The Bouncy Castle FIPS Java API in 100 Examples (Final Draft)

David Hook

Copyright (c) 2016 David Hook

Published by CERTOSS, Inc,
10846 Via San Marino, Cupertino, CA 95014, United States of America

For permission to reproduce parts or all of this work, please contact CERTOSS, Inc.

Table of Contents

Introduction

About this Book

In order to keep this brief and to the point, this booklet is not about cryptography so much, as about the BC FIPS Java API and how it presents cryptography. To get the most out of this book you should have some understanding of the principals of cryptography. Having an existing understanding of the Java Cryptography Architecture, the Java Cryptography Extension, and the Java Secure Socket Extension would not hurt either, although you can probably pick a lot of that up by working through the examples.

While the booklet is also primarily written with the BCFIPS provider in mind, where possible the examples have been written for the standard Java APIs for cryptography, so most of the examples will also be usable with the regular BC provider. The examples are not meant to be definitive, but they should give you a good overview of what can be done with the BCFIPS provider and its associated APIs. So if you run into a situation where the example does not quite fit what you want to do, hopefully a look around the other classes referenced in the same packages used in the example will get you there. For brevity the examples do not include import statements, but you can find the full source for them as well as some small examples of use at https://www.bouncycastle.org/fips-java.

The examples do make use of some predefined sample values as well, these are defined in a class called ExValues. You can safely make them up, but to avoid them becoming a possible point of concern they are defined below:

```
public class ExValues
{
    public static final long THIRTY_DAYS = 1000L * 60 * 60 * 24 * 30;
    public static final SecretKey SampleAesKey =
                            new SecretKeySpec(Hex.decode("000102030405060708090a0b0c0d0e0f"), "AES");
    public static final SecretKey SampleTripleDesKey =
       new SecretKeySpec(Hex.decode("000102030405060708090a0b0c0d0e0f1011121314151617"), "TripleDES");
    public static final SecretKey SampleHMacKey =
              new SecretKeySpec(Hex.decode("000102030405060708090a0b0c0d0e0f10111213"), "HmacSHA512");
    public static final byte[] SampleInput = Strings.toByteArray("Hello World!");
    public static final byte[] SampleTwoBlockInput
                              = Strings.toByteArray("Some cipher modes require more than one block");
    public static final byte[] Nonce = Strings.toByteArray("number only used once");
    public static final byte[] PersonalizationString
                                        = Strings.toByteArray("a constant personal marker");
    public static final byte[] Initiator = Strings.toByteArray("Initiator");
    public static final byte[] Recipient = Strings.toByteArray("Recipient");
    public static final byte[] UKM = Strings.toByteArray("User keying material");
}
```

Why FIPS 140?

The Federal Information Processing Standards (FIPS) 140 standards were originally put together in 1994, with a further revision, the current one, FIPS 140-2, being released in 2001 (funnily enough on the day of Bouncy Castle's first birthday, May 25th). At this time, the FIPS 140-2 standards form the basis of the requirements for any application involved in the transmission of sensitive data in all US

Government Departments and agencies. The validation program is known as the CMVP (Cryptographic Module Validation Program) and it is managed by the National Institute of Standards and Technology (NIST). There are a lot more acronyms that could follow as well!

Leaving the acronyms aside, apart from opening a door for the development and sale of products to the US Government which require FIPS 140. FIPS 140 has also gone on to become the basis of other similar standards outside of the US, and can be used as a step in gaining a Common Criteria certification as well. In addition many industry groups inside and outside the US have modeled their security requirements on the FIPS standards and if you spend time reading through the FIPS standards you will understand why. The standards are very thorough and as you would hopefully expect from a government standards body, the FIPS standards are also widely discussed and studied, and in most cases also come with a testing procedure to make sure they have been correctly followed.

For us at the Legion of the Bouncy Castle, in trying to produce and maintain a sound cryptography API and in trying to find some independent way of validating the API, the FIPS 140-2 certification process was the most obvious choice. Mind you, when we started we did not appreciate it was going to take two years to get through!

So does the BC FIPS API mean I do not need to know what I am doing?

Sadly, any assurance about the quality of the API does not make it idiot proof. A FIPS 140-2 API really comes in two parts. The first part is the actual code, as in the jar file you use, and the second part is a document called the Security Policy. The Security Policy is important as it is the other thing that the CMVP sign off on. The Security Policy dictates how the security module should be used, what it supports, and what guidance is required for the safe use of the module in “approved mode”. The document can be a little awkward to read at first, partly because the language it uses comes from FIPS 140's origins as primarily a hardware oriented standard.

The Security Policy is worth persisting with though. When a thread in the BC FIPS API is running in “approved mode” only FIPS approved algorithms are accessible to it, while this provides some level of certainty to a developer, or a code reviewer, as to how cryptographic services are handled by an application, there are still a variety of ways to shoot yourself in the foot, and in some cases, such as with the digest algorithm “SHA-1”, or even with the use of the same RSA key for signing certification requests and then encryption, there are also only specific uses allowed – uses which are impossible to monitor from the APIs point of view, but mean something, both from the point of view of an auditor, and also from the point of view of how secure your usage of the API really is.

So, read the Security Policy, save your bullets for the problems facing your project, rather than wasting them on your foot. Sanity check what you are doing as well, none of us are perfect, that includes the authors of the BC FIPS APIs. It never hurts to improve one's knowledge, and the application of cryptography is definitely an area where ignorance will build the path to disaster.

And Finally...

Enjoy the examples, and if you find an error, please let me know! Thanks.

Getting Started

This chapter provides a quick look at set up and configuration of the Bouncy Castle FIPS Java provider.

Provider Installation

The Bouncy Castle FIPS Java provider can either be installed via the java.security JVM configuration file or during execution. If you install it via the java.security file you will need to add:

```
security.provider.X=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
```

where X is the priority number for the Bouncy Castle FIPS Java provider.

You can add the provider during execution by using the following imports:

```
import java.security.Security
import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
```

and then adding a line similar to:

```
Security.addProvider(new BouncyCastleFipsProvider())
```

Once the provider is added, it can be referenced in your code using the provider name “BCFIPS”.

Provider Configuration

If no SecureRandom has been specified using CryptoServicesRegistrar.setSecureRandom() the provider class will generate a FIPS compliant DRBG based on SHA-512. It is also possible to configure the DRBG by passing a string as a constructor argument to the provider via code, or the java.security configuration file.

At the moment the configuration string is limited to setting the DRBG. The configuration string must always start with "C:" and finish with "ENABLE{ALL};". The command for setting the actual DRBG type is DEFRND so a configuration string requesting the use of a SHA1 DRBG would look like:

C:DEFRND[SHA1];ENABLE{ALL};

Possible values for the DRBG type are "SHA1", "SHA224", "SHA256", "SHA384", "SHA512", "SHA512(224)", "SHA512(256)", "HMACSHA1", "HMACSHA224", "HMACSHA256", "HMACSHA384", "HMACSHA512", "HMACSHA512(224)", "HMACSHA512(256)", "CTRAES128", "CTRAES192", CTRAES256", and "CTRDESEDE".

The default DRBG will be setup with prediction resistance set to true. In situations where the amount of entropy is constrained the default DRBG can be configured to use an entropy pool based on a SHA-512 SP 800-90A DRBG. To configure this use:

C:HYBRID;ENABLE{ALL};

or include the string "HYBRID;" in the previous command string setting the DRBG. After initial seeding the entropy pool will start a reseeding thread which it will begin polling once 20 samples have been taken since the last seeding and will do a reseed as soon as new entropy bytes are returned.

The provider will also normally try to prevent RSA keys being used for both signing and encryption. This check can be turned off by running the JVM with the property

```
org.bouncycastle.rsa.allow_multi_use=true
```

There are other configuration properties available, as well as several permissions that can be set if the provider is being run under a security manager. Details on this can be found in the appendices of the User Guide.

Finally

The provider jar itself has no external dependencies, but it cannot be used in the same JVM as the regular Bouncy Castle provider. The classes in the two jar files do not get along.

There are also FIPS specific versions of the bcpkix, bcpg, and bcmail jars. Strictly speaking these are only required in a situation where the lack of the classes required to resolve the ".bc." operators in the jars may cause issues, such as in use with OSGI containers. We still recommend using the FIPS specific versions to avoid any accidental "class not found" issues being introduced during the application development process. As with the regular bcmail jar, the FIPS specific bcmail jar also requires the JavaMail API and its dependencies.

Random Numbers

In many ways the topic of random numbers is the most important thing in cryptography. Where we get random values from and what we do with them in key and IV generation is fundamental to the security of any application. You might be surprised to see IV generation mentioned, but even careless generation of IVs can cause trouble – with algorithms like GCM it is even regarded as fatal.

The primary NIST standard dealing with random numbers is NIST Special Publication 800-90A "Recommendation for Random Number Generation Using Deterministic Random Bit Generators"- now at revision 1. SP 800-90A describes three different deterministic random bit generators (DRBGs), as well as the security levels that you can expect with them. At the current time the DRBGs described in SP 800-90A can be treated as "equals" in the respect of how good they are where both implementations have the same security level. Which DRBG you choose is in many ways dependent on what the algorithm support for a particular platform is. This is worth keeping in mind, as while in the case of this book, we are discussing the full release of the BC FIPS APIs for Java, the BC APIs are designed to be easy to subset, so you may run into reduced versions of them in practice. In addition to the NIST DRBG the BC APIs also offer the DRBG from X9.31 (this is switched off in approved mode).

The three DRBGs types defined fall into the classes of hash (message digest) based, HMAC based, and cipher based. In all cases the primitives involved are used to define mixing functions designed to stretch out seed material and (hopefully) remove any biases that might have been in the seed material to start with.

The Bouncy Castle FIPS API provides implementations of all three. In this case we will just look at the use of the HMAC SHA512 based one. In terms of setup the others are similar, and as the HMAC SHA512 one is a DRBG with 256 bits of security associated with it, it is a good one to look at.

Creating DRBG Based SecureRandoms

NIST approaches the problem of creating a DRBG from the respect of providing an entropy source and then a mixing function to stretch it out. In the following example, we are creating a DRBG which might be suitable for use with the creation of initialisation vectors (IVs) or other similar random data, such as nonces.

Example 1 – Creating a FIPS Approved SecureRandom

```
public static SecureRandom buildDrbg()
{
    EntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    FipsDRBG.Builder drgbBldr = FipsDRBG.SHA512_HMAC.fromEntropySource(entSource)
                                    .setSecurityStrength(256)
                                    .setEntropyBitsRequired(256);
    return drgbBldr.build(ExValues.Nonce, false);
}
```

In this case the generateSeed() method of a regular SecureRandom is used as the entropy source. This

should make use of whatever entropy source the JVM running the code is configured for.

The next thing to note in the example is that the SecureRandom is not created via the Java provider mechanism. This is because the SecureRandom returned is an extension of the regular SecureRandom class called FipsSecureRandom. In this case an extension class was necessary as a NIST DRBG requires methods such as FipsSecureRandom.reseed() which are not available on SecureRandom (in this case even SecureRandom.setSeed() is not really a suitable candidate).

The second thing to note is the false parameter value on the FipsDRBG.Builder.build() method. This refers to the “prediction resistance” required of the constructed DRBG. In this case we are willing to assume that the DRBG function will do a good job producing a random stream and that's enough. In the case of keys or components of keys we need a higher standard to be reached so we set “prediction resistance” to true as in the following example.

Example 2 – Creating a FIPS Approved SecureRandom for Keys

```
public static SecureRandom buildDrbgForKeys()
{
    EntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    FipsDRBG.Builder drgbBldr = FipsDRBG.SHA512_HMAC.fromEntropySource(entSource)
                                .setSecurityStrength(256)
                                .setEntropyBitsRequired(256)
                                .setPersonalizationString(ExValues.PersonalizationString);
    return drgbBldr.build(ExValues.Nonce, true);
}
```

In this case there are two interesting differences between the construction of the SecureRandom in example 1 and the construction of the DRBG in example 2. The first is the passing of true as the parameter value for prediction resistance and the second is the use of the setPersonalizationString method. The personalization string is an example “hedging ones bet” so as to reduce the likelihood of two DRBGs somehow producing the same key stream where the entropy source turns out to be similar. As a value the personalization string can be secret, although needn't be. Primarily it needs to be unique. SP 800-90A discusses this further, and also the topic of other fields, one called additional input, which can also be used to enhance the basic entropy being provided to the DRBG on a reseed, and the nonce which helps further distinguish the DRBG.

Configuring a Default SecureRandom

In the case where the JCA/JCE provider is not used, it is difficult for the API to work out what to use for a source of randomness. Examples of where unexpected randomness requirements can come up include things like RSA blinding and, in the case of the BC FIPS API, rather than relying on “new SecureRandom()” the API executes a call to CryptoServicesRegistrar.getSecureRandom() which will throw an exception if a default has not been provided. A default SecureRandom can be set using CryptoServicesRegistrar.setSecureRandom as shown below.

Example 3 – Configuring the Default SecureRandom

```
public static void setDefaultDrbg()
{
```

```
    EntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    FipsDRBG.Builder drgbBldr = FipsDRBG.SHA512.fromEntropySource(entSource)
                                    .setSecurityStrength(256)
                                    .setEntropyBitsRequired(256);
    CryptoServicesRegistrar.setSecureRandom(drgbBldr.build(ExValues.Nonce, true));
}
```

Finally, where a default random is provided, it will also be used by the provider rather than the provider building its own. You can get access to the default JCE SecureRandom by using:

```
SecureRandom defaultRandom = SecureRandom.getInstance("DEFAULT", "BCFIPS");
```

Symmetric Key Encryption

The BC FIPS API offers both approved mode symmetric ciphers, AES and TripleDES, and also a number of other symmetric ciphers that appear in IETF and ISO standards such as ARC4, Blowfish, Camellia, CAST5, DES, GOST28147, IDEA, RC2, SEED, Serpent, SHACAL2, and Twofish.

The most basic place to start with symmetric key encryption is the generation of actual keys. Keys for symmetric ciphers are simply bit strings, often without structure (DES and TripleDES are an exception here). Broadly a key's strength is related to it bit length so it's common to see APIs referring to keys by bits involved rather than bytes, even in a language like Java where things are predominately byte aligned.

Key Generation

Generating a key in the JCE is quite simple, at its most basic you just provide a key size in bits and away you go, as follows:

Example 4 – Generating an AES Key

```
public static SecretKey generateKey()
    throws GeneralSecurityException
{
    KeyGenerator keyGenerator = KeyGenerator.getInstance("AES", "BCFIPS");

    keyGenerator.init(256);

    return keyGenerator.generateKey();
}
```

In the example whatever is the default SecureRandom for the provider will be used. The KeyGenerator.init() method can also be passed a SecureRandom where you wish to specify what source of randomness you want and the Java API also provides provision for the use of AlgorithmParameterSpec classes to be used to initialise key generators. In our case the bit length of the key size will do.

Key Construction

Sometimes it is necessary to simply construct a key from a byte array that has been provided. The simplest way to do this in the JCE is by using a SecretKeySpec.

Example 5 – Key Construction with a SecretKeySpec

```
public static SecretKey defineKey(byte[] keyBytes)
{
    if (keyBytes.length != 16 && keyBytes.length != 24 && keyBytes.length != 32)
    {
        throw new IllegalArgumentException("keyBytes wrong length for AES key");
    }
    return new SecretKeySpec(keyBytes, "AES");
}
```

The result of the defineKey() method can then be used anywhere a generated AES key would work.

Basic Modes and Padding

Having created a key, there are a variety of ways it can be used to encrypt things. All approaches are fragile in some way - it depends on what you are doing. Doing a bit of research that's relevant to your application is a good idea here as blanket statements such as “mode X is the only way to encrypt something” really tell you more about the person making the claim, than whether mode X is a good idea for you.

The most basic mode is ECB, Electronic Code Book mode. The example below shows methods using ECB mode for encryption and decryption. Note that the Cipher is created with a getInstance() method rather than a constructor. Stock standard ECB mode, like many block cipher modes is unpadded so the input has to be aligned on the block boundaries of the cipher - in this case 128 bits. In the example we have also made use of padding to get around this restriction, so rather than specifying “NoPadding” we have specified a specific format of padding “PKCS7Padding” to allow for non-block aligned data. PKCS7Padding is often also referred to as PKCS5Padding and the BC APIs provide a number of other padding mechanisms such as ISO10126-2, X9.23, ISO7816-4, and TBC (trailing bit compliment) padding.

Example 6 – ECB Mode Encryption

```
public static byte[] ecbEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return cipher.doFinal(data);
}

public static byte[] ecbDecrypt(SecretKey key, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key);
    return cipher.doFinal(cipherText);
}
```

The nice thing about ECB mode is it is obviously uncomplicated to program. Unfortunately if you are not trying to encrypt otherwise random data the lack of complication can be a defect. It's worth running this example with a few multi block strings, preferably some which repeat as well to give you an idea of what the issues are.

In this next example we are using CBC (Cipher Block Chaining) mode.

Example 7 – CBC Mode Encryption

```
public static byte[][] cbcEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return new byte[][] { cipher.getIV(), cipher.doFinal(data) };
```

```
}

public static byte[] cbcDecrypt(SecretKey key, byte[] iv, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
    return cipher.doFinal(cipherText);
}
```

Like ECB mode, CBC mode is block aligned so we need to specify padding. CBC mode also has an extra parameter, the initialisation vector (IV), which is used with the mode to prevent any obvious similarities that might have existed in two plain texts from showing up in the encrypted results. In the case of the example we are letting the Cipher object generate the IV for us and we are retrieving the generated IV using the getIV() method. You can also pass in the IV using the IvParameterSpec object for encryption as well if you wish to specify your own. Take care to make sure the IV is reliably random or unique if you do this, as otherwise you will end up back with ECB mode.

In the next example we are going to use a streaming block mode. The difference here is that padding no longer required as the cipher is actually used to generate a stream of “noise” to XOR with the data to be encrypted. As the XOR is done on a byte by byte basis there is no need for the data to be block aligned. The most basic block streaming mode is CFB mode.

Example 8 – CFB Mode Encryption

```
public static byte[][] cfbEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CFB/NoPadding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return new byte[][] { cipher.getIV(), cipher.doFinal(data) };
}

public static byte[] cfbDecrypt(SecretKey key, byte[] iv, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CFB/NoPadding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
    return cipher.doFinal(cipherText);
}
```

The set up for CFB mode is the same as with CBC mode – it also requires an IV which is used as the source of the stream generated by the cipher. The problems are similar too, if you use the same IV on two different encryptions, similarities might show up in the encrypted stream. You want to be careful with IVs.

Another mode which is also a block streaming mode, but offers more control, is CTR mode. In this case the IV is broken up into two parts, a random nonce, and a counter. It is also different from CFB mode in that the generated cipher stream is made by encrypting the nonce and counter. The use of the nonce and counter also mean that the cipher stream can be generated in a random access fashion.

The following example shows CTR.

Example 9 – CTR Mode Encryption

```
public static byte[][] ctrEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(Hex.decode("000102030405060708090a0b")));
    return new byte[][] { cipher.getIV(), cipher.doFinal(data) };
}

public static byte[] ctrDecrypt(SecretKey key, byte[] iv, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
    return cipher.doFinal(cipherText);
}
```

In this case the cipher has been initialized with a particular IV. The reason this is done is to indicate we have allocated 12 bytes to the nonce (which in real life should be random) and 4 bytes to the counter. A set up like this would allow us to encrypt a message of length $2^{\wedge 32}$ blocks.

The last example in this section shows how to use Ciphertext Stealing (CTS). NIST provides three definitions of cipher text stealing in an addendum to NIST SP 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”. CTS is used in conjunction with CBC mode and can be used where there are at least 2 blocks of data and has the advantage that it requires no padding, as the “stealing” process allows it to produce a cipher text which is the same length as the plain text. The most popular one is CS3, which is the same as the CTS mode described in RFC 2040, and is the one shown in example 10.

Example 10 – CBC Mode With Ciphertext Stealing

```
public static byte[][] ctsEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CBC/CS3Padding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    return new byte[][] { cipher.getIV(), cipher.doFinal(data) };
}

public static byte[] ctsDecrypt(SecretKey key, byte[] iv, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CBC/CS3Padding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
    return cipher.doFinal(cipherText);
}
```

Authenticated Modes

One of the issues with the basic modes is there is no mechanism in place to pick up actual errors or tampering attempts on decryption, other than perhaps getting back garbage. Authenticated modes such as GCM and CCM also incorporate a tag that provides a cryptographic checksum that can be used to help validate a decryption. These modes are also known as Authenticated Encryption with Associated Data (AEAD) modes as they also provide for mixing some additional clear text, or associated data, into the tag used for validation. The BC FIPS Java API also includes EAX, but the mode is not available in

the approved mode of operation.

The first of the modes we will look at is GCM, which is described in NIST SP 800-38D, “Galois/Counter Mode (GCM)”. This is based on CTR mode as well as having its own hashing function incorporated into it. In this case, as the mode also incorporates the tag, the set up for the mode is a bit more complicated and uses a GCMParameterSpec.

Example 11 – GCM Mode Encryption

```
public static Object[] gcmEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key,  // taglength, nonce
                        new GCMParameterSpec(128, Hex.decode("000102030405060708090a0b")));
    return new Object[] { cipher.getParameters(), cipher.doFinal(data), };
}

public static byte[] gcmDecrypt(SecretKey key, AlgorithmParameters gcmParameters, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, gcmParameters);
    return cipher.doFinal(cipherText);
}
```

In this case the example has specified a tag length of 128 bits (the maximum) and 12 byte nonce, giving a counter size of 4 bytes. We have also made use of the getParameters() method to retrieve the parameters used to initialize the cipher so we have something to give to the decrypt method. Note as the mode is based on a block streaming mode there's no padding. If you are planning to use GCM it is worth having a look at NIST SP 800-38D for guidance, poor choice of IVs can cause huge problems with this mode and it is not recommended to use a lower tag size unless you really know what you are doing.

The other AEAD mode available is CCM which is defined in NIST SP 800-38C. In terms of set up it is very similar to GCM and, as Java does not provide a parameter spec for it, the BC FIPS API allows the use of the GCMParameterSpec class with CCM. CCM, as defined by NIST, is also built on CTR mode, but in this case makes use of a CBC-MAC for the checksum.

Example 12 – CCM Mode Encryption

```
public static Object[] ccmEncrypt(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key,
                               new GCMParameterSpec(128, Hex.decode("000102030405060708090a0b")));
    return new Object[] { cipher.getParameters(), cipher.doFinal(data) };
}

public static byte[] ccmDecrypt(SecretKey key, AlgorithmParameters ccmParameters, byte[] cipherText)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, ccmParameters);
    return cipher.doFinal(cipherText);
}
```

The use of GCM and CCM is also discussed in RFC 5084. As mentioned at the start of the section, one of the things which distinguishes GCM and CCM from the other modes is the ability to also incorporate associated data into the checksum calculation. This can be used for a variety of things such as validating non-encrypted payload, and also, where otherwise kept secret between the two communicating parties, help test for the provenance of a message being decrypted.

Adding associated data into the calculation was not supported natively in the JCE until the arrival of Java 1.7 when the updateAAD methods were added to the Cipher class. The following example shows their use.

Example 13 – CCM With Associated Data Encryption

```
public static Object[] aeadEncrypt(SecretKey key, byte[] data, byte[] associatedData)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.ENCRYPT_MODE, key,
                      new GCMParameterSpec(128, Hex.decode("000102030405060708090a0b")));
    cipher.updateAAD(associatedData);
    return new Object[] { cipher.getParameters(), cipher.doFinal(data) };
}

public static byte[] aeadDecrypt(SecretKey key, AlgorithmParameters ccmParameters,
                                                        byte[] cipherText, byte[] associatedData)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AES/CCM/NoPadding", "BCFIPS");
    cipher.init(Cipher.DECRYPT_MODE, key, ccmParameters);
    cipher.updateAAD(associatedData);
    return cipher.doFinal(cipherText);
}
```

The BC FIPS API provides the AEADParameterSpec for dealing with associated data in Java 1.5 and Java 1.6 as well.

