SafeNet ProtectToolkit J JCA/JCE API

Overview

gemalto
security to be free

Part Number 007-008398-007
Version 5.2

Trademarks

Gemalto Rebranding

In early 2015, Gemalto NV completed its acquisition of SafeNet, Inc. As part of the process of rationalizing the product portfolios between the two organizations, the HSM product portfolio has been streamlined under the SafeNet brand. As a result, the ProtectServer/ProtectToolkit product line has been rebranded as follows:

Old product name	New product name
Protect Server External 2 (PSE2)	SafeNet ProtectServer Network HSM
Protect Server Internal Express 2 (PSI-E2)	SafeNet ProtectServer PCIe HSM
ProtectToolkit	SafeNet ProtectToolkit

Disclaimer

All information herein is either public information or is the property of and owned solely by Gemalto NV. and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Gemalto's information.

This document can be used for informational, non-commercial, internal and personal use only provided that:

- The copyright notice below, the confidentiality and proprietary legend and this full warning notice appear in all copies.
- This document shall not be posted on any network computer or broadcast in any media and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Gemalto makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Gemalto reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Gemalto hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Gemalto be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Gemalto does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Gemalto be held liable for any third party

actions and in particular in case of any successful attack against systems or equipment incorporating Gemalto products. Gemalto disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service or loss of privacy.

Technical Support

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, please contact your supplier or Gemalto support. Gemalto support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Gemalto and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact method	Contact	
Address	Gemalto NV 4690 Millennium Drive Belcamp, Maryland 21017 USA	
Phone	Global	+1 410-931-7520
	Australia	1800.020.183
	China	(86) 10 8851 9191
	France	0825 341000
	Germany	01803 7246269
	India	000.800.100.4290
	Netherlands	0800.022.2996
	New Zealand	0800.440.359
	Portugal	800.1302.029
	Singapore	800.863.499
	Spain	900.938.717
	Sweden	020.791.028
	Switzerland	0800.564.849
	United Kingdom	0800.056.3158
	United States	(800) 545-6608
Web	www.safenet-inc.com	
Support and Downloads	www.safenet-inc.com/support Provides access to the Gemalto Knowledge Base and quick downloads for various products.	
Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the Gemalto Knowledge Base.	

Revision History

Revision	Date	Reason
A	14 March 2016	Release 5.2

TABLE OF CONTENTS

1.0 SCOPE

The purpose of this document is to provide an introduction to the Java APIs that provide security and cryptographic services. These are known as the Java Cryptography Architecture (JCA) and Java Cryptography Extensions (JCE).

While reading this document, it is suggested you have both the JCA/JCE API documentation at hand. The JCA documentation can be found in the Java2 release or online at:

http://java.sun.com/products/archive/j2se/1.3.0_05/ (or later version)

JCE documentation is currently available at http://java.sun.com/products/jce/index.html.

Finally, ProtectToolkit J includes a detailed reference manual detailing the specific algorithms included, and the various parameters they accept. It also includes some extensions to the base JCA/ JCE API.

This document assumes the reader is familiar with the Java programming language.

THIS PAGE INTENTIONALLY LEFT BLANK

2.0 INTRODUCTION

The Java platform provides APIs for dealing with security and cryptographic services. The first is known as the Java Cryptography Architecture (JCA) and provides a framework for basic security functions such as certificates, digital signatures and message digests.

The JCA and a default provider (the Sun provider) are included with Java 2. The Java Cryptography Extension (JCE) extends the JCA to provide encryption, key exchange, key generation and message authentication services. The Java Cryptographic Extension (JCE) is released as a standard extension to the Java2 platform.

The JCA/ JCE do not directly provide specific implementations of the various algorithms. Rather they are an interface between the application and a number of specific implementations of the algorithms. Generally a vendor will group the algorithms they have developed into a `Provider` which may then be installed into the Java runtime environment. Once installed, an application may specifically request a particular provider's implementation or, if not the framework will choose an implementation from the highest priority `Provider` that implements the requested algorithm.

This architecture is achieved by providing "factory classes" that are used to create object instances that implement a specific algorithm. Each of the factory classes has a `private` constructor, instances can only be constructed by calling a `public static` method which returns an instance of the desired type. When an algorithm is requested, the factory class will iterate through the installed providers and return the first implementation it finds (unless the application has requested a specific `Provider`).

The `java.security.Security` class is responsible for maintaining a list of available providers. When this class is initialised it will read the `java.security` properties file (which is located at `$JAVA_HOME/lib/security/java.security`).

This properties file has a list of the installed providers, ordered by preference. For example the Sun and Acme providers could be listed as;

security.provider.1=sun.security.provider.Sun
security.provider.2=org.acme.crypto.provider.Acme

A `Provider` may also be installed dynamically by an application at runtime. This is achieved by using the `Security.addProvider()` method, passing the `Provider` instance of the vendor to be installed. For example:

```
Security.addProvider(new org.acme.crypto.provider.Acme());
```

The following packages are provided as part of the JCA:

`java.security`

Provides the interfaces for the security framework. Generally, these classes do not have `public` constructors, rather they consist of factory methods which will create `Provider` based implementations of the requested algorithms. Here you will find the `KeyFactory`, `KeyPairGenerator`, `KeyStore`, `MessageDigest` and `Signature` classes.

