The Cryptoworkshop Guide to Java Cryptography and the Bouncy Castle APIs

Acknowledgements

While it may seem a little premature to include an acknowledgements section in a preliminary draft, the truth is there have already been a number of contributions to this work.

The idea behind this project was to be able to generate documentation which included validated source code for the examples, so you'll see some source code in the book itself, which perhaps suprisingly is the same source code that appears in the example files. As you will appreciate this is not something currently possible with a conventional publishing system (TeX aside), so doing this has required writing some software on our part as well as finding a clean way of generating the output PDF. To this end we would like to acknowledge the contribution of iText Software Corp (http://www.itextpdf.com) in making an iText license available to us. Without that we doubt this idea would have made it off the table napkin.

Also to the reviewers of this document so far, thanks!

1. The Bouncy Castle Provider

1.1. Installing the Provider

The Bouncy Castle Provider can be installed either at run time, or by using static registration via the java.security file.

To install the provider at run time just add:

```
Security.addProvider(new BouncyCastleProvider());
```

To install the provider using static registration you need to add an entry to the java.security properties file (found in $JAVA_HOME/jre/lib/security/java.security, where $JAVA_HOME is the location of your JDK/JRE distribution). You'll find detailed instructions in the file but basically it comes down to adding a line:

```
security.provider.<n>=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Where <n> is the preference you want the provider at (1 being the most preferred).

Where you put the jar is up to mostly up to you, although with jdk1.3 and jdk1.4 the best (and in some cases only) place to have it is in $JAVA_HOME/jre/lib/ext. Note: under Windows there will normally be a JRE and a JDK install of Java if you think you have installed it correctly and it still doesn't work chances are you have added the provider to the installation not being used.

Note: with JDK 1.4 and later you will need to have installed the unrestricted policy files to take full advantage of the provider. If you do not install the policy files you are likely to get something like the following:

```
java.lang.SecurityException: Unsupported keysize or algorithm parameters
            at javax.crypto.Cipher.init(DashoA6275)
```

The policy files can be found at the same place you downloaded the JDK.

1.2. Bouncy Castle Provider Configuration

The Bouncy Castle provider has a number of parameters which can be used to set default parameters. The configuration parameters help cover issues like Elliptic Curve's implicitly CA, and the setting of default parameters for algorithms such as Diffie-Hellman. These can be set using the setParameter() method on the ConfiguarbleProvider interface which the Bouncy Castle provider class implements.

The parameter set is defined in the ConfigurableProvider interface. The current parameter set includes THREAD_LOCAL_EC_IMPLICITLY_CA, EC_IMPLICITLY_CA, THREAD_LOCAL_DH_DEFAULT_PARAMS, and DH_DEFAULT_PARAMS. As the names suggest the thread local versions will allow the setting of specific parameter values to specfic threads where thread local storage is available, otherwise the parameter sets a VM wide default.

The following example shows the setting of some standard Diffie-Hellman parameter values for the Bouncy Castle provider.

```
        ConfigurableProvider prov = (ConfigurableProvider)Security.getProvider("BC");

        DHParameterSpec dhSpec512 = new DHParameterSpec(
            new
BigInteger("fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df
63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17", 16),
            new
BigInteger("678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd73da1
79069b32e2935630e1c2062354d0da20a6c416e50be794ca4", 16),
            384);

        DHParameterSpec dhSpec768 = new DHParameterSpec(
```

```
            new
BigInteger("e9e642599d355f37c97ffd3567120b8e25c9cd43e927b3a9670fbec5d890141922d2c3b3ad24800
93799869d1e846aab49fab0ad26d2ce6a22219d470bce7d777d4a21fbe9c270b57f607002f3cef8393694cf45ee
3688c11a8c56ab127a3daf", 16),
            new
BigInteger("30470ad5a005fb14ce2d9dcd87e38bc7d1b1c5facbaecbe95f190aa7a31d23c4dbbcbe061745444
01a5b2c020965d8c2bd2171d3668445771f74ba084d2029d83c1c158547f3a9f1a2715be23d51ae4d3e5a1f6a70
64f316933a346d3f529252", 16),
            384);

        DHParameterSpec dhSpec1024 = new DHParameterSpec(
                    new
BigInteger("f7e1a085d69b3ddecbbcab5c36b857b97994afbbfa3aea82f9574c0b3d0782675159578ebad4594
fe67107108180b449167123e84c281613b7cf09328cc8a6e13c167a8b547c8d28e0a3ae1e2bb3a675916ea37f0b
fa213562f1fb627a01243bcca4f1bea8519089a883dfe15ae59f06928b665e807b552564014c3bfecf492a",
16),
                    new
BigInteger("fd7f53811d75122952df4a9c2eece4e7f611b7523cef4400c31e3f80b6512669455d402251fb593
d8d58fabfc5f5ba30f6cb9b556cd7813b801d346ff26660b76b9950a5a49f9fe8047b1022c24fbba9d7feb7c61b
f83b57e7c6a8a6150f04fb83f6d3c51ec3023554135a169132f675f3ae2b61d72aeff22203199dd14801c7",
16),
                    512);

        prov.setParameter(ConfigurableProvider.DH_DEFAULT_PARAMS, new DHParameterSpec[] {
dhSpec512, dhSpec768, dhSpec1024 });
```

In the above example, future calls to key generators doing Diffie-Hellman key pair generation will use the provider parameter values, rather than generating new parameter values as required (which, while it is a one off cost per key size, is rather expensive).

An alternative way to prevent long parameter generation cycles is configure the KeyPairGenerator up front via the initialize() method.

1.3. Things to Watch Out For

Problems can arise if you change the order of providers in the java.security file, rather than just adding the Bouncy Castle provider to the end. This especially applies to the provider sitting in position 1. If you do change the order of providers make sure you test thoroughly with the JVMs you need to deploy on as you may find what will work on one JVM will not work on another.

The JCE is normally always picked up by the sytem class loader, the same is not necessarily true if the provider jar is not in the system class path (i.e. lib/ext). In some container based environments, such as Tomcat, you may find you get apparently impossible class cast exceptions as the provider classes may end up being picked up by a class loader the system class loader does not trust. If this does happen there are only two options, the provider jar either needs to be in the container's most trusted class path (so any war file always gets the class from the same class loader) or the provider jar needs to be in the lib/ext area for the JVM that the container is running on.

If a strange performance issue, where your application suddenly pauses for 30 seconds on startup appears, it is likely that the culprit is the generation of DSA or Diffie-Hellman key parameters. In this case you can configure the provider to have default parameter values or pre-configure the KeyPairGenerator you are using. If this sounds like you, see the section in this chapter on Bouncy Castle provider configuration.

2. Basic Operators and API Organisation

The Bouncy Castle API for generating and processing certificates, certification requests, and CMS, SMIME and CRMF messages are based around abstracting out the cryptographic operations from the classes that generate or acts as containers for the actual certificates and messages.

The cryptographic operators are defined by a series of interfaces and implementations of these interfaces can then be plugged into message handlers and generators as required. This allows the Bouncy Castle APIs to be used on a range of virtual machines and makes it possible to create operators not only using the JCA/JCE, but also the Bouncy Castle lightweight APIs as well as custom APIs that a particular environment might present. Understanding these operators and how they are used is the quickest way of learning how to deal with the Bouncy Castle APIs.

Another useful rule to follow comes from the use of "bc" and "jcajce" in final pakcage names. The use of jcajce signifies that the classes in the package are designed to be used with the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE). Other packages have a final name of bc. When you see this you are looking at a package that is specifically designed to be used with the Bouncy Castle lightweight APIs. So if you are looking for something designed to specifically handle a JCA/JCE object in a given package hierarchy you will probably find what you are looking for in the jcajce package under that hierarchy, on the other hand if you are looking for something specific to the lightweight API look in the bc packages for the hierarchy. If you are looking to develop a custom operator of your own the bc package is also the best place to look for examples as it has to deal with direct implementations.

2.1. Digest Calculators

The DigestCalculator interface is one of the simplest to implement and it is typical of the operator interfaces. A method, getAlgorithmIdentifier(), is implemented to provide an ASN.1 description of the algorithm the calculator implements. Another method, getOutputStream(), returns an OutputStream which can be written to to feed data into the calculator, and a third method, getDigest() is used to return the calculated result.

For example, a simple implementation of a calculator to create a SHA-1 digests might look as follows:

```
package cwguide;

import java.io.ByteArrayOutputStream;
import java.io.OutputStream;

import org.bouncycastle.asn1.oiw.OIWObjectIdentifiers;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.crypto.Digest;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.operator.DigestCalculator;

public class SHA1DigestCalculator
    implements DigestCalculator
{
    private ByteArrayOutputStream bOut = new ByteArrayOutputStream();

    public AlgorithmIdentifier getAlgorithmIdentifier()
    {
        return new AlgorithmIdentifier(OIWObjectIdentifiers.idSHA1);
    }

    public OutputStream getOutputStream()
    {
        return bOut;
    }

    public byte[] getDigest()
    {
        byte[] bytes = bOut.toByteArray();

        bOut.reset();

        Digest sha1 = new SHA1Digest();

        sha1.update(bytes, 0, bytes.length);

        byte[] digest = new byte[sha1.getDigestSize()];

        sha1.doFinal(digest, 0);

        return digest;
    }
}
```

Probably the only "mystery" in trying to implement one of these is the body of the getAlgorithmIdentifier() method. With a little bit of background it is actually quite straightforward. An algorithm identifier consists of an object identifier which uniquely identifies the algorithm being used and it includes an optional parameters field which is used to carry any public parameters required to configure the implementation of the algorithm. While this values in some way appear to be magic, you will find them in them defined in whatever standard it might be you are trying to conform to.

As it happens Bouncy Castle has providers and builders for DigestCalculators that conform to the standards we have actually had to deal with. These exist both for the JCA and the lightwieght API. The general provider for the lightweight API is the BcDigestCalculatorProvider, and a JCA based one can be created using the JcaDigestCalculatorProviderBuilder class.

2.2. Content Signers

The ContentSigner interface is directly comparable to the DigestCalculator interface, and similar rules apply.

Here is the interface:

```
package org.bouncycastle.operator;

import java.io.OutputStream;

import org.bouncycastle.asn1.x509.AlgorithmIdentifier;

public interface ContentSigner
{
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * a signature. Use org.bouncycastle.util.io.TeeOutputStream if you want to accumulate
     * the data on the fly as well.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * Returns a signature based on the current data written to the stream, since the
     * start or the last call to getSignature().
     *
     * @return bytes representing the signature.
     */
    byte[] getSignature();
}
```

Construction of ContentSigner implementations is slightly more complex than the situation with DigestSigner, the reason being that unlike a message digest a signature algorithm always requires the use of a private key as well as a message digest.

For the general case Bouncy Castle provides builder objects. In a situation where you're using the JCA a builder is provided for creating ContentSigner implementations which make use of the JCA. The class that does this is the JcaContentSignerBuilder and in the case where we wanted to create a SHA1withRSA signer, we might create s signer sigGen as follows:

```
    sigGen = new JcaContentSignerBuilder("SHA1withRSA").setProvider("BC").build(privKey);
```

In the above case privKey is simply the PrivateKey object representing the private key we are signing with.

Of course we may actually be wanting to do this using the Bouncy Castle lightweight API. This is slightly more involved as it requires providing algorithm identifiers for the signature encryption algorithm and the signature digest algorithm. For an RSA based algorithm we can use the BcRSAContentSignerBuilder. In this case the code for creating a SHA1withRSA signature might look as follows:

```
        AlgorithmIdentifier sigAlgId = new
DefaultSignatureAlgorithmIdentifierFinder().find("SHA1withRSA");
        AlgorithmIdentifier digAlgId = new
```

```
DefaultDigestAlgorithmIdentifierFinder().find(sigAlgId);

        sigGen = new BcRSAContentSignerBuilder(sigAlgId, digAlgId).build(lwPrivKey);
```

where lwPrivKey is an RSA private key as described in the lightweight API.

2.3. Content Verifiers

Signatures generated by ContentSigner classes are verified using objects implementing the ContentVerifier interface. The ContentVerifier interface is defined along the same lines as the ContentSigner interface.

```
package org.bouncycastle.operator;

import java.io.OutputStream;

import org.bouncycastle.asn1.x509.AlgorithmIdentifier;

public interface ContentVerifier
{
    /**
     * Return the algorithm identifier describing the signature
     * algorithm and parameters this expander supports.
     *
     * @return algorithm oid and parameters.
     */
    AlgorithmIdentifier getAlgorithmIdentifier();

    /**
     * Returns a stream that will accept data for the purpose of calculating
     * a signature for later verification. Use org.bouncycastle.util.io.TeeOutputStream if
you want to accumulate
     * the data on the fly as well.
     *
     * @return an OutputStream
     */
    OutputStream getOutputStream();

    /**
     * @param expected expected value of the signature on the data.
     * @return true if the signature verifies, false otherwise
     */
    boolean verify(byte[] expected);
}
```

Generally you will not pass in a ContentVerifier directly as normally you will construct a general message parser which will generate a specific ContentVerifier when the time is right. The reason for this is that, unlike the situation you are in when generating a signature where you are able to determine the signature algorithm used, if you are dealing with having to verify other peoples' signatures you may not be certain which variants of a group of related signature algorithms people send you messages may have chosen to use. To deal with this, in the Bouncy Castle APIs we generally talk in terms of objects implementing ContentVerifierProvider.

The ContentVerifierProvider is also a simple interface:

```
package org.bouncycastle.operator;

import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.cert.X509CertificateHolder;

/**
 * General interface for providers of ContentVerifier objects.
 */
public interface ContentVerifierProvider
{
    /**
     * Return whether or not this verifier has a certificate associated with it.
     *
     * @return true if there is an associated certificate, false otherwise.
     */
    boolean hasAssociatedCertificate();

    /**
     * Return the associated certificate if there is one.
     *
     * @return a holder containing the associated certificate if there is one, null if
there is not.
     */
    X509CertificateHolder getAssociatedCertificate();
```

```
    /**
     * Return a ContentVerifier that matches the passed in algorithm identifier,
     *
     * @param verifierAlgorithmIdentifier the algorithm and parameters required.
     * @return a matching ContentVerifier
     * @throws OperatorCreationException if the required ContentVerifier cannot be created.
     */
    ContentVerifier get(AlgorithmIdentifier verifierAlgorithmIdentifier)
        throws OperatorCreationException;
}
```

As you can see the get() method is the one returning the actual ContentVerifier that is required. The get() method is passed the AlgorithmIdentifier that is associated with the signature a ContentVerifier is required for. The other two
methods defined allow the implementor to tell a user of the provider if there is an underlying certificate associated with the provider, and if necessary, enable the user to fetch a copy of the underlying certificate so the user check other attributes that might be associated with the public key being used by the ContentVerifier.

In most cases you will probably not have to worry about the intrinsics of implementing one of these. In the case of the JCA, the BC APIs provide the JcaContentVerifierProviderBuilder, which will allow you to build a ContentVerifierProvider bound to a specific provider. In the case where you need to use the BC lightweight APIs and are not worried about dragging in unncessary classes you can use the BcContentVerifierProviderBuilder. If do need to create your own custom implementation, the source of the BcContentVerifierProviderBuilder is most likely the best place to look for ideas.

2.4. Things to Watch Out For

Some care has to be taken with AlgorithmIdentifier objects as there are two areas of potential error to be aware of. The first is that while an object identifier is unique, algorithms do sometimes have more than one object identifier associated with them depending on which body or company developed the standard the algorithm is being used with. The second is that you will often see NULL for the parameters definition, if you do, this means the parameters need to be explicitly set to ASN.1 NULL (DERNull.INSTANCE in Bouncy Castle) rather than being empty.

Another potential trap is that for the most part objects implementing cryptographic algorithms are not thread safe. It is not safe to assume that operator objects are thread safe either.

3. X.509 Certificates and Certification Requests

3.1. Certificate Representation in Bouncy Castle

The Bouncy Castle APIs generate X509CertificateHolder objects which can then be converted into Java X509Certificate objects if required. The reason this is done is to allow the provision and manipulation of certificates in a JVM independent manner. The holder class allows for the generation of the DER encoding of certificates, provides a uniform way of getting access to the SubjectPublicKey via its ASN.1 encoding, and also provides methods to use Bouncy Castle ContentVerifier operators to check the certificate's signature.
A JcaX509CertificateConverter class is provided to allow conversion of X509CertificateHolder objects into actual java.security.X509Certificate objects. If you wish to convert an X509Certificate object into an X509CertificateHolder object a JcaX509CertificateHolder class is provided as well.

3.2. Basic Certificate Generation

X.509 Certificates come in two forms, Version 1 certificates which generally only appear as trust anchors, and Version 3 certificates which are used as intermediate certificates and for what are referred to as end-entity certificates. An end-entity in this context is usual an organisation or individual that is using the certificate for some purpose, either to publish their public encryption key or to provide others with a way of verifying signatures.

3.2.1. Distinguished Names

An X.509 certificate is basically a signed blob that ties a public key to some other form of identity (the subject) as well as providing some key to who it was that assserts it is so (the issuer). The core of this was initially provided by X.500 which is a naming structure built around the idea of a directory. The main ASN.1 structure in X.500 is the Name.

```
Name ::= CHOICE {
             RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET SIZE (1..MAX) OF AttributeTypeAndValue

AttributeTypeAndValue ::= SEQUENCE {
                            type  OBJECT IDENTIFIER,
                            value ANY }
```

The JCA relies on X500Principal for supporting X.500 Name structures. In Bouncy Castle the support for X.500 names is provided under the package hierarchy org.bouncycastle.asn1.x500 with the Name structure been represented by the X500Name class. The packages provide for defining a style for an X.500 name, as well as methods for associating that style with X.500 names that have been read in. Styles are used to determine how X500Name objects are built from strings, how they generate strings, and how they are compared for equality. You can convert an X500Principal to an X500Name by using X500Name.getInstance(x500Principal.getEncoded()).

If you want to construct a X500Name from scratch with using a string representation of the distinguished name you can use the X500NameBuilder class.

The default style is defined in the class BCStyle. As far as we can tell this is the most generally "correct" one, however depending on the requirements and restrictions of the profile, or standard, you are implementing against, you may want to implement your own style to enable uniform and compliant processing of X.500 names in your application.

3.2.2. Things to Watch Out For

There are a two major sources of trouble with X.500 name processing.

There are two ways of converting an X.500 name to a string, depending on which direction you think it is more important to traverse the sequence. The ISO

order converts the X.500 name in the order that it finds the elements of the name, the IETF format converts the name by traversing the sequence in reverse order.

There are also multiple ways of comparing X.500 names for equality, the most general one compares RDNs for equality regardless of their position in the sequence only checking that the RDN is present rather than in the same spot in both cases. The elements of the RDN containing text may also be compared case insensitive, or case sensitive.

In Bouncy Castle's case the default toString() presents the X.500 name in ISO order, and uses the most general equality comparison. What you want to do will depend on what profile you have to use. In the face of this you can normally simplify your life a lot by avoiding using string representations of X.500 names for anything other than debugging, relying on the actual ASN.1 structure for anything else.

Two other things to watch out for.

If you have to add multiple ATVs is that X500 names are normally always DER encoded. In this case the set containing the ATVs will always be encoded sorted based on the values of the encodings of each ATV so that any two sets of the same group of ATVs will always generate the same encodings (and thus the same input for any signature created). This does mean that if you build a set of ATVs it may encode in a different order to how you added them. The important thing to remember here is that only an ASN.1 sequence can be used to specify element in a particular order, if you make assumptions about the order of elements in a set you do so at your own peril.

It is also worth noting that the Name structure's ASN.1 definition means it is a CHOICE item. For this reason X.500 names cannot be implicitly tagged, they must always be tagged explicitly, even in a situation where an ASN.1 module may appear to suggest otherwise.

3.3. Version 1 Certificate Generation

The first version of X.509 primarily solved the problem of how to say who issued a certificate, whose public key the certificate contained, and how to do it in a fashion that meant that, all things being equal, the certificate that contains the information cannot be tampered with.

Consider the following code that generates a root certificate:

```
/**
 * Build a sample V1 certificate to use as a CA root certificate
 */
public static X509Certificate buildRootCert(KeyPair keyPair)
    throws Exception
{
    X509v1CertificateBuilder certBldr = new JcaX509v1CertificateBuilder(
                new X500Name("CN=Test Root Certificate"),
                BigInteger.valueOf(1),
               new Date(System.currentTimeMillis()),
                new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
                new X500Name("CN=Test Root Certificate"),
                keyPair.getPublic());

    ContentSigner signer = new JcaContentSignerBuilder("SHA1withRSA")
                                    .setProvider("BC").build(keyPair.getPrivate());

    return new JcaX509CertificateConverter().setProvider("BC")
                    .getCertificate(certBldr.build(signer));
}
```

The certBldr is first constructed using the mandatory fields required for the TBSCertificate block. Next, a ContentSigner is created to generate the signature required for final certificate building. Finally, the JcaX509CertificateConverter is used to convert the X509CertificateHolder class into an actual java.security.X509Certificate object for the specified provider.

3.4. Version 3 Certificate Generation

The principal innovation with the release of X.509 version 3 was the introduction of certificate extensions. The type of a particular extension is identified by the object identifier, or OID, associated with it.
Object identifiers for commonly used extensions are defined in the class org.bouncycastle.asn1.x509.Extension.

Some extensions require calculations that are defined in RFC 5280. If you are using JCA keys, the JcaX509ExtensionUtils class can be used to do these calculations for you and create the appropriate extension.

Consider the two methods following, one of which creates an intermediate certificate which can be used to sign other certificates, and one of which creates an end entity certificate which might be used to verify one of the subject's signatures or to encrypt data to be sent to the entity represented by the certificate's subject. Both methods make use of the JcaX509ExtensionUtils class and create a number of extensions, with some small differences reflecting the differences in the roles the two certificates would be used for.

```
/**
 * Build a sample V3 certificate to use as an intermediate CA certificate
 */
public static X509Certificate buildIntermediateCert(
    PublicKey intKey, PrivateKey caKey, X509Certificate caCert)
    throws Exception
{
    X509v3CertificateBuilder certBldr = new JcaX509v3CertificateBuilder(
            caCert.getSubjectX500Principal(),
            BigInteger.valueOf(1),
           new Date(System.currentTimeMillis()),
            new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
            new X500Principal("CN=Test CA Certificate"),
            intKey);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    certBldr.addExtension(Extension.authorityKeyIdentifier,
                          false, extUtils.createAuthorityKeyIdentifier(caCert))
            .addExtension(Extension.subjectKeyIdentifier,
                          false, extUtils.createSubjectKeyIdentifier(intKey))
            .addExtension(Extension.basicConstraints,
                          true, new BasicConstraints(0))
            .addExtension(Extension.keyUsage,
                          true, new KeyUsage(KeyUsage.digitalSignature
                                           | KeyUsage.keyCertSign
                                           | KeyUsage.cRLSign));

    ContentSigner signer = new JcaContentSignerBuilder("SHA1withRSA")
                                    .setProvider("BC").build(caKey);

    return new JcaX509CertificateConverter().setProvider("BC")
                   .getCertificate(certBldr.build(signer));
}

/**
 * Build a sample V3 certificate to use as an end entity certificate
 */
public static X509Certificate buildEndEntityCert(
    PublicKey entityKey, PrivateKey caKey, X509Certificate caCert)
    throws Exception
{
    X509v3CertificateBuilder   certBldr = new JcaX509v3CertificateBuilder(
        caCert.getSubjectX500Principal(),
        BigInteger.valueOf(1),
        new Date(System.currentTimeMillis()),
        new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
        new X500Principal("CN=Test End Entity Certificate"),
        entityKey);

    JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();

    certBldr.addExtension(Extension.authorityKeyIdentifier,
                          false, extUtils.createAuthorityKeyIdentifier(caCert))
            .addExtension(Extension.subjectKeyIdentifier,
                          false, extUtils.createSubjectKeyIdentifier(entityKey))
            .addExtension(Extension.basicConstraints,
                          true, new BasicConstraints(false))
            .addExtension(Extension.keyUsage,
                          true, new KeyUsage(KeyUsage.digitalSignature
                                           | KeyUsage.keyEncipherment));

    ContentSigner signer = new JcaContentSignerBuilder("SHA1withRSA")
                                    .setProvider("BC").build(caKey);

    return new JcaX509CertificateConverter().setProvider("BC")
                   .getCertificate(certBldr.build(signer));
}
```

3.5. Certificate Extensions

Having seen extensions getting added, it is worth devoting a bit of time to understanding what the common ones actually mean.

3.5.1. Basic Constraints

The basicConstraints extension is identified by the constant Extension.basicConstraints which has OID value "2.5.29.19" (id-ce-basicConstraints). Its ASN.1 definition looks like this:

```
BasicConstraints ::= SEQUENCE {
     cA                  BOOLEAN DEFAULT FALSE,
     pathLenConstraint   INTEGER (0..MAX) OPTIONAL }
```

The basicConstraints extension helps you to determine if the certificate containing it is allowed to sign other certificates, and if so what depth this can go to. So, for example, if cA is TRUE and the pathLenConstraint is 0, then the certificate, as far as this extension is concerned, is allowed to sign other certificates, but none of the certificates so signed can be used to sign other certificates.

To recover a BasicConstraints using a X509CertificateHolder class you can use:

```
 BasicConstraints basicConstraints = BasicConstraints.fromExtensions(
                                                    certHldr.getExtensions());
```

to recover the object. In the case of Java's X509Certificate class, a method getBasicConstraints() is provided which returns an int. The int value represents the pathLenConstraint unless cA is FALSE in which case the return value of getBasicConstraints() is -1.

3.5.2. Authority Key Identifier

The authorityKeyIdentifier extension is identified by the constant Extension.keyUsage which has OID value "2.5.29.35" (id-ce-authorityKeyIdentifier). Its ASN.1 definition looks like this:

```
AuthorityKeyIdentifier ::= SEQUENCE {
   keyIdentifier             [0] KeyIdentifier           OPTIONAL,
   authorityCertIssuer       [1] GeneralNames            OPTIONAL,
   authorityCertSerialNumber [2] CertificateSerialNumber OPTIONAL  }

KeyIdentifier ::= OCTET STRING
```

The object of this extension is to identify the public key that can be used to verify the signature on the certificate, or put another way, to verify the signature on the certificate which ties the public key on the certificate to the subject of the certificate and any associated extensions. Because of this if you are willing to accept the public key identified by this extension as been authoritive, it makes sense for you to accept the public key at its associated attributes stored in the certificate as been valid as well. On the other hand, if you can't make sense of this extension or verify the signer of the certificate, accepting anything in the certificate is more of an act of blind faith, rather than an act based on the validity of the signing algorthim used.

To recover a AuthorityKeyIdentifier using a X509CertificateHolder class you can use:

```
 AuthorityKeyIdentifier authorityKeyIdentifier = AuthorityKeyIdentifier.fromExtensions(
    certHldr.getExtensions());
```

to recover the object.

3.5.3. Subject Key Identifier

The subjectKeyIdentifier extension is identified by the constant Extension.keyUsage which has OID value "2.5.29.14" (id-ce-subjectKeyIdentifier). Its ASN.1 definition looks like this:

```
SubjectKeyIdentifier ::= KeyIdentifier
```

The subjectKeyIdentifier is simply a string of octets which is used to provide an identifier for the public key the certificate contains. For example if you were looking for the issuing certificate for a particular certificate and the AuthorityKeyIdentifier for the particular certificate had the keyIdentifier field set you would expect to find the value stored in keyIdentifier in the SubjectKeyIdentifier extension of the issuing certificate. RFC 5280, section 4.2.1.2, gives 2 common methods for calculating the key identifier value.

To recover a SubjectKeyIdentifier using a X509CertificateHolder class you can use:

```
SubjectKeyIdentifier subjectKeyIdentifier = SubjectKeyIdentifier.fromExtensions(
    certHldr.getExtensions());
```

to recover the object.

3.5.4. Subject Alternative Name

The subjectAltName extension is identified by the constant Extension.keyUsage which has OID value "2.5.29.17" (id-ce-subjectAltName). Its ASN.1 definition looks like this:

```
SubjectAltName ::= GeneralNames

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName
```

The subjectAltName is used to store alternate, or alias, subject names to associate with a certificate. If you want to associate an email address with a certificate, strictly speaking, this is the best place to put it. It is worth noting that the PKIX profile allows for certificates where the subject name field in the TBSCertificate is an empty sequence with the actual subject details stored in the subjectAltName extension.

To recover the GeneralNames object for a subjectAltName extension using a X509CertificateHolder class you can use:

```
GeneralNames subjectAltName = GeneralNames.fromExtensions(
                                        certHldr.getExtensions(),
                                        Extension.subjectAlternativeName);
```

to recover the object.

3.5.5. Issuer Alternative Name

The issuerAltName extension is identified by the constant Extension.keyUsage which has OID value "2.5.29.18" (id-ce-issuerAltName). Its ASN.1 definition looks like this:

```
IssuerAltName ::= GeneralNames
```

Like the subjectAltName the issuerAltName is used to store alternate names that can be associated with the certificate issuer. Unlike the subjectAltName extension this extension cannot act as a substitute for the contents of the issuer filed in the TBSCertificate structure.

To recover the GeneralNames object for a issuerAltName extension using a X509CertificateHolder class you can use:

```
```

```
GeneralNames issuerAltName = GeneralNames.fromExtensions(
                                          certHldr.getExtensions(),
                                          Extension.issuerAlternativeName);
```

to recover the object.

3.5.6. Key Usage

The keyUsage extension is identified by the constant Extension.keyUsage which has OID value "2.5.29.15" (id-ce-keyUsage). Its ASN.1 definition looks like this:

```
KeyUsage ::= BIT STRING {
     digitalSignature        (0),
     nonRepudiation          (1),
     keyEncipherment         (2),
     dataEncipherment        (3),
     keyAgreement            (4),
     keyCertSign             (5),
     cRLSign                 (6),
     encipherOnly            (7),
     decipherOnly            (8) }
```

The keyUsage extension is the most general way of restricting the uses of the key contained in the certificate. Which bits must, or must not be set, depends largely on the profile the certificate is been used with and what purpose it has. RFC 5280, section 4.2.1.3 discusses the range of possibilities as well as providing some pointers to other RFCs that place specific interpretations on keyUsage.

To recover a KeyUsage using a X509CertificateHolder class you can use:

```
KeyUsage keyUsage = KeyUsage.fromExtensions(certHldr.getExtensions());
```

to recover the object.

3.5.7. Extended Key Usage

The extKeyUsage extension is identified by the constant Extension.extendedKeyUsage which has OID value "2.5.29.37" (id-ce-extKeyUsage). Its ASN.1 definition looks like this:

```
ExtKeyUsageSyntax ::= SEQUENCE SIZE (1..MAX) OF KeyPurposeId

KeyPurposeId ::= OBJECT IDENTIFIER
```

If this extension is present the certificate is meant to be used for only one of the purposes listed in it, unless the special KeyPurposeId anyExtendedKeyUsage is included in the ExtKeyUsageSyntax sequence. Commmonly this extension is used to lock a certificate down to a specific purpose in a more rigid way than allowed by the key usage extension.

3.5.8. Things to Watch Out For

People often get confused about the AuthorityKeyIdentifier and what it should contain. The important thing to remember about this one is that it is supposed to identify the issuer certificate, i.e. the certificate which can be used to verify the certificate containing the extension. This is the reason why if the authorityCertIssuer and authorityCertSerialNumber fields are set they are taken from the issuer field of the issuer certificate, and the serial number of the issuer certificate, as it is those two things together which will always uniquely identify the issuer certificate.

3.6. Lightweight Certificate Generation

In some cases the JVM you will be using will not have any of the JCA related classes in it, or you may be dealing with a hardware crypto adaptor that is not compliant with the JCA. In this case you can use Bouncy Castle's lightwieght API, your own implementation of ContentSigner, or a combination of both. As you would expect, doing this does require some more in-depth knowledge of what you are doing as you will need to understand what is required in AlgorithmIdentifier strcutures associated with the content, however the use of the Bouncy Castle API is largely the same.

3.6.1. Version 1 Certificate Generation

For version 1 certificates, it takes a few minor tweaks, and we can avoid using JCA classes altogether.

Consider the following code for creating a self-signed certificate:

```
    /**
     * Build a sample V1 certificate to use as a CA root certificate
     */
    public static X509CertificateHolder buildRootCert(AsymmetricCipherKeyPair keyPair)
        throws Exception
    {
    X509v1CertificateBuilder certBldr = new X509v1CertificateBuilder(
                new X500Name("CN=Test Root Certificate"),
                BigInteger.valueOf(1),
              new Date(System.currentTimeMillis()),
                new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
                new X500Name("CN=Test Root Certificate"),

SubjectPublicKeyInfoFactory.createSubjectPublicKeyInfo(keyPair.getPublic()));

        AlgorithmIdentifier sigAlg = algFinder.find("SHA1withRSA");
        AlgorithmIdentifier digAlg = new
DefaultDigestAlgorithmIdentifierFinder().find(sigAlg);

        ContentSigner signer = new BcRSAContentSignerBuilder(sigAlg,
digAlg).build(keyPair.getPrivate());

    return certBldr.build(signer);
    }
```

Like the earlier example this method generates a self-signed trust anchor. In this case though, it uses the raw building blocks to put the certificate together as well as some helper methods designed to make it easier to convert lightweight keys into their ASN.1 encoded equivalents.

The differences with the JCA example tell the story. The X500Name class replaces any use of X500Principal and an actual SubjectPublicKeyInfo structure is passed to the constructor to provide the encoding of the subject's public key. After that a ContentSigner is built up using lightweight primitives. Finally we just return a X509CertificateHolder object rather than converting the result into a certificate bound to a particular provider.

3.6.2. Version 3 Certificate Generation

The Version 3 certificate generation follows a similar pattern. The difference in this case is more due to the need to pass a DigestCalculator to the X509ExtensionUtils constructor. The DigestCalculator is required to generate values like the keyIdentifier referred to in the authorityKeyIdentifier extension. If we're following the standard PKIX profile this will always be a calculator based on SHA-1. Other than that, you can see from the following two methods that after making allowances for the differences seen in the Version 1 example, the lightweight pattern is the same as that used with the JCA based method.

```
    /**
     * Build a sample V3 certificate to use as an intermediate CA certificate
     */
    public static X509CertificateHolder buildIntermediateCert(AsymmetricKeyParameter
intKey, AsymmetricKeyParameter caKey, X509CertificateHolder caCert)
        throws Exception
    {
        SubjectPublicKeyInfo intKeyInfo =
SubjectPublicKeyInfoFactory.createSubjectPublicKeyInfo(intKey);

        X509v3CertificateBuilder certBldr = new X509v3CertificateBuilder(
                caCert.getSubject(),
                BigInteger.valueOf(1),
               new Date(System.currentTimeMillis()),
                new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
                new X500Name("CN=Test CA Certificate"),
                intKeyInfo);
```

```
        X509ExtensionUtils extUtils = new X509ExtensionUtils(new SHA1DigestCalculator());

        certBldr.addExtension(Extension.authorityKeyIdentifier,
                              false, extUtils.createAuthorityKeyIdentifier(caCert))
                .addExtension(Extension.subjectKeyIdentifier,
                              false, extUtils.createSubjectKeyIdentifier(intKeyInfo))
                .addExtension(Extension.basicConstraints,
                              true, new BasicConstraints(0))
                .addExtension(Extension.keyUsage,
                              true, new KeyUsage(KeyUsage.digitalSignature
                                                 | KeyUsage.keyCertSign
                                                 | KeyUsage.cRLSign));

        AlgorithmIdentifier sigAlg = algFinder.find("SHA1withRSA");
        AlgorithmIdentifier digAlg = new
DefaultDigestAlgorithmIdentifierFinder().find(sigAlg);

        ContentSigner signer = new BcRSAContentSignerBuilder(sigAlg, digAlg).build(caKey);

        return certBldr.build(signer);
    }

    /**
     * Build a sample V3 certificate to use as an end entity certificate
     */
    public static X509CertificateHolder buildEndEntityCert(AsymmetricKeyParameter
entityKey, AsymmetricKeyParameter caKey, X509CertificateHolder caCert)
    throws Exception
    {
        SubjectPublicKeyInfo entityKeyInfo =
SubjectPublicKeyInfoFactory.createSubjectPublicKeyInfo(entityKey);

        X509v3CertificateBuilder   certBldr = new X509v3CertificateBuilder(
            caCert.getSubject(),
            BigInteger.valueOf(1),
            new Date(System.currentTimeMillis()),
            new Date(System.currentTimeMillis() + VALIDITY_PERIOD),
            new X500Name("CN=Test End Entity Certificate"),
            entityKeyInfo);

        X509ExtensionUtils extUtils = new X509ExtensionUtils(new SHA1DigestCalculator());

        certBldr.addExtension(Extension.authorityKeyIdentifier,
                              false, extUtils.createAuthorityKeyIdentifier(caCert))
                .addExtension(Extension.subjectKeyIdentifier,
                              false, extUtils.createSubjectKeyIdentifier(entityKeyInfo))
                .addExtension(Extension.basicConstraints,
                              true, new BasicConstraints(false))
                .addExtension(Extension.keyUsage,
                              true, new KeyUsage(KeyUsage.digitalSignature
                                                 | KeyUsage.keyEncipherment));

        AlgorithmIdentifier sigAlg = algFinder.find("SHA1withRSA");
        AlgorithmIdentifier digAlg = new
DefaultDigestAlgorithmIdentifierFinder().find(sigAlg);

        ContentSigner signer = new BcRSAContentSignerBuilder(sigAlg, digAlg).build(caKey);

        return certBldr.build(signer);
    }
```

3.7. Certificate Processing

3.7.1. The X509CertificateHolder

The Bouncy Castle APIs provide two approaches to processing X.509 certificates. One is the provider independent X509CertificateHolder class, the other is using JCA where certificates are constructed bound to a specific provider. As you will notice, if you have not already, most of the BC APIs, such as the certificate builder APIs we have just looked at, are written to produce and process the provider independent certificate objects, with converters and extension classes made available to produce and process the provider dependent JCA class java.cert.X509Certificate.

The X509CertificateHolder class is the JCA provider-independent holder for a X.509 certificate. The class allows you to use the operator based methods for verifying the enclosed certificate's signature and also provides methods for accessing certificate extensions which return actual extension objects, rather than the encoded octet strings the JCA X509Certificate object does.

3.7.2. Converting to a JCA Certificate

The org.bouncycastle.cert.jcajce.JcaX509CertificateConverter class is provided to convert X509CertificateHolder objects into regular JCA X509Certificate objects.

3.8. Certification Requests

As the term suggests Certification Requests are used to send public keys and other details to a certification authority (CA) so that the CA can issue a certificate containing the public key linked with the other details.

The Bouncy Castle APIs offer two methods for generating Certification Requests.
The first of these is using PKCS#10 and is probably the most common method in use today. The second is using Certificate Request Message Format (CRMF) which is documented in RFC 4211. While less common, one of the benefits of CRMF is that it does not require that the key used in the certification request can be used for verifying a signature. CRMF can be used for encryption only keys, such as ElGamal and other Diffie-Hellman variants and can deal with the problem of confirming that the sender of the request has the request's corresponding private key by returning the certificate encrypted and delaying actual issue of the certificate until the certificate's recipient has confirmed they could decrypt certificate they were sent.

3.8.1. Using PKCS#10

PKCS#10 certification requests are constructed using the PKCS10CertificationRequestBuilder class. The constructor for the builder takes the subject information and the public key (as a SubjectPublicKeyInfo object) that should be contained in the certificate the request is for and the request itself is build using an appropriate ContentSigner which is passed into the build method.

PKCS#10 also allows for other attributes to be included in the certification request, such as certificate extensions. These can be added to the builder using the addAttribute() method.

As with most things in Bouncy Castle, extension classes exist to take advantage of JCA classes where you wish to make use of them. You can see how this can be done in the example that follows:

```
package cwguide;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;

import org.bouncycastle.asn1.pkcs.PKCSObjectIdentifiers;
import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.asn1.x500.X500NameBuilder;
import org.bouncycastle.asn1.x500.style.BCStyle;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.ExtensionsGenerator;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.GeneralNames;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.operator.jcajce.JcaContentVerifierProviderBuilder;
import org.bouncycastle.pkcs.PKCS10CertificationRequest;
import org.bouncycastle.pkcs.PKCS10CertificationRequestBuilder;
import org.bouncycastle.pkcs.jcajce.JcaPKCS10CertificationRequest;
import org.bouncycastle.pkcs.jcajce.JcaPKCS10CertificationRequestBuilder;

/**
 * A simple example showing generation and verification of a PKCS#10 request.
 */
public class JcaPKCS10Example
{
    public static void main(String[] args)
            throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        String sigName = "SHA1withRSA";

        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "BC");

        kpg.initialize(1024);

        KeyPair kp = kpg.genKeyPair();

        X500NameBuilder x500NameBld = new X500NameBuilder(BCStyle.INSTANCE);

        x500NameBld.addRDN(BCStyle.C, "AU");
        x500NameBld.addRDN(BCStyle.ST, "Victoria");
        x500NameBld.addRDN(BCStyle.L, "Melbourne");
        x500NameBld.addRDN(BCStyle.O, "The Legion of the Bouncy Castle");
```

```
        X500Name subject = x500NameBld.build();

        PKCS10CertificationRequestBuilder requestBuilder =
                    new JcaPKCS10CertificationRequestBuilder(subject, kp.getPublic());

        ExtensionsGenerator extGen = new ExtensionsGenerator();

        extGen.addExtension(Extension.subjectAlternativeName,
                            false, new GeneralNames(
                                        new GeneralName(
                                            GeneralName.rfc822Name,
                                            "feedback-crypto@bouncycastle.org")));

        requestBuilder.addAttribute(
                            PKCSObjectIdentifiers.pkcs_9_at_extensionRequest,
                            extGen.generate());

        PKCS10CertificationRequest req1 = requestBuilder.build(
                            new JcaContentSignerBuilder(sigName).setProvider("BC")
                            .build(kp.getPrivate()));

        if (!req1.isSignatureValid(new JcaContentVerifierProviderBuilder()
                                    .setProvider("BC").build(kp.getPublic())))
        {
            System.out.println(sigName + ": Failed verify check.");
        }
        else
        {
            System.out.println(sigName + ": PKCS#10 request verified.");
        }
    }
}
```

Hopefully by now the example does not contain anything that looks too suprising.

In brief, it is constructing a certification request for the subject "c=AU,st=Victoria,l=Melbourne,o=The Legion of the Bouncy Castle", with an attribute attached to the request to add the email address "feedback-crypto@bouncycastle.org" to the certificate the CA will generate. As with the X509CertificateHolder object earlier, the signature on the built request can be verified by passing in a ContentVerifierProvider which will be used to provide the ContentVerifier required to verify the particular signature algorithm used when the request was generated.

3.8.2. Using CRMF

CRMF, like PKCS#10, deals only with the formatting and encoding of certificate requests. This can be problematic as some messages it defines are clearly initiators for a multi-phase protocol. Keeping this in mind it is useful to be aware that there is another RFC, RFC 4210, entitled Certificate Management Protocol (CMP), which provides the remaining protocol required to fill in the gaps.

There are two things central to the CRMF protocol. The first is the use of a template structure to tell the CA what should be present in the issued certificate. The second is the setting out of the criteria that the CA can expect to meet to verify that the certificate requestor actually has possession of the private key which corresponds to the public key they are requesting a certificate for.

3.8.2.1. CRMF with Signature Proof-of-Possession

The most basic form of proof-of-possession is the same as the one outlined in PKCS#10 - if I can generate a signature that will verify with a given public key, I must have possesion of the corresponding private key.

```
package cwguide;

import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;

import javax.security.auth.x500.X500Principal;

import org.bouncycastle.cert.crmf.jcajce.JcaCertificateRequestMessage;
import org.bouncycastle.cert.crmf.jcajce.JcaCertificateRequestMessageBuilder;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.operator.jcajce.JcaContentVerifierProviderBuilder;

/**
 * Basic example of CRMF using a signature for proof-of-possession
 */
public class JcaBasicCRMFExample
{
    public static void main(String[] args)
```

```
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        KeyPairGenerator kGen = KeyPairGenerator.getInstance("RSA", "BC");

        kGen.initialize(512);

        KeyPair kp = kGen.generateKeyPair();

        JcaCertificateRequestMessageBuilder certReqBuild = new
JcaCertificateRequestMessageBuilder(BigInteger.ONE);

        certReqBuild.setPublicKey(kp.getPublic())
                    .setSubject(new X500Principal("CN=Test"))
                    .setProofOfPossessionSigningKeySigner(new
JcaContentSignerBuilder("SHA1withRSA").setProvider("BC").build(kp.getPrivate()));

        JcaCertificateRequestMessage certReqMsg = new
JcaCertificateRequestMessage(certReqBuild.build().getEncoded());

        // check that internal check on popo signing is working okay

        if (certReqMsg.isValidSigningKeyPOP(new
JcaContentVerifierProviderBuilder().setProvider("BC").build(kp.getPublic())))
        {
            System.out.println("CRMF message verified");
        }
        else
        {
            System.out.println("CRMF verification failed");
        }
    }
}
```

Other than the use of the JcaCertificateRequestMessageBuilder rather than the use of the equivalent PKCS#10 builder, you can see that this example follows the same pattern. The public key and the subject name are added, a ContentSigner is set and the result is used to generate the request.

The CRMF template also allows us to request serial numbers, extensions, and even the issuer name (at this point it should be pointed out that the issuer is free to reject, ignore, or even modify such requests) and the CertificateRequestMessageBuilder reflects this. For example in the PKCS#10 example we requested that an extension be added to the issued certificate. In CRMF we would add the same extension using:

```
        certReqBuild.addExtension(
               Extension.subjectAlternativeName,
               false,
               new GeneralNames(
                    new GeneralName(GeneralName.rfc822Name,
                                    "feedback-crypto@bouncycastle.org")
        ));
```

3.8.2.2. CRMF with Two Phase Proof of Possession

The two phase approach is designed to deal with the situation where the algorithm that is represented by the public key on the certificate is one such as El Gamal for example and cannot be used for signing. CRMF deals with this by requesting that the CA send an encrypted copy of the certificate back with the encryption based on the public key sent in the original request. The response in this situation will normally contain some additional information which will allow the the certificate requester to prove to the CA that it was able to decrypt the response thus proving that it has posession of the private key.

A basic initial CRMF request for this approach looks as follows:

```
package cwguide;

import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;

import javax.security.auth.x500.X500Principal;

import org.bouncycastle.asn1.crmf.SubsequentMessage;
import org.bouncycastle.cert.crmf.jcajce.JcaCertificateRequestMessage;
import org.bouncycastle.cert.crmf.jcajce.JcaCertificateRequestMessageBuilder;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

/**
```

```
 * Basic example of CRMF which tells a CA to send the certificate back encrypted.
 */
public class JcaTwoPhaseCRMFExample
{
    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        KeyPairGenerator kGen = KeyPairGenerator.getInstance("RSA", "BC");

        kGen.initialize(512);

        KeyPair kp = kGen.generateKeyPair();

        JcaCertificateRequestMessageBuilder certReqBuild = new
JcaCertificateRequestMessageBuilder(BigInteger.ONE);

        certReqBuild.setPublicKey(kp.getPublic())
                    .setSubject(new X500Principal("CN=Test"))
                    .setProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert);

        JcaCertificateRequestMessage certReqMsg = new
JcaCertificateRequestMessage(certReqBuild.build().getEncoded());

        // check that proof-of-possession is present.
        if (certReqMsg.hasProofOfPossession())
        {
            System.out.println("Proof-of-Possession found");
        }
        else
        {
            System.out.println("No proof-of-possession found");
        }
    }
}
```

In this example we have used setProofOfPossessionSubsequentMessage() which signals to the CA that there will be another phase required to show proof-of-possession. As you can see from the request point of view there is not much to it, other than the use of SubsequentMessage.encrCert.

CRMF leaves how this second phase is conducted as an exercise to the reader, generally though you can fill in the gaps on this one using CMP (RFC 4210).

4. Key and Personal Credential Storage

This chapter looks at private key and certificate storage for situations where you are using keys living in X.509 and PKCS worlds. We'll look at the manipulation of keys and certifications for OpenPGP in a different chapter.

What format you use depends a lot on what you're trying to store and where you are trying to access it from. For a lot of purposes the JKS format, which is usually accessed via the java.security.KeyStore class, is fine, but there are also a few other formats that can be used under the KeyStore API which in some cases offer better security or cross-plaform convenience. Bouncy Castle offers two alternatives itself, but if you are looking for cross platform compatibility and a higher level of security you should probably use PKCS#12. PKCS#12 has become the standard for personal credential interchange and it offers a lot of flexibilty, so much so that Bouncy Castle now has API specifically for handling it.

4.1. The PKCS#12 API

The PKCS#12 API lives in the org.bouncycastle.pkcs package. With JCA/JCE operators and light weight operators living the jcajce and bc sub-packages.

The important thing to remember with a PKCS#12 file is that it is designed to have a very lose file structure but that the ASN.1 structures that are stored in the file need to have enough information, in this case attributes, attached to them to allow a reader to re-establish the relationships between private keys and certificates, as well as the names used to identify any important certificates that might be stored in the PKCS#12 file as well.

PKCS#12 files are made up of nested sequences of Protocol Data Units (PDUs). The key structure used to hold data in a PKCS#12 file is called the SafeBag. SafeBag structures are the containers for individual certificates, private keys, CRLs and whatever else you might want to store. SafeBags are then accumulated into an AuthenticatedSafe structure, and then the AuthenticatedSafe has a MAC calculated on it and the AthenticatedSafe and the MAC are then used to construct an object referred to as the PFX (which is the name given to the top-level PDU). The contents of a SafeBag may be encrypted by default, or one or more SafeBags might be added to an AuthenticatedSafe by wrapping them in a CMS EnvelopedData object (itself a PDU for the purposes of PKCS#12).

4.1.1. Creating a PKCS#12 File - JCE Style

Other than the number of acronyms, it does not sound that bad does it? Well, while no doubt well intentioned, the flexibility of the PFX and its underlying structures has led programmers on many adventures, not all of which ended well. That being said, some standard patterns have evolved for storing private/public key infromation in PFX files, so it would now be worth spending some time looking at one example.

```
    private static void createPKCS12File(OutputStream pfxOut, PrivateKey key, Certificate[]
chain)
        throws Exception
    {
        OutputEncryptor encOut = new
JcePKCSPBEOutputEncryptorBuilder(NISTObjectIdentifiers.id_aes256_CBC).setProvider("BC").bui
ld(JcaUtils.KEY_PASSWD);

        PKCS12SafeBagBuilder taCertBagBuilder = new
JcaPKCS12SafeBagBuilder((X509Certificate)chain[2]);

        taCertBagBuilder.addBagAttribute(PKCS12SafeBag.friendlyNameAttribute, new
DERBMPString("Bouncy Primary Certificate"));

        PKCS12SafeBagBuilder caCertBagBuilder = new
JcaPKCS12SafeBagBuilder((X509Certificate)chain[1]);

        caCertBagBuilder.addBagAttribute(PKCS12SafeBag.friendlyNameAttribute, new
DERBMPString("Bouncy Intermediate Certificate"));

        JcaX509ExtensionUtils extUtils = new JcaX509ExtensionUtils();
        PKCS12SafeBagBuilder eeCertBagBuilder = new
JcaPKCS12SafeBagBuilder((X509Certificate)chain[0]);

        eeCertBagBuilder.addBagAttribute(PKCS12SafeBag.friendlyNameAttribute, new
DERBMPString("Eric's Key"));
        SubjectKeyIdentifier pubKeyId =
extUtils.createSubjectKeyIdentifier(chain[0].getPublicKey());
        eeCertBagBuilder.addBagAttribute(PKCS12SafeBag.localKeyIdAttribute, pubKeyId);

        PKCS12SafeBagBuilder keyBagBuilder = new JcaPKCS12SafeBagBuilder(key, encOut);

        keyBagBuilder.addBagAttribute(PKCS12SafeBag.friendlyNameAttribute, new
DERBMPString("Eric's Key"));
```

```
        keyBagBuilder.addBagAttribute(PKCS12SafeBag.localKeyIdAttribute, pubKeyId);

        PKCS12PfxPduBuilder builder = new PKCS12PfxPduBuilder();

        builder.addData(keyBagBuilder.build());

        builder.addEncryptedData(new
JcePKCSPBEOutputEncryptorBuilder(PKCSObjectIdentifiers.pbeWithSHAAnd128BitRC2_CBC).setProvi
der("BC").build(JcaUtils.KEY_PASSWD), new PKCS12SafeBag[]{eeCertBagBuilder.build(),
caCertBagBuilder.build(), taCertBagBuilder.build()});

        PKCS12PfxPdu pfx = builder.build(new
JcePKCS12MacCalculatorBuilder(NISTObjectIdentifiers.id_sha256), JcaUtils.KEY_PASSWD);

        // make sure we don't include indefinite length encoding
        pfxOut.write(pfx.getEncoded(ASN1Encoding.DL));

        pfxOut.close();
    }
```

The example shows all the basic building blocks for what would now be regarded as a typical PKCS#12 file containing a private key, and a set of associated certificates. Constructing it requires setting up a number of PKCS12SafeBagBuilder objects and then a final PKCS12PfxPduBuilder which is used to create the actual key and certificate store. In the case of the example, another tradition has been followed of encrypting the certificates seperately from the private key (and using 128 bit RC2 for the certificates with 256 bit AES for the key), and finally it should be noted that everything has been encrypted using the same password. This might seem a little odd at first (indeed it is not required by the standard), however these files are specifically for personal credential storage and consequently most implementations assume a common password for each item. If you really want to store two things under different passwords in the PKCS#12 format, the best thing to do is to create two files rather than one.

As you have probably guessed, the PKCS12SafeBag.friendlyNameAttribute is used to associated a string alias with the object enclosed in the SafeBag. The PKCS12SafeBag.localKeyIdAttribute is normally used by the reader of the PKCS#12 PFX file to work out which certificate has the public key associated with the private key.

4.1.2. PBE Encryption in PKCS#12 Files

Tradtionally PKCS#12 files have been created with 40 bit RC2 protecting the certificates and 3 key DES-EDE protecting the private key. These are both algorithms that specify PKCS#12 password conversion and are available, along with other PKCS#12 algorithms, in Bouncy Castle using the OIDs below:

```
    PKCSObjectIdentifiers.pbeWithSHAAnd128BitRC4
    PKCSObjectIdentifiers.pbeWithSHAAnd40BitRC4
    PKCSObjectIdentifiers.pbeWithSHAAnd3_KeyTripleDES_CBC
    PKCSObjectIdentifiers.pbeWithSHAAnd2_KeyTripleDES_CBC
    PKCSObjectIdentifiers.pbeWithSHAAnd128BitRC2_CBC
    PKCSObjectIdentifiers.pbeWithSHAAnd40BitRC2_CBC
```

Traditionally the MAC algorithm is SHA-1. This is the default for JcePKCS12MacCalculatorBuilder if nothing explicit is specified.

Things have moved on a bit since then and in our example we are also using NISTObjectIdentifiers.id_aes256_CBC to encrypt the private key and we are using NISTObjectIdentifiers.id_sha256 to calculate the MAC protecting the PFX from tampering. One thing that you need to be aware of here is that the approach at the moment is to use PKCS#5 for algorithm like AES as while the PKCS#12 MAC calcuation is defined in an algorithm independent fashion, the RSA OIDs defined do not include AES. The Bouncy Castle project has allocated same OIDs off its branch for this purpose but if you are in a situation where you are dealing with other applicaitons using your PFX files, the most general way of handling an algorithm not listed in the RSA set is to apply PKCS#5 scheme 2 to the PBE generation.

It should be pointed out here, that as PKCS#5 predates the days when UTF-8 was in common use and that it did not define a process for converting a string to bytes. For this reason most PKCS#5 implementations think the world is 8 bit ascii, so multi-byte characters either have the top byte ignored or get treated as two bytes. PKCS#12 did not make the same mistake, but unfortunately this means that if you are mixing PKCS#5 and PKCS#12 algorithms in the same PFX and you are using a character set that requires unicode, the quality of the PBE key derived to generate the MAC will probably be of higher quality than the PBE key derived to generate any encryption keys used, or you may find yourself with a PKCS#12 file you can read, but that no-one else can, even though they can verify the MAC! If you are not sure where your files are likely to end up, what systems they need to work on, or even you are just not sure what your PKCS#5 implementation does, it is better to Base64 encode the UTF-8 representation of the password first and then use that as the PFX key.

```
    char[] asciiPass = Strings.asCharArray(Base64.encode(Strings.toUTF8ByteArray(passwd)));
```

4.1.3. Creating a PKCS#12 File - Lightweight Style

4.1.4. Reading a PKCS#12 File

Depending on what you need to do, reading a PKCS#12 file can be more complicated than creating one. The simplest way to read one using Bouncy Castle is to use the KeyStore class. After that, whether you are using BC's lightweight primitives or JCA/JCE based classes it is about the same level of difficulty. We will have a look at an example using the PKCS#12 API from BC first, even if you never expect to try it that way it is useful to know how you would. As you can imagine, the BC PKCS#12 KeyStore class hides most of the complexity that will appear in the next example, however it does have to make assumptions to do this, and occasionally you will run into someone producing PKCS#12 files for which those assumptions are not correct. In a situation like that having some understanding of what is required to interpret a PKCS#12 file will help you recognise the problem early and some knowledge of the PKCS#12 API in Bouncy Castle may also turn out to be the only way you can solve the problem.

4.1.4.1. Reading PKCS#12 Files Using the BC API

As the PKCS#12 file is loosely structured, if you want to rebuild associations between certificates and keys it is something you have to do yourself, likewise the presence of encrypted data changes the way things get handled. As further consideration it is also necessary to bare in mind that what presents as encrypted data may contain other PKCS#12 structures that require further interpretation.

The example below shows a general reader for a PKCS#12 file which builds a Map of certificates and keys and attemtps to identifies which certificates are the end-entity certificates containing the public key which corresponds to the private key. A basic dump of the data recovered is done at the end of the method.

```
    private static PKCS12PfxPdu readPKCS12File(InputStream pfxIn)
        throws Exception
    {
        PKCS12PfxPdu pfx = new PKCS12PfxPdu(Streams.readAll(pfxIn));

        if (!pfx.isMacValid(new
BcPKCS12MacCalculatorBuilderProvider(BcDefaultDigestProvider.INSTANCE),
JcaUtils.KEY_PASSWD))
        {
            System.err.println("PKCS#12 MAC test failed!");
        }

        ContentInfo[] infos = pfx.getContentInfos();

        Map certMap = new HashMap();
        Map certKeyIds = new HashMap();
        Map privKeyMap = new HashMap();
        Map privKeyIds = new HashMap();

        InputDecryptorProvider inputDecryptorProvider = new
JcePKCSPBEInputDecryptorProviderBuilder()

.setProvider("BC").build(JcaUtils.KEY_PASSWD);
        JcaX509CertificateConverter  jcaConverter = new
JcaX509CertificateConverter().setProvider("BC");

        for (int i = 0; i != infos.length; i++)
        {
            if (infos[i].getContentType().equals(PKCSObjectIdentifiers.encryptedData))
            {
                PKCS12SafeBagFactory dataFact = new PKCS12SafeBagFactory(infos[i],
inputDecryptorProvider);

                PKCS12SafeBag[] bags = dataFact.getSafeBags();

                for (int b = 0; b != bags.length; b++)
                {
                    PKCS12SafeBag bag = bags[b];

                    X509CertificateHolder certHldr =
(X509CertificateHolder)bag.getBagValue();
                    X509Certificate       cert = jcaConverter.getCertificate(certHldr);

                    Attribute[] attributes = bag.getAttributes();
                    for (int a = 0; a != attributes.length; a++)
                    {
                        Attribute attr = attributes[a];

                        if (attr.getAttrType().equals(PKCS12SafeBag.friendlyNameAttribute))
```

```
                        {
certMap.put(((DERBMPString)attr.getAttributeValues()[0]).getString(), cert);
                        }
                        else if
(attr.getAttrType().equals(PKCS12SafeBag.localKeyIdAttribute))
                        {
                            certKeyIds.put(attr.getAttributeValues()[0], cert);
                        }
                    }
                }
            }
            else
            {
                PKCS12SafeBagFactory dataFact = new PKCS12SafeBagFactory(infos[i]);

                PKCS12SafeBag[] bags = dataFact.getSafeBags();

                PKCS8EncryptedPrivateKeyInfo encInfo =
(PKCS8EncryptedPrivateKeyInfo)bags[0].getBagValue();
                PrivateKeyInfo info =
encInfo.decryptPrivateKeyInfo(inputDecryptorProvider);

                KeyFactory keyFact = KeyFactory
.getInstance(info.getPrivateKeyAlgorithm().getAlgorithm().getId(), "BC");
                PrivateKey privKey = keyFact.generatePrivate(new
PKCS8EncodedKeySpec(info.getEncoded()));

                Attribute[] attributes = bags[0].getAttributes();
                for (int a = 0; a != attributes.length; a++)
                {
                    Attribute attr = attributes[a];

                    if (attr.getAttrType().equals(PKCS12SafeBag.friendlyNameAttribute))
                    {
privKeyMap.put(((DERBMPString)attr.getAttributeValues()[0]).getString(), privKey);
                    }
                    else if (attr.getAttrType().equals(PKCS12SafeBag.localKeyIdAttribute))
                    {
                        privKeyIds.put(privKey, attr.getAttributeValues()[0]);
                    }
                }
            }
        }

        System.out.println("########## PFX Dump");
        for (Iterator it = privKeyMap.keySet().iterator(); it.hasNext();)
        {
            String alias = (String)it.next();

            System.out.println("Key Entry: " + alias + ", Subject: " +
(((X509Certificate)certKeyIds.get(privKeyIds.get(privKeyMap.get(alias)))).getSubjectDN()));
        }

        for (Iterator it = certMap.keySet().iterator(); it.hasNext();)
        {
            String alias = (String)it.next();

            System.out.println("Certificate Entry: " + alias + ", Subject: " +
(((X509Certificate)certMap.get(alias)).getSubjectDN()));
        }
        System.out.println();

        return pfx;
    }
```

4.1.4.2. Reading PKCS#12 Files Using the KeyStore Class

The following code fragment does something similar to the code in the previous section - it generates a basic dump of the data stored in the PKCS#12 file. In this case it is based on the KeyStore class, so the parsing section is obviously substantially simpler, and for the most part you will get back a correct certificate chain as the KeyStore implementation does most of the work for you.

```
        KeyStore pkcs12Store = KeyStore.getInstance("PKCS12", "BC");

        pkcs12Store.load(new FileInputStream("id.p12"), JcaUtils.KEY_PASSWD);

        System.out.println("########## KeyStore Dump");

        for (Enumeration en = pkcs12Store.aliases(); en.hasMoreElements();)
        {
            String alias = (String)en.nextElement();

            if (pkcs12Store.isCertificateEntry(alias))
            {
                System.out.println("Certificate Entry: " + alias + ", Subject: " +
(((X509Certificate)pkcs12Store.getCertificate(alias)).getSubjectDN()));
            }
```

```
            else if (pkcs12Store.isKeyEntry(alias))
            {
                System.out.println("Key Entry: " + alias + ", Subject: " +
(((X509Certificate)pkcs12Store.getCertificate(alias)).getSubjectDN()));
            }
        }

        System.out.println();
```

The one occasional downfall of this approach, other than the implementation of KeyStore running into a totally unexpected structure in the PKCS#12 file, is that you lose the PKCS#12 attributes. The BC KeyStore implementation does attempt to preserve friendlyName and uses the localKeyId to rebuild associations between certificates and private keys, however these and other attributes can appear in PKCS#12 files and may sometimes be meaningful. If you need to be able to interpret attributes yourself the best thing to do is to use the Bouncy Castle PKCS#12 API instead.

4.1.5. Things to Watch Out For

Apart from been aware that the use of more than one password on any given component is a risk, and the UTF-8 issue, the other thing to watch with PKCS#12 is that traditionally the data was written out in BER format. The BC APIs reflect this tradition, however it is often the case now that a PFX needs to be saved in definite-length encoding. This is partly due to concerns that a BER parser may indavertently run an application out of memory without warning, and also because it appears BER is not often implemented in do-it-yourslef ASN.1 parsing libraries. In order to make sure you produce definite-length output when dumping the PFX, you need to call the getEncoded() method passing it ASN1Encoding.DL as in:

```
        FileOutputStream fOut = new FileOutputStream("id.p12");
        fOut.write(pfx.getEncoded(ASN1Encoding.DL));
        fOut.close();
```

This will give you a nice definite-length friendly encoded file.

5. Cryptographic Message Syntax (CMS)

This chapter looks at the secure messaging standard that was originally defined in PKCS#7 and is now defined by RFC 5652. A lot of thought has been put into CMS. As it uses BER encoding, it supports both streaming and packet style protocols, consequently it is possible to use it to create very large messages. As it follows the practice of separating the encryption of the data from the sharing of the key used it can also be comfortably used with standard public key algorithms, including those using key agreement, as well as shared keys such as passwords or specified symmetric keys.

CMS allows you to sign, encrypt, and compress messages. It is used to support a number of other standards as well, the most notable of these being S/MIME.

5.1. Getting Started

The Bouncy Castle CMS API is included under the org.bouncycastle.cms package tree. This is distributed in the distribution jars starting with bcpkix for the major JDKs and is also included in the J2ME distribution. Currently the J2ME version supports all of the functionality that can be dealt with using the lightweight API, pretty well everything other than the handling of compressed data.

The RFC defines that for a minimal CMS implemtation you need to be able to support at forms of signing and encryption.

5.2. The Basic Structures

The basic structure of CMS is the ContentInfo type

```
ContentInfo ::= SEQUENCE {
        contentType ContentType,
        content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER
```

The contentType field is used to tell a reader of the ContentInfo sequence how to interpret the explicitly tagged object contained in the content field. As we go through the API you will see that ContentType identifiers are defined for the core CMS structures used to contain signed, enveloped, and compressed data. The Bouncy Castle classes such as CMSSignedData and CMSEnvelopedData actually take ContentInfo objects on construction, likewise the generators for typical CMS messages also produce ContentInfo objects when they complete.

The catch-all ContentType in CMS is id-data. This is defined as:

```
id-data OBJECT IDENTIFIER ::= { iso(1) member-body(2)
         us(840) rsadsi(113549) pkcs(1) pkcs7(7) 1 }
```

When you see one of these in a ContentInfo object it is telling you that the content field contains an arbitrary octet string. You need to be forewarned if you wish to assign any interpretation to the octets, there is nothing to prevent this octet string from being the product of an ASN.1 encoding itself, likewise it might just be a stream of bits such as an image.

In Bouncy Castle speak the classes that capture the need to represent arbitrary ContentInfo structures are the CMSTypedData and the CMSTypedStream classes which are used to return data read from the packet and stream based parsers respectively, and to pass in data where detached messages that do not encapsulate the data, such as some signatures, are used. Whether you wish to choose the packet or sream based classes for processing CMS is largely up to you, the main things to note are that while the stream based classes for both generation and parsing allow you to handle arbitrarily large messages, as you are using a streaming model the order in which operations are done in either parsing or generating becomes significant.

5.3. Message Signing

CMS provides for signatures on messages to either encapsulate, or contain, the message or to be detached from the message. The decision to use either is largely a matter of convenience - for example S/MIME makes use of detached signatures so that a message is always readable even if the recipient does not have the facilities required to process the signature. Where the decision is made to use detached signatures care must be taken to ensure that the message being signed will not be modified during transport in a manner that will cause verification of its associated detached signature to fail.

Section 5 of RFC 5652 covers signing of data. If you are really curious about the internal workings of the protocol we would reading it. We will have a look at the basic structures here as well as the Bouncy Castle API reflects how the messages are structured, so having some knowledge of the internals of the protocol does make it easier to understand what the Bouncy Castle APIs are trying to do and how to use them.

5.3.1. Basic Concepts

The core structure for supporting message signing is the SignedData structure, which is then placed in a ContentInfo structure labeled with the content type id-signedData. The ASN.1 definition for the structure and its identifier is:

```
id-signedData OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2 }

SignedData ::= SEQUENCE {
    version CMSVersion,
    digestAlgorithms DigestAlgorithmIdentifiers,
    encapContentInfo EncapsulatedContentInfo,
    certificates [0] IMPLICIT CertificateSet OPTIONAL,
    crls [1] IMPLICIT RevocationInfoChoices OPTIONAL,
    signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

DigestAlgorithmIdentifier ::= AlgorithmIdentifier

CertificateSet ::= SET OF CertificateChoices

RevocationInfoChoices ::= SET OF RevocationInfoChoice

SignerInfos ::= SET OF SignerInfo
```

It is the SignedData structure that the CMSSignedData and CMSSignedDataParser classes are designed to take. In the event the data that was signed is encapsulated in the SignedData it lives in the encapContentInfo field. The RFC defines EncapsulatedConentInfo as:

```
EncapsulatedContentInfo ::= SEQUENCE {
    eContentType ContentType,
    eContent [0] EXPLICIT OCTET STRING OPTIONAL }
```

If the data is encapsulated, it will be in the eContent field, annotated with the appropriate type in eContentType (usually id-data). If the data is not encapsulated eContentType will give the expected type of the data, and eContent will be missing. Where eContent is missing you will need to explicitly pass the data to be associated with the SignedData using one of the constructors for CMSSignedData or CMSSignedDataParser that takes a signedContent argument.

You can get access to the signed content using the getSignedContent() method on the CMSSignedData, or CMSSignedDataParser, classes. Note that in the event the data in the SignedData structure is not encapsulated, getSignedContent() will just return what was passed to the objects constructor.

The SignerInfos set is where the actual signatures to be verified live, one signature for each SignerInfo structure. In Bouncy Castle the SignerInfo structure is captured in the SignerInformation object. You can get access to SignerInformation objects by calling the getSignerInfos() method on the CMSSignedData, or CMSSignedParser class. The getSignerInfos() method returns a SignerInformationStore which can be used to retrieve one, all, or a subset of the SignerInformation objects associated with the signed data depending on what is passed to the get() or getSigners() method.

5.3.2. Creating and Validating a Detached Signature

Generating a signature is based around the use of the CMSSignedDataGenerator, or CMSSignedDataStreamGenerator, class depending on what suits your

purpose. We will look at the simplest case first, which is using the CMSSignedDataGenerator, as in the case of the DataStreamGenerator some operations can only be done in a particular order.

As with certificate generation, the CMS SignedData generation is based around operators, so a general set of operators are available if you have access to the JCA, and a more specific set of operators are available if you are using the BC lightweight API.

5.3.2.1. Detached Signature - JCA Style

The first step to being able to construct a detached signature is setting up a generator to create the SignerInfo structure you wish to include. If you are using the JCA the Bouncy Castle APIs provide two ways of creating signer info generator: the JcaSimpleSignerInfoGeneratorBuilder class, which assumes the both the signature algorithm and any associated digests can be sourced from a single provider, and the JcaSignerInfoGeneratorBuilder which allows for configuring the DigestCalculator used and the ContentSigner used seperately.

The second consideration is whether you wish to include the signing certificate in the message. This will not remove the requirement that the recipient should verify that the certificate is legitimate, but it will mean that the recipient does not need to have the certificate on hand when the message is verified. To pass this information into the message we construct a Store object containing the certificates and pass it to the CMSSignedDataGenerator.setCertificates() method.

Once a CMS SignedData message has been generated, verifying it follows a similar pattern to what we saw with certificates, the difference being that rather than using a ContentVerifierProvider to verify the signature we use a SignerInfoVerifier which has a ContentVerifierProvider contained in it. As we are using the JCA we can use the JcaSimpleSignerInfoVerifierBuilder to make one of these, although if more flexibility was required we could also use a JcaSignerInfoVerifierBuilder.

An example of creating a CMS SignedData message for a detached signature follows. The main body of the class shows how the message is constructed and the isValid() method shows how the message is verified. Note that as the message is detached, creating a CMSSignedData object for verification purposes requires passing the message data into the constructor.

```
package cwguide;

import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.Security;
import java.security.cert.CertStore;
import java.security.cert.Certificate;
import java.security.cert.PKIXCertPathBuilderResult;
import java.security.cert.X509CertSelector;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.Iterator;

import org.bouncycastle.asn1.x509.KeyUsage;
import org.bouncycastle.cert.jcajce.JcaCertStore;
import org.bouncycastle.cert.jcajce.JcaCertStoreBuilder;
import org.bouncycastle.cms.CMSProcessableByteArray;
import org.bouncycastle.cms.CMSSignedData;
import org.bouncycastle.cms.CMSSignedDataGenerator;
import org.bouncycastle.cms.CMSTypedData;
import org.bouncycastle.cms.SignerInformation;
import org.bouncycastle.cms.SignerInformationStore;
import org.bouncycastle.cms.jcajce.JcaSimpleSignerInfoGeneratorBuilder;
import org.bouncycastle.cms.jcajce.JcaSimpleSignerInfoVerifierBuilder;
import org.bouncycastle.cms.jcajce.JcaX509CertSelectorConverter;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Store;

/**
 * JCA example of generating a detached signature.
 */
public class JcaSignedDataExample
{
    /**
     * Take a CMS SignedData message and a trust anchor and determine if
     * the message is signed with a valid signature from a end entity
     * entity certificate recognized by the trust anchor rootCert.
     */
    public static boolean isValid(
        CMSSignedData   signedData,
        X509Certificate rootCert)
        throws Exception
    {
        CertStore certsAndCRLs = new
JcaCertStoreBuilder().setProvider("BC").addCertificates(signedData.getCertificates()).build
();
```

```
        SignerInformationStore signers = signedData.getSignerInfos();
        Iterator it = signers.getSigners().iterator();

        if (it.hasNext())
        {
            SignerInformation signer = (SignerInformation)it.next();
            X509CertSelector signerConstraints = new
JcaX509CertSelectorConverter().getCertSelector(signer.getSID());

            signerConstraints.setKeyUsage(JcaUtils.getKeyUsage(KeyUsage.digitalSignature));

            PKIXCertPathBuilderResult result = JcaUtils.buildPath(rootCert,
signerConstraints, certsAndCRLs);

            return signer.verify(new JcaSimpleSignerInfoVerifierBuilder().setProvider("BC")

.build((X509Certificate)result.getCertPath().getCertificates().get(0)));
        }

        return false;
    }

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        KeyStore        credentials = JcaUtils.createCredentials();
        PrivateKey      key = (PrivateKey)credentials.getKey(JcaUtils.END_ENTITY_ALIAS,
JcaUtils.KEY_PASSWD);
        Certificate[]   chain = credentials.getCertificateChain(JcaUtils.END_ENTITY_ALIAS);

        X509Certificate cert = (X509Certificate)chain[0];
        Store certs = new JcaCertStore(Arrays.asList(chain));

        // set up the generator
        CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

        gen.addSignerInfoGenerator(new
JcaSimpleSignerInfoGeneratorBuilder().setProvider("BC").build("SHA256withRSA", key, cert));
        gen.addCertificates(certs);

        // create the signed-data object
        CMSTypedData data = new CMSProcessableByteArray("Hello World!".getBytes());

        CMSSignedData signed = gen.generate(data);

        // recreate
        signed = new CMSSignedData(data, signed.getEncoded());

        // verification step
        X509Certificate rootCert =
(X509Certificate)credentials.getCertificate(JcaUtils.ROOT_ALIAS);

        if (isValid(signed, rootCert))
        {
            System.out.println("verification succeeded");
        }
        else
        {
            System.out.println("verification failed");
        }
    }
}
```

If you run this example you should see the output

```
    verification succeeded
```

The JcaSimpleSignerInfoGeneratorBuilder class the example uses assumes that the same provider can be used to source all the algorithms required to add the SignerInfo structure to the CMS message. This is not always the case, so a more long winded way of doing this would have been to say:

```
        gen.addSignerInfoGenerator(
              new JcaSignerInfoGeneratorBuilder(
                    new JcaDigestCalculatorProviderBuilder().setProvider("BC")
                    .build())
              .build(new JcaContentSignerBuilder("SHA256withRSA").setProvider("BC")
                          .build(key), cert));
```

instead. In general you should only need to use the more verbose approach if you need to make use of different providers for digest calculation and signature creation.

5.3.2.2. Detached Signature - Lightweight Style

Writing a lightweight version for generating and verifying a detached signature requires a bit more plumbing. In the JCA case it is easy to create finders for the appropriate AlgorithmIdentifier structures on the fly, although the price you pay is that the support code to do this is a lot bigger. In the lightweight case some useful default classes are provided for looking up AlgorithmIdentifier structures and generating DigestCalculator objects depending on the signature algorithm requested, these could easily be slimmed down though as at the end of the day the only requirement for creating a SignerInfo structure is being able to create a SignerInfoGeneratorBuilder object. Likewise verifying a SignedData message only requires creating a SignerInformationVerifier which matches one of the SignerInfo structures attached to the message.

An example of creating a CMS SignedData message for a detached signature using the lightweight API follows. The example has been constructed along the same lines as the JCA example. The differences between the two are largely due to the need to use more fundamental classes to build the operators required for the creation and verification of SignerInfo objects.

```
package cwguide;

import java.util.Arrays;
import java.util.Iterator;

import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.cert.X509CertificateHolder;
import org.bouncycastle.cms.CMSProcessableByteArray;
import org.bouncycastle.cms.CMSSignedData;
import org.bouncycastle.cms.CMSSignedDataGenerator;
import org.bouncycastle.cms.CMSTypedData;
import org.bouncycastle.cms.DefaultCMSSignatureAlgorithmNameGenerator;
import org.bouncycastle.cms.SignerInfoGeneratorBuilder;
import org.bouncycastle.cms.SignerInformation;
import org.bouncycastle.cms.SignerInformationStore;
import org.bouncycastle.cms.SignerInformationVerifier;
import org.bouncycastle.cms.bc.BcRSASignerInfoVerifierBuilder;
import org.bouncycastle.crypto.params.AsymmetricKeyParameter;
import org.bouncycastle.operator.DefaultDigestAlgorithmIdentifierFinder;
import org.bouncycastle.operator.DefaultSignatureAlgorithmIdentifierFinder;
import org.bouncycastle.operator.bc.BcDigestCalculatorProvider;
import org.bouncycastle.operator.bc.BcRSAContentSignerBuilder;
import org.bouncycastle.util.CollectionStore;
import org.bouncycastle.util.Store;

/**
 * Lightweight example of generating a detached signature.
 */
public class BcSignedDataExample
{
    /**
     * Take a CMS SignedData message and a trust anchor and determine if
     * the message is signed with a valid signature from a end entity
     * entity certificate recognized by the trust anchor rootCert.
     */
    public static boolean isValid(
        CMSSignedData         signedData)
        throws Exception
    {
        Store certs = signedData.getCertificates();
        SignerInformationStore signers = signedData.getSignerInfos();
        Iterator it = signers.getSigners().iterator();

        if (it.hasNext())
        {
            SignerInformation signer = (SignerInformation)it.next();
            X509CertificateHolder cert =
(X509CertificateHolder)certs.getMatches(signer.getSID()).iterator().next();

            SignerInformationVerifier verifier = new BcRSASignerInfoVerifierBuilder(
                new DefaultCMSSignatureAlgorithmNameGenerator(),
                new DefaultSignatureAlgorithmIdentifierFinder(),
                new DefaultDigestAlgorithmIdentifierFinder(),
                new BcDigestCalculatorProvider()).build(cert);

            return signer.verify(verifier);
        }

        return false;
    }

    public static void main(String[] args)
        throws Exception
    {
        BcCredential           credentials = BcUtils.createCredentials();
        AsymmetricKeyParameter  key = credentials.getPrivateKey();
        X509CertificateHolder[] chain = credentials.getCertificateChain();
```

```
        X509CertificateHolder cert = chain[0];
        Store certs = new CollectionStore(Arrays.asList(chain));

        // set up the generator
        CMSSignedDataGenerator gen = new CMSSignedDataGenerator();

        AlgorithmIdentifier sigAlg = new
DefaultSignatureAlgorithmIdentifierFinder().find("SHA256withRSA");
        AlgorithmIdentifier digAlg = new
DefaultDigestAlgorithmIdentifierFinder().find(sigAlg);

        gen.addSignerInfoGenerator(new SignerInfoGeneratorBuilder(new
BcDigestCalculatorProvider()).build(new BcRSAContentSignerBuilder(sigAlg,
digAlg).build(key), cert));

        gen.addCertificates(certs);

        // create the signed-data object
        CMSTypedData data = new CMSProcessableByteArray("Hello World!".getBytes());

        CMSSignedData signed = gen.generate(data);

        // recreate
        signed = new CMSSignedData(data, signed.getEncoded());

        // verification step
        if (isValid(signed))
        {
            System.out.println("verification succeeded");
        }
        else
        {
            System.out.println("verification failed");
        }
    }
}
```

Running this example should also produce the output

```
    verification succeeded
```

5.3.3. Things to Watch Out For

It should be noted the EncapsulatedContentInfo is not identical to what it is in PKCS#7 (eContent is defined as ANY rather than OCTET STRING). In the event you are working with a legacy application, you may find it necessary to use the work around described in Section 5.2.1 in RFC 5652.