Message Digest, MACs, and HMACs

Prior to the introduction of authenticated modes with ciphers the only way to tell if decrypted data was correct was to have a separate message digest or MAC associated with it. Message digests also provide a valuable tool for constructing signatures. MACs and HMACs are still relevant as there are still plenty of cases where messages need to be tamper resistant, even when the content in them might be publicly readable.

Message Digests

Message digest, or hash, functions differ from a regular checksum, such as CRC32, in that changing any bit in an input stream to a digest calculator has an unpredictable affect on the resulting output. This is one of the things we look for in a cryptographic checksum as opposed to a regular one. The associated feature of this, which is that it is overwhelmingly difficult to predict the value of a digest and produce two documents which produce a “collision” in digest values, is fundamental to the idea that if two documents verify to the same digital signature, they are basically the same document.

As with symmetric ciphers, there have been a few goes at producing digests now, some such as SHA-1 and MD5 are no longer really in use and are getting phased out where they are (FIPS only allows SHA-1 for compliance with existing protocols). The current recommended digests come from the SHA-2 (described in FIPS PUB 180-4) and the SHA-3 (described in FIPS PUB 202) family.

The SHA-2 family digests are still regarded as safe, although there is now a trend towards the use of SHA-384 and SHA-512 and its two smaller variants instead of SHA-224 and SHA-256. This is happening largely due to the size of the internal buffer in SHA-224/SHA-256 that is used to store data for the digest calculation and concerns about whether that may be a problem if/when quantum computers start making themselves felt. The SHA-3 family is based on a different approach to the SHA-2 family to doing the digest construction and also includes two expandable functions (XOFs), SHAKE-128 and SHAKE-256.

While the SHA-3 family is different internally, the digests contained in it produce digests of the same size as the SHA-2 family and they can be used as drop in replacements for each other. The first example shows how to create a simple digest using the SHA-2 and SHA-3 variants that produce 512 bits of output from the input data.

Example 14 – Two Digest Examples

```
public static byte[] calculateDigest(byte[] data)
    throws GeneralSecurityException
{
    MessageDigest hash = MessageDigest.getInstance("SHA512", "BCFIPS");
    return hash.digest(data);
}

public static byte[] calculateSha3Digest(byte[] data)
    throws GeneralSecurityException
{
    MessageDigest hash = MessageDigest.getInstance("SHA3-512", "BCFIPS");
```

```
    return hash.digest(data);
}
```

Expandable Output Functions

Expandable output functions are completely new on the scene – the first ones standardized were announced in the SHA-3 standard. Primarily, we can probably expect them to replace things like KDFs and mask functions as they have the feature producing “almost indefinite length” output while still offering a specific level of security. Being so new on the scene there isn't really any precedent for these functions in the JCA/JCE and so there is no API support for them directly at the moment.

At the moment, if you wish to make use of SHAKE-128 and SHAKE-256, you need to use the BC low-level API that comes with the BC FIPS module. The first example here shows the use of SHAKE-256 in producing 32 bytes of output.

Example 15 – Basic Use of an XOF

```
public static byte[] calculateShakeOutput(byte[] data)
    throws IOException
{
    FipsXOFOperatorFactory<FipsSHS.Parameters> factory =
                                                    new FipsSHS.XOFOperatorFactory<FipsSHS.Parameters>();
    OutputXOFCalculator<FipsSHS.Parameters> calculator =
                                                     factory.createOutputXOFCalculator(FipsSHS.SHAKE256);

    OutputStream digestStream = calculator.getFunctionStream();
    digestStream.write(data);
    digestStream.close();

    return calculator.getFunctionOutput(32);
}
```

The BC API also allows for the continuous “squeezing” of the function to produce more output. Example 15 also produces 32 bytes of output but does it by requesting output from the XOF object twice.

Example 16 – Multiple Returns from an XOF

```
public static byte[] calculateShakeOutputContinuous(byte[] data)
    throws IOException
{
    FipsXOFOperatorFactory<FipsSHS.Parameters> factory =
                                                    new FipsSHS.XOFOperatorFactory<FipsSHS.Parameters>();
    OutputXOFCalculator<FipsSHS.Parameters> calculator =
                                                     factory.createOutputXOFCalculator(FipsSHS.SHAKE256);

    OutputStream digestStream = calculator.getFunctionStream();
    digestStream.write(data);
    digestStream.close();

    // note in this case we are calling getFunctionOutput twice.
    return Arrays.concatenate(calculator.getFunctionOutput(16), calculator.getFunctionOutput(16));
}
```

If you run the two examples for the same input and examine the return value, you will find they produce the same byte stream.

Message Digest Based MACs

Macs based on keyed digests can be used to authenticate data. The most popular method for doing this at the moment is the HMAC, defined in FIPS PUB 198-1 and RFC 2104.

HMAC support at the moment is not provided for the SHA-3 family. HMAC constructions have recently being added to the NIST standards, so watch out for these in a future release of BCFIPS. It is also likely that some new digest based MACs other than HMAC will be defined for the SHA-3 family as it is not vulnerable to the length extension issue of the digests that the original HMAC construction was designed to deal with.

As with a symmetric cipher, the first thing required to use a HMAC is key, this can be created using a SecretKeySpec as we saw with AES or by using a KeyGenerator specific to the algorithm as in the following example.

Example 17 – HMAC Key Generation

```
public static SecretKey generateKey()
    throws GeneralSecurityException
{
    KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacSHA512", "BCFIPS");
    keyGenerator.init(256);
    return keyGenerator.generateKey();
}
```

HMAC calculation is done using the Mac class in the JCE. As you might imagine the main difference between the Mac class and the MessageDigest class is the need to call Mac.init() to initialize the MAC with the key and any other parameters that are required.

In the case of a HMAC only a key is required as we can see in the following example.

Example 18 – HMAC Calculation

```
public static byte[] calculateHmac(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Mac hmac = Mac.getInstance("HMacSHA512", "BCFIPS");
    hmac.init(key);
    return hmac.doFinal(data);
}
```

Note: the doFinal() method in the example fulfills the role of the equivalent digest() method in the MessageDigest class. As the Mac class is often used with cipher based algorithms, the naming conventions in it follow the Cipher class.

Symmetric Cipher Based MACs

There are a number of different approaches to calculating MACs based around symmetric ciphers. The most common in the FIPS world now are CMAC and GMAC. CCM also gets used in a MAC mode as well occasionally – as with everything else what you end up using may depend on the restrictions of the environments you are dealing with.

The first one we will look at is CMAC, defined in NIST SP 800-38B. CMAC can be used with both 128 bit and 64 bit block ciphers, so it can be used with Triple-DES as well as AES. In terms of initialization its the same as with a HMAC – only a key is required.

Example 19 – MAC Calculation using CMAC

```
public static byte[] generateMacCMAC(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Mac mac = Mac.getInstance("AESCMAC", "BCFIPS");
    mac.init(key);
    return mac.doFinal(data);
}
```

GMAC is a little different. It is defined in SP 800-38D, the same document that defines GCM. In this case the MAC also requires an IV and it should be noted that as it is really a specialisation of GCM without encrypted data, all constraints on GCM, such as the need for uniqueness of IVs, also apply to GMAC.

Example 20 – MAC Calculation using GMAC

```
public static byte[] generateMacGMAC(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Mac mac = Mac.getInstance("AESGMAC", "BCFIPS");
    mac.init(key, new IvParameterSpec(Hex.decode("000102030405060708090a0b")));
    return mac.doFinal(data);
}
```

Just as GMAC is a specialisation of GCM, it is also possible to use CCM purely for MAC calculation.

Example 21 – MAC Calculation using CCM

```
public static byte[] generateMacCCM(SecretKey key, byte[] data)
    throws GeneralSecurityException
{
    Mac mac = Mac.getInstance("AESCCMMAC", "BCFIPS");
    mac.init(key, new IvParameterSpec(Hex.decode("000102030405060708090a0b")));
    return mac.doFinal(data);
}
```

Finally, it should also be noted that for both GMAC and CCMMAC, the GCMParameterSpec class can also be used to define the IV and to modify the size of the MAC tag being produced on a call to Mac.doFinal().

Signatures

It's interesting to note that even with a lot of the arguments that go on about encryption and what sort of access people should have to encryption technology, no-one has ever argued the case that people do not need to be able to produce digital signatures. You could almost say that digital signing keeps the world turning.

The main document in the FIPS world concerning digital signing appears under the FIPS PUB 186 banner and is now at FIPS PUB 186-4. In practice, due to the long life of some signatures you may also find yourself verifying signatures produced under FIPS PUB 186-2 and FIPS PUB 186-3.

FIPS PUB 186-4 discusses approaches to digital signing based around three algorithms: DSA, RSA, and the Elliptic Curve DSA equivalent, ECDSA. There are also some variations on how signatures can be done in RSA. The BC FIPS APIs offer support for all the algorithms detailed. We will look at DSA first.

The DSA Algorithm

Signature algorithms require key pairs. In the case of DSA, the two sizes that can be used for the generation of signatures is 2048 and 3072. The following example will generate a DSA key pair with a key size of 3072 bits.

Example 22 – Key Pair Generation

```
public static KeyPair generateKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DSA", "BCFIPS");
    keyPair.initialize(3072);
    return keyPair.generateKeyPair();
}
```

Note: the example will take a while to run. The reason for this is that in addition to generating the key pair the key pair generator will also generate DSA parameters suitable for 3072 bit keys. In the case of 2048 bit keys there are actually default parameters, but in the case of 3072 with specific parameters, or where you might need to specify parameters in general, example 25 is the one to look at for generating key pairs.

For signing and verifying we use the Signature class. Like most of the provider based classes Signature objects are instanced using a getInstance() method rather than constructor. For signatures the common format in the JCA is to define the algorithm as <digest>with<public key algorithm>, in this case DSA. In the example we are generating and verifying signatures which are based on the SHA-384 digest and the DSA public key algorithm.

Example 23 – Signing and Verifying

```
public static byte[] generateSignature(PrivateKey dsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withDSA", "BCFIPS");
    signature.initSign(dsaPrivate);
    signature.update(input);
    return signature.sign();
}

public static boolean verifySignature(PublicKey dsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withDSA", "BCFIPS");
    signature.initVerify(dsaPublic);
    signature.update(input);
    return signature.verify(encSignature);
}
```

We can also change the digest to any other SHA-2 family digest so for example, SHA-512 would be “SHA512withDSA”, likewise SHA-256 would be “SHA256withDSA”, and so on. Further details on the algorithms supported with DSA can be found in the BC FIPS Java User Guide.

The numbers representing the public and private components of a DSA key are only meaningful for a given set of domain parameters. NIST define algorithms for generating and validating domain parameters. The BC FIPS API supports both situations, but only the parameter generation is exposed at the JCA level as there isn't really full API support for validation in the JCA. As we saw in example 22 generating parameters every time we create a key pair is expensive, so if you're not given parameters it is worth generating the parameters separately and then explicitly passing them in for key pair generation.

The following example shows how to generate a set of DSA parameters in the BC FIPS APIs.

Example 24 – Parameter Generation

```
public static DSAParameterSpec generateParameters()
    throws GeneralSecurityException
{
    AlgorithmParameterGenerator algGen = AlgorithmParameterGenerator.getInstance("DSA", "BCFIPS");
    algGen.init(new DSADomainParametersGenerationParameterSpec(3072, 256, 112));
    AlgorithmParameters dsaParams = algGen.generateParameters();
    return dsaParams.getParameterSpec(DSAParameterSpec.class);
}
```

Note the DSADomainParametersGenerationSpec is a BC specific class as there is not current a ready equivalent in the JCA for doing this.

Once we have generated our parameters it is easy to start to generate key pairs from them.

Example 25 – Generating Key Pairs using Parameters

```
public static KeyPair generateKeyPairUsingParameters(DSAParameterSpec dsaParameterSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DSA", "BCFIPS");
    keyPair.initialize(dsaParameterSpec);
```

```
    return keyPair.generateKeyPair();
}
```

Try running example 25 and comparing it to the speed of example 22. There is quite a difference in the time taken.

The RSA Algorithm

The RSA algorithm takes a different approach to DSA when generating signatures. The main reason for this is that RSA can also be used to encrypt a block of data – so a digital signature generated with RSA is really an encryption of a data block with private key that anyone with the public key can then decrypt and verify. Keep this in mind as it will help explain why, other than for the purpose of generating a certification request, an RSA key used for encryption should never be used for signing and visa-versa.

Generation of RSA signatures also requires key pairs of the sizes 2048 and 3072 bits. Another difference with RSA is the ability to specify a public exponent for the public key component. A good choice of public exponent can help as we tend to verify signatures more often than we create them, so by choosing an appropriate public exponent we can make the verification process reasonably efficient. A few standard public exponents are provided by the RSAKeyGenParameterSpec class which is part of the JCA. In the example below we've used the smallest acceptable value for FIPS purposes. The number is labeled as F4, as it also happens to be the 4th Fermat prime, and has a value of 0x10001.

Example 26 – Key Pair Generation

```
public static KeyPair generateKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("RSA", "BCFIPS");
    keyPair.initialize(new RSAKeyGenParameterSpec(3072, RSAKeyGenParameterSpec.F4));
    return keyPair.generateKeyPair();
}
```

You will probably also note that RSA key pair generation is also fairly slow. It will also grind through a lot of entropy as a large number of bits will get consumed trying to generate random primes.

Having generated a key pair we can now look at generating signatures. FIPS PUB 186-4 draws on 2 different other standards for generating RSA signatures: X9.31 and PKCS#1 v2.1 (the PKCS#1.5 format and the PSS format).

The first of the algorithms which we will look is the original PKCS#1.5 format.

Example 27 – The PKCS#1.5 Signature Format

```
public static byte[] generatePkcs1Signature(PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSA", "BCFIPS");
    signature.initSign(rsaPrivate);
    signature.update(input);
    return signature.sign();
}
```

```
public static boolean verifyPkcs1Signature(PublicKey rsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSA", "BCFIPS");
    signature.initVerify(rsaPublic);
    signature.update(input);
    return signature.verify(encSignature);
}
```

As you can see it looks just like DSA. Likewise you can see that the same naming convention for the algorithm applies, in this case <digest>withRSA, and all the SHA-2 variants are supported for it.

The X9.31 signature format has been with us for at least as long as the PKCS#1.5 format and dates back to the late 90s. We had a little bit of trouble working out how to name this as the JCA documentation on standard naming mainly concerns itself with the PKCS algorithms when it is discussing RSA. That said, we settled on “RSA/X9.31” as the naming since the use of a “/” to establish that something is a subset under a particular algorithm seems to follow the broader naming conventions used in Java, and that is used in the following example.

Example 28 – The X9.31 Signature Format

```
public static byte[] generateX931Signature(PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSA/X9.31", "BCFIPS");
    signature.initSign(rsaPrivate);
    signature.update(input);
    return signature.sign();
}

public static boolean verifyX931Signature(PublicKey rsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSA/X9.31", "BCFIPS");
    signature.initVerify(rsaPublic);
    signature.update(input);
    return signature.verify(encSignature);
}
```

Once again the full range of SHA-2 digests, and others (if not in FIPS approved mode) is supported by the BC FIPS API for X9.31.

In 2002 RSA announced a new signature format based on the Probabilistic Signature Scheme (PSS). It should be used in preference to the PKCS#1.5 format as PSS is regarded as provably secure, in the sense that trying to forge a signature in PSS can be shown to reduce to the problem of breaking RSA. While there have not been any successful attacks on properly formatted PKCS#1.5 signatures done carefully, the assurances offered by PSS are very nice to have.

The following example shows the use of RSA PSS.

Example 29 – The PSS Signature Format

```
public static byte[] generatePssSignature(PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSAandMGF1", "BCFIPS");
    signature.initSign(rsaPrivate);
```

```
    signature.update(input);
    return signature.sign();
}

public static boolean verifyPssSignature(PublicKey rsaPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withRSAandMGF1", "BCFIPS");
    signature.initVerify(rsaPublic);
    signature.update(input);
    return signature.verify(encSignature);
}
```

Note the change in the naming convention. It is still <digest>withRSAandMGF1 (so the full range of SHA-2 digests can be used), but the curious might want to know what is going on with the MGF1. The “andMGF1” is added as PSS uses a mask function as part of its signature construction and allows for different mask functions to be used. At the moment the mask function is an algorithm called MGF1, but sometime in the future, for example, this might become SHAKE256, as in “andSHAKE256”. We shall see.

The PSS scheme also allows for the use of different digests with MGF1 and for the use of zero length seeds, or fixed ones. This can be useful where you always want a signature for the same document to appear to be the same signature, or where you always want a signature generated for the same document by the same organisation to appear to be the same signature. The JCA allows for different length salts and other digests by using the PSSParameterSpec and the MGF1ParameterSpec as we can see in the following example.

Example 30 – PSS Signatures with Parameters

```
public static byte[][] generatePssSignatureWithParameters(PrivateKey rsaPrivate, byte[] input)
    throws GeneralSecurityException, IOException
{
    Signature signature = Signature.getInstance("SHA384withRSAandMGF1", "BCFIPS");

    signature.setParameter(new PSSParameterSpec("SHA-384", "MGF1",
                  new MGF1ParameterSpec("SHA-384"), 0, PSSParameterSpec.DEFAULT.getTrailerField()));

    signature.initSign(rsaPrivate);
    signature.update(input);

    AlgorithmParameters pssParameters = signature.getParameters();

    return new byte[][] { signature.sign(), pssParameters.getEncoded() };
}

public static boolean verifyPssSignatureWithParameters(PublicKey rsaPublic, byte[] input,
                                                       byte[] encSignature, byte[] encParameters)
    throws GeneralSecurityException, IOException
{
    AlgorithmParameters pssParameters = AlgorithmParameters.getInstance("PSS", "BCFIPS");
    pssParameters.init(encParameters);

    PSSParameterSpec pssParameterSpec = pssParameters.getParameterSpec(PSSParameterSpec.class);

    Signature signature = Signature.getInstance("SHA384withRSAandMGF1", "BCFIPS");

    signature.setParameter(pssParameterSpec);
    signature.initVerify(rsaPublic);
    signature.update(input);

    return signature.verify(encSignature);
```

```
}
```

Note, in this case, the digest used with the signature, the PSS generation, and the mask function are all the same. This is an established convention that should be followed as it means that all components of the system will be operating at an equivalent level of security. While there is not a well known attack for this case either, you will probably feel very depressed if it turns out that setting the MGF1ParameterSpec to SHA-1 actually opened the possibility of an attack on your SHA-384 PSS signatures - better to avoid that.

Using Elliptic Curve – ECDSA

The final signature type looked at in FIPS PUB 186-4 is ECDSA, which is a DSA style signature based on Elliptic Curves.

In a similar way to DSA there are two ways of generating EC keys for use with ECDSA. In one case you can specify the key size required and a default curve will be used if available, otherwise you can specify the actual curve you want to use and a key pair suitable for that curve will be generated.

In the first example we have just specified the key size required.

Example 31 – Key Pair Generation

```
public static KeyPair generateKeyPair()
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC", "BCFIPS");
    keyPair.initialize(384);
    return keyPair.generateKeyPair();
}
```

The example will result in a key pair on the curve P-384. Other default curves available include P-224, P-256, and P-521.

When specifying an actual curve to key pair generation we use the JCA class ECGenParameterSpec, which just takes the name of the curve to be used as a parameter.

Example 32 – Key Pair for a Named Curve

```
public static KeyPair generateKeyPairUsingCurveName(String curveName)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("EC", "BCFIPS");
    keyPair.initialize(new ECGenParameterSpec(curveName));
    return keyPair.generateKeyPair();
}
```

You can find a full list of the available curve names in the BC FIPS Java API User Guide. As an example if you wished to generate a key pair for the same curve that example 31 would use you would specify curveName to have the value “P-384”.

Once you have generated key pair, signing and verifying follow exactly the same procedure as they do with DSA.

Example 33 – ECDSA Signing and Verifying

```
public static byte[] generateSignature(PrivateKey ecPrivate, byte[] input)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withECDSA", "BCFIPS");
    signature.initSign(ecPrivate);
    signature.update(input);
    return signature.sign();
}

public static boolean verifySignature(PublicKey ecPublic, byte[] input, byte[] encSignature)
    throws GeneralSecurityException
{
    Signature signature = Signature.getInstance("SHA384withECDSA", "BCFIPS");
    signature.initVerify(ecPublic);
    signature.update(input);
    return signature.verify(encSignature);
}
```

Finally

It is worth having a look at the BC FIPS Java API User Guide about signatures as well. In addition to other varieties not mentioned here, there are even constructions for DSA and ECDSA which produce deterministic signatures, which while not FIPS approved at the moment, are starting to appear in some standards outside of the original one - RFC 6979. These can be useful to know about.

Key Wrapping

Key wrapping is what we are doing when we encrypt one key using another. As this is a common occurrence it is not surprising there are some standard ways of doing it. NIST provide standards for use with symmetric key algorithms such as AES and the RSA algorithm – although in the case of RSA, the NIST terminology used describes the wrapping function in connection with key transport.

The main thing to keep in mind with key wrapping is that it is madness to wrap a key which is expected to have a particular security strength with one that does not at least have the equivalent security strength (a higher one is clearly better, but sometimes we have to do what we can).

Using Symmetric Keys for Wrapping

The simplest techniques for key wrapping are based around the use of symmetric keys. They are documented in NIST SP 800-38F. The approach is superior to simply encrypting a key as it also includes some checksum information, the idea being there is a reasonable chance that an error will be detected if an attempt is made to unwrap a previously wrapped key with the wrong key, or an attempt is made to feed garbage into the key unwrapping algorithm instead.

Like regular encryption the key wrapping algorithms have to contend with the fact that block ciphers work with block aligned data. For that reason SP 800-38F offers two alternatives, one without padding which requires block aligned keys (where a block is half the block size of the cipher), and one with padding for non-block aligned keys.

This example shows the use of AES key wrapping, as defined in SP 800-38F.

Example 34 – Wrapping without Padding

```
public static byte[] wrapKey(SecretKey key, SecretKey keyToWrap)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AESKW", "BCFIPS");
    cipher.init(Cipher.WRAP_MODE, key);
    return cipher.wrap(keyToWrap);
}

public static Key unwrapKey(SecretKey key, byte[] wrappedKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AESKW", "BCFIPS");
    cipher.init(Cipher.UNWRAP_MODE, key);
    return cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

This example shows the use of AES key wrapping with padding, also as defined in SP 800-38F.

Example 35 – Wrapping with Padding

```
public static byte[] wrapKeyWithPadding(SecretKey key, SecretKey keyToWrap)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AESKWP", "BCFIPS");
    cipher.init(Cipher.WRAP_MODE, key);
```

```
    return cipher.wrap(keyToWrap);
}

public static Key unwrapKeyWithPadding(SecretKey key, byte[] wrappedKey)
    throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance("AESKWP", "BCFIPS");
    cipher.init(Cipher.UNWRAP_MODE, key);
    return cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

Note that in both cases we are using Cipher.WRAP_MODE and Cipher.UNWRAP_MODE rather than Cipher.ENCRYPT and Cipher.DECRYPT. This allows us to pass an actual key on wrapping and to return a key from the cipher doing the unwrapping, rather than just a block of bytes which we would then need to convert into a key.

Using RSA OAEP for Wrapping

The RSA Optimal Asymmetric Encryption Padding (OAEP) algorithm was originally introduced in PKCS#1 Version 2 at the same time as PSS and is described in NIST SP 800-56B, “Recommendations for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography”. I think the reason for this is that the assumption is that this technique would only be used to send a symmetric key to another party. In our case we are referring to it as key wrapping as that is how we talk about it in the JCE. The technique is not as general as AESKWP either, a key being transported, or wrapped, must fit in a single RSA block, along with any padding involved.

Basic OAEP wrapping allows you to wrap a symmetric key using an RSA public key, and then the other party can use the corresponding private key to recover it. In the same way as we did with AESKW we use WRAP_MODE and UNWRAP_MODE on the cipher class as shown in the following example.

Example 36 – OAEP Wrapping

```
public static byte[] oaepKeyWrap(PublicKey rsaPublic, SecretKey secretKey)
    throws GeneralSecurityException
{
    Cipher c = Cipher.getInstance("RSA/NONE/OAEPPadding", "BCFIPS");
    c.init(Cipher.WRAP_MODE, rsaPublic);
    return c.wrap(secretKey);
}

public static Key oaepKeyUnwrap(PrivateKey rsaPrivate, byte[] wrappedKey)
    throws GeneralSecurityException
{
    Cipher c = Cipher.getInstance("RSA/NONE/OAEPPadding", "BCFIPS");
    c.init(Cipher.UNWRAP_MODE, rsaPrivate);
    return c.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

Unlike AESKWP, but like PSS, OAEP can also be used in conjunction with parameters that allow you to use a higher level cipher than the default (currently SHA-1).

Example 37 – OAEP Wrapping with Parameters

```
public static byte[][] oaepKeyWrapWithParameters(PublicKey rsaPublic, SecretKey secretKey)
    throws GeneralSecurityException, IOException
{
    Cipher c = Cipher.getInstance("RSA/NONE/OAEPPadding", "BCFIPS");
    c.init(Cipher.WRAP_MODE, rsaPublic,
                new OAEPParameterSpec("SHA-384", "MGF1", new MGF1ParameterSpec("SHA-384"),
                                                                PSource.PSpecified.DEFAULT));

    return new byte[][] { c.wrap(secretKey), c.getParameters().getEncoded() };
}

public static Key oaepKeyUnwrapWithParameters(PrivateKey rsaPrivate, byte[] wrappedKey,
                                                                    byte[] encParameters)
    throws GeneralSecurityException, IOException
{
    Cipher c = Cipher.getInstance("RSA/NONE/OAEPPadding", "BCFIPS");

    AlgorithmParameters algorithmParameters = AlgorithmParameters.getInstance("OAEP", "BCFIPS");
    algorithmParameters.init(encParameters);

    c.init(Cipher.UNWRAP_MODE, rsaPrivate, algorithmParameters);

    return c.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

As we can see there are small differences, there's no salt value, and instead we have PSource, which provides a way of passing some additional parameters which can be used to provide a value for the label L used in the construction of the OAEP padding – see RFC 3447 for details.

Using RSA KEM for Wrapping

The second RSA based technique given in SP 800-56B is the one based on Key Encapsulation Mechanism (KEM). The difference between the RSA KEM and the RSA OAEP is that is the case of RSA KEM it is key material which is transmitted to the recipient by the initiator, not an actual key, or keys. When RSA KEM is used, both sides use the generated key material to them make the keys that they use. As the RSA key involved is encrypting a block of random data RSA KEM actually has a tighter security proof than RSA OAEP.

RSA KEM key transport also uses the cipher API, but in this case a wrapping key is generated from the key material and used to wrap the actual key that is sent. The example below shows the use of AES-256 KW with RSA KEM.

Example 38 – RSA KEM Based Key Wrapping

```
public static byte[] kemKeyWrap(PublicKey rsaPublic, SecretKey secretKey)
    throws GeneralSecurityException
{
    Cipher c = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");

    c.init(Cipher.WRAP_MODE, rsaPublic,
                            new KTSParameterSpec.Builder(
                                   NISTObjectIdentifiers.id_aes256_wrap.getId(), 256).build());

    return c.wrap(secretKey);
}
```

```
public static Key kemKeyUnwrap(PrivateKey rsaPrivate, byte[] wrappedKey)
    throws GeneralSecurityException
{
    Cipher c = Cipher.getInstance("RSA-KTS-KEM-KWS", "BCFIPS");

    c.init(Cipher.UNWRAP_MODE, rsaPrivate,
                               new KTSParameterSpec.Builder(
                                     NISTObjectIdentifiers.id_aes256_wrap.getId(), 256).build());

    return c.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
}
```

In this case, the secret key passed in will be wrapped using an AES key generated as a result of the RSA KEM process. This is also the technique documented in RFC 5990 and is available in the BC CMS API found in the bcpkix jar.

Key Establishment and Agreement

Key Establishment Using RSA

There are two additional options for RSA OAEP and RSA KEM detailed SP 800-56B which do not actually mesh very well with the Cipher API in Java. These are both related to the version of the algorithms using key confirmation. Both these approaches require the use of BC FIPS specific classes as the overlap between the approach and the JCE API is a little bit tenuous.

We will look at the RSA OAEP. In this case, from the BC FIPS Java world view, RSA OAEP crosses the line between key wrapping and key establishment when key confirmation is added. Key confirmation also involves transferring a MAC key with the key being transported. In the case of key confirmation the MAC key is then used by the recipient to send a MAC back to the send of the wrapped key to allow the sender of the key to confirm that the recipient (and possibly only the recipient) correctly received the key.

The following example shows OAEP with key confirmation. Note in this case the API used is based around the SecretKeyFactory as we generate both the key to be established and the MAC key as part of the OAEP key transport operation. Also note the call to the macKey.zeroize() method – this is the only place in the BC FIPS Java API where a key object is mutable. This is required by SP 800-56B as ideally the MAC key is forcibly erased as soon as it has been recovered and used.

Example 39 – OAEP Key Establishment with Key Confirmation

```
public static byte[][] initiatorOaepKeyEstablishWithKeyConfirmation(PublicKey rsaPublic)
    throws GeneralSecurityException
{
    SecretKeyFactory kemFact = SecretKeyFactory.getInstance("RSA-KTS-OAEP", "BCFIPS");
    KTSGenerateKeySpec kemParams = new KTSGenerateKeySpec.Builder(
                                            rsaPublic, "AES", 256).withMac("HmacSHA384", 384).build();

    KTSKeyWithEncapsulation encapsKey = (KTSKeyWithEncapsulation)kemFact.generateSecret(kemParams);

    ZeroizableSecretKey macKey = encapsKey.getMacKey();

    Mac mac = Mac.getInstance(macKey.getAlgorithm(), "BCFIPS");
    mac.init(macKey);

    DERMacData macData = new DERMacData.Builder(DERMacData.Type.UNILATERALU,
        ExValues.Initiator, ExValues.Recipient, null, encapsKey.getEncapsulation()).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    macKey.zeroize(); // FIPS requirement

    return new byte[][] { encapsKey.getEncoded(), encapsKey.getEncapsulation(), encMac };
}

public static byte[][] recipientOaepKeyEstablishWithKeyConfirmation(PrivateKey rsaPrivate,
                                                                    byte[] encapsulation)
    throws GeneralSecurityException
{
    SecretKeyFactory kemFact = SecretKeyFactory.getInstance("RSA-KTS-OAEP", "BCFIPS");
    KTSExtractKeySpec kemParams = new KTSExtractKeySpec.Builder(
                                                rsaPrivate, encapsulation,
                                                "AES", 256).withMac("HmacSHA384", 384).build();
```

```
    KTSKeyWithEncapsulation encapsKey = (KTSKeyWithEncapsulation)kemFact.generateSecret(kemParams);

    ZeroizableSecretKey macKey = encapsKey.getMacKey();

    Mac mac = Mac.getInstance(macKey.getAlgorithm(), "BCFIPS");
    mac.init(macKey);

    DERMacData macData = new DERMacData.Builder(DERMacData.Type.UNILATERALU,
        ExValues.Initiator, ExValues.Recipient, null, encapsKey.getEncapsulation()).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    macKey.zeroize(); // FIPS requirement

    return new byte[][] { encapsKey.getEncoded(), encapsKey.getEncapsulation(), encMac };
}
```

You can see that the process divides into 3 steps. On the initiator side there is the use of the KTSGenerateKeySpec and the SecretKeyfactory to produce a secret key and a MAC key as well as the data to send to the receiver. On the recipient side the encapsulation and the recipient private key is used to create a KTSExtractKeySpec and the SecretKeyFactory is then used to recover the secret key and the MAC key. As this is for an example, both the initiator and the recipient then go ahead and calculate MAC which are returned by the function.

If you try this example and check the return values you should find the MAC calculations produced the same tag. In real use the recipient would send the MAC back to the initiator in order to show that the key has now being successfully established. There are additional guidelines, which are worth reading, for constructing the MAC data that are given in SP 800-56B. The BC DERMacData class is provided to make the construction of the MAC data easier to perform.

Diffie-Hellman Key Agreement

The second approach available to us for establishing keys between multiple parties is Diffie-Hellman (or more completely Diffie-Hellman-Merkle) key agreement (DH). While we will not look at it here, one interesting aspect of the traditional DH scheme is it can be used between more than two parties to allow all participants to arrive at a shared secret. Details about the Diffie-Hellman scheme implemented in the BC FIPS Java API are given in NIST SP 800-56A, now at revision 2.

In our case we will just concentrate on two parties, but as with DSA, the DH algorithm requires domain parameters and also key pairs for all participants.

Domain parameters for DH are just as expensive to generate as DSA ones and as it is a multi-party algorithm it is very important that everyone agrees on the same domain parameter set first. Domain parameters can be generated in a pure JCE fashion as follows.

Example 40 – DH Domain Parameter Generation

```
public static DHParameterSpec generateParameters()
    throws GeneralSecurityException
{
    AlgorithmParameterGenerator algGen = AlgorithmParameterGenerator.getInstance("DH", "BCFIPS");
    algGen.init(3072);
```

```
    AlgorithmParameters dsaParams = algGen.generateParameters();

    return dsaParams.getParameterSpec(DHParameterSpec.class);
}
```

There are also techniques for validating DH parameters – if you need access to the validation parameters you need to make use of the BC specific class DHDomainParameterSpec instead of the DHParameterSpec.

Having agreed on a set of parameters the next step is to generate a key pair using them. This is also very similar to what is done for DSA.

Example 41 – DH Key Pair Generation

```
public static KeyPair generateKeyPair(DHParameterSpec dhParameterSpec)
    throws GeneralSecurityException
{
    KeyPairGenerator keyPair = KeyPairGenerator.getInstance("DH", "BCFIPS");
    keyPair.initialize(dhParameterSpec);
    return keyPair.generateKeyPair();
}
```

Now that we have our key pair, and hopefully others have generated theirs, we can now try to generate a shared secret key.

Example 42 – Basic DH Key Agreement

```
public static byte[] initiatorAgreementBasic(PrivateKey initiatorPrivate, PublicKey recipientPublic)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DH", "BCFIPS");

    agreement.init(initiatorPrivate);
    agreement.doPhase(recipientPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}

public static byte[] recipientAgreementBasic(PrivateKey recipientPrivate, PublicKey initiatorPublic)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DH", "BCFIPS");

    agreement.init(recipientPrivate);
    agreement.doPhase(initiatorPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}
```

Note that the private key involved is passed to the init() method and the public key of the other party is passed to the doPhase() method with the second parameter to doPhase(), the value true, indicating the end of the protocol (as in that all public keys are now provided).

While this works and produces a 256 bit AES key, it really only works “properly” if at least one of the keys is an ephemeral key. The symmetric key is derived directly from the public private key calculation

between the two parties so the derivation will always produce the same value if the same key pairs are involved. Another issue with this approach is the key size is restricted to the size of the shared secret calculated in the protocol. A fact which can also limit the usefulness of the result.

The answer in this case is to make use of what is called user keying material and a KDF. SP 800-56B specifies a KDF construction called the Concatenation KDF, or in our case "CKDF" for short. The user keying material and the use of the KDF both help mask the secret, randomise the result and allow for as much data as is required to be generated (for sensible values of required).

Example 43 – DH Key Agreement with a KDF

```
public static byte[] initiatorAgreementWithKdf(
                    PrivateKey initiatorPrivate, PublicKey recipientPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DHwithSHA384CKDF", "BCFIPS");

    agreement.init(initiatorPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(recipientPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}

public static byte[] recipientAgreementWithKdf(
                    PrivateKey recipientPrivate, PublicKey initiatorPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DHwithSHA384CKDF", "BCFIPS");

    agreement.init(recipientPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(initiatorPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}
```

Note in practice you would want at least some of the user keying material to be random, or at least unique to the exchange, otherwise you will wind up back where you started.

Another common KDF is the one from X9.63. In the case of the BC FIPS Java API this is simply referred to as KDF, so rather than saying "DHwithSHA384CKDF" you would drop the C and say "DHwithSHA384KDF" if you wanted to use the KDF from X9.63.

Diffie-Hellman, like the other key establishment schemes, can also be used with key confirmation, but in this case the variant with key confirmation actually fits reasonably well with the KeyAgreement API provided with the JCE (the odd use of a type cast aside) although the way it is done is again very BC centric. For a hint have a look at generateSecret().

Example 44 – DH Key Agreement with Key Confirmation

```
public static byte[][] initiatorAgreeKeyEstablishWithKeyConfirmation(
                    PrivateKey initiatorPrivate, PublicKey recipientPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DHwithSHA384CKDF", "BCFIPS");
```

```
    agreement.init(initiatorPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(recipientPublic, true);

    AgreedKeyWithMacKey agreedKey = (AgreedKeyWithMacKey)agreement
                                                        .generateSecret("CMAC[128]/AES[256]");

    Mac mac = Mac.getInstance("CMAC", "BCFIPS");
    mac.init(agreedKey.getMacKey());

    DERMacData macData = new DERMacData.Builder(DERMacData.Type.UNILATERALU,
        ExValues.Initiator, ExValues.Recipient, null, null).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    agreedKey.getMacKey().zeroize(); // FIPS requirement – zero out MAC key immediately after use.

    return new byte[][] { agreedKey.getEncoded(), encMac };
}

public static byte[][] recipientAgreeKeyEstablishWithKeyConfirmation(
                    PrivateKey recipientPrivate, PublicKey initiatorPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("DHwithSHA384CKDF", "BCFIPS");

    agreement.init(recipientPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(initiatorPublic, true);

    AgreedKeyWithMacKey agreedKey = (AgreedKeyWithMacKey)agreement
                                                        .generateSecret("CMAC[128]/AES[256]");

    Mac mac = Mac.getInstance("CMAC", "BCFIPS");
    mac.init(agreedKey.getMacKey());

    DERMacData macData = new DERMacData.Builder(DERMacData.Type.UNILATERALU,
        ExValues.Initiator, ExValues.Recipient, null, null).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    agreedKey.getMacKey().zeroize(); // FIPS requirement – zero out MAC key immediately after use.

    return new byte[][] { agreedKey.getEncoded(), encMac };
}
```

As you have probably guessed, the above example is using the Diffie-Hellman key agreement to generate a 128 bit CMAC key for use with the key confirmation MAC step and a 256 bit AES key as the shared encryption key.

Elliptic Curve Diffie-Hellman

The Diffie-Hellman protocol can also be used with Elliptic Curve. Strictly speaking Elliptic Curve Diffie-Hellman, or ECDH, is really only correct for curves with a co-factor of 1. Another formulation based on incorporating the curve's co-factor is actually the one described by NIST SP 800-56A which also details the use of Elliptic Curves with Diffie-Hellman as well as the older finite field method. The variant incorporating the curve's co-factor is known as Elliptic Curve Co-factor Diffie-Hellman or ECCDH for short.

Generation of key pairs for Elliptic Curve Cryptography was covered by examples 31 and 32 in the chapter on digital signatures.

Once you have a key pair the most basic expression of the ECCDH looks like the following.

Example 45 – Basic ECCDH Key Agreement

```
public static byte[] initiatorAgreementBasic(PrivateKey initiatorPrivate, PublicKey recipientPublic)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDH", "BCFIPS");

    agreement.init(initiatorPrivate);
    agreement.doPhase(recipientPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}

public static byte[] recipientAgreementBasic(PrivateKey recipientPrivate, PublicKey initiatorPublic)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDH", "BCFIPS");

    agreement.init(recipientPrivate);
    agreement.doPhase(initiatorPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}
```

Not surprisingly it is the same as for Finite Field Diffie-Hellman. We will again get a 256 bit AES key (assuming a large enough curve) and we again have a problem on our hands if both the keys involved are for long term use.

Fortunately the answer to these problems is the same as before, use some additional user keying material (with at least some unique properties) and a KDF, in the case below an SP 800-56A Concatenation KDF, rather than the X9.63 one which would just be specified by using “KDF” rather than “CKDF”.

Example 46 – Basic ECCDH Key Agreement with a KDF

```
public static byte[] initiatorAgreementWithKdf(
                    PrivateKey initiatorPrivate, PublicKey recipientPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDHwithSHA384CKDF", "BCFIPS");

    agreement.init(initiatorPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(recipientPublic, true);

    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}

public static byte[] recipientAgreementWithKdf(
                    PrivateKey recipientPrivate, PublicKey initiatorPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDHwithSHA384CKDF", "BCFIPS");

    agreement.init(recipientPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(initiatorPublic, true);
```

```
    SecretKey agreedKey = agreement.generateSecret("AES[256]");

    return agreedKey.getEncoded();
}
```

Finally, to bring this chapter to an end, we can also use key confirmation with ECCDH as the following example shows.

Example 47 – ECCDH Key Agreement with Key Confirmation

```
public static byte[][] initiatorAgreeKeyEstablishWithKeyConfirmation(
                    PrivateKey initiatorPrivate, PublicKey recipientPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDHwithSHA384CKDF", "BCFIPS");

    agreement.init(initiatorPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(recipientPublic, true);

    AgreedKeyWithMacKey agreedKey = (AgreedKeyWithMacKey)agreement
                                                            .generateSecret("CMAC[128]/AES[256]");

    Mac mac = Mac.getInstance("CMAC", "BCFIPS");
    mac.init(agreedKey.getMacKey());

    DERMacData macData = new DERMacData.Builder(
              DERMacData.Type.UNILATERALU, ExValues.Initiator, ExValues.Recipient, null, null).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    agreedKey.getMacKey().zeroize(); // FIPS requirement

    return new byte[][] { agreedKey.getEncoded(), encMac };
}

public static byte[][] recipientAgreeKeyEstablishWithKeyConfirmation(
                   PrivateKey recipientPrivate, PublicKey initiatorPublic, byte[] userKeyingMaterial)
    throws GeneralSecurityException
{
    KeyAgreement agreement = KeyAgreement.getInstance("ECCDHwithSHA384CKDF", "BCFIPS");

    agreement.init(recipientPrivate, new UserKeyingMaterialSpec(userKeyingMaterial));
    agreement.doPhase(initiatorPublic, true);

    AgreedKeyWithMacKey agreedKey = (AgreedKeyWithMacKey)agreement
                                                            .generateSecret("CMAC[128]/AES[256]");

    Mac mac = Mac.getInstance("CMAC", "BCFIPS");
    mac.init(agreedKey.getMacKey());

    DERMacData macData = new DERMacData.Builder(
              DERMacData.Type.UNILATERALU, ExValues.Initiator, ExValues.Recipient, null, null).build();

    byte[] encMac = mac.doFinal(macData.getMacData());

    agreedKey.getMacKey().zeroize(); // FIPS requirement

    return new byte[][] { agreedKey.getEncoded(), encMac };
}
```

Certification Requests, Certificates, and Revocation

One of the issues with public key cryptography is while having a public key is all very well, how do you make sure someone actually knows it belongs to you?

Part of the answer to this, at least for some people, is the use of certificates. More particularly X.509 certificates. Cryptographic Message Syntax (CMS), S/MIME which is built on top of CMS, Time-Stamp Protocol (TSP) and a host of others all rely on the use of X.509 certificates which represent a common method of tying some identity information as well as some assurance of validity to a public key.

The BC FIPS Java API does not provide direct support in the FIPS module for the support of certification requests, certificates, and revocations, however it does work with the BC extensions APIs that provide support for these things.

Certification Requests

Not surprisingly the first step in getting a certificate for a public key is to request one. Apart from any paperwork involved this also normally involves some way of sending your public key to the certificate authority (CA) that will issue you with a certificate. Two popular ways of doing this rely on a standard called PKCS#10 described in RFC 2986 and also on another standard called Certificate Request Message Format (CRMF) described in RFC 4211. Another standard called KMIP is also starting to gain a bit of ground, but the support in BC is still at a very primitive stage.

The first one we will look at is PKCS#10. Generating a basic PKCS#10 certification request is very simple, just specify the public key and how you want to be known, and then sign the result and send it off. The resulting message should then be verifiable by anyone who is capable of decoding the public key the certification request contains.

A minimal PKCS#10 request is described in the following example.

Example 48 – A Basic PKCS#10 Request

```
public static PKCS10CertificationRequest createPkcs10Request()
    throws GeneralSecurityException, OperatorCreationException
{
    KeyPair ecKeyPair = EC.generateKeyPair();
    ContentSigner signer = new JcaContentSignerBuilder("SHA384withECDSA")
                                                .setProvider("BCFIPS").build(ecKeyPair.getPrivate());

    return new JcaPKCS10CertificationRequestBuilder(
                    new X500Name("CN=PKCS10 Example"), ecKeyPair.getPublic()).build(signer);
}


public static boolean verifyPkcs10Request(PKCS10CertificationRequest pkcs10Request)
    throws GeneralSecurityException, OperatorCreationException, PKCSException
{
    ContentVerifierProvider verifierProvider = new JcaContentVerifierProviderBuilder()
                                .setProvider("BCFIPS").build(pkcs10Request.getSubjectPublicKeyInfo());

    return pkcs10Request.isSignatureValid(verifierProvider);
}
```

In this case we can see the certification request is for an Elliptic Curve public key and it is signed using a SHA-384 digest and the ECDSA algorithm. The subject name requested for the certificate a CA might produce is the X.500 name “CN=PKCS10 Example”.

Often it is also necessary for a client to communicate to a CA some values for extensions that should appear in the issued certificate. The most common of these is the subjectAltName (subject alternative name) extension. The next example shows the steps required to add a subject alternative name to a certification request.

Example 49 – A PKCS#10 Request with Extensions

```
public static PKCS10CertificationRequest createPkcs10RequestWithSubjectAltName()
    throws GeneralSecurityException, OperatorCreationException, IOException
{
    KeyPair ecKeyPair = EC.generateKeyPair();
    ContentSigner signer = new JcaContentSignerBuilder("SHA384withECDSA")
                                .setProvider("BCFIPS")
                                .build(ecKeyPair.getPrivate());

    Extension subjectAltName = new Extension(Extension.subjectAlternativeName, false,
                new DEROctetString(new GeneralNames(new GeneralName(new X500Name("CN=Alt Name")))));

    return new JcaPKCS10CertificationRequestBuilder(
                    new X500Name("CN=PKCS10 Example"), ecKeyPair.getPublic())
                    .addAttribute(
                            PKCSObjectIdentifiers.pkcs_9_at_extensionRequest,
                            new Extensions(subjectAltName)).build(signer);
}
```

In general, the reason PKCS#10 uses a signature based on the private key of the public key being requested is to provide what is called “proof of possession” – that is by presenting a signature that can be verified by the public key in the certification the CA can be sure that the party that generated the certification request was in possession of the private key associated with the public key in the request. This is also the one exception to the rule concerning never using an encryption key for signing that FIPS allows.