`java.security.cert`

The interfaces for parsing and managing certificates, in particular X.509 v3 certificates.

`java.security.interfaces`

The provider-independent interfaces for dealing with RSA and DSA public/private keys.

`java.security.spec`

The provider-independent interfaces for key and algorithm specifications.

The following packages are provided as part of the JCE:

`javax.crypto`

The core services provided by the JCE. Here you will find the `Cipher`, `KeyGenerator` and `Mac` classes.

`javax.crypto.interfaces`

Provider-independent interfaces for Diffie-Hellman keys.

`javax.crypto.spec`

Provider independent specifications for DES, DESede, Diffie-Hellman and various other keys and algorithm parameters.

The following is a very simple program that will encrypt the string "hello world" with a randomly generated key, and then decrypt the cipher text using the same key. Note for this program to run successfully a JCE provider that includes the CAST128 algorithm must be installed.

```
import javax.crypto.*;
import javax.crypto.spec.*;

public class HelloJCE
{
   static final plainText = "hello world".getBytes();

   public static void main(String args)
   throws Exception
   {
           KeyGenerator keyGen = KeyGenerator.getInstance("CAST128");
           keyGen.init(128);
           Key key = keyGen.generateSecret();

           Cipher cipher =
                   Cipher.getInstance("CAST128/CBC/PKCS5Padding");
           cipher.init(Cipher.ENCRYPT_MODE, key);
           byte[] cipherText = cipher.doFinal(plainText);

           IvParameterSpec param =
                   new IvParameterSpec(cipher.getIV());
           cipher.init(Cipher.DECRYPT_MODE, key, param);
           byte[] text = cipher.doFinal(cipherText);

           System.out.println("decrypted text: " + new String(text));
   }
}
```

3.0 ENCRYPTION/DECRYPTION

The JCE supports encryption and decryption using symmetric algorithms (such as DES and RC4) and asymmetric algorithms (such as RSA and ElGamal). The algorithms may be stream or block ciphers, with each algorithm supporting different modes, padding or even algorithm-specific parameters.

3.1 The Cipher Class

The basic interface used to encipher or decipher data is the `javax.crypto.Cipher` class. The class provides the necessary mechanism for encrypting and decrypting data using arbitrary algorithms from any of the installed providers.

To create a `Cipher` instance, use one of the `Cipher.getInstance()` methods. This method will accept a transformation string and an optional provider name. The transformation string is used to specify the encryption algorithm as well as the cipher mode and padding. The transformation is specified in the form;

- "algorithm"
- "algorithm/mode/padding"

In the first instance, we are requesting the algorithm with its default mode and padding mechanism. The second instance fully qualifies all options. For a list of support algorithms consult the provider's documentation. Some common transformations are;

- "RC4"
- "DES/CBC/PKCS5Padding"
- "RSA/ECB/PKCS1Padding"

The following code will create a cipher for performing RC4 encryption or decryption, a cipher for doing RSA in ECB mode with PKCS#1 padding provided by the ABA provider and a cipher for performing DESede encryption/decryption in CBC mode with PKCS#5 padding:

```
Cipher rc4Cipher = Cipher.getInstance("RC4");
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
Cipher desEdeCipher =
                Cipher.getInstance("DESede/CBC/PKCS5Padding");
```

Once we have a `Cipher` instance, we will need to initialise the `Cipher` for encryption or decryption. We will also need to provide a `Key`, see section 8.0 for a discussion of key management.

```
Key desKey, rsaKey;

desCipher.init(Cipher.ENCRYPT_MODE, desKey);
rsaCipher.init(Cipher.DECRYPT_MODE, rsaKey);
```

As you can see, the first value passed to the `Cipher.init()` method indicates whether we are initialising for encryption or decryption. The second argument provides the key to use during encryption or decryption.

There are a number of other initialisation methods for providing algorithm specific parameters (such as Initialisation Vectors, the number of rounds to use etc.). See section 4.4 for a discussion on algorithm parameters.

Now that our `Cipher` is initialised, we can start processing data. To do so we use the `Cipher.update()` and `Cipher.doFinal()` methods. The `Cipher.update()` methods may be used to incrementally process data. Once all the data is processed, one of the `Cipher.doFinal()` methods must be called.

In the simplest usage, a single `Cipher.doFinal()` call may be passed all the data:

```
byte[] plainText = "hello world".getBytes();
byte[] cipherText = desCipher.doFinal(plainText);
```

Once the `Cipher.doFinal()` method has been called, the `Cipher` instance will be reset to the state it was in after the last call to the `Cipher.init()` method. That means the `Cipher` may be reused to encipher or decipher more data using the same `Key` and parameters that were specified in the initialisation.

3.2 Cipher Input and Output Streams

Rather than deal with the complications of buffering enciphered or deciphered data produced by the `Cipher.update()` methods, it may be desirable to use a Java Input/Output Stream type interface. Fortunately, the JCE provides us with such a mechanism.

The `javax.crypto.CipherInputStream` and `javax.crypto.CipherOutputStream` are based on the Java IO filter streams. This allows them to process data and pass on that data to an underlying stream.

To create a cipher stream, firstly create and initialise a `javax.crypto.Cipher` instance and the underlying stream and then instantiate the required stream type with these two arguments.

For example, the following code fragment will create a `CipherOutputStream` that will encipher its data (using DES) and pass the result to a `ByteArrayOutputStream`. We can access the ciphertext by calling `ByteArrayOutputStream.toByteArray()`.

```
Key desKey;
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);

ByteArrayOutputStream bout = new ByteArrayOutputStream();
CipherOutputStream cout =
                    new CipherOutputStream(bout,  cipher);
cout.write("hello world".getBytes());
cout.close();

byte[] cipherText = bout.toByteArray();
```

Once we can encipher and decipher data using a simple stream, interface, we can create much more complicated scenarios. For example the `OutputStream` could just as easily be a `SocketOutputStream` or we could construct an `ObjectOutputStream` on top of our cipher stream and encipher Java objects directly.

3.3 SealedObject

The `javax.crypto.SealedObject` class provides the mechanism to encipher a `Serializable` object. This class allows the application to encipher a Java object and then recover the object all through a simple interface. The `SealedObject` is also `Serializable` to simplify the transport and storage of the enciphered objects.

A `SealedObject` can be constructed through either serialisation or by its constructor. The constructor is used to create a new enciphered object. The constructor's arguments are the object to encipher and the `Cipher` to use. The provided `Cipher` instance must be initialised for encryption before the `SealedObject` is created. This means calling a `Cipher.init()` method with `Cipher.ENCRYPT_MODE` as the mode, the required encryption `Key` and any algorithm parameters.

The following fragment will create a new `SealedObject` containing the enciphered `String` "hello world":

```
Key desKey = ...
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, deskey);

SealedObject so = new SealedObject("hello world", cipher);
```

To recover the original object, the `SealedObject.getObject()` methods may be used. These methods take either a `Cipher` or `Key` object. When providing the `Cipher` parameter, the instance must be initialised in the `Cipher.DECRYPT_MODE` mode, with the appropriate decryption key and the same algorithm parameters as the original `Cipher`. When providing a `Key` parameter, the encryption algorithm and algorithm parameters are extracted from the `SealedObject`.

The following fragment will extract a `SealedObject` object from an `ObjectInputStream` and then recover the protected object:

```
ObjectInputStream oin ...
Key desKey = ...

SealedObject so = (SealedObject)oin.readObject();
String plainText = (String)so.getObject(deskey);
```

One important security aspect to note with this class is that it does not use a digital signature to ensure the object is not tampered with in its serialised form. It is therefore possible that the object could be altered in storage or transport without detection. Fortunately, the JCA provides the `java.security.SignedObject` mechanism which can be used in conjunction with the `SealedObject` class to avoid this problem. (See section 7.2 for a discussion on the `SignedObject` class).

3.4 Algorithm Parameters

Some cipher algorithms support parameterisation, for example the DES cipher in CBC mode can have an initialisation vector as an algorithm parameter and other ciphers may have a selectable block size or round count. The JCE provides support for algorithm-independent initialisation via the `java.security.spec.AlgorithmParameterSpec` and `java.security.AlgorithmParameters` classes.

The `java.security.spec.AlgorithmParameterSpec` derived classes can be constructed programatically by an application. The following classes are provided by the JCA/JCE:

java.security.spec	
`DSAParameterSpec`	Used to specify the parameters used with the DSA algorithm. The parameters consist of the base g, prime p and sub-prime q.
javax.crypto.spec	
`DHGenParameterSpec`	The set of parameters used for generating Diffie-Hellman parameters for use in Diffie-Hellman key agreement.
`DHParameterSpec`	The set of parameters used with Diffie-Hellman as specified in PKCS#3.
`IvParameterSpec`	An initialisation vector for use with a feedback cipher. That is an array of bytes of length equal to the block size of the cipher.
`RC2ParameterSpec`	Parameters for the RC2 algorithm. The parameters are the effective key size and an optional 8-byte initialisation vector (only in feedback mode).
`RC5ParameterSpec`	Parameters for the RC5 algorithm. The parameters are a version number, number of rounds, a word size and an optional initialisation vector (only in feedback mode).

Your provider may also include further classes for passing parameters to the algorithms it implements.

The JCA also has mechanisms for dealing with the provider-dependent `AlgorithmParameters`. This class is used as an opaque representation of the parameters for a given algorithm and allows an application to store persistently the parameters used by a `Cipher`.

