System.out.println("Token model: " + info.getTokenModel());
    System.out.println("Token serial number: " +
info.getTokenSerialNumber());
    System.out.println("Slot description: " +
info.getSlotDescription());
    System.out.println("Slot manufacturer: " +
info.getSlotManufacturer());
    System.out.println("Slot ID: " + info.getSlotId());
    System.out.println("Slot index: " + info.getSlotIndex());
  }
}
```

Getting PKCS#11 provider information

JCAPI can also provide you detailed information about the PKCS#11 library used by a specific PKCS#11 CSP through the immutable class `JCAPIPKCS11ProviderInfo`. This class encapsulates the CK_INFO structure as defined in PKCS#11 v2.20, and it contains general information about an available (i.e. user added or JCAPI supported) PKCS#11 library, such as the PKCS#11 version number that this library supports, the manufacturer identity of this library etc. In addition to the CK_INFO structure, this class also provides information about the MS CAPI CSP name for this provider, and the file name of the loaded PKCS#11 DLL.

An instance of the `JCAPIPKCS11ProviderInfo` class can be retrieved through any of the following static methods located in the class `JCAPIPKCS11Util`.

- `JCAPIPKCS11ProviderInfo getPKCS11ProviderInfoByAlias(String alias)` - Returns general provider information about the PKCS#11 library that is used for accessing the key/certificate entry as given by parameter `alias`.

- `JCAPIPKCS11ProviderInfo getPKCS11ProviderInfoByCSP(String cspName)` - Returns general provider information about the PKCS#11 library that is used by the CSP as given by parameter `cspName`. Note that you don't have to attach a hardware token to your computer in order to get this information; installed CAPI and PKCS#11 drivers (DLL files) will suffice.

This example will list detailed information of the PKCS#11 library used for each JCAPI supported PKCS#11 CSP.

```
Security.addProvider(new JCAPIProvider());
System.out.println("List detailed PKCS#11 library information for
each JCAPI supported PKCS#11 CSP.");
String[] csps = JCAPIPKCS11Util.getSupportedPKCS11CSPs();
for(int i = 0; i < csps.length; i++)
{
  JCAPIPKCS11ProviderInfo info =
JCAPIPKCS11Util.getPKCS11ProviderInfoByCSP(csps[i]);
  if(info != null)
  {
    System.out.println("\n* Found the following information about
CSP: " + csps[i]);
    System.out.println("\nCryptoki version: " +
info.getCryptokiVersion());
    System.out.println("MS CAPI CSP: " + info.getCSP());
    System.out.println("PKCS#11 DLL file: " + info.getDLLName());
    System.out.println("Flags: " + info.getFlags());
    System.out.println("Library description: " +
info.getLibraryDescription());
    System.out.println("Library version: " +
info.getLibraryVersion());
    System.out.println("Manufacturer ID: " +
info.getManufacturerId());
  }
  else
    System.out.println("\nCould not find any information about
PKCS#11 CSP: " + csps[i]);
}
```

Chapter

8

SSL/TLS

Learn how to use JCAPI for SSL/TSL purposes. You will get to know how to configure your java program, or system, with JCAPI for seamless SSL/TLS support even though the private key(s) in your MS CAPI stores are protected (i.e. not exportable).
We will also discuss and explain how to use the JCAPI key store types to make it easier for you to define your default key- and trust stores.

JCAPI can be seamlessly used together with other SSL/TLS frameworks, such as JSSE, when MS CAPI based trust- and key stores are required. It can even be used for keys that are protected by MS CAPI i.e. private keys that are not exportable.

There are two ways of configuring JCAPI for SSL/TLS; either externally through JSSE defined properties, or programmatically in the source code. Configuring JCAPI through properties has the advantage that the user does not have re-compile the program.

Configuring JCAPI through properties

Here is how to configure JSSE to use JCAPI as its key store. We will use the MS CAPI system store *MY* as the key store i.e. where JSSE can find the required private keys from:

```
javax.net.ssl.keyStoreProvider="JCAPI"
javax.net.ssl.keyStoreType="msks-MY"
```

Here is how to configure JSSE to use JCAPI as its trust store. We will use the MS CAPI system store *ROOT* as the trust store i.e. where JSSE can find all trusted certificates from:

```
javax.net.ssl.trustStoreProvider="JCAPI"
javax.net.ssl.trustStoreType="msks-ROOT"
```

For more information, please examine and execute the various SSL example programs located in the JCAPI examples directory `examples/ssl`.

Configuring JCAPI programmatically

Here is how to create an `SSLSocketFactory` instance which will use JCAPI as its key store and trust store. We will use the MS CAPI system store *MY* as the key store, and the MS CAPI system store *ROOT* as the trust store:

```
SSLContext c = SSLContext.getInstance("TLS");

KeyStore keyStore = KeyStore.getInstance("msks-MY", "JCAPI");
keyStore.load(null, null);
KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgori
thm());
kmf.init(keyStore, null);

KeyStore trustStore = KeyStore.getInstance("msks-ROOT",
"JCAPI");
trustStore.load(null, null);
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAl
gorithm());
tmf.init(trustStore);