Not all algorithms can be used for signing as well as encryption though, managing keys for ElGamal (or more correctly Elgamal) and DH key agreement really deserves something better and this is where CRMF steps in.

The first example of CRMF given here is basically the equivalent to what is happening in PKCS#10 – the proof of possession is given using a signature generated with the private key associated with the public key contained in the request.

Example 50 – A Basic CRMF Request

```
public static byte[] createCertificateRequestMessage(KeyPair keyPair)
    throws IOException, OperatorCreationException, CRMFException
{
    // the ONE here is the certificate request ID.
    JcaCertificateRequestMessageBuilder certReqBuild =
                                            new JcaCertificateRequestMessageBuilder(BigInteger.ONE);

    certReqBuild.setPublicKey(keyPair.getPublic())
```

```
        .setAuthInfoSender(new X500Principal("CN=CRMF Example"))
        .setProofOfPossessionSigningKeySigner(new JcaContentSignerBuilder("SHA384withECDSA")
                                                .setProvider("BCFIPS").build(keyPair.getPrivate()));

    return certReqBuild.build().getEncoded();
}

public static boolean isValidCertificateRequestMessage(byte[] msgEncoding, PublicKey publicKey)
    throws OperatorCreationException, CRMFException
{
    JcaCertificateRequestMessage certReqMsg = new JcaCertificateRequestMessage(msgEncoding)
                                                                        .setProvider("BCFIPS");

    return certReqMsg.isValidSigningKeyPOP(new JcaContentVerifierProviderBuilder()
                                                            .setProvider("BCFIPS").build(publicKey));
}
```

As you can see from the second method in the example, the validation of the request follows a similar process to that of PKCS#10 as well.

Where things become different is where a key might not be usable for signing, such as DH, ElGamal, or even RSA where you absolutely do not want to even do signing once. In this case we formulate the request differently, by specifying we want proof of possession to be dealt with in a subsequent message.

Example 51 – A CRMF Request for Encryption Only Keys

```
public static byte[] createEncCertificateRequestMessage(KeyPair keyPair)
    throws IOException, OperatorCreationException, CRMFException
{
    JcaCertificateRequestMessageBuilder certReqBuild =
                                            new JcaCertificateRequestMessageBuilder(BigInteger.ONE);

    certReqBuild.setPublicKey(keyPair.getPublic())
        .setAuthInfoSender(new X500Principal("CN=CRMF Example"))
        .setProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert);

    return certReqBuild.build().getEncoded();
}
```

The subsequent message type given in the example tells the CA that it should assume we have the private key so we are able to decrypt a message sent to us using the public one. In this case a CA would still send you back your certificate, but it would be in a message encrypted using your private key – normally derived from CMS, which we will look at later. Getting access to the certificate would then involve decrypting the message from the CA, only possible if you have the private key.

Certificate Construction

X.509 certificates generally come in 2 styles. Version 1 certificates, which are normally self signed and used as trust anchors, and version 3 certificates which also incorporate certificate extensions.

In itself the version 1 certificate provides a basic way of associating identity and validity time to a public key. The example following creates a basic self signed certificate.

Example 52 – Building a Version 1 X.509 Certificate

```
public static X509Certificate makeV1Certificate(PrivateKey caSignerKey, PublicKey caPublicKey)
    throws GeneralSecurityException, IOException, OperatorCreationException
{
    X509v1CertificateBuilder v1CertBldr = new JcaX509v1CertificateBuilder(
        new X500Name("CN=Issuer CA"),
        BigInteger.valueOf(System.currentTimeMillis()),
        new Date(System.currentTimeMillis() - 1000L * 5),
        new Date(System.currentTimeMillis() + ExValues.THIRTY_DAYS),
        new X500Name("CN=Issuer CA"),
        caPublicKey);

    JcaContentSignerBuilder signerBuilder = new JcaContentSignerBuilder("SHA384withECDSA")
                                                                        .setProvider("BCFIPS");

    return new JcaX509CertificateConverter().setProvider("BCFIPS")
                                .getCertificate(v1CertBldr.build(signerBuilder.build(caSignerKey)));
}
```

Reading the example we can see that the two X.500 names are provided to the builder. In this case they are both the same as the certificate is self issued. The reason for there being two is that the first name is associated with the certificate's issuer, and the second name is associated with the certificate's subject – that is the X.500 name the certificate is for and the public key in the certificate should be associated with. We are also using the current time in milliseconds as the certificate serial number – any unique number will do here. As the serial number is also used in connection with revocation as well as being part of the certificate identity information in protocols such as CMS it is very important you make sure two certificates cannot get handed out with the same serial number (so for any real multi-threaded application a counter would be a much better choice than the current time).

The second thing you will notice in the example is the use of the JcaX509CertificateConverter class, the converter object is used to convert a certificate object from the BC local one (X509CertificateHolder) to the JCA one (X509Certificate).

In the case of a certificate issued under another one, the issuer in the second certificate should be the same as the subject for the certificate that issued it. The next example shows the construction of version 3 certificate with same basic extensions and issued under a CA certificate (the caCertificate in the example).

Example 53 – Building a Version 3 X.509 Certificate

```
public static X509Certificate makeV3Certificate(
                        X509Certificate caCertificate, PrivateKey caPrivateKey, PublicKey eePublicKey)
    throws GeneralSecurityException, CertIOException, OperatorCreationException
{
    X509v3CertificateBuilder v3CertBldr = new JcaX509v3CertificateBuilder(
        caCertificate.getSubjectX500Principal(),                                    // issuer
        BigInteger.valueOf(System.currentTimeMillis())                              // serial number
                        .multiply(BigInteger.valueOf(10)),
        new Date(System.currentTimeMillis() - 1000L * 5),                           // start time
        new Date(System.currentTimeMillis() + ExValues.THIRTY_DAYS),                // expiry time
        new X500Principal("CN=Cert V3 Example"),                                    // subject
        eePublicKey);                                                               // subject public key

    //
    // extensions
    //
```

```
    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();
    v3CertBldr.addExtension(
        Extension.subjectKeyIdentifier,
        false,
        extUtils.createSubjectKeyIdentifier(eePublicKey));
    v3CertBldr.addExtension(
        Extension.authorityKeyIdentifier,
        false,
        extUtils.createAuthorityKeyIdentifier(caCertificate));
    v3CertBldr.addExtension(
        Extension.basicConstraints,
        true,
        new BasicConstraints(false));

    JcaContentSignerBuilder signerBuilder = new JcaContentSignerBuilder("SHA384withECDSA")
                                                                    .setProvider("BCFIPS");

    return new JcaX509CertificateConverter().setProvider("BCFIPS")
                              .getCertificate(v3CertBldr.build(signerBuilder.build(caPrivateKey)));
}
```

We can see three extensions in the example, the subject key identifier which provides a hashed value that should uniquely identify the public key, an authority key identifier which provides similar information for looking up the certificate issuer, and the "basicConstraints" extension, which in this case identifies this certificate as being one that cannot have other certificates issued under it. Other extensions can also be added, such as those related to key usage, and where to look for revocation information. You can find a full list of the possibilities in RFC 5280 and its updates. Interpreting some of these can be a bit of a black art though, so if you are building certificate paths, it is worth keeping the extension set required as simple as possible.

Certificate Revocation

At this stage we have worked out how to associated a public key with an identity and start handing it out. The question then gets raised as to what you do if it turns out the key on the certificate being handed out somehow becomes invalid before its time, either through compromise of the public key, accidental destruction of the private key, or just simply that the public key owner would prefer to be known by a different X.500 name. If you are going to start deploying certificates this is a question worth asking at the start – leaving concerns about revocation until an event like key compromise occurs is likely to result in widespread depression.

The basic answer to this question is the use of certificate revocation lists, or CRLs, to revoke certificates. A slightly more sophisticated answer is the use of protocols like Online Certificate Status Protocol (OCSP).

In the following example we are constructing a basic CRL.

Example 54 – Creating a CRL

```
public static X509CRL makeV2Crl(
                  X509Certificate caCert, PrivateKey caPrivateKey, X509Certificate revokedCertificate)
    throws GeneralSecurityException, CertIOException, OperatorCreationException
{
    Date now = new Date();
```

```
    X509v2CRLBuilder crlGen = new JcaX509v2CRLBuilder(caCert.getSubjectX500Principal(), now);

    crlGen.setNextUpdate(new Date(System.currentTimeMillis() + ExValues.THIRTY_DAYS));

    // this is the actual certificate we are revoking
    crlGen.addCRLEntry(revokedCertificate.getSerialNumber(), now, CRLReason.privilegeWithdrawn);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();
    crlGen.addExtension(
        Extension.authorityKeyIdentifier,
        false,
        extUtils.createAuthorityKeyIdentifier(caCert.getPublicKey()));

    X509CRLHolder crl = crlGen.build(new JcaContentSignerBuilder("SHA384withECDSA")
                                                        .setProvider("BCFIPS").build(caPrivateKey));

    return new JcaX509CRLConverter().setProvider("BCFIPS").getCRL(crl);
}
```

The basic elements of a CRL are similar to those of a certificate – who issued it (passed to the builder constructor), how long is the CRL up to date for (specified with setNextUpdate()), and extensions, in this case also providing additional information about the issuer. The difference is that rather than being for a particular subject the CRL identifies which certificates for a particular issuer have been revoked, when they were revoked, and why they were revoked. It is important to note that both a CRL and protocols like OCSP use the serial number of the certificate being revoked as the key to identify it. This is another reason to make sure you cannot hand out the same serial number twice when creating certificates.

OCSP, now described in RFC 6960, was originally designed as an on-line protocol built on top of CRLs, these days you can find all sorts of things in the backend, but it still has a CRL centered view of the world in terms of how it communicates. OCSP provides revocation support through the use of responders, servers which respond to OCSP requests. OCSP requests are simply status requests for given certificates under a particular issuer.

We will have a look at a simple OCSP request first.

Example 55 – Creating an OCSP Request

```
public static OCSPReq makeOcspRequest(X509Certificate caCert, X509Certificate certToCheck)
    throws OCSPException, OperatorCreationException, CertificateEncodingException
{
    DigestCalculatorProvider digCalcProv = new JcaDigestCalculatorProviderBuilder()
                                                                .setProvider("BCFIPS").build();

    // general id value for our test issuer cert and a serial number.
    CertificateID certId = new JcaCertificateID(
                      digCalcProv.get(CertificateID.HASH_SHA1), caCert, certToCheck.getSerialNumber());

    // basic request generation
    OCSPReqBuilder gen = new OCSPReqBuilder();
    gen.addRequest(certId);

    return gen.build();
}
```

As you can see a basic request requires some information about the CA and also the serial number of the certificate under investigation. Some of the information is hashed for easy processing before being

sent to the responder. In this case the request is going to use SHA-1 for the hashing, what gets used may vary from server to server though.

Once the responder gets the request, it will create a status message giving what it knows about the certificate and send the status message back to the client that made the request.

Example 56 – Creating an OCSP Response

```
public static OCSPResp makeOcspResponse(
                                    X509Certificate caCert, PrivateKey caPrivateKey, OCSPReq ocspReq)
    throws OCSPException, OperatorCreationException, CertificateEncodingException
{
    DigestCalculatorProvider digCalcProv = new JcaDigestCalculatorProviderBuilder()
                                                                    .setProvider("BCFIPS").build();
    BasicOCSPRespBuilder respGen = new JcaBasicOCSPRespBuilder(
                                           caCert.getPublicKey(), digCalcProv.get(RespID.HASH_SHA1));
    CertificateID certID = ocspReq.getRequestList()[0].getCertID();

    // magic happens…

    respGen.addResponse(certID, CertificateStatus.GOOD);

    BasicOCSPResp resp = respGen.build(
        new JcaContentSignerBuilder("SHA384withECDSA").setProvider("BCFIPS").build(caPrivateKey),
        new X509CertificateHolder[]{new JcaX509CertificateHolder(caCert)},
        new Date());

    OCSPRespBuilder rGen = new OCSPRespBuilder();

    return rGen.build(OCSPRespBuilder.SUCCESSFUL, resp);
}
```

In this case it is not really possible to provide a full example, but if you imagine that the place where the “magic happens” comment appears is the place where the revocation status of the certificate represented by certID is checked, and it turns out that the certificate is still in good standing, what follows after “magic happens” is what would probably happen.

The final link in the exchange is the client interpreting the message sent back by the responder for the original certificate status request.

Example 57 – Checking an OCSP Response

```
public static boolean isGoodCertificate(
                                   OCSPResp ocspResp, X509Certificate caCert, X509Certificate eeCert)
    throws OperatorCreationException, OCSPException, CertificateEncodingException
{
    DigestCalculatorProvider digCalcProv = new JcaDigestCalculatorProviderBuilder()
                                                                        .setProvider("BCFIPS").build();

    // SUCCESSFUL here means the OCSP request worked, it doesn't mean the certificate is valid.
    if (ocspResp.getStatus() == OCSPRespBuilder.SUCCESSFUL)
    {
        BasicOCSPResp resp = (BasicOCSPResp)ocspResp.getResponseObject();

        // make sure response is signed by the appropriate CA
        if (resp.isSignatureValid(new JcaContentVerifierProviderBuilder()
                                                .setProvider("BCFIPS").build(caCert.getPublicKey())))
        {
            // return the actual status of the certificate - null means valid.
            return resp.getResponses()[0].getCertID().matchesIssuer(
                                                  new JcaX509CertificateHolder(caCert), digCalcProv)
                && resp.getResponses()[0].getCertID().getSerialNumber()
```

```
                                                                .equals(eeCert.getSerialNumber())
                && resp.getResponses()[0].getCertStatus() == null;
        }
    }
    throw new IllegalStateException("OCSP Request Failed");
}
```

For the purposes of demonstration we have reduced the example to a method which just indicates whether or not a certificate is valid and assumes a single response (BasicOCSPResp). That said, if you were doing this in an application environment you would probably expect to make use of the status value returned in the request should the status value turn out to not be null – indicating that there was an issue with the certificate.

CertPath Validation

Now that we have looked at checking individual certificates, the next problem to solve is how to validate a chain of them, as generally you will be presented with a minimum of a CA certificate and a client, or end entity certificate (EE), as well as a trust anchor. Trust anchors, as we have already mentioned are generally self-signed, so you have to accept them at face value. If you are going to do that it is obviously worth making sure a CA certificate that purports to be issued by a particular trust anchor really is, and having done that you want to make sure that any client issued under the CA really is as well.

The next example shows how to validate a certificate path in the typical style using the PKIX validator in the BC FIPS Java Provider. The PKIX validator is so called as it is based on the certificate path validation algorithm described in RFC 5280.

Example 58 – Basic CertPath Validation

```
public static PKIXCertPathValidatorResult validateCertPath(
                                X509Certificate taCert, X509Certificate caCert, X509Certificate eeCert)
    throws GeneralSecurityException
{
    List<X509Certificate> certchain = new ArrayList<X509Certificate>();

    certchain.add(eeCert);
    certchain.add(caCert);

    CertPath certPath = CertificateFactory.getInstance("X.509", "BCFIPS")
                                                        .generateCertPath(certchain);

    Set<TrustAnchor> trust = new HashSet<TrustAnchor>();
    trust.add(new TrustAnchor(taCert, null));

    CertPathValidator certPathValidator = CertPathValidator.getInstance("PKIX", "BCFIPS");

    PKIXParameters param = new PKIXParameters(trust);
    param.setRevocationEnabled(false);
    param.setDate(new Date());

    return (PKIXCertPathValidatorResult)certPathValidator.validate(certPath, param);
}
```

Note that the trust anchor is passed as a separate parameter to the CA and EE certificate. Note also that in this case as CRLs are not being referenced as PKIXParameters.setRevocationEnabled() is called

with the value false.

Had we wanted to process CRLs as well we would have also needed to pass the CRLs to the PKIXParameters object using a CertStore which can be used to contain certificates and CRLs. You can see in the following example that a CertStore is introduced and then used to make the CRLs available for validating certificates issued under the trust anchor and the CA certificate.

Example 59 – Basic CertPath Validation with CRLs

```
public static PKIXCertPathValidatorResult validateCertPathWithCrl(
                                    X509Certificate taCert, X509CRL taCrl, X509Certificate caCert,
                                                                X509CRL caCrl, X509Certificate eeCert)
    throws GeneralSecurityException
{
    List<X509Certificate> certchain = new ArrayList<X509Certificate>();

    certchain.add(eeCert);
    certchain.add(caCert);

    CertPath certPath = CertificateFactory.getInstance("X.509","BCFIPS")
                                                        .generateCertPath(certchain);

    Set<TrustAnchor> trust = new HashSet<TrustAnchor>();
    trust.add(new TrustAnchor(taCert, null));

    Set crls = new HashSet();
    crls.add(caCrl);
    crls.add(taCrl);

    CertStore crlsStore = CertStore.getInstance("Collection",
                                          new CollectionCertStoreParameters(crls), "BCFIPS");

    CertPathValidator certPathValidator = CertPathValidator.getInstance("PKIX","BCFIPS");

    PKIXParameters param = new PKIXParameters(trust);
    param.addCertStore(crlsStore);
    param.setDate(new Date());

    return (PKIXCertPathValidatorResult)certPathValidator.validate(certPath, param);
}
```

Supporting OCSP is a little more complicated, we will not going into it here, but you can do it using a PKIXCertPathChecker. Be careful with this one – putting a network access into a path validation can also be a great way to put yourself under a denial of service attack (possibly self-induced). It is important to only use URLs you trust, and to make sure you have an appropriate fall back position when an OCSP responder is down.

Password Based Encryption and Key Storage

The BC FIPS Java module is software only, unlike a system based on a hardware adapter, key storage becomes a developer's problem. Fortunately there are a couple of different storage formats supported by the provider including one which is fully FIPS compliant (at the time of writing) for protecting private keys, and there is already a commonly used format based on PEM for saving certificates and even unprotected public/private keys where there is simply a need to write them to disk. In our case key stores are always protected by passwords, which brings up the topic of password based encryption as well.

Password Based Encryption

There are a number of mechanisms available for converting passwords into secret keys, they appear in association with applications like OpenSSL, as well as standards such as PKCS#5 and PKCS#12. The BC FIPS Java module offers just NIST standards in approved mode, as well as a few of the others in non-approved mode. The NIST standard for converting passwords to keys, specifically for key storage is described in NIST SP 800-132, part 1. It is the same algorithm as described in PKCS#5 Scheme 2 (RFC 2898).

The NIST password based key derivation function (PBKDF) is based around the use of HMACs with both SHA-1 and the SHA-2 family digests regarded as acceptable. The best way to generate one of these keys in the JCE is via a SecretKeyFactory.

Example 60 – Password Based Key Generation

```
public static SecretKey makePbeKey(char[] password)
    throws GeneralSecurityException
{
    SecretKeyFactory keyFact = SecretKeyFactory.getInstance("HmacSHA384", "BCFIPS");

    SecretKey hmacKey = keyFact.generateSecret(
                new PBEKeySpec(password,  Hex.decode("0102030405060708090a0b0c0d0e0f10"), 1024, 256));

    return new SecretKeySpec(hmacKey.getEncoded(), "AES");
}
```

In this example we are using a HMAC based on SHA-384 to generate a 256 bit AES key, which we create via a SecretKeySpec. The example shows a fixed salt and iteration count as well. While you might have a fixed iteration count it would be unusual to have a fixed salt as you would run the risk of having the two things encrypted with the same password producing the same encryption key, something which may well result in a pattern that would stand out to a third-party observer.

Encoding Public and Private Keys

The most basic thing to do with a public or private key is to convert it into a byte array and then back again.

In Java public keys are normally expected to be encodable to the same format used in an X.509

certificate. You can create this encoding using PublicKey.getEncoded(). Going the other way is a little more involved as you need a KeyFactory appropriate for the algorithm the key represents. The following example will work with RSA public keys.

Example 61 – Public Key Encoding

```
public static byte[] encodePublic(PublicKey publicKey)
{
    return publicKey.getEncoded();
}

public static PublicKey producePublicKey(byte[] encoding)
    throws GeneralSecurityException
{
    KeyFactory keyFact = KeyFactory.getInstance("RSA", "BCFIPS");

    return keyFact.generatePublic(new X509EncodedKeySpec(encoding));
}
```

Note that we are using an X509EncodedKeySpec to tell the KeyFactory what format the byte array is encoded in.

Private keys are similar, except in this case PrivateKey.getEncoded() will normally return a PKCS#8 format encoding of the private key.

Example 62 – Private Key Encoding

```
public static byte[] encodePrivate(PrivateKey privateKey)
{
    return privateKey.getEncoded();
}

public static PrivateKey producePrivateKey(byte[] encoding)
    throws GeneralSecurityException
{
    KeyFactory keyFact = KeyFactory.getInstance("RSA", "BCFIPS");

    return keyFact.generatePrivate(new PKCS8EncodedKeySpec(encoding));
}
```

As with the public key, KeyFactory.generatePrivate() is passed a PKCS8EncodedKeySpec to tell it what format the byte array representing the encoding is in.

Note that if you are not using a software based Java cryptography provider, but using a hardware based one instead, it is quite likely any attempt to recover a private key except via wrapping will result in an exception.

PEM Format

PEM format is used to store binary objects encoded as Base64. It originated with Privacy Enhanced Mail (RFC 1421) and was more recently formalized in RFC 7468 “Textual Encodings of PKIX, PKCS, and CMS Structures”. The headers are ASCII and it is even possible to include additional attributes in the header information to allow for descriptive information concerning encryption and the like.

The simplest thing to store in a PEM formatted file is a certificate. You will see PEM certificates a lot

in the TLS configuration for servers and just about anywhere else a certificate is required. The PEM writer and parser for BC lives in the org.bouncycastle.openssl package in the bcpkix jar. The following example shows method for writing a PEM certificate to a String and then reading it back.

Example 63 – Writing a Certificate in PEM Format

```
public static String writeCertificate(X509Certificate certificate)
    throws IOException
{
    StringWriter sWrt = new StringWriter();
    JcaPEMWriter pemWriter = new JcaPEMWriter(sWrt);

    pemWriter.writeObject(certificate);
    pemWriter.close();

    return sWrt.toString();
}

public static X509Certificate readCertificate(String pemEncoding)
    throws IOException, CertificateException
{
    PEMParser parser = new PEMParser(new StringReader(pemEncoding));

    X509CertificateHolder certHolder = (X509CertificateHolder)parser.readObject();

    return new JcaX509CertificateConverter().getCertificate(certHolder);
}
```

Note that while it is possible to write an X509Certificate with the JcaPEMWriter, parsing the file produces the local BC equivalent which can then be converted.

Private keys can also be written in PEM format and the JcaPEMWriter has the ability to handle them explicitly. As you will see in the example a different object is returned here as well – a PEMKeyPair.

Example 64 – Writing a Private Key in PEM Format

```
public static String writePrivateKey(PrivateKey privateKey)
    throws IOException
{
    StringWriter sWrt = new StringWriter();
    JcaPEMWriter pemWriter = new JcaPEMWriter(sWrt);

    pemWriter.writeObject(privateKey);
    pemWriter.close();

    return sWrt.toString();
}

public static PrivateKey readPrivateKey(String pemEncoding)
    throws IOException, CertificateException
{
    PEMParser parser = new PEMParser(new StringReader(pemEncoding));
    PEMKeyPair pemKeyPair = (PEMKeyPair)parser.readObject();

    return new JcaPEMKeyConverter().getPrivateKey(pemKeyPair.getPrivateKeyInfo());
}
```

The PEMKeyPair is returned where the private key is specifically noted in the PEM file, in the case where the PEM header is “-----BEGIN EC PRIVATE KEY-----” or “-----BEGIN RSA PRIATE KEY-----”. The reasons for this are largely historic and relate to how OpenSSL originally used PEM files. In the situation where the key type is unknown the header will be “-----BEGIN PRIVATE

KEY-----” and the return value will be a PKCS#8 PrivateKeyInfo object (an ASN.1 defined structure).

The PEM format also offers two ways of storing a private key that has been encrypted. The most recent one uses the encoding of PKCS#8 EncryptedPrivateKeyInfo structure, the original one uses a format that also originated with OpenSSL. Both formats are still fairly common, but it is better to use the most recent one where possible.

The following example shows how to write and read an encrypted private key using the FIPS preferred PBKDF and the AES-256 cipher in CBC mode. Note that in this case it is a two stage process, an EncryptedPrivateKeyInfo structure is created and then it is written out. Likewise on reading an EncryptedPrivateKeyInfo structure is read back, and then properly decrypted and converted into a key.

Example 65 – Writing an Encrypted Private Key in PEM Format

```
public static String writeEncryptedKey(char[] passwd, PrivateKey privateKey)
    throws IOException, OperatorCreationException
{
    StringWriter sWrt = new StringWriter();
    JcaPEMWriter pemWriter = new JcaPEMWriter(sWrt);

    PKCS8EncryptedPrivateKeyInfoBuilder pkcs8Builder =
                                              new JcaPKCS8EncryptedPrivateKeyInfoBuilder(privateKey);
    pemWriter.writeObject(pkcs8Builder.build(new JcePKCSPBEOutputEncryptorBuilder(
                          NISTObjectIdentifiers.id_aes256_CBC).setProvider("BCFIPS").build(passwd)));
    pemWriter.close();

    return sWrt.toString();
}

public static PrivateKey readEncryptedKey(char[] password, String pemEncoding)
    throws IOException, OperatorCreationException, PKCSException
{
    PEMParser parser = new PEMParser(new StringReader(pemEncoding));
    PKCS8EncryptedPrivateKeyInfo encPrivKeyInfo = (PKCS8EncryptedPrivateKeyInfo)parser.readObject();

    InputDecryptorProvider pkcs8Prov = new JcePKCSPBEInputDecryptorProviderBuilder()
                                                                .setProvider("BCFIPS").build(password);

    JcaPEMKeyConverter   converter = new JcaPEMKeyConverter().setProvider("BCFIPS");

    return converter.getPrivateKey(encPrivKeyInfo.decryptPrivateKeyInfo(pkcs8Prov));
}
```

The older OpenSSL style uses a character string rather than an ASN.1 object identifier to identify the encryption algorithm used. As it is using the OpenSSL PBKDF, the encryption is not FIPS approved. The character string defining the algorithm is of the format “<algorithm>-<keysize>-<mode>”, so in the example that follows we are using AES in CBC mode with a 256 bit key. The cipher will be configured to use PKCS#5/PKCS#7 padding where padding is required.

Example 66 – Writing an Encrypted Private Key (OpenSSL Style)

```
public static String writeEncryptedKeyOpenSsl(char[] passwd, PrivateKey privateKey)
    throws IOException, OperatorCreationException
{
    StringWriter sWrt = new StringWriter();
    JcaPEMWriter pemWriter = new JcaPEMWriter(sWrt);

    pemWriter.writeObject(privateKey,
                      new JcePEMEncryptorBuilder("AES-256-CBC").setProvider("BCFIPS").build(passwd));
```

```
    pemWriter.close();

    return sWrt.toString();
}

public static PrivateKey readEncryptedKeyOpenSsl(char[] passwd, String pemEncoding)
    throws IOException, OperatorCreationException
{
    PEMParser parser = new PEMParser(new StringReader(pemEncoding));
    PEMEncryptedKeyPair pemEncryptedKeyPair = (PEMEncryptedKeyPair)parser.readObject();

    PEMDecryptorProvider pkcs8Prov = new JcePEMDecryptorProviderBuilder()
                                                              .setProvider("BCFIPS").build(passwd);
    JcaPEMKeyConverter   converter = new JcaPEMKeyConverter().setProvider("BCFIPS");

    return converter.getPrivateKey(pemEncryptedKeyPair.decryptKeyPair(pkcs8Prov).getPrivateKeyInfo());
}
```

The OpenSSL cipher set also includes Triple-DES (“DES-EDE”), Blowfish (“BF”), DES (“DES”) and RC2 (“RC2”). The modes CFB (“CFB”), ECB (“ECB”), and OFB (“OFB”) are also supported. If you have a look at the output file for a PEM encoded private key you will see how the additional information to recognize the encryption is encoded as PEM attributes just after the header.

KeyStores

Java KeyStores are a problem area with FIPS as, strictly speaking, most of them are not FIPS compliant when it comes to storing private keys. The BC FIPS Java Provider supports two types of KeyStore locally, the BCFKS type, which has an ASN.1 based format specifically designed with the current FIPS standards in mind, and the PKCS12 type, which is described originally in the RSA Labs standard PKCS#12 and is also listed in RFC 7292.

We will have a look at the BCFKS type first.

Unlike PKCS12, the BCFKS format can store certificates, private keys, and some types of secret key. The key store is built around AES-256, SHA-512, and the NIST PBKDF, with PKCS#8 EncryptedPrivateKeyInfo structures being used to carry private keys and octet strings containing wrapped secret keys where required. Further details on the store format can be found in an Appendix in the BC FIPS Java User Guide.

The following example shows how to store a certificate in a BCFKS KeyStore.

Example 67 – Storing a Certificate in a BCFKS KeyStore

```
public static byte[] storeCertificate(char[] storePassword, X509Certificate trustedCert)
    throws GeneralSecurityException, IOException
{
    KeyStore keyStore = KeyStore.getInstance("BCFKS", "BCFIPS");
    keyStore.load(null, null);

    keyStore.setCertificateEntry("trustedca", trustedCert);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    keyStore.store(bOut, storePassword);

    return bOut.toByteArray();
}
```

This next example shows how to store a PrivateKey and its associated certificate chain in a BCFKS KeyStore.

Example 68 – Storing a PrivateKey in a BCFKS KeyStore

```
public static byte[] storePrivateKey(char[] storePassword, char[] keyPass,
                                                    PrivateKey eeKey, X509Certificate[] eeCertChain)
    throws GeneralSecurityException, IOException
{
    KeyStore keyStore = KeyStore.getInstance("BCFKS", "BCFIPS");
    keyStore.load(null, null);

    keyStore.setKeyEntry("key", eeKey, keyPass, eeCertChain);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    keyStore.store(bOut, storePassword);

    return bOut.toByteArray();
}
```

Finally, this last example is one way of storing a secret key in a BCFKS KeyStore.

Example 69 – Storing a Secret Key in a BCFKS KeyStore

```
public static byte[] storeSecretKey(char[] storePassword, char[] keyPass, SecretKey secretKey)
    throws GeneralSecurityException, IOException
{
    KeyStore keyStore = KeyStore.getInstance("BCFKS", "BCFIPS");
    keyStore.load(null, null);

    keyStore.setKeyEntry("secretkey", secretKey, keyPass, null);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    keyStore.store(bOut, storePassword);

    return bOut.toByteArray();
}
```

Note that in the case of a secret key the certificate chain parameter of the setKey() method is set to null.

PKCS12 is most commonly used to store a single set of public private credentials, in the Java world this is extended a bit and its not uncommon to find KeyStore implementations that can store multiple private keys and stand alone certificates as well. The one thing you need to be aware is that only the KeyStore password is significant with PKCS12. While there are some implementations which will allow individual key passwords as well, the vast majority do not, and if you wish to produce PKCS12 files that are compatible across different platforms it is best to avoid having passwords associated with keys in PKCS12.

The following example shows how to store a stand alone certificate in a PKCS12 KeyStore with the BC FIPS Java Provider. Keep in mind it may not work with all other providers.

Example 70 – Storing a Certificate in a PKCS#12 KeyStore

```
public static byte[] storeCertificatePkcs12(char[] storePassword, X509Certificate trustedCert)
    throws GeneralSecurityException, IOException
{
    KeyStore keyStore = KeyStore.getInstance("PKCS12", "BCFIPS");
    keyStore.load(null, null);

    keyStore.setCertificateEntry("trustedca", trustedCert);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    keyStore.store(bOut, storePassword);

    return bOut.toByteArray();
}
```

This example shows how to store a private key and its certificate chain in a PKCS12 KeyStore using the BC FIPS Java Provider.

Example 71 – Storing a Private Key in a PKCS#12 KeyStore

```
public static byte[] storePrivateKeyPkcs12(char[] storePassword, PrivateKey eeKey,
                                                                X509Certificate[] eeCertChain)
    throws GeneralSecurityException, IOException
{
    KeyStore keyStore = KeyStore.getInstance("PKCS12", "BCFIPS");
    keyStore.load(null, null);

    keyStore.setKeyEntry("key", eeKey, null, eeCertChain);

    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    keyStore.store(bOut, storePassword);

    return bOut.toByteArray();
}
```

In this case you can see a difference from the BCFKS example (Example 69) – the password parameter to the setKey() method is set to null. The PKCS#12 implementation that underlies the BC FIPS Java provider ignores the password parameter for keys.

Finally, as you have probably noticed, there is a bit of variation in what can be done, or what is expected to be done with files following the PKCS#12 standard. Originally we at BC thought we would be able to capture all the variations under the KeyStore API however this has proved to be impossible.

This last example shows the use of the BC PKCS API (found in the bcpkix jar) to create a file using the format defined in PKCS#12. The method is called createPfxPdu() as that is the actual name given in the PKCS#12 standard to the ASN.1 structure that contains the certificate and key data in the resulting KeyStore.

Example 72 – Using the BC API to create a PKCS#12 KeyStore

```
public static byte[] createPfxPdu(char[] passwd, PrivateKey privKey, X509Certificate[] certs)
    throws GeneralSecurityException, OperatorCreationException, PKCSException, IOException
{
    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();
    PKCS12SafeBagBuilder caCertBagBuilder = new JcaPKCS12SafeBagBuilder(certs[1]);

    caCertBagBuilder.addBagAttribute(
                PKCSObjectIdentifiers.pkcs_9_at_friendlyName, new DERBMPString("CA Certificate"));
```

```
    // store the key certificate
    PKCS12SafeBagBuilder eeCertBagBuilder = new JcaPKCS12SafeBagBuilder(certs[0]);

    eeCertBagBuilder.addBagAttribute(
                    PKCSObjectIdentifiers.pkcs_9_at_friendlyName, new DERBMPString("End Entity Key"));
    eeCertBagBuilder.addBagAttribute(
                    PKCSObjectIdentifiers.pkcs_9_at_localKeyId,
                                        extUtils.createSubjectKeyIdentifier(certs[0].getPublicKey()));

    // store the private key
    PKCS12SafeBagBuilder keyBagBuilder = new JcaPKCS12SafeBagBuilder(privKey,
        new JcePKCSPBEOutputEncryptorBuilder(
                                    PKCSObjectIdentifiers.pbeWithSHAAnd3_KeyTripleDES_CBC)
                                                        .setProvider("BCFIPS").build(passwd));

    keyBagBuilder.addBagAttribute(
                    PKCSObjectIdentifiers.pkcs_9_at_friendlyName, new DERBMPString("End Entity Key"));
    keyBagBuilder.addBagAttribute(
                    PKCSObjectIdentifiers.pkcs_9_at_localKeyId,
                                        extUtils.createSubjectKeyIdentifier(certs[0].getPublicKey()));

    // create the actual PKCS#12 blob.
    PKCS12PfxPduBuilder pfxPduBuilder = new PKCS12PfxPduBuilder();
    PKCS12SafeBag[] safeBags = new PKCS12SafeBag[2];
    safeBags[0] = eeCertBagBuilder.build();
    safeBags[1] = caCertBagBuilder.build();
    pfxPduBuilder.addEncryptedData(new JcePKCSPBEOutputEncryptorBuilder(
                                            PKCSObjectIdentifiers.pbeWithSHAAnd3_KeyTripleDES_CBC)
                                                    .setProvider("BCFIPS").build(passwd), safeBags);
    pfxPduBuilder.addData(keyBagBuilder.build());

    return pfxPduBuilder.build(new JcePKCS12MacCalculatorBuilder()
                                                        .setProvider("BCFIPS"), passwd).getEncoded();
}
```

The example is basically performing the same task as Example 72. While it looks more complicated than the original one using the KeyStore API, once you have read it a few times you will realize it is not actually more complicated, it is just more verbose. That said, it does also allow for much finer grained control over how the PKCS#12 file is constructed and what is placed in it. Additional classes are also available in the PKCS package to read PKCS#12 formatted files as well, and it is worth taking a little time to become familiar with them as they can help you work through any issues you might have with PKCS#12 formatted files that the KeyStore API seems unable to handle.

CMS, S/MIME, and TSP

There are two common formats for encrypted messaging: Cryptographic Message Syntax (CMS) and OpenPGP. Both formats are useful to know about, most of the time they will save you from having to invent your own format, and even where you do end up having to come up with something of your own, they provide useful case studies of how these things can be done.

This chapter will look at messaging based on CMS. We will look at the CMS protocol first as, in addition to forming the basis of Secure MIME (S/MIME), and Time-Stamp Protocol (TSP), it also makes an appearance in protocols like CRMF which we looked at in the chapter covering certification requests. CMS provides fundamental structures and approaches for doing signing, encryption, and the sending and interpreting of authenticated data.

The Bouncy Castle APIs support both an in-memory model for CMS objects as well as a streaming model to make large data object processing possible. We will look at the use of the in-memory model. The streaming model is similar in use in most respects and details on that can be found in the documentation for the CMS APIs. You will need to use the streaming model if you are handling very large files.

CMS Signatures and Counter Signatures

CMS signatures appear in CMS signed data structures. There are two types, one which contains the data that has been signed – the encapsulated type – and one which does not contain the data that was signed but expects that the data will be sent separately and made available if a verification of the CMS signature is required – the detached type. Detached signatures have a variety of uses, the most obvious being S/MIME as they make it possible to keep the data that was signed “human readable”, rather than converting it into an ASN.1 BER encoded blob. CMS signatures can support multiple signers, and there is also provision for third parties to act as a counter signer for particular signers as well.

This first example shows how to generate an encapsulated signed object and then how to verify one. The algorithm used for signing is ECDSA using the SHA-384 digest and there is a single signer involved.

Example 73 – Generating a CMS Encapsulated Signature

```
public static byte[] createSignedObject(
                        PrivateKey signingKey, X509Certificate signingCert, byte[] data)
    throws GeneralSecurityException, OperatorCreationException, CMSException, IOException
{
    List<X509Certificate> certList = new ArrayList<X509Certificate>();

    CMSTypedData msg = new CMSProcessableByteArray(data);
    certList.add(signingCert);

    Store certs = new JcaCertStore(certList);

    DigestCalculatorProvider digProvider =
                             new JcaDigestCalculatorProviderBuilder().setProvider("BCFIPS").build();
    JcaSignerInfoGeneratorBuilder signerInfoGeneratorBuilder =
                                                    new JcaSignerInfoGeneratorBuilder(digProvider);
```

```
    ContentSigner signer = new JcaContentSignerBuilder("SHA384withECDSA")
                                                          .setProvider("BCFIPS").build(signingKey);

    CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

    gen.addSignerInfoGenerator(signerInfoGeneratorBuilder.build(signer, signingCert));
    gen.addCertificates(certs);

    // true indicates the data used to create the signature is to be included as well
    return gen.generate(msg, true).getEncoded();
}

public static boolean verifySignedObject(byte[] cmsSignedData)
    throws GeneralSecurityException, OperatorCreationException, CMSException
{
    CMSSignedData          signedData = new CMSSignedData(cmsSignedData);
    Store                  certStore = signedData.getCertificates();
    SignerInformationStore signers = signedData.getSignerInfos();

    Collection c = signers.getSigners();
    Iterator it = c.iterator();

    while (it.hasNext())
    {
        SignerInformation     signer = (SignerInformation)it.next();
        Collection            certCollection = certStore.getMatches(signer.getSID());
        Iterator              certIt = certCollection.iterator();
        X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

        if (!signer.verify(new JcaSimpleSignerInfoVerifierBuilder().setProvider("BCFIPS").build(cert)))
        {
            return false;
        }
    }
    return true;
}
```

The passing of the parameter value true to the gen.generate() method in the createSignedObject() method is what determines that the data is encapsulated in the signature. You can also see in createSignedObject() that the signer is added by creating a JcaSignerInfoGeneratorBuilder and using the private key and its associated certificate to provide the arguments to the build() method on the builder. In the case of createSignedObject() we also add the signer's certificate to the generator so that it will be included in the CMS signed data. This will result in CMS signed data object that is self contained, at least to the point where the signer's signature can be verified. Note you would still want to check the provenance of the certificate.

The verifySignedObject() method in the example shows one approach for verifying at least one signature in a CMS signed data object. CMSSignedData.getCertificates() returns the certificates that were stored as part of the signature generation process. Note that that there are no guarantees that certificates appropriate to verifying the signer signatures will be in a CMS signed data object unless you have a prior agreement with the party that created the original signed data object. In this case we do have the certificate and we can use the Signer Identifier (SID) attached to the SignerInformation object to retrieve the correct certificate for verifying the signature. Once we have done that we can use an appropriately configured SignerInfoVerifierBuilder (in this case the JcaSimpleSignerInfoVerifierBuilder) to verify the signature (or not).

Creating a detached signature follows the same steps as creating an encapsulated one. You can see the

difference at the end of the createDetachedSignature() method – we do not pass true to generate(). Likewise there is a small difference in the verifyDetachedData() method from what we saw earlier. The data that was signed has to be passed in to the constructor of the CMSSignedData object.

Example 74 – Generating and Verifying a CMS Detached Signature

```
public static byte[] createDetachedSignature(
                                PrivateKey signingKey, X509Certificate signingCert, byte[] data)
    throws GeneralSecurityException, OperatorCreationException, CMSException, IOException
{
    List<X509Certificate> certList = new ArrayList<X509Certificate>();

    CMSTypedData msg = new CMSProcessableByteArray(data);
    certList.add(signingCert);

    Store certs = new JcaCertStore(certList);

    DigestCalculatorProvider digProvider = new JcaDigestCalculatorProviderBuilder()
                                                            .setProvider("BCFIPS").build();
    JcaSignerInfoGeneratorBuilder signerInfoGeneratorBuilder =
                                                        new JcaSignerInfoGeneratorBuilder(digProvider);
    ContentSigner signer = new JcaContentSignerBuilder("SHA384withECDSA")
                                                                .setProvider("BCFIPS").build(signingKey);

    CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

    gen.addSignerInfoGenerator(signerInfoGeneratorBuilder.build(signer, signingCert));
    gen.addCertificates(certs);

    return gen.generate(msg).getEncoded();
}

public static boolean verifyDetachedData(byte[] cmsSignedData, byte[] data)
    throws GeneralSecurityException, OperatorCreationException, CMSException
{
    CMSSignedData          signedData = new CMSSignedData(
                                                new CMSProcessableByteArray(data), cmsSignedData);
    Store                  certStore = signedData.getCertificates();
    SignerInformationStore signers = signedData.getSignerInfos();

    Collection c = signers.getSigners();
    Iterator it = c.iterator();

    while (it.hasNext())
    {
        SignerInformation     signer = (SignerInformation)it.next();
        Collection            certCollection = certStore.getMatches(signer.getSID());
        Iterator              certIt = certCollection.iterator();
        X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

        if (!signer.verify(new JcaSimpleSignerInfoVerifierBuilder().setProvider("BCFIPS").build(cert)))
        {
            return false;
        }
    }

    return true;
}
```

We will meet the SignerInfoGeneratorBuilder and SignerInfoVerifierBuilder objects again when we look at S/MIME later in this chapter.

Counter signatures are different from regular ones as the process of creating one involves the signature of another signer, rather than the data that was signed in the first place. We can see this reflected in the

first line of the example – a CMSSignedData object is created first, and then the SignerInformation object, representing the original signer, is extracted. After that a generator is constructed but given the original signer as input instead of data, and then the signer resulting from that is retrieved and added to the original signer. Finally a new generator is created to bundle the modified signer and the data together and the resulting CMSSignedData object is encoded and returned.

Example 75 – Generating a CMS Counter Signature

```
public static byte[] createCounterSignedData(
          PrivateKey signingKey, X509Certificate signingCert,
          byte[] data, PrivateKey counterSignerKey, X509Certificate counterSignerCert)
    throws OperatorCreationException, GeneralSecurityException, CMSException, IOException
{
    CMSSignedData signedData = new CMSSignedData(createSignedObject(signingKey, signingCert, data));

    SignerInformation signer = signedData.getSignerInfos().iterator().next();
    CMSSignedDataGenerator counterSignerGen = new CMSSignedDataGenerator();
    DigestCalculatorProvider digProvider = new JcaDigestCalculatorProviderBuilder()
                                                                    .setProvider("BCFIPS").build();
    JcaSignerInfoGeneratorBuilder signerInfoGeneratorBuilder =
                                                        new JcaSignerInfoGeneratorBuilder(digProvider);

    counterSignerGen.addSignerInfoGenerator(signerInfoGeneratorBuilder.build(
                            new JcaContentSignerBuilder("SHA384withRSA")
                                        .setProvider("BCFIPS").build(counterSignerKey),
                            counterSignerCert));

    SignerInformationStore counterSigners = counterSignerGen.generateCounterSigners(signer);

    signer = SignerInformation.addCounterSigners(signer, counterSigners);

    CMSSignedDataGenerator signerGen = new CMSSignedDataGenerator();

    signerGen.addCertificate(new JcaX509CertificateHolder(signingCert));
    signerGen.addCertificate(new JcaX509CertificateHolder(counterSignerCert));
    signerGen.addSigners(new SignerInformationStore(signer));

    return signerGen.generate(new CMSProcessableByteArray(data), true).getEncoded();
}

public static boolean verifyCounterSignature(byte[] cmsSignedData)
    throws OperatorCreationException, GeneralSecurityException, CMSException, IOException
{
    CMSSignedData         signedData = new CMSSignedData(cmsSignedData);
    SignerInformation     signer = signedData.getSignerInfos().iterator().next();
    SignerInformation     counterSigner = signer.getCounterSignatures().iterator().next();
    Collection            certCollection = signedData.getCertificates()
                                                        .getMatches(counterSigner.getSID());
    Iterator              certIt = certCollection.iterator();
    X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

    return counterSigner.verify(new JcaSimpleSignerInfoVerifierBuilder()
                                                        .setProvider("BCFIPS").build(cert));
}
```

You can see verifying a counter signature is a two step process as well, starting with the CMS signed data, you find the signer of interest, and then get the counter signatures present on it. After that it is like verifying any other CMS signer.

CMS Encrypted Data

Encrypted data in CMS is always encapsulated and is stored in what are called CMS enveloped data

structures. With enveloped data we talk about recipients, rather than signers, and we use generators to create recipient information objects as well. When decrypting we need to create a recipient that is targeted to the encryption used and there are a range of recipient objects to accommodate this. Like signed data can support multiple signers, enveloped data can support multiple recipients.

The first example we are going to look at is for a recipient using a key transport algorithm, such as RSA OAEP. The example below shows the encryption and decryption of data where the data itself is encrypted using an AES 256 bit session key and AES in CBC mode. This process is carried out in the createKeyTransEnvelopedObject() method. While it is not immediately obvious, what is also done under the covers is that the AES session key is encrypted using RSA OAEP so the owner of the private key associated with the recipient certificate can then recover the original data as seen in the extractKeyTransEnvelopedData() method.

Example 76 – CMS Encryption using RSA

```
public static byte[] createKeyTransEnvelopedObject(X509Certificate encryptionCert, byte[] data)
    throws GeneralSecurityException, CMSException, IOException
{
    CMSEnvelopedDataGenerator envelopedGen = new CMSEnvelopedDataGenerator();
    JcaAlgorithmParametersConverter paramsConverter = new JcaAlgorithmParametersConverter();

    envelopedGen.addRecipientInfoGenerator(
                    new JceKeyTransRecipientInfoGenerator(
                            encryptionCert,
                            paramsConverter.getAlgorithmIdentifier(PKCSObjectIdentifiers.id_RSAES_OAEP,
                            OAEPParameterSpec.DEFAULT)).setProvider("BCFIPS"));

    return envelopedGen.generate(
        new CMSProcessableByteArray(data),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                    .setProvider("BCFIPS").build()).getEncoded();
}

// encryption certificate is used to identify the recipient associated with the private key
public static byte[] extractKeyTransEnvelopedData(
                     PrivateKey privateKey, X509Certificate encryptionCert, byte[] encEnvelopedData)
    throws CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    Collection c = recipients.getRecipients(new JceKeyTransRecipientId(encryptionCert));
    Iterator it = c.iterator();
    if (it.hasNext())
    {
        RecipientInformation recipient = (RecipientInformation)it.next();
        return recipient.getContent(new JceKeyTransEnvelopedRecipient(privateKey)
                                                                    .setProvider("BCFIPS"));
    }
    throw new IllegalArgumentException("recipient for certificate not found");
}
```

You can see in the example, that CMS also carries the idea of a Recipient Identifier (RID) similar to the Signer Identifier (SID). In this case we construct a RID from a certificate using the JceKeyTransRecipientId constructor, and then use that to find the recipient matching the private key we have.

CMS also allows for the generation of recipients using key agreement. In this example we see the use of a recipient based on the key agreement algorithm EC CDH using the X9.63 KDF. In this case the

under-the-covers operation using the agreed key that will be generated by the recipient to create an AES-256 bit wrapping key, and the wrapping key is then used to recover the AES session key that was used to encrypt the data.

Example 77 – CMS Encryption using Key Agreement

```
public static byte[] createKeyAgreeEnvelopedObject(
    PrivateKey initiatorKey, X509Certificate initiatorCert, X509Certificate recipientCert, byte[] data)
    throws GeneralSecurityException, CMSException, IOException
{
    CMSEnvelopedDataGenerator envelopedGen = new CMSEnvelopedDataGenerator();

    envelopedGen.addRecipientInfoGenerator(new JceKeyAgreeRecipientInfoGenerator(
                                                    CMSAlgorithm.ECCDH_SHA384KDF,
                                                     initiatorKey,
                                                     initiatorCert.getPublicKey(),
                                                     CMSAlgorithm.AES256_WRAP)
                                            .addRecipient(recipientCert).setProvider("BCFIPS"));

    return envelopedGen.generate(
        new CMSProcessableByteArray(data),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                       .setProvider("BCFIPS").build()).getEncoded();
}

public static byte[] extractKeyAgreeEnvelopedData(
                      PrivateKey recipientKey, X509Certificate recipientCert, byte[] encEnvelopedData)
    throws GeneralSecurityException, CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);

    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new JceKeyAgreeRecipientId(recipientCert);
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(new JceKeyAgreeEnvelopedRecipient(recipientKey).setProvider("BCFIPS"));
}
```

Other than the use of a different recipient information generator, and a different style RID to recover the recipient object for decryption, you can see the process for encrypting and decrypting data using key agreement is basically the same as for key transport.

In the next example we look at the use of a password to create a CMS enveloped structure. The recipient information generator is a bit more complicated than what we have seen previously, largely because there are so many more optional and variable parameters.

In the example given here the parameter set has been chosen specifically with FIPS approved algorithms in mind, and a UTF-8 character encoding has been selected for converting the password into a byte array to allow the broadest range of character sets to be used and taken advantage of.

Example 78 – CMS Encryption using a Password

```
public static byte[] createPasswordEnvelopedObject(char[] passwd, byte[] salt,
                                                                    int iterationCount, byte[] data)
    throws GeneralSecurityException, CMSException, IOException
{
    CMSEnvelopedDataGenerator envelopedGen = new CMSEnvelopedDataGenerator();

    envelopedGen.addRecipientInfoGenerator(
        new JcePasswordRecipientInfoGenerator(CMSAlgorithm.AES256_CBC, passwd)
```

```
        .setProvider("BCFIPS")
        .setPasswordConversionScheme(PasswordRecipient.PKCS5_SCHEME2_UTF8)
        .setPRF(PasswordRecipient.PRF.HMacSHA384)
        .setSaltAndIterationCount(salt, iterationCount));

    return envelopedGen.generate(
        new CMSProcessableByteArray(data),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                        .setProvider("BCFIPS").build()).getEncoded();
}

public static byte[] extractPasswordEnvelopedData(char[] passwd, byte[] encEnvelopedData)
    throws GeneralSecurityException, CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new PasswordRecipientId();
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(
        new JcePasswordEnvelopedRecipient(passwd)
        .setProvider("BCFIPS")
        .setPasswordConversionScheme(PasswordRecipient.PKCS5_SCHEME2_UTF8));
}
```

The RID associated with passwords, while it exists, is not really as useful as the other RIDs we have seen. The issue being that there is nothing to really hang onto as an identifier. Other than that, once you identify the use of the recipient generator and the RID, example 79 follows the same process as examples 77 and 78 do.

In this last example we will look at the use of a symmetric key, known as a key encryption key, to encrypt the session key. This is straight forward, but requires the use of an agreed key id between the sender of the enveloped data and the recipient that is meant to process it as well as the sharing of the key itself.

Example 79 – CMS Encryption using a Key Encryption Key

```
public static byte[] createKekEnvelopedObject(byte[] keyID, SecretKey keyEncryptionKey, byte[] data)
    throws GeneralSecurityException, CMSException, IOException
{
    CMSEnvelopedDataGenerator envelopedGen = new CMSEnvelopedDataGenerator();

    envelopedGen.addRecipientInfoGenerator(
        new JceKEKRecipientInfoGenerator(keyID, keyEncryptionKey));

    return envelopedGen.generate(
        new CMSProcessableByteArray(data),
        new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                        .setProvider("BCFIPS").build()).getEncoded();
}

public static byte[] extractKekEnvelopedData(
                                    byte[] keyID, SecretKey keyEncryptionKey, byte[] encEnvelopedData)
    throws GeneralSecurityException, CMSException
{
    CMSEnvelopedData envelopedData = new CMSEnvelopedData(encEnvelopedData);

    RecipientInformationStore recipients = envelopedData.getRecipientInfos();
    RecipientId rid = new KEKRecipientId(keyID);
    RecipientInformation recipient = recipients.get(rid);

    return recipient.getContent(
                            new JceKEKEnvelopedRecipient(keyEncryptionKey).setProvider("BCFIPS"));
```

```
}
```

In this case the RID has some meaning as there is a keyID to use. Otherwise it follows the same procedure as the other recipient types.

Multiple recipients, including different recipient types, can be included in the same enveloped data. Just bare in mind if you are using passwords that there's no way to distinguish which recipient is using which password, if you have more than one password recipient you will need some other way of determining which recipient to use for a particular user.

CMS Authenticated Data

Authenticated data is different from signed data in the sense that there is just a MAC associated with the data, but it is similar to enveloped data in the sense that we use recipients again.

The following only provides an example of authenticated data using a key transport (KeyTrans) recipient. That said, any recipient that can be used with enveloped data can be used with authenticated data. The CMSAuthenticatedDataGenerator.generate() method is unique to the type though as in this case the data is transmitted in the clear, but the a MAC needs to be calculated on it using a session key.

Example 80 – Creating and Verifying CMS Authenticated Data

```
public static byte[] createAuthenticatedData(
            X509Certificate originatorCertificate, X509Certificate recipientCertificate, byte[] data)
    throws GeneralSecurityException, CMSException, IOException
{
    ASN1ObjectIdentifier macAlg = CMSAlgorithm.DES_EDE3_CBC;
    CMSAuthenticatedDataGenerator authDataGenerator = new CMSAuthenticatedDataGenerator();
    X509CertificateHolder origCert = new JcaX509CertificateHolder(originatorCertificate);

    authDataGenerator.setOriginatorInfo(new OriginatorInfoGenerator(origCert).generate());
    authDataGenerator.addRecipientInfoGenerator(
                                        new JceKeyTransRecipientInfoGenerator(recipientCertificate)
                                        .setProvider("BCFIPS"));

    return authDataGenerator.generate(
                            new CMSProcessableByteArray(data),
                            new JceCMSMacCalculatorBuilder(macAlg)
                                                    .setProvider("BCFIPS").build()).getEncoded();
}

public static byte[] extractAuthenticatedData(
                    PrivateKey recipientPrivateKey, X509Certificate recipientCert, byte[] encAuthData)
    throws GeneralSecurityException, CMSException
{
    CMSAuthenticatedData authData = new CMSAuthenticatedData(encAuthData);

    RecipientInformationStore recipients = authData.getRecipientInfos();
    RecipientInformation recipient = recipients.get(new JceKeyTransRecipientId(recipientCert));

    if (recipient != null)
    {
        byte[] recData = recipient.getContent(
                    new JceKeyTransAuthenticatedRecipient(recipientPrivateKey).setProvider("BCFIPS"));

        if (Arrays.constantTimeAreEqual(authData.getMac(), recipient.getMac()))
        {
            return recData;
        }
```

```
        else
        {
            throw new IllegalStateException("MAC check failed");
        }
    }
    throw new IllegalStateException("no recipient found");
}
```

In the example the MAC is based around the Triple-DES CBC MAC. You will also note that a constant time comparison is done when it is checked. These days it is regarded as best to make you cryptography code as boring and as constant-time as possible, rejecting MACs at the earliest time possible can be provide a hint to an outside observer about what a valid MAC might look like, so be careful where you choose to optimize.

S/MIME Signed Data

S/MIME is built directly on top of CMS. Signed S/MIME objects can be either encapsulated or detached in which case a MIME multipart object is used and the content being signed is in one part, with the signature following in the next part. The advantage of the multipart format is that a reader of the message does not have to be able to decode the actual signed message, so a reader without the ability to process the signature can still read the signed data.

Detached signatures have a MIME type of “application/pkcs7-signature” and encapsulated signatures have a MIME type of “application/pkcs7-mime”.

The following example shows how to create a S/MIME signed object with a detached signature, so a multipart message. As part of this process the createSignedMultipart() method also includes an SMIMECapabilitiesAttribute. The capabilities attribute is used to tell the recipient of the S/MIME message some details about what the capabilities of the originator of the message are. This information is useful where the receiver wishes to send a reply. The capabilities attribute is created in the generateSignedAttributes() method.

Example 81 – Creating and Verifying an S/MIME Signed Multipart

```
private static ASN1EncodableVector generateSignedAttributes()
{
    ASN1EncodableVector signedAttrs = new ASN1EncodableVector();

    SMIMECapabilityVector caps = new SMIMECapabilityVector();
    caps.addCapability(SMIMECapability.aES128_CBC);
    caps.addCapability(SMIMECapability.aES192_CBC);
    caps.addCapability(SMIMECapability.aES256_CBC);

    signedAttrs.add(new SMIMECapabilitiesAttribute(caps));

    return signedAttrs;
}

public static MimeMultipart createSignedMultipart(
                            PrivateKey signingKey, X509Certificate signingCert, MimeBodyPart message)
    throws GeneralSecurityException, OperatorCreationException, SMIMEException, IOException
{
    List<X509Certificate> certList = new ArrayList<X509Certificate>();
    certList.add(signingCert);
```

```
    Store certs = new JcaCertStore(certList);

    ASN1EncodableVector signedAttrs = generateSignedAttributes();

    signedAttrs.add(new Attribute(CMSAttributes.signingTime, new DERSet(new Time(new Date()))));

    SMIMESignedGenerator gen = new SMIMESignedGenerator();

    gen.addSignerInfoGenerator(new JcaSimpleSignerInfoGeneratorBuilder()
                                    .setProvider("BCFIPS")
                                    .setSignedAttributeGenerator(new AttributeTable(signedAttrs))
                                    .build("SHA384withRSAandMGF1", signingKey, signingCert));
    gen.addCertificates(certs);

    return gen.generate(message);
}

public static boolean verifySignedMultipart(MimeMultipart signedMessage)
    throws GeneralSecurityException, OperatorCreationException,
                                                    CMSException, SMIMEException, MessagingException
{
    SMIMESigned            signedData = new SMIMESigned(signedMessage);
    Store                  certStore = signedData.getCertificates();
    SignerInformationStore signers = signedData.getSignerInfos();

    Collection c = signers.getSigners();
    Iterator it = c.iterator();

    while (it.hasNext())
    {
        SignerInformation     signer = (SignerInformation)it.next();
        Collection            certCollection = certStore.getMatches(signer.getSID());
        Iterator              certIt = certCollection.iterator();
        X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

        if (!signer.verify(new JcaSimpleSignerInfoVerifierBuilder().setProvider("BCFIPS").build(cert)))
        {
            return false;
        }
    }
    return true;
}
```

Not surprisingly, once you take into account the use of the JavaMail library, the create and verify method look just like the equivalents for the CMS signed data case. It is just the case that the underlying carrier for the data has changed as the data is now one part of a two part message.

S/MIME Encrypted Data

Creating encrypted, or enveloped, data using S/MIME also follows the same procedure as with CMS.

Encrypted, or enveloped, data has the MIME type “application/pkcs7-mime”. You will probably notice this is the same as for encapsulated signed data. In this case the two are distinguished using an additional attribute “smime-type”. For this situation smime-type will be set equal to “enveloped-data”, whereas in the case of encapsulated signed data the smime-type attribute will be equal to “signed-data”.

In this example we are using a key transport recipient but any valid CMS recipient, or recipients, can be used instead if the receivers of the S/MIME message are capable of processing them.

Example 82 – Creating and Processing S/MIME Encrypted Messages

```
public static MimeBodyPart createEnvelopedBodyPart(
                                            X509Certificate encryptionCert, MimeBodyPart message)
    throws GeneralSecurityException, SMIMEException, CMSException, IOException
{
    SMIMEEnvelopedGenerator gen = new SMIMEEnvelopedGenerator();
    gen.addRecipientInfoGenerator(new
JceKeyTransRecipientInfoGenerator(encryptionCert).setProvider("BCFIPS"));
    return gen.generate(message, new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                                        .setProvider("BCFIPS").build());
}

// as this is based on CMS we again use encryptionCert to identify the recipient for the private key
public static MimeBodyPart extractEnvelopedBodyPart(
                 PrivateKey privateKey, X509Certificate encryptionCert, MimeBodyPart envelopedBodyPart)
    throws SMIMEException, CMSException, MessagingException
{
    SMIMEEnveloped envelopedData = new SMIMEEnveloped(envelopedBodyPart);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();

    Collection c = recipients.getRecipients(new JceKeyTransRecipientId(encryptionCert));

    Iterator it = c.iterator();
    if (it.hasNext())
    {
        RecipientInformation recipient = (RecipientInformation)it.next();

        return SMIMEUtil.toMimeBodyPart(
                            recipient.getContent(new JceKeyTransEnvelopedRecipient(privateKey)
                                                                    .setProvider("BCFIPS")));
    }

    throw new IllegalArgumentException("recipient for certificate not found");
}
```

It is worth also noting that often with S/MIME enveloped messages there will normally be at least two recipients – usually the originator of the message will add themselves as a recipient as well as it will make it possible for them to decrypt the message they sent at a later date. While it is not done in the example, for the key transport originator, it is just a matter of adding another JceKeyTransRecipientInfoGenerator built around the originator's certificate.

Occasionally you will also want to sign a message, encrypt the result, and then send that. In this case you need to create the signed multipart first, and then encrypt it - following the principal of “sign what you mean, encrypt what you mean to protect”. The main thing to deal with here is how JavaMail needs to present the signed multipart for encryption – it needs to be placed into another MimeBodyPart object and then that is passed for encryption. If you look at the create method in the example you will see how that is done.

Example 83 – Using Signing and Encryption together with S/MIME

```
public static MimeBodyPart createSignedEncryptedBodyPart(
                     PrivateKey signingKey, X509Certificate signingCert,
                                              X509Certificate encryptionCert, MimeBodyPart message)
    throws GeneralSecurityException, SMIMEException, CMSException, IOException,
                                                OperatorCreationException, MessagingException
{
    SMIMEEnvelopedGenerator gen = new SMIMEEnvelopedGenerator();
    gen.addRecipientInfoGenerator(new JceKeyTransRecipientInfoGenerator(encryptionCert)
                                                                        .setProvider("BCFIPS"));

    MimeBodyPart bodyPart = new MimeBodyPart();

    bodyPart.setContent(createSignedMultipart(signingKey, signingCert, message));
```

```
    return gen.generate(bodyPart, new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                                                    .setProvider("BCFIPS").build());
}

public static boolean verifySignedEncryptedBodyPart(
                 PrivateKey privateKey, X509Certificate encryptionCert, MimeBodyPart envelopedBodyPart)
    throws SMIMEException, CMSException, GeneralSecurityException, OperatorCreationException,
                                                                    MessagingException, IOException
{
    SMIMEEnveloped envelopedData = new SMIMEEnveloped(envelopedBodyPart);
    RecipientInformationStore recipients = envelopedData.getRecipientInfos();

    Collection c = recipients.getRecipients(new JceKeyTransRecipientId(encryptionCert));
    Iterator it = c.iterator();
    if (it.hasNext())
    {
        RecipientInformation recipient = (RecipientInformation)it.next();
        MimeBodyPart signedPart = SMIMEUtil.toMimeBodyPart(
            recipient.getContent(new JceKeyTransEnvelopedRecipient(privateKey).setProvider("BCFIPS")));

        return verifySignedMultipart((MimeMultipart)signedPart.getContent());
    }
    throw new IllegalArgumentException("recipient for certificate not found");
}
```

On decryption you then need to be able to recover the multipart so you can verify the signature. The BC S/MIME API includes a utility class SMIMEUtil with a toMimeBodyPart() method to enable this to be done. As you can see in the verify method of the example, once the content of the enveloped data has been converted back into a MIME body part, the multipart representing the signature is just the content of it, as it was when we originally created the message.

Time-Stamp Protocol

Time-Stamp Protocol (TSP) is defined in RFC 3161 and has become increasing important over the last few years as all us have to grapple with the “horror” of valid certificates expiring. The idea behind it is fairly simple, you just get a trusted third-party to issue a longer term signature testifying that you sent it a particular hash at a particular time. Assuming the hash represents a particular blob or document, and assuming the hash function is otherwise secure, you can then testify that the blob or document really was that value on the particular time.

The feature of TSP is you do not have to send the data that the hash the time-stamp server signs is based on. It keeps the protocol small and makes it easier to both request and provide time-stamps. It is easy to time-stamp previously time-stamped data as well.

Our first example shows how to create a basic TSP request.

Example 84 – Creating a TSP Request

```
public static byte[] createTspRequest(byte[] sha384Hash)
    throws IOException
{
    TimeStampRequestGenerator reqGen = new TimeStampRequestGenerator();
    return reqGen.generate(TSPAlgorithms.SHA384, sha384Hash).getEncoded();
}
```

As you can see it is quite straight forward. What hash you might use depends on what you have

available and may also depend on what the server is willing to process, but in this case we are using SHA-384. Sometimes you want the server to include its TSP verification certificate in the response as well. To do this you should also add:

```
reqGen.setCertReq(true);
```

to the request generation. TSP responses are based on CMS signed data and some applications are not capable of verifying a CMS signed data object where the signer has not included its signing certificates. If you are having an issue with a TSP response and a third party application, the first thing worth checking is the presence of the TSP server's time-stamp verification certificate.

After some initial set up creating a TSP response involves creating a TimeStampTokenGenerator and then using that to create the TSP response with a TimeStampResponseGenerator.

Example 85 – Creating a TSP Response

```
public static byte[] createTspResponse(
                          PrivateKey tspSigningKey, X509Certificate tspSigningCert, byte[] encRequest)
    throws TSPException, OperatorCreationException, GeneralSecurityException, IOException
{
    AlgorithmIdentifier digestAlgorithm = new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha384);
    DigestCalculatorProvider digProvider = new JcaDigestCalculatorProviderBuilder()
                                                                    .setProvider("BCFIPS").build();

    TimeStampTokenGenerator tsTokenGen = new TimeStampTokenGenerator(
            new JcaSimpleSignerInfoGeneratorBuilder()
                                            .build("SHA384withRSA", tspSigningKey, tspSigningCert),
            digProvider.get(digestAlgorithm),
            new ASN1ObjectIdentifier("1.2"));
    tsTokenGen.addCertificates(new JcaCertStore(Collections.singleton(tspSigningCert)));

    TimeStampResponseGenerator tsRespGen = new TimeStampResponseGenerator(
                                                                tsTokenGen, TSPAlgorithms.ALLOWED);
    return tsRespGen.generate(new TimeStampRequest(encRequest),
                                                        new BigInteger("23"), new Date()).getEncoded();
}
```

In our example we have added our TSP server's certificate to the response, so this example would be suitable for a client requesting the server to include it's certificate. The only piece of real weirdness is the example is the OBJECT IDENTIFIER with the value 1.2 that is passed to the token generator's constructor.

As part of creating a TSP token a TSP server is also meant to specify what policy it created the TSP token under. This is usually server specific and defined using an OBJECT IDENTIFIER issued under an OID branch controlled by the server's owners. The policy oid will normally be associated with an actual document that details how the TSP server will deal with a particular time-stamp request. RFC 3628 provides requirements for a baseline time-stamp policy for a TSP server.

Verifying a TSP response is also very simple code wise. The validate method will also check the criteria given in RFC 3161 to ensure that the time-stamp response is properly created.

Example 86 – Verifying a TSP Response

```
public static boolean verifyTspResponse(X509Certificate tspCertificate, byte[] encResponse)
    throws IOException, TSPException, OperatorCreationException
{
    TimeStampResponse tsResp = new TimeStampResponse(encResponse);
    TimeStampToken tsToken = tsResp.getTimeStampToken();

    tsToken.validate(new JcaSimpleSignerInfoVerifierBuilder()
                                                    .setProvider("BCFIPS").build(tspCertificate));
    return true;
}
```

The last thing to look at with TSP is how to add a TSP response to a CMS signature as they can be used to provide an additional warranty for the validity of one or more of the signers on the signature. In the case of time-stamping a signer, adding a time-stamp involves the addition of an unsigned attribute with the OID label of id-aa-signatureTimeStampToken.

The following example shows how to create the attribute, add it to a signer, and then verify the time-stamp on the signer at a later stage.

Example 87 – Adding a TSP Response to a CMS Signature

```
public static Attribute createTspAttribute(
                            PrivateKey tspSigningKey, X509Certificate tspSignerCert, byte[] data)
    throws GeneralSecurityException, OperatorCreationException, TSPException, IOException
{
    MessageDigest digest = MessageDigest.getInstance("SHA-384", "BCFIPS");
    TimeStampResponse response = new TimeStampResponse(
      Tsp.createTspResponse(tspSigningKey, tspSignerCert, Tsp.createTspRequest(digest.digest(data))));

    TimeStampToken timestampToken = response.getTimeStampToken();

    return new Attribute(
                        PKCSObjectIdentifiers.id_aa_signatureTimeStampToken,
                        new DERSet(timestampToken.toCMSSignedData().toASN1Structure()));
}


public static byte[] createTimeStampedSigner(
                            PrivateKey signingKey, X509Certificate signingCert, byte[] data,
                                            PrivateKey tspSigningKey, X509Certificate tspSignerCert)
    throws OperatorCreationException, GeneralSecurityException, CMSException, TSPException, IOException
{
    CMSSignedData signedData = new CMSSignedData(createSignedObject(signingKey, signingCert, data));

    SignerInformation signer = signedData.getSignerInfos().iterator().next();
    ASN1EncodableVector timestampVector = new ASN1EncodableVector();
    timestampVector.add(createTspAttribute(tspSigningKey, tspSignerCert, signer.getSignature()));

    AttributeTable at = new AttributeTable(timestampVector);

    // create replacement signer
    signer = SignerInformation.replaceUnsignedAttributes(signer, at);

    // create replacement SignerStore
    SignerInformationStore newSignerStore = new SignerInformationStore(signer);

    // replace the signers in the signed data object
    return CMSSignedData.replaceSigners(signedData, newSignerStore).getEncoded();
}
```

```
public static boolean verifyTimeStampedSigner(byte[] cmsSignedData)
    throws OperatorCreationException, GeneralSecurityException, CMSException, IOException, TSPException
{
    CMSSignedData         signedData = new CMSSignedData(cmsSignedData);
    SignerInformation     signer = signedData.getSignerInfos().iterator().next();
    TimeStampToken        tspToken = new TimeStampToken(
            ContentInfo.getInstance(signer.getUnsignedAttributes()
                .get(PKCSObjectIdentifiers.id_aa_signatureTimeStampToken).getAttributeValues()[0]));
    Collection            certCollection = tspToken.getCertificates().getMatches(tspToken.getSID());
    Iterator              certIt = certCollection.iterator();
    X509CertificateHolder cert = (X509CertificateHolder)certIt.next();

    // this method throws an exception if validation fails.
    tspToken.validate(new JcaSimpleSignerInfoVerifierBuilder().setProvider("BCFIPS").build(cert));

    return true;
}
```

OpenPGP

OpenPGP is currently defined in RFC 4880, with several updates, adding ciphers like Camellia (RFC 5581) and Elliptic Curve Cryptography (RFC 6637). The protocol is now over 20 years old and like CMS is widely used.

Dealing with OpenPGP and FIPS can be a little difficult though. The key ring format defined in OpenPGP is not FIPS compliant, and some of the encryption techniques used are not either. It is possible though to construct some OpenPGP messages using FIPS techniques, so a FIPS compliant application of OpenPGP would not make use of PGP keyrings but it might make use of some PGP messaging. In the non-FIPS world you will almost certainly find yourself having to use OpenPGP.

The other thing to keep in mind reading this chapter is that the BC OpenPGP API is a streaming API, unlike the BC CMS API it does not provide an in-memory model as well.

Key Rings

The OpenPGP protocol defines its own key storage format, broadly referred to as a key ring. A basic key ring is a single master key with a collection of sub-keys, in the BC OpenPGP API these are supported by PGPKeyRing and its sub-classes, PGPPublicKeyRing (for public keys) and PGPSecretKeyRing (for private keys). It is also possible to have more than one key ring in a file, in the BC OpenPGP API support for these is provided using the PGPPublicKeyRingCollection and the PGPSecretKeyRingCollection classes.

The OpenPGP protocol requires the master key to be a signing key and a basic set for a usable key ring requires a master key used for signing key and one sub-key used for encryption.

Example 88 – Generating a Basic Key Ring

```
public static byte[][] generateKeyRing(String identity, char[] passphrase)
    throws GeneralSecurityException, PGPException, IOException
{
    KeyPair dsaKp = Dsa.generateKeyPair();
    KeyPair rsaKp = Rsa.generateKeyPair();
    PGPKeyPair dsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.DSA, dsaKp, new Date());
    PGPKeyPair rsaKeyPair = new JcaPGPKeyPair(PGPPublicKey.RSA_ENCRYPT, rsaKp, new Date());
    PGPDigestCalculator sha1Calc = new JcaPGPDigestCalculatorProviderBuilder()
                                                                .build().get(HashAlgorithmTags.SHA1);
    PGPKeyRingGenerator keyRingGen = new PGPKeyRingGenerator(
        PGPSignature.POSITIVE_CERTIFICATION, dsaKeyPair, identity, sha1Calc, null, null,
        new JcaPGPContentSignerBuilder(
                            dsaKeyPair.getPublicKey().getAlgorithm(), HashAlgorithmTags.SHA384),
        new JcePBESecretKeyEncryptorBuilder(PGPEncryptedData.AES_256, sha1Calc)
                            .setProvider("BCFIPS").build(passphrase));

    keyRingGen.addSubKey(rsaKeyPair);

    // create an encoding of the secret key ring
    ByteArrayOutputStream secretOut = new ByteArrayOutputStream();
    keyRingGen.generateSecretKeyRing().encode(secretOut);
    secretOut.close();
```

```
    // create an encoding of the public key ring
    ByteArrayOutputStream publicOut = new ByteArrayOutputStream();
    keyRingGen.generatePublicKeyRing().encode(publicOut);
    publicOut.close();

    return new byte[][] { secretOut.toByteArray(), publicOut.toByteArray() };
}
```

Note that a JcaPGPKeyPair is used to convert the JCA/JCE key pairs into something more OpenPGP friendly. The extra class is required as OpenPGP has its own way of flagging the key type, it stores a data with the object as well, and OpenPGP also associates an 8 byte keyID with a public key. The JcaPGPKeyPair is also very useful where you have to directly convert JCA/JCE keys into OpenPGP, such as where you might be generate FIPS compliant PGP signatures, but not be able to use the PGP key ring format for storing the keys.

OpenPGP Signed Data

The OpenPGP protocol has a number of packet types which are involved in the creation of signed data. The data itself is stored in a literal-data packet, which can also include a file name, or the string "_CONSOLE" indicating no file name is provided. Literal-data might also be in a compressed data packet. For an equivalent to a CMS encapsulated signature the OpenPGP signed data consists of one or more one-pass-signature packets (which contain header information) before the literal-data followed by one or more signature packets. In the case of a detached signature, the signature block will just be one or more signature packets.

In the first example we will look at the encapsulated style. After creating an appropriate output stream for OpenPGP formatted objects, a signature generator is initialized, the one-pass-signature packet is written out to it and then a stream from a PGPLiteralDataGenerator is opened on top of the output stream and the real data is written out, updating the signature along the way. At the end the literal data stream is closed off and the resulting signature is written out as an encoding onto the output stream.

Example 89 – Generating and Verifying a Signed Object

```
public static byte[] createSignedObject(int signingAlg, PGPPrivateKey signingKey, byte[] data)
    throws PGPException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    BCPGOutputStream bcOut = new BCPGOutputStream(bOut);

    PGPSignatureGenerator sGen = new PGPSignatureGenerator(
        new JcaPGPContentSignerBuilder(signingAlg, PGPUtil.SHA384).setProvider("BCFIPS"));

    sGen.init(PGPSignature.BINARY_DOCUMENT, signingKey);

    sGen.generateOnePassVersion(false).encode(bcOut);

    PGPLiteralDataGenerator lGen = new PGPLiteralDataGenerator();

    OutputStream lOut = lGen.open(
        bcOut,
        PGPLiteralData.BINARY,
        "_CONSOLE",
        data.length,
        new Date());
```

```
    for (int i = 0; i != data.length; i++)
    {
        lOut.write(data[i]);
        sGen.update(data[i]);
    }

    lGen.close();

    sGen.generate().encode(bcOut);

    return bOut.toByteArray();
}



public static boolean verifySignedObject(PGPPublicKey verifyingKey, byte[] pgpSignedData)
    throws PGPException, IOException
{
    JcaPGPObjectFactory        pgpFact = new JcaPGPObjectFactory(pgpSignedData);

    PGPOnePassSignatureList    onePassList = (PGPOnePassSignatureList)pgpFact.nextObject();
    PGPOnePassSignature        ops = onePassList.get(0);

    PGPLiteralData literalData = (PGPLiteralData)pgpFact.nextObject();
    InputStream dIn = literalData.getInputStream();

    ops.init(new JcaPGPContentVerifierBuilderProvider().setProvider("BCFIPS"), verifyingKey);

    int ch;
    while ((ch = dIn.read()) >= 0)
    {
        ops.update((byte)ch);
    }

    PGPSignatureList sigList = (PGPSignatureList)pgpFact.nextObject();
    PGPSignature sig = sigList.get(0);

    return ops.verify(sig);
}
```

Verifying is a similar process, the main difference being that a PGPObjectFactory is used to parse the signed data stream and that list objects are returned when parsing the stream where the one-pass-signature packet and the signature packet are found. At the end the PGPSignature object is then passed to the PGPOnePassSignature for verification. In some ways you can think of the PGPSignature object as being the equivalent to the SignerInformation object in CMS. Note that as the verification step is being carried out using a streaming API the literal-data must be fully read before trying to call pgpFact.nextObject() to return the PGPSignatureList. If the literal-data is not fully read the parser will start reading somewhere in the literal-data packet rather than at the next packet boundary. Nothing good will come of that!

Detached signatures just consist of one or more PGPSignature objects, with the data that was signed assumed to have come from somewhere else. In this case no one-pass-signature object is encoded to the stream and the verification step is carried out directly on the PGPSignature object.

Example 90 – Generating and Verifying Detached Signatures

```
public static byte[] createDetachedSignature(int signingAlg, PGPPrivateKey signingKey, byte[] data)
    throws PGPException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    PGPSignatureGenerator sGen = new PGPSignatureGenerator(
        new JcaPGPContentSignerBuilder(signingAlg, PGPUtil.SHA384).setProvider("BCFIPS"));
    sGen.init(PGPSignature.BINARY_DOCUMENT, signingKey);

    for (int i = 0; i != data.length; i++)
    {
        sGen.update(data[i]);
    }

    sGen.generate().encode(bOut);

    return bOut.toByteArray();
}

public static boolean verifyDetachedSignature(
                                    PGPPublicKey verifyingKey, byte[] pgpSignature, byte[] data)
    throws PGPException, IOException
{
    JcaPGPObjectFactory        pgpFact = new JcaPGPObjectFactory(pgpSignature);
    PGPSignatureList           sigList = (PGPSignatureList)pgpFact.nextObject();
    PGPSignature               sig = sigList.get(0);

    sig.init(new JcaPGPContentVerifierBuilderProvider().setProvider("BCFIPS"), verifyingKey);

    sig.update(data);

    return sig.verify();
}
```

OpenPGP Encrypted Data

OpenPGP offers two key encryption methods to add recipients to its encrypted data object – recipients can be either password based or public key based. In terms of public key based OpenPGP allows for key transport style encryption, or the use of key agreement.

In the first example we will look at one way of constructing an encrypted message in OpenPGP using RSA. In this case, as well as the two examples after we need to know the length of the literal data prior to encryption. There is also a variation of the PGPEncryptedDataGenerator.open() which takes a byte buffer as its argument and allows creation of an encrypted data object in a streaming fashion.

Example 91 – OpenPGP Encryption using RSA

```
public static byte[] createRsaEncryptedObject(PGPPublicKey encryptionKey, byte[] data)
    throws PGPException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    PGPLiteralDataGenerator lData = new PGPLiteralDataGenerator();

    OutputStream  pOut = lData.open(bOut,
                                    PGPLiteralData.BINARY,
                                    PGPLiteralData.CONSOLE,
                                    data.length,
                                    new Date());
    pOut.write(data);
```

```
    pOut.close();

    byte[] plainText = bOut.toByteArray();

    ByteArrayOutputStream encOut = new ByteArrayOutputStream();
    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
                                            new JcePGPDataEncryptorBuilder(
                                                                SymmetricKeyAlgorithmTags.AES_256)
                                                .setWithIntegrityPacket(true)
                                                .setSecureRandom(new SecureRandom())
                                                .setProvider("BCFIPS"));

    encGen.addMethod(new JcePublicKeyKeyEncryptionMethodGenerator(encryptionKey)
                                                                        .setProvider("BCFIPS"));

    OutputStream cOut = encGen.open(encOut, plainText.length);
    cOut.write(plainText);
    cOut.close();

    return encOut.toByteArray();
}

public static byte[] extractRsaEncryptedObject(PGPPrivateKey privateKey, byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpEncryptedData);
    PGPEncryptedDataList encList = (PGPEncryptedDataList)pgpFact.nextObject();

    // note: we can only do this because we know we match the first encrypted data object
    PGPPublicKeyEncryptedData encData = (PGPPublicKeyEncryptedData)encList.get(0);

    PublicKeyDataDecryptorFactory dataDecryptorFactory = new JcePublicKeyDataDecryptorFactoryBuilder()
                                                            .setProvider("BCFIPS").build(privateKey);

    InputStream clear = encData.getDataStream(dataDecryptorFactory);
    byte[] literalData = Streams.readAll(clear);
    if (encData.verify())
    {
        PGPObjectFactory litFact = new JcaPGPObjectFactory(literalData);
        PGPLiteralData litData = (PGPLiteralData)litFact.nextObject();
        byte[] data = Streams.readAll(litData.getInputStream());
        return data;
    }

    throw new IllegalStateException("modification check failed");
}
```

You can apply the same code to use ElGamal as the recipient's encryption key as well.

As with signed data, there can be multiple recipients so in the extract method you will notice that the parser returns list objects for the encrypted data. In the case of the example, as we know there is just one recipient, we just grab the first PGPPublicKeyEncryptedData object and use that with our PublicKeyDataDecryptorFactory. PGPPublicKeyEncrytedData objects also store the keyID of the recipient's encryption key – it is accessible via the getKeyID() method. You can make use of the keyID in the situation where there is more than one recipient and you are trying to identify which one is associated with the PGPPrivateKey you have.

Note also that the example includes a call to setWithIntegrityPacket() with the value true. The has two affects on the encryption. The first is it will include a SHA-1 hash at the end of the encrypted data stream, the second is that it will use regular CFB mode (the same as the FIPS one) instead of OpenPGP's variation on it. The SHA-1 hash is what is checked by the call to encData.verify() in the

extractRsaEncryptedObject() method.

As of RFC 6637, OpenPGP also offers public key encryption with key agreement using Elliptic Curves. You can see these examples are the same as those that are presented for RSA keys.

Example 92 – OpenPGP Encryption using Elliptic Curve

```
public static byte[] createKeyAgreeEncryptedObject(PGPPublicKey recipientKey, byte[] data)
    throws PGPException, IOException
{
    // we save the data to be encrypted in PGP format here
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    PGPLiteralDataGenerator lData = new PGPLiteralDataGenerator();

    OutputStream  pOut = lData.open(bOut,
                                    PGPLiteralData.BINARY,
                                    PGPLiteralData.CONSOLE,
                                    data.length,
                                    new Date());
    pOut.write(data);
    pOut.close();

    byte[] plainText = bOut.toByteArray();

    // now we encrypt it
    ByteArrayOutputStream encOut = new ByteArrayOutputStream();
    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
                                          new JcePGPDataEncryptorBuilder(
                                                                  SymmetricKeyAlgorithmTags.AES_256)
                                              .setWithIntegrityPacket(true)
                                              .setSecureRandom(new SecureRandom())
                                              .setProvider("BCFIPS"));

    encGen.addMethod(new JcePublicKeyKeyEncryptionMethodGenerator(recipientKey).setProvider("BCFIPS"));

    OutputStream cOut = encGen.open(encOut, plainText.length);
    cOut.write(plainText);
    cOut.close();

    return encOut.toByteArray();
}

public static byte[] extractKeyAgreeEncryptedObject(
                                          PGPPrivateKey recipientPrivateKey, byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpEncryptedData);
    PGPEncryptedDataList encList = (PGPEncryptedDataList)pgpFact.nextObject();

    PGPPublicKeyEncryptedData encData = (PGPPublicKeyEncryptedData)encList.get(0);

    PublicKeyDataDecryptorFactory dataDecryptorFactory =
                                             new JcePublicKeyDataDecryptorFactoryBuilder()
                                                   .setProvider("BCFIPS").build(recipientPrivateKey);

    InputStream clear = encData.getDataStream(dataDecryptorFactory);
    byte[] literalData = Streams.readAll(clear);
    if (encData.verify())
    {
        PGPObjectFactory litFact = new JcaPGPObjectFactory(literalData);
        PGPLiteralData litData = (PGPLiteralData)litFact.nextObject();
        byte[] data = Streams.readAll(litData.getInputStream());
        return data;
    }

    throw new IllegalStateException("modification check failed");
}
```

Finally OpenPGP also provides the capability for using a password to encrypt data. There is nothing FIPS compliant about the password-to-key scheme here, but you will run into password encrypted OpenPGP data every now and then. Using a password is very similar to a public key in other respects, the only difference is the use of a different key encryption method object – in this case the JcePBEKeyEncryptionMethodGenerator.

Example 93 – OpenPGP Encryption using a Password

```
public static byte[] createPbeEncryptedObject(char[] passwd, byte[] data)
    throws PGPException, IOException
{
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    PGPLiteralDataGenerator lData = new PGPLiteralDataGenerator();

    OutputStream  pOut = lData.open(bOut,
                                    PGPLiteralData.BINARY,
                                    PGPLiteralData.CONSOLE,
                                    data.length,
                                    new Date());
    pOut.write(data);
    pOut.close();

    byte[] plainText = bOut.toByteArray();

    ByteArrayOutputStream encOut = new ByteArrayOutputStream();
    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
                                          new JcePGPDataEncryptorBuilder(
                                                                SymmetricKeyAlgorithmTags.AES_256)
                                              .setWithIntegrityPacket(true)
                                              .setSecureRandom(new SecureRandom())
                                              .setProvider("BCFIPS"));

    encGen.addMethod(new JcePBEKeyEncryptionMethodGenerator(passwd).setProvider("BCFIPS"));

    OutputStream cOut = encGen.open(encOut, plainText.length);
    cOut.write(plainText);
    cOut.close();

    return encOut.toByteArray();
}

public static byte[] extractPbeEncryptedObject(char[] passwd, byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpEncryptedData);
    PGPEncryptedDataList encList = (PGPEncryptedDataList)pgpFact.nextObject();
    PGPPBEEncryptedData encData = (PGPPBEEncryptedData)encList.get(0);

    PBEDataDecryptorFactory dataDecryptorFactory = new JcePBEDataDecryptorFactoryBuilder(
                                       new JcaPGPDigestCalculatorProviderBuilder()
                                                   .setProvider("BCFIPS").build())
                                       .setProvider("BCFIPS").build(passwd);

    InputStream clear = encData.getDataStream(dataDecryptorFactory);
    byte[] literalData = Streams.readAll(clear);
    if (encData.verify())
    {
        PGPObjectFactory litFact = new JcaPGPObjectFactory(literalData);
        PGPLiteralData litData = (PGPLiteralData)litFact.nextObject();
        byte[] data = Streams.readAll(litData.getInputStream());
        return data;
    }

    throw new IllegalStateException("modification check failed");
}
```

The OpenPGP format is very flexible in terms of how it allows messages to be put together so they are easy to nest. In the case of trying to sign and then encrypt data all that is required is to nest a signed data message inside an encrypted one. In this final example we create a signed message which is then encrypted.

Example 94 – Using Signing and Encryption together with OpenPGP

```
public static byte[] createSignedEncryptedObject(
                                    PGPPublicKey encryptionKey, PGPPrivateKey signingKey, byte[] data)
    throws PGPException, IOException
{
    byte[] plainText = Sign.createSignedObject(PublicKeyAlgorithmTags.ECDSA, signingKey, data);

    ByteArrayOutputStream encOut = new ByteArrayOutputStream();

    PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
        new JcePGPDataEncryptorBuilder(SymmetricKeyAlgorithmTags.AES_256)
            .setWithIntegrityPacket(true)
            .setSecureRandom(new SecureRandom())
            .setProvider("BCFIPS"));

    encGen.addMethod(new JcePublicKeyKeyEncryptionMethodGenerator(encryptionKey)
                                                                        .setProvider("BCFIPS"));
    OutputStream cOut = encGen.open(encOut, plainText.length);
    cOut.write(plainText);
    cOut.close();

    return encOut.toByteArray();
}

public static boolean verifySignedEncryptedObject(
                PGPPrivateKey decryptionKey, PGPPublicKey verificationKey, byte[] pgpEncryptedData)
    throws PGPException, IOException
{
    PGPObjectFactory pgpFact = new JcaPGPObjectFactory(pgpEncryptedData);
    PGPEncryptedDataList encList = (PGPEncryptedDataList)pgpFact.nextObject();
    PGPPublicKeyEncryptedData encData = (PGPPublicKeyEncryptedData)encList.get(0);
    PublicKeyDataDecryptorFactory dataDecryptorFactory =
              new JcePublicKeyDataDecryptorFactoryBuilder().setProvider("BCFIPS").build(decryptionKey);

    InputStream clear = encData.getDataStream(dataDecryptorFactory);
    byte[] signedData = Streams.readAll(clear);
    if (encData.verify())
    {
        return Sign.verifySignedObject(verificationKey, signedData);
    }

    throw new IllegalStateException("modification check failed");
}
```

Note that often you will find the signed message is actually nested inside a compressed one as well, and sometimes, if you are accepting messages from multiple parties, it might be sometimes compressed, sometimes not.

TLS

The main high level API for TLS in Java is the JSSE which follows a similar pattern to the JCA/JCE in the respect that it is also provider based. The standard JSSE provider also has a FIPS mode in which case it will guarantee to only get its cryptographic service classes from a specified provider, regardless of what might be configured in terms of available providers or provider priority.

To use the JSSE in FIPS mode, you can either configure a provider for the JSSE in the java.security file or pass the name of the provider, or an actual provider, as a constructor argument to the JSSE provider when you are setting it up at execution time. In the case of the java.security file for the JVM, the BC FIPS Java provider can be configured by including the line:

```
security.provider.X=com.sun.net.ssl.internal.ssl.Provider BCFIPS
```

Where X is the priority of the JSSE provider.

Otherwise you can add the JSSE provider explicitly at execution time by including something like:

```
Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider("BCFIPS"));
```

inside your code.

Utility Methods for the Examples

There is a bit of plumbing required to get a TLS client or server up and running, so in this case we have defined a utility class that contains a few things to make our examples simpler.

```
public class Util
{
    public interface BlockingCallable
        extends Callable
    {
        void await() throws InterruptedException;
    }

    public static class Task
        implements Runnable
    {
        private final Callable callable;
        public Task(Callable callable)
        {
            this.callable = callable;
        }
        public void run()
        {
            try
            {
                callable.call();
            }
            catch (Exception e)
            {
                e.printStackTrace(System.err);
                if (e.getCause() != null)
                {
                    e.getCause().printStackTrace(System.err);
                }
            }
        }
```

```
    }

    public static void runClientAndServer(BlockingCallable server, BlockingCallable client)
        throws InterruptedException
    {
        new Thread(new Util.Task(server)).start();
        server.await();
        new Thread(new Util.Task(client)).start();
        client.await();
    }

    public static void doClientProtocol(
        Socket sock,
        String text)
        throws IOException
    {
        OutputStream out = sock.getOutputStream();
        InputStream in = sock.getInputStream();
        out.write(Strings.toByteArray(text));
        out.write('!');
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            System.out.print((char)ch);
        }
        System.out.println((char)ch);
    }

    public static void doServerProtocol(
        Socket sock,
        String text)
        throws IOException
    {
        OutputStream out = sock.getOutputStream();
        InputStream in = sock.getInputStream();
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            System.out.print((char)ch);
        }
        out.write(Strings.toByteArray(text));
        out.write('!');
        System.out.println((char)ch);
    }
}
```

You will note the utility class makes use of a CountDownLatch – this is not necessarily required by using TLS, in our case it is there so that it is possible to synchronize the client and server examples in the same JVM so they can be run using the Util.runClientAndServer() method. You'll see that the latch is triggered by either the client or the server when it is safe for anyone that may be waiting on them to proceed.

The Basics

The first bridge to cross with TLS is just to get a client running that will recognize a server and a server that can identify itself.

The JSSE uses the KeyManager and TrustManager classes for dealing with these two issues. Essentially a KeyManager provides a connection with any credentials it might need to identify itself with, and a TrustManager provides a connection with a mechanism for validating the credentials

presented the by the other end of the connection.

Both KeyManager and TrustManager classes can be produced by factories which are provided in the same manner as other JCE/JCA objects, by the use of getInstance() methods, rather than constructors. The simplest way to initialize them is with KeyStore objects. The standard TrustManager is based on the use of the PKIX certificate path API as well, so it is also possible to initialize the JSSE PKIX TrustManager using PKIXBuilderParameters via the init() method taking ManagerFactoryParameters.

In our first two examples we look at the basic client which needs to identify a server, and a basic server which needs to be able to identify itself to the client.

For the client case, we need a TrustManager to be associated with the connection so we can identify the server. In this case we initialize a TrustManagerFactory object using a KeyStore which contains the server certificate and then initialize an SSLContext object with the TrustManagers produced by the factory, and a SecureRandom (in this case the FIPS provider default one).

Example 95 – A Basic TLS Client

```
public class SimpleClient
    implements Util.BlockingCallable
{
    private final KeyStore trustStore;
    private final CountDownLatch latch;

    public SimpleClient(KeyStore trustStore)
    {
        this.trustStore = trustStore;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance("SunX509");
        trustMgrFact.init(trustStore);

        SSLContext clientContext = SSLContext.getInstance("TLS");
        clientContext.init(null, trustMgrFact.getTrustManagers(),
                                                SecureRandom.getInstance("DEFAULT", "BCFIPS"));

        SSLSocketFactory fact = clientContext.getSocketFactory();
        SSLSocket cSock = (SSLSocket)fact.createSocket(HOST, PORT_NO);
        Util.doClientProtocol(cSock, "Hello");

        // signal we are finished
        latch.countDown();
        return null;
    }

    public void await()
        throws InterruptedException
    {
        latch.await();
    }
}
```

For the server case, the server needs to be able to identify itself, so we need a KeyManager to be associated with the connection so we can prove we are what we say we are to a client. So in this case we initialize a KeyManagerFactory with a KeyStore and the password needed to recover the private

key and then we use the KeyManager produced by the factory to initialize an SSLContext object.

Example 96 – A Basic TLS Server

```
public class SimpleServer
    implements Util.BlockingCallable
{
    private final KeyStore serverStore;
    private final char[] keyPass;
    private final CountDownLatch latch;

    public SimpleServer(KeyStore serverStore, char[] keyPass)
    {
        this.serverStore = serverStore;
        this.keyPass = keyPass;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
        keyMgrFact.init(serverStore, keyPass);

        SSLContext serverContext = SSLContext.getInstance("TLS");
        serverContext.init(
                 keyMgrFact.getKeyManagers(), null, SecureRandom.getInstance("DEFAULT", "BCFIPS"));

        SSLServerSocketFactory fact = serverContext.getServerSocketFactory();
        SSLServerSocket sSock = (SSLServerSocket)fact.createServerSocket(PORT_NO);

        // signal we are up and running
        latch.countDown();

        SSLSocket sslSock = (SSLSocket)sSock.accept();
        Util.doServerProtocol(sslSock, "World");

        return null;
    }

    public void await()
        throws InterruptedException
    {
        latch.await();
    }
}
```

You can run these two in the same JVM using the Util.runClientAndServer() method. The use of the CountDownLatches will keep everything in sync. If all goes well you will be greeted with the strings “Hello!” and “World!” and then the process will exit.

Client Authentication

So can we follow the same principals to implement client authentication as well? The answer is yes.

The first thing we need to do is provide the client with a means of identifying itself. As we have already seen in the JSSE that involves the use of a KeyManager, so the first difference you will notice between this example and the previous client example is that there is a client KeyStore and a client key password handed to the constructor. The second difference you will notice is that a KeyManagerFactory is created using these and the resulting KeyManager is passed to the SSLContext

object.

The result of this is a context which can be used to create an SSLSocketFactory which is capable of both identifying itself to the other end point, as well as having some means of identifying the other end point.

Example 97 – A TLS Client with Client Authentication

```
public class ClientAuthClient
    implements Util.BlockingCallable
{
    private final KeyStore trustStore;
    private final KeyStore clientStore;
    private final char[] clientKeyPass;
    private final CountDownLatch latch;

    public ClientAuthClient(KeyStore trustStore, KeyStore clientStore, char[] clientKeyPass)
    {
        this.trustStore = trustStore;
        this.clientStore = clientStore;
        this.clientKeyPass = clientKeyPass;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance("SunX509");
        trustMgrFact.init(trustStore);

        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
        keyMgrFact.init(clientStore, clientKeyPass);

        SSLContext clientContext = SSLContext.getInstance("TLS");
        clientContext.init(keyMgrFact.getKeyManagers(),
                     trustMgrFact.getTrustManagers(), SecureRandom.getInstance("DEFAULT", "BCFIPS"));

        SSLSocketFactory fact = clientContext.getSocketFactory();
        SSLSocket cSock = (SSLSocket)fact.createSocket(HOST, PORT_NO);
        Util.doClientProtocol(cSock, "Hello");

        // signal we are finished
        latch.countDown();

        return null;
    }

    public void await()
        throws InterruptedException
    {
        latch.await();
    }
}
```

The server was already identifying itself, but was not previously trying to identify a client, so as you would expect by now, we need to add the use of a TrustManager to the server side, and we also need to tell the server that we expect the client to authenticate itself.

As the client did originally, we now pass the server a KeyStore that serves as a trust store for the credentials presented by the other end point (in this case the client), and we add the use of a TrustManagerFactory which is used to produce TrustManagers for the SSLContext associated with the

connections the server is creating.

Finally, we tell the server to expect the client credentials and we do this by calling SSLServerSocket.setNeedClientAuth() with the parameter value true.

Example 98 – A TLS Server with Client Authentication

```
public class ClientAuthServer
    implements Util.BlockingCallable
{
    private final KeyStore serverStore;
    private final char[] keyPass;
    private final KeyStore trustStore;
    private final CountDownLatch latch;

    public ClientAuthServer(KeyStore serverStore, char[] keyPass, KeyStore trustStore)
    {
        this.serverStore = serverStore;
        this.keyPass = keyPass;
        this.trustStore = trustStore;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
        keyMgrFact.init(serverStore, keyPass);

        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance("SunX509");
        trustMgrFact.init(trustStore);

        SSLContext serverContext = SSLContext.getInstance("TLS");
        serverContext.init(keyMgrFact.getKeyManagers(),
                    trustMgrFact.getTrustManagers(), SecureRandom.getInstance("DEFAULT", "BCFIPS"));
        SSLServerSocketFactory fact = serverContext.getServerSocketFactory();

        SSLServerSocket sSock = (SSLServerSocket)fact.createServerSocket(PORT_NO);
        sSock.setNeedClientAuth(true);

        // signal we are up and running
        latch.countDown();

        SSLSocket sslSock = (SSLSocket)sSock.accept();
        Util.doServerProtocol(sslSock, "World");
        sslSock.close();

        return null;
    }

    public void await()
        throws InterruptedException
    {
        latch.await();
    }
}
```

It is worth noting the setNeedClientAuth() method has the effect of making client authentication compulsory. Sometimes this is not desirable and you may have some services which are available to all clients, even anonymous ones, and some which are restricted according to who the client is. In that case you can use SSLServerSocker.setWantClientAuth() to tell the server socket that it should make use of client authentication where it is available, but require all clients to provide it.

A common way to use TLS from a Java client is via the URL class in java.net using URL.openConnection(). If we want to make use of one of these we also need a way of telling the underlying connection to make use of our TLS settings. Doing this normally requires two things: configuring a specific socket factory to use and providing a mechanism for verifying the hostname presented to the client. We have already seen how to create an SSLSocketFactory from an SSLContext configured for client authentication and happily there is a method on HttpsURLConnection (the object type returned by URL.openConnection() when the HTTPS protocol is specified which allows us to set the SSLSocketFactory to use – setSSLSocketFactory(). Hostname verification is a little bit different and requires us to provide our own implementation of the HostnameVerifier interface. A simple one is provided in a private class at the bottom of the example.

Example 99 – TLS Authenticated Client Using HttpsURLConnection

```
public class HttpsAuthClient
    implements Util.BlockingCallable
{
    private final KeyStore trustStore;
    private final KeyStore clientStore;
    private final char[] clientKeyPass;
    private final CountDownLatch latch;

    public HttpsAuthClient(KeyStore trustStore, KeyStore clientStore, char[] clientKeyPass)
    {
        this.trustStore = trustStore;
        this.clientStore = clientStore;
        this.clientKeyPass = clientKeyPass;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance("SunX509");
        trustMgrFact.init(trustStore);

        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
        keyMgrFact.init(clientStore, clientKeyPass);

        SSLContext clientContext = SSLContext.getInstance("TLS");
        clientContext.init(keyMgrFact.getKeyManagers(),
                    trustMgrFact.getTrustManagers(), SecureRandom.getInstance("DEFAULT", "BCFIPS"));
        SSLSocketFactory fact = clientContext.getSocketFactory();

        URL url = new URL("https://"+ HOST + ":" + PORT_NO);
        HttpsURLConnection httpsConnection = (HttpsURLConnection)url.openConnection();
        httpsConnection.setSSLSocketFactory(fact);
        httpsConnection.setHostnameVerifier(new LocalHostVerifier());
        httpsConnection.connect();

        InputStream in = httpsConnection.getInputStream();
        int ch;
        while ((ch = in.read()) >= 0)
        {
            System.out.print((char)ch);
        }

        // signal we are finished
        latch.countDown();

        return null;
    }
```

```
    public void await()
        throws InterruptedException
    {
        latch.await();
    }

    private class LocalHostVerifier
        implements HostnameVerifier
    {
        public boolean verify(String hostName, SSLSession session)
        {
            try
            {
                X500Principal hostID = (X500Principal)session.getPeerPrincipal();
                return hostName.equals("localhost") && hostID.getName().equals("CN=Issuer CA");
            }
            catch (Exception e)
            {
                return false;
            }
        }
    }
}
```

As you can see things only start looking different when we get to the process of open the URL and then configuring the HttpsURLConnection. As we are now using the URL handler we are no longer dealing with the socket directly, but getting back an InputStream.

Also as we are talking HTTP things are not quite the same as before for the server either – there is no change to the TLS related code but we need some code that will send back a HTTP response.

Example 100 – TLS Server for Client Using HttpsURLConnection

```
public static class HttpsAuthServer
    implements Util.BlockingCallable
{
    private final KeyStore serverStore;
    private final char[] keyPass;
    private final KeyStore trustStore;
    private final CountDownLatch latch;

    HttpsAuthServer(KeyStore serverStore, char[] keyPass, KeyStore trustStore)
    {
        this.serverStore = serverStore;
        this.keyPass = keyPass;
        this.trustStore = trustStore;
        this.latch = new CountDownLatch(1);
    }

    public Object call()
        throws Exception
    {
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance("SunX509");
        keyMgrFact.init(serverStore, keyPass);

        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance("SunX509");
        trustMgrFact.init(trustStore);

        SSLContext serverContext = SSLContext.getInstance("TLS");
        serverContext.init(keyMgrFact.getKeyManagers(), trustMgrFact.getTrustManagers(),
                                            SecureRandom.getInstance("DEFAULT", "BCFIPS"));

        SSLServerSocketFactory fact = serverContext.getServerSocketFactory();
        SSLServerSocket sSock = (SSLServerSocket)fact.createServerSocket(PORT_NO);
        sSock.setNeedClientAuth(true);
```

```
        // signal we are up and running
        latch.countDown();

        SSLSocket sslSock = (SSLSocket)sSock.accept();

        readRequest(sslSock.getInputStream());
        sendResponse(sslSock.getOutputStream());

        sslSock.close();

        return null;
    }

    public void await()
        throws InterruptedException
    {
        latch.await();
    }

    private static String readLine(InputStream in)
        throws IOException
    {
        StringBuilder bld = new StringBuilder();
        int ch;
        while ((ch = in.read()) >= 0 && (ch != '\n'))
        {
            if (ch != '\r')
                bld.append((char)ch);
        }
        return bld.toString();
    }

    private static void readRequest(
        InputStream in)
        throws IOException
    {
        String line = readLine(in);
        while (line.length() != 0)
        {
            System.out.println("Request: " + line);
            line = readLine(in);
        }
    }

    private static void sendResponse(
        OutputStream out)
    {
        PrintWriter pWrt = new PrintWriter(new OutputStreamWriter(out));
        pWrt.print("HTTP/1.1 200 OK\r\n");
        pWrt.print("Content-Type: text/plain\r\n");
        pWrt.print("\r\n");
        pWrt.print("Hello World!\r\n");
        pWrt.flush();
    }
}
```

You can see how there has been a slight change in the way the input/output is done on the SSLSocket created in the server. While this is not a cryptography related change, a failure to take it into account will result in exceptions and a reminder that even if the cryptography is correct, the application of it might not be.

I hope all this has been some help!

Appendix A – An Introduction to the BC ASN.1 API

The Bouncy Castle APIs have had their own ASN.1 library since there original inception. The core classes are in the org.bouncycastle.asn1 package and these are common between both the BC FIPS Java API and the regular BC Java API. Originally the ASN.1 package was just for processing small objects like signature and digital certificates and relied mainly on the use of fixed length encoding, primarily DER. With the introduction of the streaming API for CMS about 7 years ago, it was necessary to properly follow BER.

The idea of this appendix is to give a quick insight into ASN.1 and how it relates to what we normally do in the Bouncy Castle APIs. The appendix is not really about ASN.1 per se, and we recommend looking at “A Layman's Guide to a Subset of ASN.1, BER, and DER” by Burton S. Kaliski Jr. if you wish to brush up, but the appendix will give you an idea how the parser and encoders provided in Bouncy Castle see the world and why.

ASN.1 Encoding

The BC APIs provide direct support for encoding using DER (Distinguished Encoding Rules) and BER (Basic Encoding Rules). DER encoding is a subset of BER.

BER also allows for both indefinite-length and definite-length encodings. Indefinite-length encodings are those where the length of the final output is not known before hand. With a definite-length encoding you will know to the byte how much output there will be. Indefinite-length encodings are useful for large objects – the streaming CMS API relies on these – as you they also let you process more data than you can hold in memory. An example of an indefinite length object is the BEROctetString, which will encode as a constructed OCTET STRING – basically a stream of definite-length octet strings representing parts of the actual data followed by the end marker of two zero value bytes.

DER on the other hand is always definite-length and, more importantly, has very specific guidelines to ensure that any two DER encodings of the same thing will be equal. One of the interesting features of this is that ASN.1 SET structures are sorted in DER (remember SETs are unordered, do not ever rely on the order of things in a SET as one day it will lead to disappointment). Another interesting feature is that where the DEFAULT value is used in an ASN.1 definition, if a field in the definition has the default value it is left out. Finally constructed primitive types, such as a constructed OCTET STRING, are written out as their definite-length equivalents. These things need to be kept in mind and understood as one of the uses of DER is to create input for signature, hash, and MAC algorithms – the idea being that as only one encoding can be produced anyone should be able to reproduce the signature, hash, or MAC value by following the DER rules.

The Bouncy Castle API manages this distinction partly by respecting DER encoding for definite-length objects in the objects encoding method, and, where an alternative encoding might exist, making use of the type of the OutputStream the object is being written to. If a DEROutputStream is used whenever

there is an option of encoding something as DER or simply definite-length (part of BER), the extra work will be done to encode the object as DER. A DLOutputStream will always try to produce definite-length objects and a BEROutputStream or an ASN1OutputStream will produce both DER, definite-length, and BER depending on what object it is writing out.

It is worth giving some consideration to what's going to be done with the data when you are considering whether or not to accept BER data as well, some standards such as PKCS#12 allow BER encoding, even though the files involved will normally be small. If you are relying on data being small and you are dealing with ASN.1 objects from third parties, it can be a good idea to either enforce direct-length encodings on input (at least for the outermost object) or enforce a maximum size for a blob if it is indefinite-length encoded. Remember indefinite-length really does mean indefinite-length, if you have an application that really does require loading an indefinite-length object into memory, a well crafted or badly broken object will be capable of running your application out of memory if no defenses are in place.

Compatibility Issues

The most common compatibility issue you are likely to run into between the BC ASN.1 package and others is that a lot of people do not actually implement BER and can only handle definite-length data. In a case like this you can re-encode things using the DLOutputStream – just keep in mind that on occasion encodings are nested inside OCTET STRINGs that are inside other ASN.1 objects. Depending on what the other party is capable of dealing with you may need to re-encode some nested objects as part of the definite-length conversion process.

Another compatibility issue can arise with DER itself – there's a bit to remember and occasionally people forget things. This leads us to the happy place where using DER correctly means a signature fails to validate. As an example it is possible to extend the SignerInformation class in the CMS API to use definite-length rather than DER encoding for encoding signed attributes. If you are getting “mysterious” failures it is worth checking to see whether or not something like this is happening. How you get to deal with it will depend on the circumstances, if it is a well meaning but otherwise incorrect government agency you will probably need to use the work around, alternately you may be able to get the other party to fix the issue.

Using the Streaming API

Classes like DERSequence, DERSet, DLSequence, DLSet, BERSequence, and BERSet all use an in-memory model to hold and encode objects. You will also notice classes with names like BERSequenceGenerator, BEROctetStringGenerator, BERSequenceParser, and BEROctetStringParser. The second group of classes use a streaming model to process the data they are either writing (the Generator classes), or the reading (the Parser classes).

There's really only one thing to remember here. Streaming means you cannot go back so suddenly the order in which things are either accessed or written becomes very important. This is not often a cause

of error when creating objects for writing, but it is a tripping point for reading particularly where a piece of previously memory bound code is using ASN1Sequence.getObjectAt() and it gets missed that one of the getObjectAt() calls was with an index that was out of order, or is fetching the same index twice.

A Handy Hint

Finally, a look at the API will show there are a lot of classes with getInstance() methods. It is better to make use of these rather than using casts when going from something like an ASN1Encodable or an ASN1Primitive to an actual ASN.1 type, or high level ASN.1 type named after one of the structures in an ASN.1 based standard such as PKIX. In the early days a persistent issue, even within the BC project, was that an object that was expecting something like an ASN1Sequence would be presented with a higher level object that would encode to an ASN1Sequence, but whose Java type was not ASN1Sequence. The issue was always related to the situation where sometimes the code would be dealing with objects produced in the same JVM (so they would often have a higher level type) or the code would be dealing with objects which had been recently deserialized, in which case they would have a primitive type like ASN1Sequence. Unfortunately it turns out that trying to deal with this correctly just by anticipating and casting and hoping for the best is a path to disaster, occasionally a disaster that might suddenly show up in a production environment which had previously appeared to be fine. So use getInstance().

Bibliography

BC-FJA (Bouncy Castle FIPS Java API) “User Guide”, Legion of the Bouncy Castle Inc., August 2016.

BC-FJA (Bouncy Castle FIPS Java API) “Non-Proprietary FIPS 140-2 Cryptographic Module Security Policy”, Legion of the Bouncy Castle Inc., August 2016.

FIPS PUB 180-4 “Secure Hash Standard (SHS)”, NIST, August 2015.

FIPS PUB 186-4 “Digital Signature Standard (DSS)”, NIST, July 2013.

FIPS PUB 197 “Advanced Encryption Standard (AES)”, NIST, November 2001.

FIPS PUB 198-1 “The Keyed-Hash Message Authentication Code (HMAC)”, NIST, July 2008.

FIPS PUB 202 “SHA-3 Standard: Permutation-Bases Hash and Extendable-Output Functions”, NIST, August 2015.

NIST SP 800-38A “Recommendations for Block Cipher Modes of Operation”, by M. Dworkin, NIST, December 2001.

NIST SP 800-38A Addendum “Recommendations for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”, by M. Dworkin, NIST, December 2010.

NIST SP 800-38B “Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication”, by M. Dworkin, NIST, May 2005.

NIST SP 800-38C “Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality”, by M. Dworkin, NIST, May 2004.

NIST SP 800-38D “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM and GMAC)”, by M. Dworkin, NIST, May 2007.

NIST SP 800-38F “Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping”, by M. Dworkin, NIST, December 2012.

NIST SP 800-52 Revision 1 “Guidelines for the Selection Configuration, and Use of Transport Layer Security (TLS) Implementations”, by T. Polk, K. McKay and S. Chokhani, NIST, April 2014.

NIST SP 800-56A Revision 2 “Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography”, by E. Barker, L. Chen, et al., NIST, May 2013.

NIST SP 800-56B “Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography” by E. Barker, L. Chen, et al., NIST, August 2009

NIST SP 800-67 Revision 1 “Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher”, by W. Barker and E. Barker, NIST, January 2012.

NIST SP 800-90A Revision 1 “Recommendation for Random Number Generation Using Deterministic Random Bit Generators”, by E. Barker and J. Kelsey, NIST, June 2015.

NIST SP 800-132 “Recommendation for Password-Based Key Derivation – Part 1: Storage Applications”, by M. Turan, E. Barker, et al., NIST, December 2012.

RFC 1421 “Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures”, by J. Linn, IETF, February 1993.

RFC 2040 “The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms”, by R. Baldwin and R. Rivest. IETF, October 1996.

RFC 2898 “PKCS #5: Password-Based Cryptography Specification Version 2.0”, by B Kaliski, RSA Laboratories, September 2000.

RFC 2986 “PKCS #10: Certification Request Syntax Specification Version 1.7”, by M. Nystrom and B. Kaliski, RSA Security, November 2000.

RFC 3161 “Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)”, by C. Adams, P. Cain, et al., IETF, August 2001.

RFC 3447 “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1”, by J. Jonsson and B Kaliski, RSA Laboratories, February 2003.

RFC 3628 “Policy Requirements for Time-Stamping Authorities (TSAs)”, by D. Pinkas, N. Pope, and J. Ross, IETF, November 2003.

RFC 4211 “Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)”, by J. Schaad, IETF, September 2005.

RFC 4880 “Open PGP Message Format”, by J. Callas, L. Donnerhacke, et al., IETF, November 2007.

RFC 5246 “The Transport Layer Security (TLS) Protocol Version 1.2”, by T. Dierks and E. Rescorla, IETF, August 2008.

RFC 5280 “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, by D. Cooper, S. Santesson, et al., IETF, May 2008.

RFC 5581 “The Camellia Cipher in OpenPGP”, by D. Shaw, IETF, June 2009.

RFC 5652 “Cryptographic Message Syntax (CMS)”, by R. Housley, IETF, September 2009.

RFC 5751 “Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification”, by B. Ramsdell and S. Turner, IETF, January 2010.

RFC 5990 “Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)”, by J. Randall, B. Kaliski, et al., IETF, September 2010.

RFC 6637 “Elliptic Curve Cryptography (ECC) in OpenPGP, by A. Jivsov, IETF, June 2012.

RFC 6960 “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP”, by S. Santesson, M. Myers, et al., IETF, June 2013.

RFC 6979 “Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital

Signature Algorithm (ECDSA)”, by T. Pornin, IETF, August 2013.

RFC 7292 “PKCS #12: Personal Information Exchange Syntax v1.1”, by K. Moriarty, M. Nystrom, et al., IETF, July 2014.

RFC 7468 “Textual Encodings of PKIX, PKCS, and CMS Structures”, by S. Josefsson and S. Leonard, IETF, April 2015.