There are three situations where an application may encounter an `AlgorithmParameters` instance:

1. `Cipher.getParameters()`
   After a `Cipher` has been initialised, it may have generated a set of parameters (based on supplied and/or default values). The value returned by the `getParameters()` method allows the `Cipher` to be re-initialised to exactly the same state.

2. `AlgorithmParameters.getInstance()`
   Rather than generating the parameters via the `Cipher` class, it is possible to generate them either based on an encoded format or an `AlgorithmParameterSpec` instance. To do so create an uninitialised instance using the `getInstance` method and then initialise it by calling the appropriate `init()` method.

3. `AlgorithmParameterGenerator.getParameters()`
   Finally, a set of parameters can be generated using the `AlgorithmParameterGenerator`. Firstly, a generator is created for the required algorithm using the `getInstance()` method. Then the generator is initialised by calling one of the `init()` methods, finally to create the instance use the `getParameters` method.

This class provides the concept of algorithm-independent parameter generation, in that the initialisation can be based on a "size" and a source of randomness. In this case the "size" value is interpreted differently for each algorithm.

4.0 MESSAGE DIGESTS

The JCA provides support for the generation of message digests via the `java.security.MessageDigest` class. This class uses the standard factory class design, so to create a `MessageDigest` instance use the `getInstance()` method with the desired algorithm name and optional provider as parameters.

Once created use the various `update()` methods to process the message data and then finally call the `digest()` method to calculate the final digest. At this point the instance may be re-used to calculate a digest for a new message.

```
MessageDigest digest = MessageDigest.getInstance("SHA");

byte[] msg = "The message".getBytes();
digest.update(msg);

byte[] result = digest.digest();
```

THIS PAGE INTENTIONALLY LEFT BLANK

5.0 MESSAGE AUTHENTICATION CODE (MAC)

The `javax.crypto.Mac` API is used to access a "Message Authentication Code" (MAC) algorithm. These algorithms are used to check the integrity of messages upon receipt. There are two classes of MAC algorithms in general, those that are based on message digests (known as HMAC algorithms) and those on encryption algorithms. In both cases a shared secret is required.

A `Mac` is used in the same fashion as a `Cipher`. First, use the factory method `Mac.getInstance()` to get the provider implementation of the required algorithm, then initialise the algorithm with the appropriate key via the `Mac.init()` method. Then, use the `Mac.update()` method to process the message and finally use the `Mac.doFinal()` method to calculate the MAC for the message.

To verify the message, follow the same procedure and compare the supplied MAC with the calculated MAC.

Note that it is not necessary to use the `Mac.init()` method to check multiple messages if the shared secret has not changed. The `Mac` will be reset after the call to `Mac.doFinal()` (or a call to `Mac.reset()`).

```
/*
 * on the sender
 */
Mac senderMac = Mac.getInstance("HMAC-SHA1");
senderMac.init(shaMacKey);
byte[] mac = senderMac.doFinal(data);

/*
 * now transmit message and mac to receiver
 */
Mac recMac = Mac.getInstance("HMAC-SHA1");
recMac.init(shaMacKey);
byte[] calcMac = recMac.doFinal(data);

for (int i = 0; i < calcMac.length; i ++)
{
   if (calcMac[i] != mac[i])
   {
      /* bogus MAC! */
      return false;
   }
}

/* all okay */
return true;
```

6.0 AUTHENTICATION

6.1 Digital Signatures

The `java.security.Signature` class provides the functionality of a digital signature algorithm. Digital signatures are the digital equivalent of the traditional pen and paper signature. They can be used to authenticate the originator of a document, as well as to prove that a person signed the document. Generally, digital signatures are based on public-key encryption which means that, unlike a MAC, anyone that has access to the public key (and the document) can check the validity of the document.

The `Signature` interface supports generation and verification of signatures. Once a signature instance has been created using the `Signature.getInstance()` method, it needs to be initialised with the `Signature.initSign()` method for creation of a signature, or `Signature.initVerify()` method for verification of a signature.

Once initialised, the document to be processed should be passed to the signature via the `Signature.update()` methods. Once the entire document has been processed, the `Signature.sign()` method may be called to generate the signature, or the `Signature.verify()` method to verify a supplied signature against a previously generated signature.

After a signature has been generated or verified, the `Signature` instance is reset to the state it was in after it was last initialised, allowing another signature to be generated or verified using the same key.

One such signature algorithm is "MD5 with RSA" and is defined in PKCS#1. This algorithm specifies that the document to be signed is passed through the MD5 digest algorithm and then an ASN.1 block containing the digest, along with a digest algorithm identifier, is enciphered using RSA.

To create such a signature;

```
/*
 * Assume this private key is initialised
 */
PrivateKey rsaPrivKey;

/*
 * Create the Signature instance and initialise
 * it for signing with our private key
 */
Signature rsaSig = Signature.getInstance("MD5withRSA");
rsaSig.initSign(rsaPrivKey);

/*
 * Pass in the document data via the update() methods
 */
byte[] document = "The document".getBytes();
rsaSig.update(document);

/*
 * Generate the signature
 */
byte[] signature = rsaSig.sign();
```

To verify the generated signature:

```
/*
 * Assume this public key is initialised
 */
PublicKey rsaPubKey;

/*
 * Create the Signature instance and initialise
 * it for signature verification with the public key
 */
Signature rsaSig = Signature.getInstance("MD5withRSA");
rsaSig.initVerify(rsaPubKey);

/*
 * Pass in the document data via the update() methods
 */
byte[] document = "The document".getBytes();
rsaSig.update(document);

/*
 * Check the generated signature against the supplied
 * signature
 */
if (rsaSig.verify(signature))
{
   // signature okay
}
else
{
   // signature fails
}
```

6.2 Object Signing

The `java.security.SignedObject` provides a mechanism for ensuring that a Java object can be authenticated and cannot be tampered with without detection. The mechanism used is similar to the `SealedObject` in that the object to be protected is serialised and then a signature is attached. The `SealedObject` is `Serializable` so it may be stored or transmitted via the object streams.

To create a `SignedObject`, firstly create an instance of the signature algorithm to use via the `Signature.getInstance()` method, then create the new `SignedObject` instance by providing the object to be signed, the signing key and the `Signature` instance. Note that there is no need to initialise the `Signature` instance; the `SignedObject` constructor will perform that function.

```
Signature signingEngine = Signature.getInstance(
   "MD5withRSA");
SignedObject so = new SignedObject("hello world",
   privateKey, signingEngine);
```

To verify a `SignedObject`, simply create the `Signature` instance for the required algorithm and then use the `SignedObject.verify()` method with the appropriate `PublicKey`. Again, there is no need to initialise the `Signature` instance.

```
Signature verifyEngine = Signature.getInstance(
   "MD5withRSA");
if (so.verify(publicKey, verifyEngine))
{
   // object okay, extract it
   Object obj = so.getObject();
}
else
{
   // object not authenticated
}
```

Note that this class only provides a mechanism for authentication and verification, it does not provide confidentiality (i.e. encryption). The `SealedObject` may be used for this purpose (see section 4.3). The following example combines these two classes to provide a confidential, authenticated, tamper-proof object:

```
/*
 * sealedObj will contain the signed, enciphered data
 */
SignedObject signedObj = new SignedObject(
   "hello world", privateKey, signingEngine);
SealedObject sealedObj = new SealedObject(
   signedObj, cipher);

/*
 * to verify and recover the original object
 */
SignedObject newObj = sealedObject.getObject(cipher);
if (newObj.verify(publicKey, verificationEngine))
{
   // object verified tampered
   String str = (String)newObj.getObject();
}
else
{
   // object tampered with!
}
```

7.0 KEY MANAGEMENT

The JCA/JCE framework manages keys in two forms, a provider-dependent format and a provider-independent format.

The provider-dependent keys will implement either the `java.security.Key` interface (or one of its sub classes) for public-key algorithms or the `javax.crypto.SecretKey` interface for secret-key algorithms. Provider keys can be generated randomly, via a key agreement algorithm or from their associated provider-independent format.

The provider-independent formats will implement the `java.security.spec.KeySpec` interface. Subclasses of this type exist for both specific key types and for different encoding types. For example, the `java.security.spec.RSAPublicKeySpec` can be used to construct an RSA public key from its modulus and exponent and a `java.security.spec.PKCS8EncodedKeySpec` can be used to construct a private key encoded using PKCS#8.

Each `Provider` will supply a number of mechanisms that will create the provider-dependent keys or convert the provider-independent keys into provider based keys.

7.1 Generating Random Keys

The simplest mechanism to create keys for a given provider is to use their random key generators. Random keys are most often generated for use as "session-keys" which will be used for a given dialogue or session and are then no longer required. In the case of public-key algorithms, however, they may be generated once and then stored for later use. The JCE framework provides key generation via the following classes:

`javax.crypto.KeyGenerator`

Generation of symmetric keys (ie DES, IDEA, RC4)

`java.security.KeyPairGenerator`

Generation of public/private key pairs (ie RSA)

For instance, to create a random 128-bit key for RC4 and initialise a `Cipher` for encryption with this key;

```
/*
 * Create the key generator for the desired algorithm,
 * and then initialise it for the required key size.
 */
KeyGenerator rc4KeyGen = KeyGenerator.getInstance("RC4");
rc4KenGen.init(128);

/*
 * Generate the key and then initialise the Cipher
 */
SecretKey rc4Key = rc4KeyGen.generateKey();
Cipher rc4Cipher = Cipher.getInstance("RC4");
rc4Cipher.init(Cipher.ENCRYPT_MODE, rc4Key);
```

Here, the `SecretKey` returned by the `KeyGenerator.generateKey()` method is a provider-dependent key. The returned key can then be used with that provider's algorithms.

Some algorithms have keys that are considered *weak*, for example with a weak DES key the ciphertext may be the same as the plaintext! Generally the `KeyGenerator` will not generate those keys, however it is best to check the provider documentation for details on the specific algorithm.

The code to generate a public/private key pair is quite similar;

```
   KeyPairGenerator rsaKeyGen =
KeyPairGenerator.getInstance("RSA");
   rsaKeyGen.initialize(1024);

   KeyPair rsaKeyPair = rsaKeyGen.generateKeyPair();
   Cipher rsaCipher = Cipher.getInstance("RSA");
   rsaCipher.init(Cipher.ENCRYPT_MODE,
rsaKeyPair.getPrivate());
```

7.2 Key Conversion

Two interfaces are provided to convert between a provider-dependent `Key` and the provider-independent `KeySpec`; `java.security.KeyFactory` and `javax.crypto.SecretKeyFactory`. The `KeyFactory` class is used for public-key algorithms and the `SecretKeyFactory` class for secret-key algorithms.

An application may choose to store its keys in some way and then re-create the key using a `KeySpec`. For example, the application may contain an embedded RSA public key as two integers; the `RSAPublicKeySpec` along with a `KeyFactory` that can process `RSAPublicKeySpec` instances could then be used to create the provider-dependent key.

Each provider will generally supply a number of `KeyFactory`/`SecretKeyFactory` classes that will accept the various `KeySpec` classes and produce `Key` instances that may be used with the provider algorithms. These factories are not likely to support all `KeySpec` types, so the provider documentation should provide the details as to what conversions will be accepted.

There are a number of `KeySpec` classes provided by the JCA/JCE;

java.security.spec	
`PKCS8EncodedKeySpec`	A DER encoding of a private key according to the format specified in the PKCS#8 standard.
`X509EncodedKeySpec`	A DER encoding of a public or private key, according to the format specified in the X.509 standard.
`RSAPublicKeySpec`	An RSA public key
`RSAPrivateKeySpec`	An RSA private key
`RSAPrivateCrtKeySpec`	An RSA private key, with the Chinese Remainder Theorem (CRT) values
`DSAPublicKeySpec`	A DSA public key
`DSAPrivateKeySpec`	A DSA private key
javax.crypto.spec	
`DESKeySpec`	A DES secret key
`DESedeKeySpec`	A DESede secret key
`PBEKeySpec`	A user-chosen password that can be used with password base encryption (PBE)
`SecretKeySpec`	A key that can be represented as a byte array and have no associated parameters. The encoding type is known as RAW.

To convert a `KeySpec` instance into a provider based `Key`, firstly create a `KeyFactory` or `SecretKeyFactory` of the appropriate type using the `getInstance()` method. Once the instance has been created, use the `KeyFactory.generatePrivate()`, `KeyFactory.generatePublic()` or `SecretKeyFactory.generateSecret()` method.

In the following example we will create a `Key` from a `KeySpec` and then recover the `KeySpec` from the `Key`.

```
   /*
    * This is the raw key
    */
   byte[] keyBytes = { (byte)0x1, (byte)0x02, (byte)0x03,
      (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08
};

   /*
    * Create the provider independent KeySpec
    */
   DESKeySpec desKeySpec = new DESKeySpec(keyBytes);

   /*
    * Create the KeyFactory to do the Key<->KeySpec translation
    */
   SecretKeyFactory keyFact = KeyFactory.getInstance("DES");

   /*
    * Create the provider based SecretKey
    */
   SecretKey desKey = keyFact.generateSecret(desKeySpec);

   /*
    * Convert the provider Key into a generic KeySpec
    */
   DESKeySpec desKeySpec2 = keyFact.getKeySpec(desKey,
      DESKeySpec.class);
```

7.3 Key Agreement Protocols

Keys may also be generated using the `javax.crypto.KeyAgreement` API. This interface provides the functionality of a key agreement (or key exchange) protocol. For example, a Diffie-Hellman `KeyAgreement` instance would allow two or more parties to generate a shared Diffie-Hellman Key.

To generate the key, it is necessary to call `KeyAgreement.doPhase()` for each party in the exchange with a `Key` object that represents the current state of the key agreement. The last call to `KeyAgreement.doPhase()` should have the `lastPhase` set to `true`.

Once all the key agreement phases have been processed, the shared `SecretKey` may be generated by calling the `KeyAgreement.generateSecret()` method.

The `KeyAgreement` API does not define how each of the parties communicates the necessary information for each exchange in the protocol. The required information is passed to the `KeyAgreement.doPhase()` method as a `Key`. This `Key` will generally be generated using either a `KeyGenerator` or a `KeyFactory`. The provider documentation will detail the specific steps required for a given protocol.

```
/*
 * Create the KeyAgreement instance for the required
 * protocol and initialise it with our key.  In the
 * case of Diffie-Hellman this would be our private
 * key.
 */
KeyAgreement keyAg = KeyAgreement.getInstance("DH");
keyAg.init(ourKey);

/*
 * Exchange information as per the key exchange
 * protocol.  For DH we would exchange public keys.
 * Note since there is only two parties in this
 * case the return value is not relevant.
 */
keyAg.doPhase(remotePubKey, true);

/*
 * Create the shared secret-key
 */
SecretKey key = keyAg.generateSecret("DES");
```

7.4 Key Storage

Once a `Key` has been generated you may wish to store it for future use. Generally, you'll be saving public/private keys so that you can reuse them at a later date in a key exchange.

The `java.security.KeyStore` API provides one mechanism for management of a number of keys and certificates. There are two types of entries in a `KeyStore`; `Key` entries and `Certificate` entries. Key entries are sensitive information whereas certificates are not.

As `Key` entries are sensitive, they are therefore are protected by the `KeyStore`. The API allows for a password, or pass phrase, to be attached to each key entry. What the actual implementation does with the password is not defined, although it may be used to encipher the entry. A key entry may either be a `SecretKey`, or a `PrivateKey`. In the case of a `PrivateKey`, the entry is saved along with a `Certificate` chain which is the chain of trust. The chain of trust starts with the `Certificate` containing the corresponding `PublicKey` and ends with a self-signed certificate.

A certificate entry represents a "trusted certificate entry", that is a `Certificate` whose identity we trust. This type of entry can be used to authenticate other parties.

To create a `KeyStore` instance, use the `KeyStore.getInstance()` method. This will return an empty key store which may then be populated by calling the `KeyStore.load()` method. This method accepts an `InputStream` instance and an optional password. Each individual `KeyStore` will treat these parameters differently, so check the provider documentation for details.

The Sun provider supplies a `KeyStore` known as "JKS". This `KeyStore` is used by the `keytool` and `jarsigner` applications.

```
/*
 * Create an instance of the Java Key Store (defined by Sun)
 */
KeyStore keyStore = KeyStore.getInstance("JKS");
```

To add a new entry into the `KeyStore`, use either `setCertificateEntry()` or one of the `setKeyEntry()` methods. This will add the new entry with the associated alias.

```
char[] myPass;
  SecretKey secretKey;

  /*
   * Store a SecretKey in the KeyStore, with "mypass"
   * as the password.
   */
  keyStore.setKeyEntry("mysecretkey", secretKey,
                                          myPass, null);

  /*
   * assume that privateKey contains my PrivateKey
   * and myCert contains a Certificate with the
   * corresponding PublicKey
   */
  PrivateKey privateKey;
  Certificate myCert;

  keyStore.setKeyEntry("myprivatekey", privateKey,
                                        myPass, myCert);
```

To extract an entry, use the `getKey()` method to extract a `Key` or `getCertificate()` for a `Certificate`.

```
  /*
   * recover the SecretKey
   */
  SecretKey key = (SecretKey)keyStore.getKey("mysecretkey",
                                                    myPass);

  /*
   * recover the PrivateKey
   */
  PrivateKey privKey =
        (PrivateKey)keyStore.getKey("myprivatekey", myPass);

  /*
   * recover the Certificate (containing the PublicKey)
   * corresponding to our PrivateKey
   */
  Certificate cert = keyStore.getCertificate("myprivatekey");
```

If the `KeyStore` supports persistence via the `store()` and `load()` methods, the provider documentation will explain what types of `Key` types may be stored.

7.5 Certificates

The JCA framework provides support for generic certificates, as well as X.509v3 certificates. Certificates may be stored using the `KeyStore` API, or they may be generated from their encoded format (either the PEM or PKCS#7 encoding).

To create a `java.security.cert.Certificate` instance from its encoded format, firstly create a `java.security.cert.CertificateFactory` instance of the required type (eg X.509). Then use the `generateCertificate()` or `generateCertificates()` methods to convert your `InputStream` into `Certificate` instances.

```
CertificateFactory cf =
      CertificateFactory.getInstance("X.509");
X509Certificate cert =
     (X509Certificate)cf.generateCertificate(inputStream);
```

Two useful methods of the `Certificate` class are `getPublicKey()` and `verify()`. The first of these allows access to the `PublicKey` of the certificate's owner and the second allows an application to verify that the certificate was signed using the private key that corresponds to the provided public key.

The `java.security.cert.X509Certificate` class, which extends the `Certificate` class, provides methods to access the other attributes of a X.509 certificate such as the Issuer's distinguished name or its validity period.

The `keytool` application provided with JDK1.2 can be used to generate certificates and store them in a `KeyStore`. Check the JDK documentation for information on how to use this application.

8.0 ERROR HANDLING AND EXCEPTIONS

The JCA/JCE framework includes a number of specialised exception classes:

java.security	
`DigestException`	Thrown if an error occurs during the final computation of the digest. Generally this indicates that the output buffer is of insufficient size.
`InvalidAlgorithmParameterException`	Thrown by classes that use `AlgorithmParameters` or `AlgorithmParameterSpec` instances where the supplied instance is not compatible with the algorithm or the supplied parameter was `null` and the algorithm requires a non-null parameter.
`InvalidKeyException`	Thrown by the various classes that use `Key` objects, such as `Signature`, `Mac` and `Cipher` when the provided `Key` is not compatible with the given instance.
`InvalidParameterException`	Only used in the deprecated interfaces in the `Signature` class and the deprecated class `Signer`.
`KeyStoreException`	Thrown by the `KeyStore` class when the object has not been initialised properly.
`NoSuchAlgorithmException`	Thrown by the `getInstance()` methods when the requested algorithm is not available.
`NoSuchProviderException`	Thrown by the `getInstance()` methods when the requested provider is not available.
`SignatureException`	Thrown by the `Signature` class during signature generation or validation if the object has not been initialised correctly or an error occurs in the underlying ciphers.
javax.crypto	
`BadPaddingException`	Thrown by the `Cipher` class (or classes which use a `Cipher` class to process data) if this cipher is in decryption mode, (un)padding has been requested, and the deciphered data is not bounded by the appropriate padding bytes.
`IllegalBlockSizeException`	Thrown by the `Cipher` class (or classes which use a `Cipher` class to process data) if this cipher is a block cipher, no padding has been requested (only in encryption mode), and the total input length of the data processed by this cipher is not a multiple of block size
`NoSuchPaddingException`	Thrown by the `Cipher` class by the `getInstance()` method when a transformation is requested that contains a padding scheme that is not available.
`ShortBufferException`	Thrown by the `Cipher` class when an output buffer is supplied that is too small to hold the result.

THIS PAGE INTENTIONALLY LEFT BLANK

GLOSSARY

API

Application Programming Interface, an interface used by an application developer to interface to a set of functionality provided by a third party.

Asymmetric Cipher

An encryption algorithm that uses different keys for encryption and decryption. These ciphers are usually also known as public-key ciphers as one of the keys is generally public and the other is private. RSA and ElGamal are two asymmetric algorithms.

Block Cipher

A cipher that processes input in a fixed block size greater than 8 bits. A common block size is 64 bits.

Certificate

A binding of an identity (individual, group, etc.) to a public key which is generally signed by another identity. A certificate chain is a list of certificates that indicates a chain of trust, i.e. the second certificate has signed the first, the third has signed the second and so on.

Decryption

The process of recovering the plaintext from the ciphertext.

DES

Data Encryption Standard as defined in FIPS PUB 46-2 which may be found at http://www.itl.nist.gov/div897/pubs/fip46-2.htm.

Digital Signature

A mechanism that allows a recipient or third party to verify the originator of a document and to ensure that the document has not be altered in transit.

DSA

Digital Signature Algorithm as defined in FIPS PUB 186 which may be found at http://www.itl.nist.gov/div897/pubs/fip186.htm.

Encryption

The process of converting the plaintext data into the ciphertext so that the content of the data is no longer obvious. Some algorithms perform this function in such a way that there is no known mechanism, other than decryption with the appropriate key, to recover the plaintext. With other algorithms there are known flaws which reduce the difficulty in recovering the plaintext.

JCA

Java Cryptography Architecture.

JCE

Java Cryptography Extension.

MAC

Message authentication code. A mechanism that allows a recipient of a message to determine if a message has been tampered with. Broadly there are two types of MAC algorithms, one is based on symmetric encryption algorithms and the second is based on Message Digest algorithms. This second class of MAC algorithms are known as HMAC algorithms. A DES based MAC is defined in FIPS PUB 113, see http://www.itl.nist.gov/div897/pubs/fip113.htm. For information on HMAC algorithms see RFC-2104 at http://www.ietf.org/rfc/rfc2104.txt.

Message Digest

A condensed representation of a data stream. A message digest will convert an arbitrary data stream into a fixed size output. This output will always be the same for the same input stream however the input cannot be reconstructed from the digest.

Padding

A mechanism for extending the input data so that it is of the required size for a block cipher. The PKCS documents contain details on the most common padding mechanisms of PKCS#1 and PKCS#5.

PEM

Privacy Enchanced Mail, includes standards for certificates, see RFC1422 http://www.ietf.org/rfc/rfc1422.txt.

PKCS

Public Key Cryptography Standards. A set of standards (currently PKCS#1 to PKCS#15) developed by RSA Laboratories, see http://www.rsa.com/rsalabs/pubs/PKCS/.

RFC

Request for Comments, proposed specifications for various protocols and algorithms archived by the Internet Engineering Task Force (IETF), see http://www.ietf.org.

RSA

A public-key encryption algorithm, see http://www.rsa.com.

Symmetric Cipher

An encryption algorithm that uses the same key for encryption and decryption. DES, RC4 and IDEA are all symmetric algorithms.

X.509 Certificate

Section 3.3.3 of X.509v3 defines a certificate as: "user certificate; public key certificate; certificate: The public keys of a user, together with some other information, rendered unforgeable by encipherment with the private key of the certification authority which issued it".

END OF DOCUMENT