SecureRandom sr = SecureRandom.getInstance("RNG", "JCAPI");
c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), sr);
SSLSocketFactory sf = c.getSocketFactory();
```

For more information, please examine and execute the various SSL example programs located in the JCAPI examples directory `examples/ssl`.

Key store types

JCAPI provide four different key store types which are suitable for SSL/TLS purposes:

Type	Description
msks-MY	Access and store certificates, and private keys only from the existing MS CAPI system store *MY*.
msks-ROOT	Access and store certificates, and private keys only from the existing MS CAPI system store *ROOT*.
msks-KEYSTORE	Used as a reserved key store for holding user defined key entries when SSL/TLS is used. Usually when we want to setup an SSL/TLS connection, we want the server and client to define and use their own keys and certificates, and thus exclude everything else that are stored in the MS CAPI default stores *MY* and *ROOT*.

	When a `KeyStore` instance is created from this type, it will automatically create a MS CAPI system store named *KEYSTORE* (note: you can override this name with the Java property `jcapi.ssl.keystore`) if it does not already exist. This store can then be used for you to only hold your own specific key entries in SSL/TLS.
msks-TRUSTSTORE	Used as a reserved trust store for holding user defined root CA certificates when SSL/TLS is used. Usually when we want to setup an SSL/TLS connection, we want the server and client to define and use their own keys and certificates, and thus exclude everything else that are stored in the MS CAPI default stores *MY* and *ROOT*. When a `KeyStore` instance is created from this type, it will automatically create a MS CAPI system store named *TRUSTSTORE* (note: you can override this name with the Java property `jcapi.ssl.truststore`) if it does not already exist. This store can then be used for you to only hold your own specific certificate entries in SSL/TLS.

For more information, please examine and execute the various SSL example programs located in the JCAPI examples directory `examples/ssl`.

Chapter
9

Dealing with errors

Learn how to enable the logging features and to prepare the procedure of gathering environmental information needed for error reporting when you encounter an error in JCAPI.

When you encounter an error when using JCAPI and suspect it is related to JCAPI itself, then please first do the following to find an eventual solution to your problem:

1. Read the JCAPI *Frequently Asked Questions* (FAQ) document that is included in the installation package. You will find this HTML document in your program menu by clicking *Start -> All Programs -> JCAPI -> FAQ*

2. Visit the Pheox JCAPI forum (www.pheox.com/forums) and search for problem descriptions similar to yours. The problem might have already been solved, for example, in a more recent version of JCAPI than yours.

If your problem can still not be solved, then prepare an error report which should include the following information:

1. Write down a detailed error description including existing stack trace(s).

2. Create the smallest possible test program which triggers the error.

3. Enable JCAPI internal logging by calling the following method in your test program and include the resulting log output:
   `JCAPIProperties.setLogging(true);`

4. Include the output from the following method call in your test program:
   `JCAPIUtil.getEnvironmentInfo();`

Depending on your JCAPI support license agreement with Pheox, you can do one of the following to notify us about your error report:

- *Silver Support* – Send your error report to support@pheox.com and include your license agreement code. You can also give us your error report at www.pheox.com/support. Your report will be processed immediately.

- *Basic Support* – Visit our free support forum at www.pheox.com/forums and submit your error report at the JCAPI forum. We will process your query as soon as possible.

Chapter

10

Contact Information

Web Site

www.pheox.com

Product Home Page

www.pheox.com/products/jcapi

Support

Silver Support – Please send an e-mail including your error report or question(s) to support@pheox.com, or submit your error report or question(s) at www.pheox.com/support. Do not forget to include your license agreement code.

Basic Support – Please submit your error report or question(s) at our free support forum at www.pheox.com/forums

Sales

Send your questions to sales@pheox.com

Representative Contact

Tommy Grändefors
Phone: +46 10 150 05 33
E-mail: tommy.grandefors@pheox.com

Company Address

Pheox AB
Ställverksvägen 3
SE-37439 Karlshamn
Sweden

Chapter 11

Acknowledgements

Pheox would like to thank certain individuals and organisations for direct or indirect involvement in JCAPI in terms of provided hardware, tools, documents etc.

Eutron Infosecurity

Special thanks to Eutron Infosecurity for providing us with free software and hardware for their hardware token products:

- Cryptoidentity 5
- Cryptoidentity ITSEC-I
- Cryptoidentity ITSEC-P

Feitian Technologies

Special thanks to Feitian Technologies for providing us with free software and hardware for their FTSafe ePass2000 hardware token product.

Aladdin

Special thanks to Aladdin for providing us with free software and hardware for the Aladdin eToken PRO hardware token product.

SafeNet

Special thanks to SafeNet for providing us with free software and hardware for their Rainbow iKey 2032 USB hardware token product.

Jordan Russel

Special thanks to Jordan Russel for his excellent Inno Setup installation program which is used for packaging the JCAPI product. Keep up the good work!

RSA Security Inc.

JCAPI uses PKCS#11 header files provided by RSA Security Inc. The following is acknowledged: