The Bouncy Castle FIPS C# API in 100 Examples (Draft)

Oreste Panaia

&

David Hook

Published by Legion of the Bouncy Castle Inc.,
PO Box 398, Ascot Vale, Vic, 3032, Australia

Table of Contents

Introduction

About this Book

This book is about the Bouncy Castle (BC) Federal Information Processing Standard (FIPS) C-Sharp (C#) Application Programming Interface (API); BC FIPS C# API for short, and how it presents the cryptographic functions/algorithms. It does not directly provide details on the cryptographic algorithms used, unless required in the examples which follow.

The reader is assumed to have an understanding of C# and also to have had some exposure to the principals of cryptography. Having an existing understanding of the System.Security.Cryptography (both the Microsoft Windows .NET Framework API and the cross-platform .NET 5+ versions) is useful in following the examples below – however, the **bc-fips-1.0.2.dll** assembly is standalone and most of the interface "standards" follow the System.Security.Cryptography provider.

The examples are not meant to be definitive, but they should provide you with a good overview of what can be done with the BC FIPS provider and its associated APIs. You may run into a situation where the example shown does not quite fit with you want to do – hopefully a look around at the other classes referenced in the same namespaces used in the example will get you there. For simplicity the examples do not include the using directive statement, but you can find the full source for them as well as a user guide at https://www.bouncycastle.org/fips-csharp.

The examples below make use of some pre-defined sample values as well, these are defined in a class called ExValues. If you are following the examples, you can safely make up your own values, but to avoid them becoming a possible point of concern they are defined below:

```
public class ExValues
{
    public static readonly long thirty_days = 1000L * 60 * 60 * 24 * 30;

    public static readonly byte[] sampleAesKeyInput = Hex.Decode("000102030405060708090a0b0c0d0e0f");
    public static readonly FipsAes.Key sampleAesKey = new FipsAes.Key(sampleAesKeyInput);

    public static readonly byte[] sampleTripleDesKeyInput =
        Hex.Decode("000102030405060708090a0b0c0d0e0f1011121314151617");
    public static readonly SymmetricSecretKey sampleTripleDesKey =
        new FipsTripleDes.Key(sampleTripleDesKeyInput);

    public static readonly byte[] sampleSha512MacKeyInput =
        Hex.Decode("000102030405060708090a0b0c0d0e0f10111213");
    public static readonly SymmetricSecretKey sampleSha512MacKeyInput =
        new(FipsShs.Sha512HMac,sampleSha512MacKeyInput);

    public static readonly byte[] sampleIVnonce = Strings.ToByteArray("0123456789123456");

    public static readonly byte[] sampleIVcounter = Strings.ToByteArray("012345678912");

    public static readonly byte[] sampleInput = Strings.ToByteArray("Hello World!");

    public static readonly byte[] sampleTwoBlockInput =
        Strings.ToByteArray("Some cipher modes require more than one block");

    public static readonly byte[] sampleNonce = Strings.ToByteArray("number only used once");

    public static readonly byte[] personalizationString =
        Strings.ToByteArray("a constant personal marker");
```

```
public static readonly byte[] additionalAuthenticatedDataA =
    Strings.ToByteArray("This message was sent 29th Feb at 11.00am - does not repeat");
public static readonly byte[] additionalAuthenticatedDataB =
    Strings.ToByteArray("This message was sent 30th Feb at 11.00am - does not repeat");

public static readonly byte[] sampleSignatureMessage =
    Strings.ToByteArray("Once Upon a midnight dreary, while I pondered weak and weary");

public static readonly byte[] initiator = Strings.ToByteArray("Initiator");
public static readonly byte[] recipient = Strings.ToByteArray("Recipient");
public static readonly byte[] sampleUKM = Strings.ToByteArray("User keying material");

public static readonly byte[] sampleUnpaddedKey = Strings.ToByteArray("this is my secret key123");
public static readonly byte[] samplePaddedKey = Strings.ToByteArray("this is my secret key123456");

public static readonly BigInteger dhP =
    new BigInteger("b1793382ef28b54fb6715857c556dc8ced00dc32" +
                   "6acccb89b897cf348c6331881d7890675a67acf5" +
                   "f82c9c196924fd50db59f138602e065a018c4521" +
                   "3ec88c355d2a419b75cd32e186132cbda13da9df" +
                   "23cded3b3aa9cdda2127f98226feeb055dfeb549" +
                   "c6e7beb14d1c8a2b30b81377185468a0c05c6aa3" +
                   "410fd746ce38691d92cd08c5e08b959a16f5b33d" +
                   "3a6ec71af88397cac8ed3296ac19b98fccac57a3" +
                   "864b3a6f20873cd09f22aa66d049c20729f5e99d" +
                   "3be168cd4a0438072f868305af5452537cc462b5" +
                   "e6d9a65f9e19909cad2b37e3ad018dc954ab65b9" +
                   "1b5c42551e5fe8bad45a0b555c0afd533f98f0b1" +
                   "57e07f24dc2b9a6b465e05b029e79af1", 16);
public static readonly BigInteger dhQ =
    new BigInteger("cf3c1ad8120c10f03c63383a841f0f34c28e0c968f7aad9422715df1", 16);
public static readonly BigInteger dhG =
    new BigInteger("13cfe47aa686e1c63d029ffd6327bfa543c7fe94" +
                   "8f5dc6e64c0eaeeba521fa1e01ab56401d438b04" +
                   "06ecd134f53aaa6a64fbfa687842c6c38b973a8e" +
                   "da72b3e132f38a9aef65e368b434e2c5383baac8" +
                   "efe4d5635097cd74d6b0594c3509217da5990d75" +
                   "fe4a15709992a155a7ecbbc1bae6927a6c357e90" +
                   "910265d8902d864abd7b0a1e2cce2f42da88639a" +
                   "9329fe1faff8cc19938de37a94226fb63a3f8c1f" +
                   "b3fead755d181df4711322ae006ddc0956855ba6" +
                   "53d7f8f8636d3c3c6a2b146d321a634245b20eb4" +
                   "56e044a8fbd3d7a96e096b51a23aee1c8c162d91" +
                   "670b9d634126e626a93c3a2fc27f0d7e7865dd02" +
                   "56b12068d20f2e2f975d74e2ccbd42ac", 16);
```

Why FIPS 140?

The Federal Information Processing Standards (FIPS) 140 standards were originally put together in 1994, with a further revision, the FIPS 140-2, being released in 2001 (coincidentally on the day of Bouncy Castle's first birthday, May 25th). On March 22nd 2019 FIPS 140-3 (which now standardizes on the ISO 19790:2012 and ISO 24759:2017 specifications) was approved and in September of that year the standards were released. As of April 1st, 2022, FIPS 140-3 security requirements for Cryptographic Modules supersedes FIPS 140-2 for new submissions. However, the FIPS 140-2 standards still form the basis of the requirements for many applications involved in the transmission of sensitive data in all US Government Departments and agencies. The validation program is known as the CMVP (Cryptographic Module Validation Program) and it is managed by the National Institute of Standards and Technology (NIST). There are a lot more acronyms that could follow as well!

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

For us at the Legion of the Bouncy Castle, in trying to produce and maintain a sound cryptography API and in trying to find some independent way of validating the API, the FIPS 140-2 (and in future the FIPS 140-3) certification process was the most obvious choice.

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

Sadly, any assurance about the quality of the API does not make the product idiot proof. A FIPS 140-2 (or FIPS 140-3) API really comes in two parts. The first part is the actual code, as in the C# cryptographic assembly **bc-fips-1.0.2.dll** file you use, and the second part is a document called the Security Policy. The Security Policy is important as it is the other thing that the CMVP sign off on. The Security Policy dictates how the security module should be used, what it supports, and what guidance is required for the safe use of the module in “approved mode”. The Security Policy document can be a little awkward to read at first, partly because the language it uses comes from FIPS 140's origins as primarily a hardware oriented standard. The reading of the Security Policy is worth persisting with though.

When a thread in the BC FIPS API is running in “approved mode” only FIPS approved algorithms are accessible to it. While this provides some level of confidence to a developer or to a code reviewer, as to how cryptographic services are handled by an application, there are still a variety of ways to shoot

yourself in the foot, for example: using the digest algorithm “SHA-1”, or using the same RSA key for signing certification requests and then encryption, the FIPS standards allow only specific uses – uses which are impossible to monitor from the APIs point of view, but mean something, both from the point of view of an auditor, and also from the point of view of how secure your usage of the API really is. So, read the Security Policy, save your bullets for the problems facing your project, rather than wasting them on your foot. Sanity check what you are doing as well, perfection is difficult to achieve, that includes the authors of the BC FIPS APIs. It never hurts to improve one's knowledge, and the application of cryptography is definitely an area where additional knowledge will help to avoid disaster.

And Finally...

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

Getting Started

This chapter provides a quick look at set up and configuration of the Bouncy Castle FIPS C# cryptographic assembly **bc-fips-1.0.2.dll**. In what follows we refer to this cryptographic assembly as the “Provider” or “Bouncy Castle Fips Provider”.

Assembly Installation

The Bouncy Castle FIPS provider can either be installed via the Global Assembly Cache (GAC) or used directly during execution. You will need to find out the process involved for the **bc-fips-1.0.2.dll** assembly if you intend to install in the GAC or other known assembly path.

You can access the provider directly at compile time and make this available to your application. Under Linux ensure you have the correct version of ***dotnet*** installed. Your dotnet framework should come with the ***msc*** (Mono C# compiler). Create your application and then use msc to compile the application to use with the assembly. Under MS Windows, the setup is similar except you will use the ***csc.exe*** compiler instead. The compiler will produce an executable which can access the assembly during execution of your application.

Finally

The provider **bc-fips-1.0.2.dll** assembly itself has no external dependencies, but it cannot be used in the same assembly as the regular Bouncy Castle (non-FIPS) provider. The classes in the two assemblies files do not get along.

Random Numbers

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

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

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

The Bouncy Castle FIPS API provides implementations of all three. In the examples which follow we will use the HMAC SHA512 based DRBG. In terms of setup steps involved, the other two implementations are similar, and as the HMAC SHA512 one is a DRBG with 256 bits of security associated with it, it is a good one to look at.

Creating DRBG Based SecureRandoms

NIST approaches the problem of providing random bits (an entropy source) to an application by the use of a DRBG (also known as pseudorandom number generators PRNG, or Non-deterministic Random Bit Generators NRBGs). In simple terms a (NIST) DRBG is constructed from a validated entropy source (Random Bit Generator) and then using a mixing function to stretch out the number of random bits.

In the following examples, we are creating a DRBG which might be suitable for use with the creation of initialisation vectors (IVs) or other similar random data, such as nonces.

Example 1 – Creating a FIPS Approved SecureRandom

```
public static FipsSecureRandom buildDrbg()
{
    BasicEntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    IDrbgBuilder<FipsSecureRandom> theDrbgBuilder = CryptoServicesRegistrar.CreateService(
                                            FipsDrbg.Sha512HMac).FromEntropySource(entSource);
    theDrbgBuilder.SetSecurityStrength(256);
    theDrbgBuilder.SetEntropyBitsRequired(256);

    return theDrbgBuilder.Build(ExValues.sampleNonce,false);
}
```

In this example, the GenerateSeed() method of a regular SecureRandom object is used as the entropy source – this is implementation dependent and the OS used. Under Windows version 8+, SecureRandom is based upon the RNGCryptoServiceProvider – which is used for the initial entropy. Under Linux, the /dev/random is used but is .NET (mono) implementation dependent. This should make use of whatever entropy source the **bc-fips-1.0.2.dll** assembly running on.

The first thing to note in the example above is that SecureRandom is inherited from System.Random. Then the SecureRandom returned is an extension of the regular SecureRandom class called FipsSecureRandom. In this case an extension class is necessary as a NIST DRBG requires a method such as FipsSecureRandom.Reseed() which are not available on SecureRandom.

The second thing to note is the **`false`** parameter value on the theDrbgBuilder.Builder.Build() method (in the nested class FipsDrbg.Builder). This second paramerter refers to the “prediction resistance” required of the constructed DRBG (Refer to Section 8.8 in NIST SP 800-90A Revision 1.) In this case we are willing to assume that the DRBG function will do a good job producing a random stream and that's enough. In the case of keys or components of keys we need a higher standard to be reached so we set “prediction resistance” to true as in the following example.

Example 2 – Creating a FIPS Approved SecureRandom for Keys

```
public static FipsSecureRandom buildDrbgForKeys()
{
    BasicEntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    IDrbgBuilder<FipsSecureRandom> theDrbgBuilder = CryptoServicesRegistrar.CreateService(
                                            FipsDrbg.Sha512HMac).FromEntropySource(entSource);
    theDrbgBuilder.SetSecurityStrength(256);
    theDrbgBuilder.SetEntropyBitsRequired(256);
    theDrbgBuilder.SetPersonalizationString(ExValues.personalizationString);

    return theDrbgBuilder.Build(ExValues.sampleNonce,true);
}
```

Observe that, there are two obvious differences between the construction of the of the DRBG SecureRandom in Example 1 and the construction of the DRBG SecureRandom in Example 2. The first difference is the passing **`true`** as the parameter value for prediction resistance and the second difference is the use of the SetPersonalizationString method and passing the value ***`personalizationString`***. The personalization string is an example “hedging ones bet” so as to reduce the likelihood of two DRBGs somehow producing the same key stream where the entropy source turns out to be similar. As a value, the personalization string can be secret, although needn't be. Primarily it needs to be unique. Section 8.7.1 SP 800-90A Revision 1 discusses this further, and also the topic of other fields, one called additional input, which can also be used to enhance the basic entropy being provided to the DRBG on a reseed, and the nonce which helps further distinguish the DRBG.

Configuring a Default SecureRandom

The .NET framework provides the non-cryptgraphic secure class **Random** and the cryptographic secure abstract class **RandomNumberGenerator**. In the case where the default:

System.Security.Cryptography.RNGCryptoServiceProvider

provider is not used, it is difficult for the API to work out what to use for a source of randomness.

Examples of where unexpected randomness requirements can come up include things like RSA blinding and, in the case of the BC FIPS API, rather than relying on defining “new SecureRandom()” in these non-obvious situations, the API executes a call to CryptoServicesRegistrar.GetSecureRandom() which will throw an exception if a default has not been provided. A default SecureRandom can be set using CryptoServicesRegistrar.setSecureRandom as shown below.

Example 3 – Configuring the Default SecureRandom

```
public static void setDefaultDrbg()
{
    BasicEntropySourceProvider entSource = new BasicEntropySourceProvider(new SecureRandom(), true);
    IDrbgBuilder<FipsSecureRandom> theDrbgBuilder = CryptoServicesRegistrar.CreateService(
                                                FipsDrbg.Sha512HMac).FromEntropySource(entSource);
    theDrbgBuilder.SetSecurityStrength(256);
    theDrbgBuilder.SetEntropyBitsRequired(256);
    CryptoServicesRegistrar.SetSecureRandom(theDrbgBuilder.Build(ExValues.Nonce, true));
}
```

Symmetric Key Encryption

The BC FIPS API offers both approved mode symmetric ciphers, AES and TripleDES, and also a number of other symmetric ciphers that appear in IETF (Internet Engineering Task Force) and ISO standards such as ARC4, Camellia, SEED, and Serpent.

The basic (or fundamental) place to start with symmetric key encryption is the generation of actual keys. Keys for symmetric ciphers are simply bit strings, often without structure (DES and TripleDES are an exception here). Broadly a key's strength is related to its bit length so it is common to see APIs referring to keys by bits involved rather than bytes.

Key Generation

Generating an AES key using the BC FIPS API is (almost) straight forward. The API provides a list of KeyGenerationParameters objects with their corresponding key sizes.

Example 4 – Generating an AES Key

```
public static SymmetricSecretKey generateKey()
{
    FipsAes.KeyGenerator myKeyGen = CryptoServicesRegistrar.CreateGenerator(
                                        FipsAes.KeyGen256, new SecureRandom());
    return myKeyGen.GenerateKey();
}
```

In the example example above, we have explicitly provided a “new” SecureRandom source. Note however, that we could have used our previously defined default SecureRandom (as in Example 3 above). The statement below shows how our AES key was generated using the default SecureRandom object previously stored:

```
FipsAes.KeyGenerator myKeyGen = CryptoServicesRegistrar.CreateGenerator(FipsAes.KeyGen256);
```

Key Construction

Sometimes it is necessary to simply construct a key from a byte array that has been provided. The example below provides the simplest way to do this using the BC FIPS API.

Example 5 – Key Construction with a given SecretKey

```
public static SymmetricSecretKey defineKey(byte[] keyBytes)
{
   FipsAes.Key myAESKey = new FipsAes.Key(keyBytes);
   return myAESKey;
 }
```

In the example above, the result of the defineKey() method is a valid (not necessarily secure) key that can then be used anywhere a generated AES key would work. Also, note that the length of the byte array **`keyBytes`** is checked and must be one of the integers 16, 24 or 32.

Basic Modes and Padding

Having created a key, there are a variety of ways it can be used to encrypt things. All approaches have certain advanatages and disadvanatages – which encryption method to choose from will depend on what you are doing and what outcome expectations you have. Doing a bit of research that's relevant to your application is a good idea here, as blanket statements you may come across such as “mode X is the only way to encrypt something” aren’t generally helpful.

Encryption algorithms (or modes) can be divided into two categories based on the input type: either a block cipher or a stream cipher. We will be mainly considering block ciphers in the examples which follow.

The most basic mode is ECB, Electronic Code Book mode. (Other AES block cipher modes or schemes such as CBC – Cipher Block Chaining, CFB – Cipher FeedBack, OFB – Output FeedBack CTR – Counter are discussed further below.) The example below shows methods using ECB mode for encryption and decryption. Stock standard ECB mode, like many block cipher modes is unpadded so the input has to be aligned on the block boundaries of the cipher - in this case 128 bits since we are using AES. In the example we have also made use of padding to get around this restriction, so rather than specifying “NoPadding” we have specified a specific format of padding “PKCS7Padding” to allow for non-block aligned data. PKCS7Padding is often also referred to as (or confused with) PKCS5Padding. (PKCS5Padding was originally designed for 64 bit block sizes PKCS7Padding for arbitrary block sizes. The BC API uses PKCS7Padding.) In addition, BC APIs provide a number of other padding mechanisms such as ISO10126-2, X9.23, ISO7816-4, and TBC (trailing bit compliment) padding.

Example 6 – ECB Mode Encryption/Decryption

```
public static byte[] ecbEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateBlockEncryptorBuilder(FipsAes.Ecb);

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher pkcs7PaddedEncryptor = encryptBuilder.BuildPaddedCipher(streamOutput,new Pkcs7Padding());

    using (Stream encrypterStream = pkcs7PaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] ecbDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateBlockDecryptorBuilder(FipsAes.Ecb);

    ICipher pkcs7PaddedDecryptor = decryptBuilder.BuildPaddedCipher(
           new MemoryInputStream(cipherTextData),
           new Pkcs7Padding());

    byte[] plainTextData;
    using (Stream decrypterStream = pkcs7PaddedDecryptor.Stream)
```

```
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

In the BC API, most of the Block Cipher modes operate as in the above example. However, ECB mode is obviously simpler compared to the examples which follow below – for example no Initialisation Vector (IV) is required in ECB mode. However, ECB mode leaks information about the plaintext because identical plaintext blocks produce identical ciphertext blocks. To observe such a phenomenon, create an image with the large secret message text “THIS IS A SECRET” black on a white background. Encrypt the image using ECB mode as in the example above – your image will leak information and you should be able to read the secret message. This is due to the fact that ECB mode is subject to frequency analysis. In this next example we are using CBC (Cipher Block Chaining) mode.

Example 7 – CBC Mode Encryption/Decryption

```
public static byte[] cbcEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateBlockEncryptorBuilder(FipsAes.Cbc.WithIV(ExValues.sampleIVnonce));

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher pkcs7PaddedEncryptor = encryptBuilder.BuildPaddedCipher(streamOutput,new Pkcs7Padding());

    using (Stream encrypterStream = pkcs7PaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] cbcDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateBlockDecryptorBuilder(FipsAes.Cbc.WithIV(ExValues.sampleIVnonce));

    ICipher pkcs7PaddedDecryptor = decryptBuilder.BuildPaddedCipher(
           new MemoryInputStream(cipherTextData),
           new Pkcs7Padding());

    byte[] plainTextData;
    using (Stream decrypterStream = pkcs7PaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

Like ECB mode, CBC (Cipher Block Chaining) mode is block aligned so we need to specify padding. CBC mode also has an extra parameter, the initialisation vector (IV), which is used with the mode to prevent any obvious similarities that might have existed in two plain texts from showing up in the encrypted results. In the case of the example above we are passing the IV directly to the Cipher Builder object for encryption and decryption. Take care to make sure the IV is reliably random or unique if you do this, as otherwise you will end up back with ECB mode. Also note that, the IV must be the same size as the block that is being encrypted – an exception will occur if this is not the case. However, if we pass a FipsSecureRandom object, then the IV is generated for us. For example:

```
IBlockCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                    provider.CreateBlockEncryptorBuilder(FipsAes.Cbc.WithIV(buildDrbg()));
```

To recover the “unknown” IV (in the case the IV is generated for us using a FipsSecureRandom object) we can use:

```
byte[] cipherIV = (FipsAes.ParametersWithIV)encryptBuilder.AlgorithmDetails.GetIV();
```

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

The most basic block streaming mode is CFB (Cipher FeedBack) mode.

Example 8 – CFB Mode Encryption/Decryption

```
public static byte[] cfbEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    ICipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateEncryptorBuilder(FipsAes.Cfb128.WithIV(ExValues.sampleIVnonce));

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher noPaddedEncryptor = encryptBuilder.BuildCipher(streamOutput);

    using (Stream encrypterStream = noPaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] cfbDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    ICipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateDecryptorBuilder(FipsAes.Cfb128.WithIV(ExValues.sampleIVnonce));

    ICipher noPaddedDecryptor = decryptBuilder.BuildCipher(
                new MemoryInputStream(cipherTextData));

    byte[] plainTextData;
    using (Stream decrypterStream = noPaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

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

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

Example 9 – CTR Mode Encryption/Decryption

```
public static byte[] ctrEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    ICipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateEncryptorBuilder(FipsAes.Ctr.WithIV(ExValues.sampleIVcounter));

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher noPaddedEncryptor = encryptBuilder.BuildCipher(streamOutput);

    using (Stream encrypterStream = noPaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] ctrDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    ICipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateDecryptorBuilder(FipsAes.Ctr.WithIV(ExValues.sampleIVcounter));

    ICipher noPaddedDecryptor = decryptBuilder.BuildCipher(
                new MemoryInputStream(cipherTextData));

    byte[] plainTextData;
    using (Stream decrypterStream = noPaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

In example 9 above, observe that we have initialized the cipher with an IV less than the block size of the cipher (in this case 16 bytes). The implementation of this mode operates on having the last m bytes of the IV as the counter. In the example above we have allocated 12 bytes to the nonce (which in real life should be random) and 4 bytes to the counter. Thus in this example the maximum length of a message we could encrypt is 2^{32} blocks (or 2^{36} bytes). The BC API will throw an exception if the initialisation vector greater than 16 bytes.

The last example in this section shows how to use Ciphertext Stealing (CTS). NIST provides three definitions of cipher text stealing in an addendum to NIST SP 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”. CTS is used in conjunction with CBC mode and can be used where there are at least 2 blocks of data and has the advantage that it requires no padding, as the “stealing” process allows it to produce a cipher text which is the same length as the plain text. These variants are denoted CBC-CS1, CBC-CS2, and CBC-CS3, where “CS” indicates ciphertext stealing. The variants differ only in the ordering of the ciphertext bits. The most popular one is CS3 (where the last two encrypted blocks are interchanged), which is the same as the CTS mode described in RFC 2040, and is the one shown in example 10. Observe that we are using CBC mode and therefore padding is required.

Example 10 – CBC Mode With Ciphertext Stealing (CTS)

```
public static byte[] ctsWithCS3Encrypt(FipsAes.Key key, byte[] plainTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateBlockEncryptorBuilder(
                    FipsAes.CbcWithCS3.WithIV(ExValues.sampleIVnonce));

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher pkcs7PaddedEncryptor = encryptBuilder.BuildPaddedCipher(streamOutput,new Pkcs7Padding());

    using (Stream encrypterStream = pkcs7PaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] ctsWithCS3Decrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IBlockCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateBlockDecryptorBuilder(
                    FipsAes.CbcWithCS3.WithIV(ExValues.sampleIVnonce));

    ICipher pkcs7PaddedDecryptor = decryptBuilder.BuildPaddedCipher(
           new MemoryInputStream(cipherTextData),
           new Pkcs7Padding());

    byte[] plainTextData;
    using (Stream decrypterStream = pkcs7PaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

Authenticated Modes

One of the issues with the encryption modes discussed above, is that there is no mechanism in place to pick up actual errors or tampering attempts on decryption, other than perhaps getting back garbage. One of the main issues with the modes above is that they are susceptible to a replay attack. For example, assume both Alice and Bob have their shared secret key. Alice sends Bob the message “I will do the weekly shopping today” – Bob who decrypts the message is happy with this arrangement so he can spend more time doing cryptography at work. Eve (not happy with Bob) though, has been listening and next week resends the same message to Bob. Result: Bob is doing more cryptography but no food in the house for that week with Alice hungry and unhappy. This is an example of a “replay attack”. The use of ***Authenticated Encryption with Associated Data (AEAD)*** can mitigate this type of attack.

Authenticated modes such as GCM (Galois/Counter Mode) and CCM (Counter with Cipher block chaining message authentication Mode or counter with CBC-MAC) also incorporate a ***tag*** that provides a cryptographic checksum that can be used to help validate a decryption. These modes are known as AEAD modes as they also provide for mixing some additional clear text, or associated data, into the tag used for validation. The BC FIPS C# API also includes EAX (encrypt-then-authenticate-then-translate), as an AEAD mode, but this algorithm is not available in the approved mode of operation and is currently not thread-safe.

The first of the authenticated modes we will look at is GCM, is described in NIST SP 800-38D, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC" – 2007 and is still valid. This is based on CTR mode, so there is a "counter" involved (as explained in Example 9 above) as well as having its own hashing function incorporated into it. In this GCM example we will also provided ***Additional Authenticated Data*** (AAD), specifically we use: ***ExValues.AdditionalAuthenticatedDataA.*** Note that the clear text data is optional for GCM and CCM described below. Thus in the example below using ***ExValues.AdditionalAuthenticatedDataA***, we could use this message (unencrypted) to help with authenticating the encrypted message. AAD can also be used for routing information and also as a sequence number. Note also, that using AAD the cipher need not be decrypted before deciding whether the message is valid or not. Finally, AAD is optional as in Example 12 below.

Example 11 – GCM Mode Encryption/Decryption

```
public static byte[][] gcmEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateAeadEncryptorBuilder(FipsAes.Gcm.WithIV(ExValues.sampleIVcounter));

    MemoryOutputStream streamOutput = new MemoryOutputStream();

    IAeadCipher noPaddedEncryptor = encryptBuilder.BuildAeadCipher(AeadUsage.AAD_FIRST, streamOutput);
    noPaddedEncryptor.AadStream.Write(ExValues.additionalAuthenticatedDataA,0,
                                      ExValues.additionalAuthenticatedDataA.Length);

    using (Stream encrypterStream = noPaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[][] cipherMacData = { streamOutput.ToArray(), noPaddedEncryptor.GetMac().Collect() };

    return cipherMacData;
}

public static byte[][] gcmDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateAeadEncryptorBuilder(FipsAes.Gcm.WithIV(ExValues.sampleIVcounter));

    IAeadCipher noPaddedDecryptor = decryptBuilder.BuildAeadCipher(AeadUsage.AAD_FIRST,
                new MemoryInputStream(cipherTextData));
    noPaddedDecryptor.AadStream.Write(ExValues.additionalAuthenticatedDataA,0,
                                      ExValues.additionalAuthenticatedDataA.Length);

    byte[] plainTextData;
    using (Stream decrypterStream = noPaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    byte[][] plainMacData = { plainTextData, noPaddedDecryptor.GetMac().Collect() };

    return plainMacData;
}
```

In the example above, we have specified a tag length of 128 bits (the maximum) by default (This maximum is placed upon the GCM implementation which uses a field of size 2^{128} elements for "multiplication" hashing.) According to the NIST SP 800-38D publication, the tag length can be one of 128, 120, 112, 104, 96. Lengths of 64 and 32 bits are also available but are not recommended for use.

We could have specified a shorter tag length (say 96 bits) by using:

```
IAeadCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateAeadEncryptorBuilder(
                    FipsAes.Gcm.WithIV(ExValues.sampleIVcounter).WithMacSize(96));
```

However, it is recommended that the default tag length of 128 is always used.

Note that, if using a shorter tag, only the left most significant bits of the specified length are returned. Also observe that we have used a 12 byte nonce, giving a counter size of 4 bytes. If you are planning to use GCM it is worth having a look at NIST SP 800-38D for guidance, poor choice of IVs can cause huge problems with this mode and it is not recommended to use a lower tag size unless you really know what you are doing – therefore stick to the default tag size as in the example above. Notice that we have provided the ***AAD*** as inputs to both the encryption and decryption. The BC FIPS API will throw an exception if the tag generated upon decryption does not match the tag generated upon encryption. This exception works because the cipher output from encryption is joined with the tag. This means that if the length of the output is ***n***, the actual the cipher is ***n – m*** bits, where ***m*** is the length of the tag. By the way, this implies that we need not separately store the tag as a separate output from encryption, and we need not input the tag separately for decryption.

The other AEAD mode available is CCM which is defined in NIST SP 800-38C. In terms of set up it is very similar to GCM. CCM, as defined by NIST, is also built on CTR mode, but in this case makes use of a CBC-MAC for the checksum.

Example 12 – CCM Mode Encryption/Decryption

```
public static byte[] ccmEncrypt(FipsAes.Key key, byte[] plainTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateAeadEncryptorBuilder(FipsAes.Ccm.WithIV(ExValues.sampleIVcounter));

    MemoryOutputStream streamOutput = new MemoryOutputStream();
    ICipher noPaddedEncryptor = encryptBuilder.BuildCipher(streamOutput);

    using (Stream encrypterStream = noPaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();
    return cipherTextData;
}

public static byte[] ccmDecrypt(FipsAes.Key key, byte[] cipherTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateAeadDecryptorBuilder(FipsAes.Ccm.WithIV(ExValues.sampleIVcounter));

    ICipher noPaddedDecryptor = decryptBuilder.BuildCipher(new MemoryInputStream(cipherTextData));

    byte[] plainTextData;

    using (Stream decrypterStream = noPaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

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

Example 13 – CCM Encryption/Decryption With AAD

```
public static byte[] ccmEncryptAAD(FipsAes.Key key, byte[] plainTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> encryptBuilder =
                provider.CreateAeadEncryptorBuilder(FipsAes.Ccm.WithIV(ExValues.sampleIVcounter));

    MemoryOutputStream streamOutput = new MemoryOutputStream();

    IAeadCipher noPaddedEncryptor = encryptBuilder.BuildAeadCipher(AeadUsage.AAD_FIRST, streamOutput);
    noPaddedEncryptor.AadStream.Write(ExValues.additionalAuthenticatedDataA,0,
                                      ExValues.additionalAuthenticatedDataA.Length);

    using (Stream encrypterStream = noPaddedEncryptor.Stream)
    { encrypterStream.Write(plainTextData,0,plainTextData.Length); }

    byte[] cipherTextData = streamOutput.ToArray();

    return cipherTextData;
}

public static byte[] ccmDecryptAAD(FipsAes.Key key, byte[] cipherTextData)
{
    IAeadCipherService provider = CryptoServicesRegistrar.CreateService(key);
    IAeadCipherBuilder<IParameters<Algorithm>> decryptBuilder =
                provider.CreateAeadDecryptorBuilder(FipsAes.Ccm.WithIV(ExValues.sampleIVcounter));

    IAeadCipher noPaddedDecryptor = decryptBuilder.BuildAeadCipher(AeadUsage.AAD_FIRST,
                new MemoryInputStream(cipherTextData));
    noPaddedDecryptor.AadStream.Write(ExValues.additionalAuthenticatedDataA,0,
                                      ExValues.additionalAuthenticatedDataA.Length);


    byte[] plainTextData;
    using (Stream decrypterStream = noPaddedDecryptor.Stream)
    { plainTextData = Streams.ReadAll(decrypterStream); }

    return plainTextData;
}
```

One final observation should be mentioned regarding block and stream symmetric ciphers. As previously stated, the BC API generates an exception if the tag validation from the decryption algorithm fails. Depending on the application, it may happen that part of the cipher has already been decrypted (into plain text or garbage) prior to the generation and validation of the tag. As explicitly mentioned in the NIST documentation, no part of the decrypted data should be sent to the end user until the tag has been validated.

Message Digest, MACs, and HMACs

Prior to the introduction of authenticated modes with ciphers the only way to tell if decrypted data (or for that matter, plain text data) was correct and untampered was to have a separate message digest or ***Message Authentication Code (MAC)*** associated with it. Message digests can also provide a valuable tool for constructing signatures. (Note however, that MACs are based upon symmetric keys, therefore documents can not "shared" across third-parties for authentication – unlike digital signatures using public keys). Nevertheless, MACs and ***Hash-based Message Authentication Codes (HMAC)*** are still relevant as there are still plenty of cases where messages need to be tamper resistant, even when the content in them might be publicly readable.

Message Digests

A hash function is essentially a function which maps a large $\{0,1\}^n$ space to a much smaller space $\{0,1\}^m$, where $m \ll n$. Cryptographic hash functions also known as ***message digests***, differ from a regular checksum hashes, such as CRC32 (Cyclical Redundancy Check), in that changing any bit in an input stream to hash calculator has an unpredictable affect on the resulting output. This is in contrast to CRC32 for example, where it can happen that modifying the input can produce a known output.

In contrast, a cryptographic hash or digest, ensures that there is an overwhelmingly computational difficultly of predicting the value of a digest. Further, a digest makes it computational extremely difficult to produce two documents (or messages) which result in the same digest value – this is known as producing a ***collision***. This is fundamental to the idea that if two documents verify to the same digital signature, they are basically the same document (most signature algorithms sign a cryptographic hash rather than the message itself).

As with symmetric ciphers, there has been an evolution in producing digests. For example, SHA-1 and MD5 are no longer recommended for use and existing implementations are getting phased out where they are in use. (FIPS only permits SHA-1 for compliance with existing protocols.) The current recommended digests come from the SHA-2 (described in FIPS PUB 180-4) and the SHA-3 (described in FIPS PUB 202) families of cryptographic hash functions.

The SHA-2 family digests are still regarded as safe, although there is now a trend towards the use of SHA-384 and SHA-512 and its two smaller variants instead of SHA-224 and SHA-256. This is happening largely due to the size of the internal buffer in SHA-224/SHA-256 that is used to store data for the digest calculation and concerns about whether that may be a problem if and when quantum computers start making themselves felt. (The issue of if and when a quantum computer can be a concern for a digest has not been resolved to date – there are some papers which claim better performance than a standard birthday attack on cryptographic hashes.) Irrespective of quantum computers, there is the SHA-3 family which is based on a different approach to that of the SHA-2 family digest construction. The SHA-3 family also includes two ***eXtendable-Output Functions***

(XOFs), SHAKE-128 and SHAKE-256. While the SHA-3 family is different internally, the digests contained in it, produce digests of the same size as the SHA-2 family and they can be used as drop in replacements for each other. The first example shows how to create a simple digest using the SHA-2 and SHA-3 variants that produce 512 bits of output from the input data.

Example 14 – Two Digest Examples

```
public static byte[] sha512Digest(byte[] message)
{
    IDigestFactory<FipsShs.Parameters> provider =
        CryptoServicesRegistrar.CreateService(FipsShs.Sha512);
    IStreamCalculator<IBlockResult> calculator = provider.CreateCalculator();

    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();
    return calculator.GetResult().Collect();
}


public static byte[] sha3_512Digest(byte[] message)
{
    IDigestFactory<FipsShs.Parameters> provider =
        CryptoServicesRegistrar.CreateService(FipsShs.Sha3_512);
    IStreamCalculator<IBlockResult> calculator = provider.CreateCalculator();

    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();
    return calculator.GetResult().Collect();
}
```

The **bc-fips-1.0.2.dll** assembly does include the Sha512_224 and Sha512_256 – meaning that the output will 224 bits and 256 bits respectively. In addition, the BC API also provides Sha3_224, Sha3_256 and Sha3_384. Do not be confused with the output size and the block size that is used internally for these digests. For example, do not expect the first half of the output of a Sha512 hash to be the same as that of Sha512_256. These digests operate on different internal block sizes and thus will provide different output. This is not the case for the XOFs discussed below.

Extendable Output Functions

Extendable output functions (XOFs) are relatively new on the scene of cryptographic hash functions. The first XOFs standardized were announced in the SHA-3 standard (FIPS PUB 202). Primarily, we can probably expect them to be used in things like ***(Key derivation Functions) KDFs*** and possibly to replace existing mask functions (hash primitives which also generate a variable length output). As mentioned, XOFs have the feature producing “almost indefinite length” output while still offering a specific level of security.

Example 15 – Basic Use of an XOF

```
public static byte[] shake256Output(byte[] message, int outputLength = 32)
{
    IXofFactory<FipsShs.XofParameters> provider =
        CryptoServicesRegistrar.CreateService(FipsShs.Shake256);
    IVariableStreamCalculator<IBlockResult> calculator = provider.CreateCalculator();

    Stream messageStream = calculator.Stream;

    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult(outputLength).Collect();
}
```

The BC API also allows for the continuous “squeezing” of the function to produce more output. By the way, such extendable output functions such as Shake128 and Shake256 belong to a class of functions called cryptographic sponge functions. (Called sponge functions since these functions can be pictorially represented as “soaking” the input and “squeezing” the output.) Example 16 below also produces 32 bytes of output but does so by requesting output from the XOF object twice.

Example 16 – Multiple Returns from an XOF

```
public static byte[] shake256OutputContinuous(byte[] message, int outputLength = 16)
{
    IXofFactory<FipsShs.XofParameters> provider =
        CryptoServicesRegistrar.CreateService(FipsShs.Shake256);
    IVariableStreamCalculator<IBlockResult> calculator = provider.CreateCalculator();

    Stream messageStream = calculator.Stream;

    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult(outputLength).Collect().Concat(
                calculator.GetResult(outputLength).Collect()).ToArray();
}
```

If you run Examples 15 and 16 above, for the same input and examine the return value, you will find they produce the same byte stream. What this means is that the length requested as output plays no role in the actually hashing algorithm. By the way, this implies that to make use of such XOFs algorithms for key derivation function KDFs, one must incorporate the length or type of key as part of the message to be hashed. The document NIST PUB 202 “SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions” provides additional details on using the XOFs for KDFs.

Message Digest Based MACs

MACs based on keyed digests can be used to authenticate data. Conceptually, the authentication of the data transmitted (either encrypted or generally as plain text) is authenticated because the two parties have a shared secret which is used to authenticate the transmitted data. Currently there are three FIPS families of approved MAC algorithms: *Keyed-Hash Message Authentication Code (HMAC)*, KECCAK

Message Authentication Code (KMAC) and the Cipher-based message authentication code (CMAC). The most popular method for doing this at the moment is the HMAC, defined in "The Keyed-Hash Message Authentication Code" FIPS PUB 198-1 and RFC 2104. The BC FIPS API currently supports HMAC using the following hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384 and SHA3-512.

As with a symmetric cipher, the first thing required to use a HMAC is a symmetric key. This can be created using a FipsSecureRandom object and then using that object to extract a sequence of bytes representing the desired key. The length of a key which can be used for a MAC can be of any size. However, FIPS documentation does not approve of keys whose length is 80 bits or less. Moreover, having a key whose length exceeds the size of the underlying hash used in the MAC does not add any additional security to the key constructed. (See the NIST documents: NIST PUB 198-1 "The Keyed-Hash Message Authentication Code (HMAC)", NIST Special Publication 800-107 Revision 1 "Recommendation for Applications Using Approved Hash Algorithms" and NIST Special Publication 800-133 Revision 2 "Recommendation for Cryptographic Key Generation".)

In this example we are generating a key of size 256 bits using the SHA512 underlying hash for our HMAC.

Example 17 – HMAC Key Generation

```
public static byte[] generateHmacKey()
{
    FipsSecureRandom  hmacSecureRandom = DrbgExamples.buildDrbgForKeys();
    byte[] hmacKey = new byte[32];
    hmacSecureRandom.NextBytes(hmacKey);
    return hmacKey;
}
```

Notice that the key generation is achieved by first generating a FIPS approved sequence of random bit as in Example 1 above, and then using the FipsSecureRandom to return the necessary random bits as in this Example 17 above. In the example below we have used our symmetric key generated above to use in our generation of a HMAC.

Example 18 – HMAC Calculation

```
public static byte[] calculateHMAC(byte[] message, byte[] hmacKey)
{
    IMacFactoryService  hmacProvider =  CryptoServicesRegistrar.CreateService(
        new FipsShs.Key(FipsShs.Sha512HMac, hmacKey));
    IMacFactory<FipsShs.AuthenticationParameters>  hmacFactory =
        hmacProvider.CreateMacFactory(FipsShs.Sha512HMac);

    IStreamCalculator<IBlockResult> calculator = hmacFactory.CreateCalculator();
    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult().Collect();
}
```

Symmetric Cipher Based MACs

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

The first one we will look at is CMAC, defined in NIST SP 800-38B. CMAC can be used with both 128 bit and 64 bit block ciphers, so it can be used with Triple-DES as well as AES. In terms of initialisation its the same as with a HMAC – only a key is required. Observe however, that the underlying algorithm used is AES-CBC without the need for an initialisation vector – however we will require an AES key as in the AES examples above. Finally take note of the the obvious that in all of these cipher based MACs, there is no inverse process of decryption.

Example 19 – MAC Calculation using CMAC

```
public static byte[] cacluateCMAC(byte[] aesKeyInput, byte[] message)
{
    IMacFactoryService  cmacProvider =  CryptoServicesRegistrar.CreateService(
        new FipsAes.Key(FipsAes.CMac, aesKeyInput));

    IMacFactory<FipsAes.AuthenticationParameters>  cmacFactory =
        cmacProvider.CreateMacFactory(FipsAes.CMac.WithMacSize(128));

    IStreamCalculator<IBlockResult> calculator = cmacFactory.CreateCalculator();

    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult().Collect();
}
```

GMAC is a little different. It is defined in SP 800-38D, the same document that defines GCM. In this case the MAC also requires an IV and it should be noted that as it is really a specialisation of GCM without encrypted data, all constraints on GCM, such as the need for uniqueness of IVs, also apply to GMAC. That is, GMAC is the authenticating the ***Additional Authenticated Data*** (AAD) and not any of the encrypted data – in fact using Example 11 above without any plain text input will provide the same tag as the example below (of course using the same IV).

Example 20 – MAC Calculation using GMAC

```
public static byte[] calculateGMAC(byte[] message, byte[] aesKeyInput)
{
    IMacFactoryService  gmacProvider =  CryptoServicesRegistrar.CreateService(
        new FipsAes.Key(FipsAes.GMac, aesKeyInput));

    IMacFactory<FipsAes.AuthenticationParametersWithIV>  gmacFactory =
        gmacProvider.CreateMacFactory(FipsAes.GMac.WithIV(ExValues.sampleIVcounter).WithMacSize(128));

    IStreamCalculator<IBlockResult> calculator = gmacFactory.CreateCalculator();

    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult().Collect();
}
```

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

Example 21 – MAC Calculation using CCM

```
public static byte[] calculateCBCMAC(byte[] message, byte[] aesKeyInput)
{
    IMacFactoryService  ccmProvider =  CryptoServicesRegistrar.CreateService(
        new FipsAes.Key(FipsAes.Ccm, aesKeyInput));

    IMacFactory<FipsAes.AuthenticationParametersWithIV>  ccmFactory =
        ccmProvider.CreateMacFactory(FipsAes.Ccm.WithIV(ExValues.sampleIVcounter).WithMacSize(128));

    IStreamCalculator<IBlockResult> calculator = ccmFactory.CreateCalculator();

    Stream messageStream = calculator.Stream;
    messageStream.Write(message, 0, message.Length);
    messageStream.Close();

    return calculator.GetResult().Collect();
}
```

Finally, it should also be noted that for both GMAC and CCM, the API provides that the size of the MAC tag can be specified as in the examples above. However, an exception will occur if the specified MAC tag size exceeds the block size of the symmetric cipher.

Digital Signatures

A (secure) digital signature is a set of cryptographic algorithms (or set of cryptographic tools) which are used together to sign messages and to verify the authenticity of the signature. In simple terms a digital signature provides:

Message Authentication – proof that a certain known entity (the secret key owner, for example) has created and signed the message.

Message integrity – proof that the message received by an entity has not been altered after the signing.

Non-repudiation - the signer of a message cannot repudiate (that is, to refuse to acknowledge) the signing of the message after the signature of the message has been created. Another way of saying this according to NIST is "the recipient of signed data can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory. This is known as non-repudiation, since the signatory cannot easily repudiate the signature at a later time."

It's interesting to note that even with a lot of the arguments that go on about encryption and what sort of access people should have to encryption technology, no-one (to our knowledge) has ever argued the case that people do not need to be able to produce digital signatures. You could almost say that digital signing keeps the world turning. Today, digital signatures are extensively used in all manner of business transactions, for example authorizing money transfers (banks), exchanging sensitive electronic documents, signing transactions in the public blockchain systems (e.g. bitcoin transactions) and of course your internet browser using *https*.

Having said all of that above, this document is about creating a digital signature and how to verify the signature of a signed message. This document does not explicitly address how digital signatures are used to identify who a person or entity is, that created a certain signature. The identification issue can be solved in combination with a digital certificate which binds a public key owner with the identity (person, organization, web site or other). By design digital signatures bind messages to public keys, not to digital identities. Nor do we dwell into the vexing problems associated with asymmetric-key cryptography — the problem of key distribution – this is the realm of Public Key Infrastructure (PKI).

The main document in the FIPS world concerning digital signing appears under the FIPS PUB 186 banner and is now at FIPS PUB 186-4. In practice, due to the long life of some signatures you may also find yourself verifying signatures produced under FIPS PUB 186-2 and FIPS PUB 186-3. Currently at the time of writing, FIPS PUB 186-4 has been superseded by FIPS PUB 186-5. The FIPS PUB 186-4 while relevant today will be withdrawn early February 2024. FIPS PUB 186-5 came into play early February 2023.

In essence please look at both FIPS PUB 186-4 and FIPS PUB 186-5 documents.

FIPS PUB 186-4 discuss approaches to digital signing based around three algorithms: DSA, RSA, and the Elliptic Curve DSA equivalent, ECDSA. (The DSA and ECDSA are here described as equivalent only in the sense of the "security" posed by the known mathematical difficulty of solving the ***discrete log problem***.) There are also some variations on how signatures can be done in RSA. The BC FIPS C# APIs offer support for all the algorithms detailed. The replacement FIPS PUB 186-5 adds and additional set of digital signatures using the Edwards-Curve Digital Signature Algorithm (EDDSA). These are also available in the BC FIPS C# API. We will look at DSA first.

The DSA Algorithm

Signature algorithms require an asymmetric key pair. In the case of DSA, the *domain parameters* consist of the set ***{p,q,g}***, where ***p*** and ***q*** are prime numbers and ***g*** is a generator of a subgroup within a particular multiplicative group – details are given FIPS PUB 186-4. What is of interest to the Bouncy Castle FIPS C# API are the values ***L*** (desired length of the prime ***p*** in bits) and ***N*** (desired length of the prime ***q*** in bits) which are required to generate ***p***, ***q*** and ***g*** – again these are specified in the FIPS PUB 186-4. We reproduce these values below:

L	***N***
1024	160
2048	224
2048	256
3072	256

The following example will generate a DSA key pair given the values ***L***=2048 and ***N***=256. Note that, an exception will occur if the values of ***L*** and ***N*** do not match as in the table above. Also, take note that DSA is no longer approved for signature generation (see FIPS PUB 186-5), however, DSA can still be used for signature verification.

Example 22 – Key Pair Generation

```
public static AsymmetricKeyPair<AsymmetricDsaPublicKey, AsymmetricDsaPrivateKey>
                    generateDSAKeyPair(int L, int N)
{
    FipsDsa.DomainGenParameters dsaDomainGenParams = new FipsDsa.DomainGenParameters(L,N);
    FipsDsa.DomainParametersGenerator dsaDomainParamsGen =
        CryptoServicesRegistrar.CreateGenerator(dsaDomainGenParams,new SecureRandom());
    DsaDomainParameters dsaDomainParams = dsaDomainParamsGen.GenerateDomainParameters();

    FipsDsa.KeyGenerationParameters keyGenParameters =
        new FipsDsa.KeyGenerationParameters(dsaDomainParams);

    FipsDsa.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, new SecureRandom());

    return kpGen.GenerateKeyPair();
}
```

Note that, when you run the example above, you will observe that the execution takes a while – more than a few seconds (depending on your hardware and entropy source). The reason in this delay is not the key generation part, rather it is the fact that certain large prime numbers and a group generator need to be created – that is, the set ***{p,q,g}*** as discussed above needs to be generated.

In Example 22 above we have returned a "generic" ***AsymmetricKeyPair*** object with the constrained types ***AsymmetricDsaPublicKey*** and ***AsymmetricDsaPrivateKey*** as reference objects.

```
public static AsymmetricKeyPair<AsymmetricDsaPublicKey, AsymmetricDsaPrivateKey>
                    generateDSADomainParameters(int L, int N) {...}
```

Thus the function above returns to us an appropriate asymmetric key pair object. You will see this use of "generics" throughout the BC FIPS C# API. (Further details on the use of C# generics in the .NET framework are available online on the MSN website. If you are not familiar with the term generics, then for now think of generics as templates like in C++ – though sharing some similarities, the two implementations are fundamentally different – C++ templates operate a compile time, while C# generics operate at run-time.)

For signing the BC C# API provides a signature service which takes the algorithm's private key and then generates a stream object used for signing the message. For verification of a signature, the API provides a verification service which takes the algorithm's public key, a previously generated signature and the message which has been signed, then generates a stream object for verifying the signature. Note that, the "finalization" of either processes (signing and verifying) occurs when the stream object has been "closed" as shown below. The example below shows a DSA signature generation and a DSA signature verification which is based on a SHA-384 digest of the message. Note that, the API does not provide the message digest as part of the signature functionality – it is up to you to separately generate and provide the digest if this is required.

Example 23 – Signing and Verifying

```
public static byte[] generateDSASignature(AsymmetricDsaPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService(key, new SecureRandom());

    ISignatureFactory<FipsDsa.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsDsa.Dsa);

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}
```

```
public static bool verifyDSASignature(AsymmetricDsaPublicKey key, byte[] signature, byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider = CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsDsa.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsDsa.Dsa);

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

As indicated above, by default, the digest used is based on a SHA-384 algorithm. You can change the digest to one of the approved digests such as SHA-512. To achieve this, modify the signature and verification methods above with,

```
    ISignatureFactory<FipsDsa.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsDsa.Dsa.WithDigest(FipsShs.Sha512));
```

and

```
    IVerifierFactory<FipsDsa.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsDsa.Dsa.WithDigest(FipsShs.Sha512));
```

respectively.

As was demonstrated in executing Example 22 above, generating the domain parameters every time we wish to create a key pair is (CPU time) expensive. Therefore, if when signing a message you are not given the values ***L*** and ***N***, then it is worth generating the domain parameters separately and then explicitly passing them in for key pair generation. The following example shows how to generate a set of DSA parameters in the BC FIPS C# API.

Example 24 – Domain Parameter Generation

```
public static DsaDomainParameters generateDSADomainParameters(int L, int N)
{
    FipsDsa.DomainGenParameters dsaDomainParams = new FipsDsa.DomainGenParameters(L,N);

    FipsDsa.DomainParametersGenerator dsaDomainParamGen =
        CryptoServicesRegistrar.CreateGenerator(dsaDomainParams,new SecureRandom());

    return  dsaDomainParamGen.GenerateDomainParameters();
}
```

Once a set of domain parameters have been generated, then we can easily generate DSA key pairs from them and more importantly, reduce the time it takes to generate such key pairs.

Example 25 – Generating Key Pairs using Parameters

```
public static AsymmetricKeyPair<AsymmetricDsaPublicKey, AsymmetricDsaPrivateKey>
                    generateDSAKeyPairUsingParameters(DsaDomainParameters dsaDomainParams)
{
    FipsDsa.KeyGenerationParameters keyGenParameters =
        new FipsDsa.KeyGenerationParameters(dsaDomainParams);

    FipsDsa.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, new SecureRandom());

    return kpGen.GenerateKeyPair();
}
```

Execute Example 25 multiple times once the domain parameters have been generated. Then compare the execution speed and observe that there is a significant difference in the time taken to generate a set of domain parameters compared to that of generating a pair of keys.

The RSA Algorithm

The RSA algorithm is fundamentally different in a mathematical sense to that used in the DSA. In addition, the RSA algorithm provides for encryption and decryption of a (suitable) block of data. Compare this (if you have the inclination) to the DSA algorithm which can only make a verification of a signature – that is, it can not be used to encrypt/decrypt data. Thus in the RSA scheme of things, a RSA generated digital signature is really an "encryption" of a data block with private key that anyone with the public key can then "decrypt" and therefore verify. The encryption and decryption is written in parenthesis "…" because when RSA is used for signatures, the private key is used to encrypt the message, whereas when RSA is used for encryption, then the public key is used to encrypt the message. Keep this in mind as it will help explain why, other than for the purpose of generating a certification request, an RSA key used for encryption should never be used for signing and visa-versa.

Generation of RSA signatures also requires key pairs of the sizes 2048 and 3072 bits. (Key sizes of less than 2048 are no longer permitted. In addition, FIPS PUB 186-5 allows for maximum key sizes of up to 4096.) Note that the key sizes are not explicitly related to the security strength of the RSA algorithm. There is however, an implicit relationship between the key size and the security strength based upon the complexity of the *General Number Field Sieve*. Further details are provided in the document NIST SP 800-57 which specifies a table of key sizes and their corresponding security strengths.

A RSA public key consists of an integer $\boldsymbol{n=pq}$, where $\boldsymbol{p}$ and $\boldsymbol{q}$ are large primes and an integer $\boldsymbol{e}$ such that $\boldsymbol{2^{16} < e < \lambda(n)}$, where $\boldsymbol{\lambda}$ Carmichael's totient function, and $\boldsymbol{e}$ is co-prime with $\boldsymbol{\lambda(n)}$. Thus a RSA public key is the set $\boldsymbol{\{n,e\}}$. (Note that, $\boldsymbol{p}$ and $\boldsymbol{q}$ are kept secret, only $\boldsymbol{n}$ the product is made public.) A RSA private key consists of $\boldsymbol{d = e^{-1} \bmod \lambda(n)}$, $\boldsymbol{p}$ and $\boldsymbol{q}$. Thus a RSA private key is the set $\boldsymbol{\{p,q,d\}}$. One final note on the exponent $\boldsymbol{e}$ – this need not be randomly generated as per NIST recommendations.

A good choice of public exponent can help as we tend to verify signatures more often than we create them, so by choosing an appropriate public exponent we can make the verification process reasonably efficient. In the example below we've used the smallest acceptable value for FIPS purposes. The number happens to be the 4th Fermat prime, and has a value of 0x10001.

Example 26 – Key Pair Generation

```
public static AsymmetricKeyPair<AsymmetricRsaPublicKey, AsymmetricRsaPrivateKey>
        generateRSAKeyPair(BigInteger e, int keySize)
{
    FipsRsa.KeyGenerationParameters keyGenParameters = new FipsRsa.KeyGenerationParameters(e, keySize);

    FipsRsa.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, new SecureRandom());

    return kpGen.GenerateKeyPair();
}
```

You may also notice that the RSA key pair generation is also CPU time intensive. RSA key pair generation (similar to the DSA domain parameters generation) will grind through a lot of entropy as a large number of bits will get consumed trying to generate random primes. Once a key pair have been generated, the next step is to generate a signature. FIPS PUB 186-4 draws on 2 different other standards for generating RSA signatures: X9.31 and PKCS#1 v2.1. (PKCS#1 v2.1 defines the PKCS#1.5 and the PSS formats which are to be used). However, note that, FIPS PUB 186-5, in the section Appendix E: Revisions: X9.31 has withdrawn the X9.31 standard, and that the standard now refers to only PKCS#1 v2.2 (which again defines PKCS#1.5 and the PSS formats).

The example below will first generate RSA signature using PKCS#1.5 format which is then followed by the verification.

Example 27 – The PKCS#1.5 Signature Format

```
public static byte[] generateRSASignature(AsymmetricRsaPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService(key, new SecureRandom());

    ISignatureFactory<FipsRsa.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsRsa.Pkcs1v15);

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}
```

```
public static bool verifyRSAignature(AsymmetricRsaPublicKey key, byte[] signature, byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider = CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsRsa.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsRsa.Pkcs1v15);

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

If you look carefully at examples 23 and 27, you will notice a similarity with the API used for the DSA and RSA signature generation and verification. As with the DSA, you can specify a particular (FIPS approved) digest by making the modifications in the example above to:

```
    ISignatureFactory<FipsRsa.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512));
```

and

```
    IVerifierFactory<FipsRsa.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha512));
```

respectively.
Also take note that without an explicit digest mentioned, the default is FipsShs.Sha384.

While the X9.31 signature format in FIPS PUB 186-5 has been withdrawn, this document is using FIPS PUB 186-4 as its main source of NIST standards. The example below provides the signature generation and verification using the X9.31 padding format. For this example, the current version of the BC C# API requires that we first set a default `SecureRandom` object – Example 3 above shows how to set a default `FipsSecureRandom` object.

Example 28 – The X9.31 Signature Format

```
public static byte[] generateX931Signature(AsymmetricRsaPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService(key);

    ISignatureFactory<FipsRsa.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsRsa.X931);

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}
```

```
public static bool verifyX931Signature(AsymmetricRsaPublicKey key, byte[] signature, byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider =
        CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsRsa.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsRsa.X931);

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

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

In 2002 RSA announced a new signature format based on the Probabilistic Signature Scheme (PSS). In simple terms, PSS is randomized and will produce a different signature value each time (unless you use a zero-length salt – see examples below), while PKCSV1_5 is deterministic – that is, the same message and key will produce an identical signature value each time. PSS should be used in preference to the PKCS#1.5 format as PSS is (generally) regarded as provably secure, in the sense that trying to forge a signature in PSS can be shown to reduce to the problem of breaking RSA and at the very least is no more insecure than PKCS#1.5. While there have not been any successful attacks on properly formatted PKCS#1.5 signatures done carefully, the additional assurances offered by PSS are very nice to have. The following example shows the use of RSA PSS.

PSS signatures require a more complicated set of parameters than that of a PKCS#1.5 signature. In addition to the message digest algorithm: PSS requires a mask generation function (MGF), salt and the length of the salt. The default settings as shown in Example 29 below have:

- message digest as SHA384,
- salt as null,
- salt length as 0, and
- MGF set to MGF1 using SHA384 as the digest fed into the MGF1.

(MGF1 is the name for an algorithm which uses a digest for obtaining a mask generating function.)

Example 29 – The Default PSS Signature Format

```
public static byte[] generatePSSSignature(AsymmetricRsaPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService(key);

    ISignatureFactory<FipsRsa.PssSignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsRsa.Pss);

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}
```

```
public static bool verifyPSSSignature(AsymmetricRsaPublicKey key, byte[] signature, byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider =
        CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsRsa.PssSignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsRsa.Pss);

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

As mentioned in FIPS PUB 186-5, SHAKE128 or SHAKE256 can be used as an alternative input to the MGF1. Currently however, only MGF1 is supported in this version of the C# API.

The following example demonstrates how to set one or more PSS parameters. The parameters we can set are:

- message digest,
- MFG digest,
- salt, and
- salt length.

Note that, if the salt length is non-zero and the salt is null, then a random salt of the specified salt length will be generated for you.

Example 30 – PSS Signatures with Parameters

```
public static byte[] generatePSSSignatureWithParameters(AsymmetricRsaPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider = CryptoServicesRegistrar.CreateService(key);

    ISignatureFactory<FipsRsa.PssSignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsRsa.Pss.WithDigest(
            FipsShs.Sha384).WithMgfDigest(FipsShs.Sha384).WithSalt(ExValues.sampleInput));

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}
```

```
public static bool verifyPSSSignatureWithParameters(AsymmetricRsaPublicKey key,byte[] signature,
                                                    byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider = CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsRsa.PssSignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsRsa.Pss.WithDigest(
            FipsShs.Sha384).WithMgfDigest(FipsShs.Sha384).WithSalt(ExValues.sampleInput));

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

Note that, in the example above, the digest used with the signature, the PSS generation, and the mask function are all the same. This is an established convention that should be followed as it means that all components of the system will be operating at an equivalent level of security. While there is no well-known attack for this case either, you may end up feeling very depressed if it turns out that setting the **`WithMgfDigest`** parameter to **`FipsShs.Sha1`** actually opened the possibility of an attack on your SHA-384 PSS signatures – better to avoid that possibility.

Using Elliptic Curve – ECDSA

The final signature type looked at in FIPS PUB 186-4 is ECDSA, which is a DSA style signature based on Elliptic Curves. The **`FipsEC`** class currently contains all FIPS approved curves. Note that, FIPS PUB 186-5 and NIST SP 800-186 *Recommendations for Discrete Logarithm-based Cryptography* have some of these as being depreciated. As with the previous examples above with generating signatures, we first require a key pair to be generated. The generation of a key pair requires a specification of a curve. Supported curves are given as:

- a curve over a prime number field – these curves are given in the class **`FipsEC`** as **`PXXX`**,
- a curve over a binary field – these curves are given in the class **`FipsEC`** as **`BYYY`**, and
- for each binary curve we have a specialization called a Koblitz curve given in the class **`FipsEC`** as **`KYYY`**,

where XXX and YYY are integers providing an indication of the field size.

In the example below we will generate an EC key pair using a P-384 curve with a set of domain parameters. Details on domain parameters and what constituents a private and public key are given in the document FIPS PUB 186-4.

Example 31 – Key Pair for a Named Curve

```
public static AsymmetricKeyPair<AsymmetricECPublicKey, AsymmetricECPrivateKey> generateECKeyPair()
{
    FipsEC.KeyGenerationParameters keyGenParameters =
        new FipsEC.KeyGenerationParameters(FipsEC.DomainParams.P384);

    FipsEC.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, new SecureRandom());

    return kpGen.GenerateKeyPair();
}
```

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

Example 32 – ECDSA Signing and Verifying

```
public static byte[] generateECDSASignature(AsymmetricECPrivateKey key, byte[] message)
{
    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService(key, new SecureRandom());

    ISignatureFactory<FipsEC.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsEC.Dsa);

    IStreamCalculator<IBlockResult> calculator = signer.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().Collect();
}

public static bool verifyECDSASignature(AsymmetricECPublicKey key, byte[] signature, byte[] message)
{
    IVerifierFactoryService verifierFactoryProvider = CryptoServicesRegistrar.CreateService(key);

    IVerifierFactory<FipsEC.SignatureParameters> verifier =
        verifierFactoryProvider.CreateVerifierFactory(FipsEC.Dsa);

    IStreamCalculator<IVerifier> calculator = verifier.CreateCalculator();

    Stream sOut = calculator.Stream;
    sOut.Write(message, 0, message.Length);
    sOut.Close();

    return calculator.GetResult().IsVerified(signature);
}
```

Finally

It is worth mentioning again that both DSA and ECDSA require, for each signature generation, a fresh cryptographically secure random number. In some hardware architectures, for example, small embedded hardware, generation of cryptographically secure entropy may be a challenge. In such environments, RSA *without* PSS is often used. In non-approved FIPS mode, it is possible to produce deterministic signatures by specifying a value of "k" – refer to Section 6.3 *Secret Number Generation* in FIPS PUB 186-4 or RFC 6979 for a definition of "k". The generation of such signatures, are not FIPS approved and therefore lie outside the scope of this document. Nevertheless, the BC C# API provides mechanisms for generation of such signatures – for example, for testing the integrity of the API for signature generation and signature verification.

Key Wrapping

Key wrapping is a generic term used to describe the various methods of encrypting a given key with another key. Since there are symmetric keys and asymmetric key pairs which can be used for encryption, it follows that key wrapping can be achieved by the use of encrypting a key with a given symmetric key or with one of the asymmetric key pairs. Note that, key wrapping is a common method which is used for protecting a given key during transportation over an unsecured channel and is also common when storing a key, for example in a database. Since key wrapping is a common occurrence it is not surprising that certain standards and best practices have evolved in achieving this. NIST also provide standards for use with symmetric key algorithms such as AES and asymmetric RSA key pairs – although in the case of RSA, the NIST terminology used describes the wrapping function in connection with key transport. Note that, some documents refer to *Key Wrapping* as encrypting a symmetric key using symmetric cipher and *Key Encapsulation* as encrypting a symmetric key using a public key algorithm (e.g., for hybrid encryption). We will see examples of both later in this document.

The main security concept to keep in mind with key wrapping is that it is fool hardy to wrap a key which is expected to have a particular security strength with one that does not at least have the equivalent security strength. Using a wrapping key with higher security strength is clearly better, however, this may not always be possible.

Using Symmetric Keys for Wrapping

The simplest techniques for key wrapping are based around the use of symmetric keys. They are documented in NIST SP 800-38F. The approach presented there is superior to simply encrypting a key as it also includes some checksum information, the idea being there is a reasonable chance that an error will be detected if an attempt is made to unwrap a previously wrapped key with the wrong key, or an attempt is made to feed garbage into the key unwrapping algorithm instead.
Like regular encryption the key wrapping algorithms have to contend with the fact that block ciphers work with block aligned data. For that reason SP 800-38F offers two alternatives, one without padding which requires block aligned keys (where a key block is half the block size of the cipher), and one with padding for non-block aligned keys.

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

Example 33 – Wrapping without Padding

```
public static byte[] wrapInputWithoutPadding(FipsAes.Key key, byte[] inputToWrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyWrapper<FipsAes.WrapParameters> wrapper = provider.CreateKeyWrapper(FipsAes.KW);

    byte[] wrappedInput = wrapper.Wrap(inputToWrap).Collect();

    return wrappedInput;
}
```

Note that, if the `inputToWrap` given above is not a multiple of 8 bytes, then an exception will occur. (64

bits is exactly half of the AES block size of 128 bits.)

```
public static byte[] unwrapInputWithouPadding(FipsAes.Key key, byte[] cipherToUnwrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyUnwrapper<FipsAes.WrapParameters> unwrapper = provider.CreateKeyUnwrapper(FipsAes.KW);

    byte[] unwrappedInput = unwrapper.Unwrap(cipherToUnwrap, 0, cipherToUnwrap.Length).Collect();

    return unwrappedInput;
}
```

Again, note that, the input: `cipherToUnwrap` must be a multiple of 8 bytes. Moreover, if the length of the `cipherToUnwrap` differs from the length of what was wrapped, or, if the AES key given differs, then the BC FIPS C# API will produce an exception which you will need trap.

The example below shows the use of AES key wrapping with padding, also as defined in SP 800-38F. Therefore, while SP 800-38F provides a maximum size of the input data (key) to be wrapped, the length of the input data need not be block aligned – that is, need not be a multiple of 8 bytes.

Example 34 – Wrapping with Padding

```
public static byte[] wrapInputWithPadding(FipsAes.Key key, byte[] inputToWrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyWrapper<FipsAes.WrapParameters> wrapper = provider.CreateKeyWrapper(FipsAes.KWP);

    byte[] wrappedInput = wrapper.Wrap(keyToWrap).Collect();

    return wrappedInput;
}

public static byte[] unwrapInputWithPadding(FipsAes.Key key, byte[] cipherToUnwrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyUnwrapper<FipsAes.WrapParameters> unwrapper = provider.CreateKeyUnwrapper(FipsAes.KWP);

    byte[] unwrappedInput = unwrapper.Unwrap(cipherToUnwrap, 0, cipherToUnwrap.Length).Collect();

    return unwrappedInput;
}
```

Note that, in the examples above when a input “key” is wrapped we simply return return a block of bytes. Normally we are wrapping a symmetric key and we would like to have a symmetric key returned to us when we unwrap the input “key”. The example below demonstrates how to wrap and unwrap a FipsSecureRandom symmetric key. This example therefore, allows us to pass an actual key on wrapping and to return a key from the cipher doing the unwrapping, rather than just a block of bytes which we would then need to convert into a key.

Example 35 – Wrapping Symmetric Keys

```
public static byte[] wrapKeyWithoutPadding(FipsAes.Key key, SymmetricSecretKey keyToWrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyWrapper<FipsAes.WrapParameters> wrapper = provider.CreateKeyWrapper(FipsAes.KW);

    byte[] wrappedKey = wrapper.Wrap(keyToWrap.GetKeyBytes()).Collect();

    return wrappedKey;
}

public static SymmetricSecretKey unwrapKeyWithoutPadding(FipsAes.Key key, byte[] cipherToUnwrap)
{
    IAeadBlockCipherService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyUnwrapper<FipsAes.WrapParameters> unwrapper = provider.CreateKeyUnwrapper(FipsAes.KW);

    SymmetricSecretKey unwrappedKey =
        new FipsAes.Key(unwrapper.Unwrap(cipherToUnwrap, 0, cipherToUnwrap.Length).Collect());

    return unwrappedKey;
}
```

Using RSA OAEP for Wrapping

The RSA Optimal Asymmetric Encryption Padding (OAEP) algorithm was originally introduced in PKCS#1 Version 2 at the same time as PSS and is described in NIST SP 800-56B, “Recommendations for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography”. (See also, RFC 2437.) NIST has chosen to publish this padding scheme under the guise of a key establishment scheme rather than as a wrapping mechanism. The reason for this could be that NIST may assume that this technique would only be used to send a symmetric key to another party. However, in this version of BC FIPS C# API we are referring to RSA with OAEP as key wrapping. There are two main aims of OAEP:

- Adding random padding to plaintext (the key to be wrapped for example) will convert RSA from a deterministic encryption scheme into a probabilistic one.
- Second, OAEP can prevent leaking of the RSA encryption parameters by a chosen plaintext attack.

Since basic OAEP wrapping only allows you to wrap a symmetric key using an RSA public key, it places a restriction on the length of the input which can be wrapped.

Example 36 – OAEP Wrapping

```
public static byte[] oaepKeyWrap(AsymmetricRsaPublicKey key, byte[] inputData)
{
    IKeyWrappingService provider = CryptoServicesRegistrar.CreateService(key, new SecureRandom());

    IKeyWrapper<FipsRsa.OaepWrapParameters> wrapper = provider.CreateKeyWrapper(FipsRsa.WrapOaep);

    byte[] wrappedInput = wrapper.Wrap(inputData).Collect();

    return wrappedInput;
}
```

```
public static byte[] oaepKeyUnwrap(AsymmetricRsaPrivateKey key, byte[] cipherToUnwrap)
{
    IKeyUnwrappingService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyUnwrapper<FipsRsa.OaepWrapParameters> unwrapper =
         provider.CreateKeyUnwrapper(FipsRsa.WrapOaep);

    byte[] inputData = unwrapper.Unwrap(cipherToUnwrap, 0, cipherToUnwrap.Length).Collect();

    return inputData;
}
```

Note that, if the length your `inputData` in the example above exceeds the size of the RSA modulus then, an exception will be produced. A general rule of thumb is that the input should be at least 2 bytes (used for identification) less than the bytes representing the size of the RSA modulus. (Naturally, you have a similar issue regarding the length of `cipherToUnwrap`.)

Unlike AES key wrapping with and without padding, RSA OAEP wrapping can also be used in conjunction with parameters that allow you to use a higher level cipher than the default (currently SHA-384). The BC FIPS C# API provides for the following RSA OAEP parameters to be modified:

- hash digest,
- mask generation function, and
- additional input such as a salt or initialization vector.

Example 37 – OAEP Wrapping with Parameters

```
public static byte[] oaepKeyWrapWithParameters(AsymmetricRsaPublicKey key, byte[] inputData)
{
    IKeyWrappingService provider = CryptoServicesRegistrar.CreateService(key, new SecureRandom());

    IKeyWrapper<FipsRsa.OaepWrapParameters> wrapper =
        provider.CreateKeyWrapper(
            FipsRsa.WrapOaep.WithDigest(
                FipsShs.Sha512).WithMgfDigest(
                    FipsShs.Sha512).WithEncodingParams(ExValues.sampleNonce));

    byte[] wrappedInput = wrapper.Wrap(inputData).Collect();

    return wrappedInput;
}


public static byte[] oaepKeyUnwrapWithParameters(AsymmetricRsaPrivateKey key, byte[] cipherToUnwrap)
{
    IKeyUnwrappingService provider = CryptoServicesRegistrar.CreateService(key);

    IKeyUnwrapper<FipsRsa.OaepWrapParameters> unwrapper =
        provider.CreateKeyUnwrapper(
            FipsRsa.WrapOaep.WithDigest(
                FipsShs.Sha512).WithMgfDigest(
                    FipsShs.Sha512).WithEncodingParams(ExValues.sampleNonce));

    byte[] inputData = unwrapper.Unwrap(cipherToUnwrap, 0, cipherToUnwrap.Length).Collect();

    return inputData;
}
```

Key Establishment and Agreement

Diffie-Hellman Key Agreement

In approved mode, the BC FIPS C# API provides for establishing keys between multiple parties using the Diffie-Hellman (or more completely Diffie-Hellman-Merkle) key agreement (DH) and Elliptic Curve Diffie-Hellman (ECDH) or it's variant Elliptic Curve Co-factor Diffie-Hellman (ECCDH). While it will not be discussed in this document, it is interesting to note that with any of these finite group methods of key agreement schemes, instead of two parties agreeing, the scheme can be extended so that it can be used between more than two parties to allow all participants to arrive at a shared secret – there are different implementations of such schemes which may be of interest where use of a hardware or software key distribution node is not practical.

Details about the Diffie-Hellman scheme implemented in the BC FIPS C# API are given in NIST SP 800-56A, now at revision 3. NIST SP 800-56Ar3 (and NIST SP 800-56Ar2 is also useful) provides details on the key agreement and key establishment schemes, as well as the valid domain parameters for finite fields to be used. (See also FIPS PUB 186-4 for additional information on domain parameters for finite fields.)

As with DSA, the DH algorithm requires domain parameters and also key pairs for all participants. Recall from section **The DSA Algorithm**, Example 22, that the DH algorithm requires the set ***{p,q,g}***, where ***p*** is a large prime, ***q*** is either ***p-1*** or a large prime divisor of ***p-1*** and ***g*** is a generator for the subgroup of order ***q*** of the multiplicative group of the finite field. Thus, domain parameters for DH are just as expensive to generate as the DSA ones and as it is a multi-party algorithm it is important that everyone agrees on the same domain parameter set first. In this implementation of the BC FIPS C# API, there is a list of "pre-defined safe" values (or "safe primes") of DH domain parameters which can be used. Of course, you can define your own set of parameters. The "pre-defined safe" values provided are listed in RFC7919 and RFC3526 to create finite cyclic groups that are referred to as Finite Field Diffie-Hellman Ephemeral (***ffdhe***) and Modular Exponential Diffie-Hellman (***modp***), respectively. Observe that the ffdhe groups are useful for such key exchanges as Transport Layer Security (TLS), while the modp groups are useful for key exchanges as IPSec and VPN. The example below shows how to obtain a set of DH domain parameters using one of the pre-defined values. Thus, while with DSA we are generating the primes, for the DH key exchange algorithm, we have a pre-defined set of DH parameters – therefore you should not observe any computational delay when choosing the pre-defined domain parameters.

Example 38 – DH Domain Parameters By Selection

```
public static DHDomainParameters generateParameters(IDHDomainParametersID dhParamsID)
{
    DHDomainParameters dhParams =
        DHDomainParametersIndex.LookupDomainParameters(dhParamsID);

    return dhParams;
}
```

The pre-defined set of DH parameters are:

- ffdhe2048,
- ffdhe3072,
- ffdhe4096,
- ffdhe6144,
- ffdhe8192,
- modp2048,
- modp3072,
- modp4096,
- modp6144 and
- modp8192.

Calling the above method in Example 38 above:

```
DHDomainParameters ffdhe3072 = generateParameters(FipsDH.DomainParams.ffdhe3072);
```

returns a set of domain parameters with a security strength of 128 bits.

Of course, the API provides a manner in which you can generate your own set of DH parameters as in the two examples below.

Example 39 – DH Domain Parameters DSA Generation

```
public static DHDomainParameters generateParameters(int L, int N, SecureRandom sRandom)
{
    FipsDsa.DomainGenParameters dsaDomainGenParams = new FipsDsa.DomainGenParameters(L,N);
    FipsDsa.DomainParametersGenerator dsaDomainParamsGen =
        CryptoServicesRegistrar.CreateGenerator(dsaDomainGenParams,sRandom);
    DsaDomainParameters dsaDomainParams = dsaDomainParamsGen.GenerateDomainParameters();
    DHDomainParameters dhParams =
        new DHDomainParameters(dsaDomainParams.P, dsaDomainParams.Q, dsaDomainParams.G);

    return dhParams;
}
```

Example 40 – DH Domain Parameters Direct Generation

```
public static DHDomainParameters generateParameters(BigInteger dhP,BigInteger dhQ, BigInteger dhG)
{
    DHDomainParameters dhParams = new DHDomainParameters(dhP, dhQ, dhG);
    return dhParams;
}
```

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

Example 41 – DH Key Pair Generation

```
public static AsymmetricKeyPair<AsymmetricDHPublicKey, AsymmetricDHPrivateKey>
    generateDHKeyPairUsingParameters(DHDomainParameters dhParams)
{
    FipsDH.KeyGenerationParameters keyGenParameters = new FipsDH.KeyGenerationParameters(dhParams);

    FipsDH.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, new SecureRandom());

    return kpGen.GenerateKeyPair();
}
```

Once both parties have generated their respective key pair, then we can now try to generate a shared secret key.

Example 42 – Basic DH Key Agreement

```
public static byte[] initiatorAgreementBasic(AsymmetricDHPrivateKey initiatorPrivate,
                                             AsymmetricDHPublicKey recipientPublic)
{
    IAgreementCalculatorService agreeFact = CryptoServicesRegistrar.CreateService(initiatorPrivate);

    IAgreementCalculator<FipsDH.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(FipsDH.DH);

    return agreement.Calculate(recipientPublic);
}

public static byte[] recipientAgreementBasic(AsymmetricDHPrivateKey recipientPrivate,
                                             AsymmetricDHPublicKey initiatorPublic)
{
    IAgreementCalculatorService agreeFact = CryptoServicesRegistrar.CreateService(recipientPrivate);

    IAgreementCalculator<FipsDH.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(FipsDH.DH);

    return agreement.Calculate(initiatorPublic);
}
```

The example above will produce a 128 or 256 bit string which can be used as an AES key for encryption/decryption. It's clear that if the same pair of asymmetric keys are used, then we generate the same bit string. This can be alieavated to some extend if at least one of the keys is an ephemeral key. Even with the use of an ephemeral key, the output of the alogrithm may not produce a uniformaly distributed set of bit strings. A further issue with simply generating the shared secret as above, is that the shared secret is restricted to the 128 or 256 bits. The solution to these issues is to use a Key Derivation Function (KDF) with some user keying material. The user keying material and the use of the KDF both help mask the secret, randomise the result and allow for as much data as is required to be generated (for sensible values of required).

In the example below, we use a KDF construction referred to as Concatenation as specified in the document NIST SP 800-56B.

Example 43 – DH Key Agreement with a KDF

```
public static byte[] initiatorAgreementWithConcatKDF(AsymmetricDHPrivateKey initiatorPrivate,
                        AsymmetricDHPublicKey recipientPublic, byte[] userKeyingMaterial)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(initiatorPrivate);

    IAgreementCalculator<FipsDH.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(
            FipsDH.DH.WithKeyMaterialGenerator(
               new FipsKdfKmg(FipsKdf.Concatenation, FipsPrfAlgorithm.Sha384, userKeyingMaterial, 32)
            )
        );

    return agreement.Calculate(recipientPublic);
}

public static byte[] recipientAgreementWithConcatKDF(AsymmetricDHPrivateKey recipientPrivate,
                         AsymmetricDHPublicKey initiatorPublic, byte[] userKeyingMaterial)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(recipientPrivate);

    IAgreementCalculator<FipsDH.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(
            FipsDH.DH.WithKeyMaterialGenerator(
               new FipsKdfKmg(FipsKdf.Concatenation, FipsPrfAlgorithm.Sha384, userKeyingMaterial, 32)
            )
        );

    return agreement.Calculate(initiatorPublic);
}
```

In general you would want the keying material (the userKeyingMaterial in the example above), to be random, or at least unique to the exchange, otherwise you will simply re-produce the effect of Example 42 above without a KDF. Another common KDF is the one from X9.63. To use this KDF simply replace `FipsKdf.Concatenation` with `FipsKdf.X963` in the above example.

Elliptic Curve Diffie-Hellman

The Diffie-Hellman protocol can also be used with Elliptic Curves. The points (***x***,***y***) on an elliptic curve defined over a finite field form an abelian group. If this group has prime order, then it is a cyclic group and all points (***x***,***y***) satisfying the equation of the curve lie in the group – in this case we say it has ***co-factor 1***. If the elliptic curve group has order ***hq*** where ***q*** is a large prime, then we apply the Diffie-Hellman protocol to the subgroup of order ***q*** – in this case we say the curve has a ***co-factor of h***. Moreover, not every point that satisfies the equation of the curve lies in the subgroup. However, any such point multiplied by ***h*** will lie in the subgroup. Thus in this case the Diffie-Hellman protocol is slightly more complicated. A formulation based on incorporating the curve's co-factor is actually the one described by NIST SP 800-56A which also details the use of Elliptic Curves with Diffie-Hellman as well as the finite field method. The variant incorporating the curve's co-factor is known as Elliptic Curve Co-factor Diffie-Hellman or ECCDH for short.

Generation of key pairs for Elliptic Curve Cryptography was covered by examples 31 and 32 in the chapter on digital signatures. As previously mentioned, we can “label” the EC keys which are generated for use of signatures or Diffie-Hellman. The example below shows how to “label” such keys. (By the way, this “labeling” helps with ensuring that you have the correct type of keys since the BC FIPS C# API will type check what type of keys are used for which algorithm.)

Example 44 – ECCDH Key Generation

```
public static AsymmetricKeyPair<AsymmetricECPublicKey, AsymmetricECPrivateKey>
        generateECKeyPairForDH(SecureRandom secRand)
{
    FipsEC.KeyGenerationParameters keyGenParameters =
        new FipsEC.KeyGenerationParameters(FipsEC.DomainParams.K409).For(FipsEC.Cdh);

    FipsEC.KeyPairGenerator kpGen =
        CryptoServicesRegistrar.CreateGenerator(keyGenParameters, secRand);

    return kpGen.GenerateKeyPair();
}
```

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

Example 45 – Basic ECCDH Key Agreement

```
public static byte[] initiatorAgreementBasic(
    AsymmetricECPrivateKey initiatorPrivate, AsymmetricECPublicKey recipientPublic)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(initiatorPrivate);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(FipsEC.Cdh);

    return agreement.Calculate(recipientPublic);
}

public static byte[] recipientAgreementBasic(
    AsymmetricECPrivateKey recipientPrivate, AsymmetricECPublicKey initiatorPublic)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(recipientPrivate);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(FipsEC.Cdh);

    return agreement.Calculate(initiatorPublic);
}
```

Notice that agreement generation is similar to the Finite Field Diffie-Hellman examples above. (This isn’t surprising since we are still working in a abelian cyclic finite group.) In these examples, we will obtain a key which will depend on the size of the group generated by the elliptic curve. Again as with the Finite Field Diffie-Hellman case, we may have an issue if both the keys involved are for long term use. The answer to these type of problems is the same as before, use some additional user keying material (with at least some unique properties) and a KDF.

Example 46 – Basic ECCDH Key Agreement with a KDF

```
public static byte[] initiatorAgreementWithConcatKDF(AsymmetricECPrivateKey initiatorPrivate,
                        AsymmetricECPublicKey recipientPublic, byte[] userKeyingMaterial)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(initiatorPrivate);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(
            FipsEC.Cdh.WithKeyMaterialGenerator(
               new FipsKdfKmg(FipsKdf.X963, FipsPrfAlgorithm.Sha384, userKeyingMaterial, 32)
            )
        );

    return agreement.Calculate(recipientPublic);
}

public static byte[] recipientAgreementWithConcatKDF(AsymmetricECPrivateKey recipientPrivate,
                         AsymmetricECPublicKey initiatorPublic, byte[] userKeyingMaterial)
{
    IAgreementCalculatorService agreeFact =
        CryptoServicesRegistrar.CreateService(recipientPrivate);

    IAgreementCalculator<FipsEC.AgreementParameters> agreement =
        agreeFact.CreateAgreementCalculator(
            FipsEC.Cdh.WithKeyMaterialGenerator(
               new FipsKdfKmg(FipsKdf.X963, FipsPrfAlgorithm.Sha384, userKeyingMaterial, 32)
            )
        );

    return agreement.Calculate(initiatorPublic);
}
```

Certification Requests, Certificates, and Revocation

Public key cryptography consists of a key pair (a public key and a private key). As we have seen above, for digital signatures (in basic terms) we encrypt the message digest with our private key and for data exchange (again in basic terms) we encrypt the message with our public key. One of the issues with public key cryptography is that, while having a public key is all very well, how do you ensure the identity of the holder of the public key? Part of the solution to this problem, at least for some people, is the use of certificates. A certificate is a means of identifying the holder of the key pair. Note that, a certificate provides additional information (apart from the public key) such as a hostname, or an organization, or an individual – something that identifies the holder of the key pair. For additional security, once a certificate is created, the certificate can either signed by a certificate authority (CA) or self-signed.

A common standard used for certificates is the X.509 standard. Cryptographic Message Syntax (CMS), S/MIME (Secure Multipurpose Internet Mail Extension) which is built on top of CMS, Time- Stamp Protocol (TSP) and a host of others all rely on the use of X.509 certificates which represent a common method of binding some identity information as well as some assurance of validity to a public key. The BC FIPS C# API does not provide direct support in the FIPS module for the support of certification requests, certificates, and revocations, however it does work with the BC extensions APIs that provide support for these things.

The BC C# extension API for certificate requests, certificates, and revocation are available from the ***bc-pkix-1.0.2.dll*** assembly. In contrast to the ***bc-fips-1.0.2.dll*** assembly, this bc-pkix-1.0.2.dll assembly relies on the bc-fips-1.0.2.dll assembly – and both assemblies need to be made available to your application.

Certification Requests

The first step in obtaining a certificate for a public key is to request one (unless you are generating a certificate yourself and self-signing it). In most cases a request is made to a CA. Apart from any paperwork involved this normally involves some method of sending your public key to the certificate authority that will issue you with a certificate. Two popular ways of doing this rely on a standard called PKCS#10 described in RFC 2986 and also on another standard called Certificate Request Message Format (CRMF) described in RFC 4211. Another standard called KMIP (Key Management Interoperability Protocol) is also used, however, currently there is no support in BC FIPS C# API for KMIP .

The first example of a certificate request is the PKCS#10 standard. Generating a basic PKCS#10 certification request is relatively simple: just specify the public key, some identification on how you want to be known, finally sign the request and send it off. The resulting message should then be verifiable by anyone who is capable of decoding the public key the certification request contains. A minimal PKCS#10 request is described in the following example.

Example 47 – A Basic PKCS#10 Request

```
public static Pkcs10CertificationRequest createPkcs10Request(
                    AsymmetricRsaPublicKey pubKey, AsymmetricRsaPrivateKey priKey)
{
    Pkcs10CertificationRequestBuilder pkcs10Builder =
        new Pkcs10CertificationRequestBuilder(
            new X500Name("CN=PKCS10 Example"),pubKey.GetEncoded());

    ISignatureFactory<IParameters<Algorithm>> sigFact =
        CryptoServicesRegistrar.CreateService(priKey,
            new SecureRandom()).CreateSignatureFactory(FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha256));

    Pkcs10CertificationRequest request = pkcs10Builder.Build(new PkixSignatureFactory(sigFact));

    return request;
}

public static bool verifyPkcs10Request(Pkcs10CertificationRequest pkcs10Request)
{
    IAsymmetricPublicKey pubKey = AsymmetricKeyFactory.CreatePublicKey(
        pkcs10Request.SubjectPublicKeyInfo.GetEncoded());

    PkixVerifierFactoryProvider pro = new PkixVerifierFactoryProvider(pubKey);

    bool result = pkcs10Request.IsSignatureValid(pro);

    return result;
}
```

In the example above we see that the certificate request is for a RSA public key and it is signed using a SHA-256 digest. The subject name requested for the certificate a CA might produce is the X.500 name “CN=PKCS10 Example”. (Note that X.500 is a family of communication protocols and not just one standard.) We could have easily replaced the RSA algorithm with an appropriate elliptic curve and the ECDSA signing algorithm – refer to the examples associated with this document.

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

Example 48 – A PKCS#10 Request with Extensions

```
public static Pkcs10CertificationRequest createPkcs10RequestWithExtension(
                    AsymmetricRsaPublicKey pubKey, AsymmetricRsaPrivateKey priKey)
{
    X509ExtensionsGenerator extGen = new X509ExtensionsGenerator();

    extGen.AddExtension(X509Extensions.SubjectAlternativeName, false,
        new GeneralNames(new GeneralName(new X500Name("CN=Alt Name"))));
    extGen.AddExtension(X509Extensions.IssuerAlternativeName, false,
        new GeneralNames(new GeneralName(new X500Name("CN=Issuer Name"))));

    Pkcs10CertificationRequestBuilder pkcs10Builder =
             new Pkcs10CertificationRequestBuilder(
                 new X500Name("CN=PKCS10 Example"), pubKey.GetEncoded());

    pkcs10Builder.AddAttribute(PkcsObjectIdentifiers.Pkcs9AtExtensionRequest, extGen.Generate());

    ISignatureFactory<IParameters<Algorithm>> sigFact =
        CryptoServicesRegistrar.CreateService(
            priKey, new SecureRandom()).CreateSignatureFactory(
                FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha256));

    Pkcs10CertificationRequest request = pkcs10Builder.Build(new PkixSignatureFactory(sigFact));

    return request;
}
```

The PKCS#10 (Public Key Cryptographic Standard 10) used in the above two examples, was presented in 2000, by RSA Laboratories and, largely for this reason, has become one of the more common used certificate formats, in particular with the message exchange mechanisms of web-based applications. Looking closer at this standard a PKCS#10 certificate request consists of three parts:

- certification request information,
- a signature algorithm identifier and
- a digital signature of the certification request information data.

In general then, the reason PKCS#10 uses a signature based on the private key of the public key being requested is to provide what is called ***"proof of possession"*** (this is sometimes abbreviated as "Pop"). That is, by presenting a signature that can be verified using the embedded public key in the certification request, then the CA (or RA – Registration Authority) can be certain that the party that generated the certification request was in possession of the private key associated with the public key in the request. This is also the one exception to the rule concerning never using an encryption key for signing that FIPS allows – refer to NIST Special Publication 800-57 Part 1 Revision 5 *Recommendation for Key Management*, section 8.1.5.1.1.2.

Not all algorithms however can be used for signing as well as encryption though, managing keys for ElGamal (or more correctly Elgamal) and DH key agreement are such examples. The PKCS#10 syntax does not support such keys used for encryption only. As a security consideration, encryption keys can not be treated the same as the signing keys, and PKCS#10 is not sufficient for this case. The CRMF format (standard) has been constructed with the idea of dealing with encryption keys. The first example of CRMF given below is basically the equivalent to what is happening in PKCS#10 – the proof of possession is given using a signature generated with the private key associated with the public key embedded with in the certificate request.

Example 49 – A Basic CRMF Request

```
public static CertificateRequestMessage createCertificateRequestMessage(
    AsymmetricRsaPublicKey pubKey, AsymmetricRsaPrivateKey priKey)
{
    CertificateRequestMessageBuilder certReqBuilder =
        new CertificateRequestMessageBuilder(BigInteger.One);

    ISignatureFactory<IParameters<Algorithm>> sigFact =
        CryptoServicesRegistrar.CreateService(priKey,
            new SecureRandom()).CreateSignatureFactory(FipsRsa.Pkcs1v15.WithDigest(FipsShs.Sha256));
    PkixSignatureFactory pkixSignFact = new PkixSignatureFactory(sigFact);

    certReqBuilder = certReqBuilder.SetSubject(new X500Name("CN=CRMF Example"));
    certReqBuilder = certReqBuilder.SetPublicKey(
                        SubjectPublicKeyInfo.GetInstance(pubKey.GetEncoded()));
    certReqBuilder = certReqBuilder.SetProofOfPossessionSignKeySigner(pkixSignFact);

    return certReqBuilder.Build();
}

public static bool isValidCertificateRequestMessage(CertificateRequestMessage certReqMsg)
{
    IAsymmetricPublicKey pubKey = AsymmetricKeyFactory.CreatePublicKey(
        certReqMsg.GetCertTemplate().PublicKey.GetEncoded());

    PkixVerifierFactoryProvider pro = new PkixVerifierFactoryProvider(pubKey);
    bool result = certReqMsg.IsValidSigningKeyPop(pro);

    return result;
}
```

Observe that, in the method above: **`isValidCertificateRequestMessage(...)`**, the validation of the request follows a similar process to that of the validation of a PKCS#10 certificate request. Where differences begin to become apparent, is where a key might not be usable for signing, such as DH, ElGamal, or even RSA where you absolutely do not want to even do signing once. In this case we formulate the CRMF request differently by specifying we want proof of possession to be dealt with in a subsequent message. Further details on how a CA (or RA) can valid a CRMF request and the different types of Pop are provided in RFC 4211.

Example 50 – A CRMF Request for Encryption Only Keys

```
public static CertificateRequestMessage createEncCertificateRequestMessage(
    AsymmetricRsaPublicKey pubKey, AsymmetricRsaPrivateKey priKey)
{
    CertificateRequestMessageBuilder certReqBuilder =
        new CertificateRequestMessageBuilder(BigInteger.One);

    certReqBuilder = certReqBuilder.SetSubject(new X500Name("CN=CRMF Example"));

    certReqBuilder =
        certReqBuilder.SetPublicKey(SubjectPublicKeyInfo.GetInstance(pubKey.GetEncoded()));

    certReqBuilder = certReqBuilder.SetProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert);

    return certReqBuilder.Build();
}
```

The subsequent message type given in the example above (**`SubsequentMessage.encrCert`**) tells the CA that it should assume that the End Entity (***EE*** – the entity who generated the request) has the private key so the EE is able to decrypt a message sent to it using the public one. In this case a CA would still send the certificate back to the EE, however, it would be in a message encrypted using the EE's private key –

normally derived using the Cryptographic Message Syntax (CMS), which will be described in later examples. An End Entity getting access to a certificate (from the CRMF request) would be required to decrypt the message from the CA, which would only possible if the end entity has the private key.

Certificate Construction

Certificates are digital documents which provide some sort of proof to the binding of a public key to an End Entity. In simple terms, for a certificate to be functional, it would contain the following minimal information:

- a public key,
- name of the End Entity,
- an expiration date of the certificate,
- name of the issuing certifying authority (CA),
- a (unique) serial number,
- a digital signature of the certificate issuer.

The most common standard for a certificate format (construction) is that of International Telecommunication Union (ITU-T) X.509. The X.509 standard is part of the X.500 series which defines a directory service. X.509 certificates generally come in three styles. Version 1 X.509 certificates (first published 1988), version 2 certificates X.509 (fist published 1993) and version 3 X.509 certificates (first published 1995). Version 2 certificates are no longer used or recommended due to the deprecated re-use of the same certificate from two or more different Certificate Authorities. The BC C# API therefore only supports version 1 and version 3 X.509 certificates. All data within a certificate is encoded using ASN.1 (Abstract Syntax Notation 1), with DER (Distinguished Encoding Rules) used as the output format.

Version 1 certificates provides a basic way of associating (binding) an identity and a validity time to a public key. Version 1 certificates lack: Extensions, the Issuer Unique Identifier and Subject Unique Identifier. The example following creates a basic self signed version 1 X.509 certificate.

Example 51 – Building a Version 1 X.509 Certificate

```
public static X509Certificate makeX509V1Certificate(
    IAsymmetricPrivateKey caSignerKey, IAsymmetricPublicKey caPublicKey)
{
    X509V1CertificateGenerator certGen = new X509V1CertificateGenerator();

    certGen.SetSerialNumber(ExValues.FixedRandomBigInteger(32,1));
    certGen.SetIssuerDN(new X500Name("CN=Issuer Self_Signed_CA"));
    certGen.SetNotBefore(DateTime.UtcNow.AddSeconds(-50));
    certGen.SetNotAfter(DateTime.UtcNow.AddSeconds(50));
    certGen.SetSubjectDN(new X500Name("CN=Subject Self_Signed_CA"));
    certGen.SetPublicKey((AsymmetricECPublicKey)caPublicKey);

    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService((AsymmetricECPrivateKey)caSignerKey,
            ExValues.DHFixedRandomA(1000));
```

```
        ISignatureFactory<FipsEC.SignatureParameters> signer =
            signatureFactoryProvider.CreateSignatureFactory(FipsEC.Dsa);

        return certGen.Generate(new PkixSignatureFactory(signer));

    }
```

In the example above, we can see that the two X.500 Distinguished Name (DN) entries are provided to the `X509V1CertificateGenerator` X509 version 1 generator . In this example they are essentially the same as we have self issued the certificate – that is, we have used our own private key to sign the certificate. The reason for there being two is that the first name is associated with the certificate's issuer, and the second name is associated with the certificate's subject – that is the X.500 name the certificate is for and the public key in the certificate should be associated with. We are also using a random integer as the certificate serial number – any unique number will do here. As the serial number is also used in connection with revocation as well as being part of the certificate identity information in protocols such as CMS it is important you make sure two certificates cannot get handed out with the same serial number. (In any real multi-threaded application a counter would be a much better choice than say the current time. Further if you are using a random integer to ensure uniqueness, keep in mind that the maximum length of a serial number for interoperability is 159 bits.)

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

Example 52 – Building a Version 3 X.509 Certificate

```
public static X509Certificate makeX509V3Certificate(
        X509Certificate caCert,
        IAsymmetricPrivateKey caSignerKey,
        IAsymmetricPublicKey endEntityPublicKey)
{
    X509V3CertificateGenerator certGen = new X509V3CertificateGenerator();
    BigInteger serialNumber = ExValues.FixedRandomBigInteger(32,1);
    certGen.SetSerialNumber(serialNumber);
    certGen.SetIssuerDN(caCert.SubjectDN);
    certGen.SetNotBefore(DateTime.UtcNow.AddSeconds(-50));
    certGen.SetNotAfter(DateTime.UtcNow.AddSeconds(50));
    certGen.SetSubjectDN(new X500Name("CN=End Entity Certifcate"));
    certGen.SetPublicKey((AsymmetricECPublicKey)endEntityPublicKey);

    certGen.AddExtension(X509Extensions.AuthorityKeyIdentifier, false,
        new AuthorityKeyIdentifier(caCert.GetEncoded()));
    certGen.AddExtension(X509Extensions.SubjectKeyIdentifier, false,
        new SubjectKeyIdentifier(endEntityPublicKey.GetEncoded()));
    certGen.AddExtension(X509Extensions.BasicConstraints, true,
        new BasicConstraints(false));
    certGen.AddExtension(X509Extensions.KeyUsage, true,
        new KeyUsage(KeyUsage.DigitalSignature | KeyUsage.KeyEncipherment));

    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService((AsymmetricECPrivateKey)caSignerKey,
            ExValues.DHFixedRandomA(1000));

    ISignatureFactory<FipsEC.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsEC.Dsa);

    return certGen.Generate(new PkixSignatureFactory(signer));
```

```
}
```

Example 52 above provides a way in which to create a certificate chain. In this example we have only two levels – a *parent* certificate and a *child* certificate – the parent certificate is assumed to have been previously generated. For a self-signed certificate we only require an asymmetric key-pair. In contrast, for a certificate chain, we require the parent's certificate (i.e. the authority's certificate), the subject's public key (i.e. the end entities public key) and the parent's (authority's) private key. As previously mentioned the main difference between versions 1 and 3 X.509 certificates, is that version 3 adds a series of extensions to the certificate. These extensions now allow version 3 X.509 certificates to be chained. (Note that version 1 certificates can be chained, however chaining is less reliable. Also note that version 1 certificates are being deprecated and modern libraries are coded to read and write version 3 certificates and read only version 1 certificates.) In our example above, we see four extensions used:

- ***SubjectKeyIdentifier*** – which uniquely identifies the owner of the public key. The Subject Key Identifier is usually derived from the public key. In this example it is a hash derived from the public key of the end entity.
- ***AuthorityKeyIdentifier*** – which is a unique identifier derived from the parent's certificate. In a certificate chain such as this, the Authority Key Identifier in the child certificate must match the Subject Key Identifier in the parent certificate.
- ***BasicConstraints*** – the attribute value here is generally a boolean “true” or “false”. A false value indicates that the certificate is for use by an end entity, while a true value indicates that the certificate is issued for use by a Certificate Authority (a so call Sub-CA, for subordinate CA?).
- ***KeyUsage*** – a bit string which identifies what the key embedded within the certificate can be used for.

Additional extensions can also be added, a full list of the possible extensions can be found in RFC 5280 and its updates. Interpreting some of these can be a bit of a black art though, so if you are building certificate paths (chains), it is worth keeping the extension set required as simple as possible.

Certificate Revocation

At this stage, having followed the examples in this document, you will have worked out how to associate a public key with an identity and start issuing the X.509 certificate – you can almost start your own CA, PKI business ... however, hold off for now. One question that needs to be answered is:
“What happens if it turns out the key on the certificate being handed out somehow becomes invalid before its time, either through compromise of the public key, accidental destruction of the private key, or just simply that the public key owner would prefer to be known by a different X.500 name, or other certificate issuing mistakes?”
If you are going to start deploying/issuing certificates, then this is a question worth asking at the commencement of your project – leaving concerns about invalid certificates until an event like key compromise occurs is likely to result in widespread depression and expression – lookup Apple, Google, GoDaddy and Lets Encrypt for examples of when things go bad.

The basic answer to the question above is the use of certificate revocation lists. A revocation of a certificate is the "operation of revoking" (or the "operation of annulment") of a certificate. Therefore a Certificate Revocation List (CRL) is a list of certificates which are known to be invalid. There are various ways in which such lists are composed, updated and advertised. For example, Certificate Authorities may share their CRL on their website which can then be accessed via some transport protocol such as HTTP, LDAP etc. In fact, there is an X.509 certificate extension known as CRL Distribution Points (see RFC 5280) which allows, for example, a URI pointing to the the CRL. A slightly more sophisticated answer is the use of protocols like Online Certificate Status Protocol (OCSP).

A CRL, to be interoperable, has a particular format (or conforms to a particular standard) – this format (or standard) is provided in RFC 5280 Section 5. The current standard provided in RFC 5280 is version 2 of a CRL. In the following example we are constructing a basic version 2 CRL.

Example 53 – Creating a CRL

```
public static X509Crl CreateCRL(X509Certificate caCert,
        IAsymmetricPrivateKey caSignerKey, X509Certificate[] revokedCerts)
{
    X509V2CrlGenerator crlGen = new X509V2CrlGenerator();
    CrlNumber crlNumber = new CrlNumber(ExValues.FixedRandomBigInteger(32,1));

    crlGen.SetIssuerDN(caCert.SubjectDN);
    crlGen.SetThisUpdate(DateTime.UtcNow);
    crlGen.SetNextUpdate(DateTime.UtcNow.AddSeconds(100));

    crlGen.AddExtension(X509Extensions.AuthorityKeyIdentifier, true,
            new AuthorityKeyIdentifier(caCert.GetEncoded()));
    crlGen.AddExtension(X509Extensions.CrlNumber, false, crlNumber);

    for (int j = 0; j < revokedCerts.Length; ++j)
    {
        crlGen.AddCrlEntry(revokedCerts[j].SerialNumber, DateTime.UtcNow,
                           CrlReason.PrivilegeWithdrawn);
    }

    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService((AsymmetricECPrivateKey)caSignerKey,
            ExValues.DHFixedRandomA(1000));

    ISignatureFactory<FipsEC.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsEC.Dsa);

    return crlGen.Generate(new PkixSignatureFactory(signer));
}
```

The construction of a CRL is similar to that of a construction of a X509 certificate. We observe in the example above we have the CA which issued the certificates (`caCert`), in addition to how long the CRL is up to date for (this is specified with the `SetNextUpdate(...)` method. We have also added two extensions: the *AuthorityKeyIdentifier* (see above) as well as a unique CRL number (which is optional). Again details about CRL extensions are provided in RFC 5280. In this example we have revoked two certificates with an indicated reason for the revocation, the effective date of the revocation and it's unique serial number. It is important to note that both a CRL and protocols like OCSP use the serial number of the certificate being revoked as the key to identify it. This is another reason to make sure you cannot hand out the same serial number twice when creating certificates.

As mentioned above, the Online Certificate Status Protocol (OCSP) is an alternative to the certificate revocation list. OCSP was originally designed as an on-line protocol (Internet Protocol) built on top of CRLs. These days you can find all sorts of things in the back end, but it still has a CRL centered view of the world in terms of how it communicates. The protocol is fully described in RFC 6960. Essentially the OCSP operates as follows:

- A ***request*** is made to determine the validity of a certificate. A request is simply a question regarding the status of a given certificate. Common responses to requests are "good" (certificate is valid), "revoked" or "unknown".
- A ***response*** is made by an OCSP Responder (one or more servers) whose task is to listen to requests, contact a trusted CA and then send back the response "good", "revoked" or "unknown".

A simple OCSP request and response are provided in the next two examples.

Example 54 – Creating an OCSP Request

```
public static OcspReq makeOcspRequest(X509Certificate caCert, X509Certificate[] certsToCheck)
{
    IDigestFactory<AlgorithmIdentifier> sha1Factory =
        new PkixDigestFactoryProvider().CreateDigestFactory(CertificateID.HashSha1);
    CertificateID certID  =
        new CertificateID(sha1Factory, caCert, certToCheck.SerialNumber);

    OcspReqGenerator ocspGen = new OcspReqGenerator();
    for (int j = 0; j < certsToCheck.Length; ++j)
    {
        CertificateID certID  =
            new CertificateID(sha1Factory, caCert, certsToCheck[j].SerialNumber);
        ocspGen.AddRequest(certID);
    }

    return gen.Generate();
}
```

As you can see a basic request requires some information about the CA and also the serial number of the certificates under investigation. Some of the information is hashed for easy processing before being sent to the responder. In this example the request is going to use SHA-1 for the hashing, what hash digest is implemented may vary from server to server however.

Once the responder receives the request, the responder will create a status message comprising of what it knows about the certificate which is then returned back to the client that made the request. One thing to note with a response and that is, that the response must be digitally signed. According to the RFC 6960 documentation, there are various parties (or various private keys) who can sign the response. In the example below, we have used the private key of the CA who has issued the certificates (of the certificates whose status was requested).

Example 55 – Creating an OCSP Response

```
public static BasicOcspResp makeOcspResponse(OcspReq request, X509Certificate caCert,
        IAsymmetricPrivateKey caSignerKey)
{
    BasicOcspRespGenerator respGen = new BasicOcspRespGenerator(caCert.GetPublicKey());

    for (int j = 0; j < request.GetRequestList().Length; ++j)
    {
        CertificateID certID = request.GetRequestList()[j].GetCertID();
        // ...
        // magic happens
        // ...
        respGen.AddResponse(certID, CertificateStatus.Good);
    }

    ISignatureFactoryService signatureFactoryProvider =
        CryptoServicesRegistrar.CreateService((AsymmetricECPrivateKey)caSignerKey,
            ExValues.DHFixedRandomA(1000));

    ISignatureFactory<FipsEC.SignatureParameters> signer =
        signatureFactoryProvider.CreateSignatureFactory(FipsEC.Dsa);

    X509Certificate[] caCertChain = new X509Certificate[1]{caCert};
    return respGen.Generate(new PkixSignatureFactory(signer), caCertChain, DateTime.UtcNow);
}
```

In the example above, it is not possible to provide a full implementation of what processes an OCSP responder would perform. However, if you imagine that the place where the ***magic happens*** comment appears above is the place where the revocation status of the certificate represented by certID is checked, and it turns out that the certificate is still in good standing, what follows after ***magic happens*** is what would probably happen.

So far we have an OCSP request and an OCSP response. The third and final link in the exchange is the client receiving and interpreting the message sent back by the responder for the original certificate (or certificates as in our examples) status request.

Example 56 – Checking an OCSP Response

```
public static bool[] checkCertificates(BasicOcspResp response, X509Certificate caCert,
                                       X509Certificate[] certsToCheck)
{

    bool[] result = new bool[certsToCheck.Length];
    for (int j = 0; j < response.Responses.Length; ++j)
    {
        result[j] = false;
    }

    IVerifierFactoryProvider<AlgorithmIdentifier> verifierFactoryProvider =
        new PkixVerifierFactoryProvider(caCert.GetPublicKey());
    // Check that the signature is valid
    if (
        (!response.IsVerified(verifierFactoryProvider)) ||
        (response.Responses.Length != certsToCheck.Length)
       )
    {
        return result;
    }

    IDigestFactory<AlgorithmIdentifier> sha1Factory =
        new PkixDigestFactoryProvider().CreateDigestFactory(CertificateID.HashSha1);

    for (int j = 0; j < response.Responses.Length; ++j)
```

```
    {
        if (
            (response.Responses[j].GetCertStatus() == null) &&
            (response.Responses[j].GetCertID().SerialNumber.Equals(certsToCheck[j].SerialNumber)) &&
            (response.Responses[j].GetCertID().MatchesIssuer(sha1Factory,caCert))
           )
        {
           result[j] = true;
        }
    }

    return result;
}
```

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

Certification Path Validation

In the previous examples we have demonstrated how one can check an individual X509 certificate which has been signed by a private key, by using the embedded public key. In actual applications, things get slightly more complicated with the concept of sub CAs (subordinate Certificate Authorities). This concept then results in a certificate chain, also known as the *certification path*. This chain (or path) is a list of certificates which is used to authenticate an End Entity (EE). The path commences from a root CA certificate (a trusted anchor) – as mentioned previously, the root CA certificate is self-signed and this certificate must be accepted at face value. The validation of the End Entity starts at the End Entity's certificate and works its way up to the root CA certificate. In the simplest case we have only two certificates: the root CA certificate and the EE's (client's) certificate.

The next example shows how to validate a certificate path consisting of a Trust Anchor (root CA), a SUB-CA certificate and an End Entity certificate. The validation of the path is based upon the certificate path validation algorithm described in RFC 5280.

Example 57 – Basic Certification Path Validation

```
public static PkixCertPathValidatorResult validateCertPath(X509Certificate taCert,
                    X509Certificate caCert, X509Certificate eeCert)
{
    List<X509Certificate> certChain = new List<X509Certificate>();
    certChain.Add(eeCert);
    certChain.Add(caCert);
    PkixCertPath certPath = new PkixCertPath(certChain);

    HashSet<TrustAnchor> trustAnchors = new HashSet<TrustAnchor>();
    trustAnchors.Add(new TrustAnchor(taCert, null));

    PkixParameters param = new PkixParameters(trustAnchors);
    param.Date = new DateTimeObject(DateTime.UtcNow);
    param.IsRevocationEnabled = false;
    PkixCertPathValidator certPathValidator = new PkixCertPathValidator();
    PkixCertPathValidatorResult result = certPathValidator.Validate(certPath, param);

    return result;
}
```

As per RFC5280, observe that the trust anchor (root CA – which is a self-signed certificate) is passed as a separate parameter from the sub CA and End Entity certificates. Also note that all X509 certificates should version 3.

In the above example, you will observer that the CRLs are not being referenced due to the statement `param.IsRevocationEnabled = false;`. Had we wanted to process CRLs as well (and therefore determine their validity apart from their authentication) we would have also needed to pass the CRLs to the `PkixParameters` object using a CertStore which can be used to contain certificates and CRLs. You can see in the following example that a CertStore is introduced and then used to make the CRLs available for validating certificates issued under the trust anchor and the CA certificate.

Example 58 – Basic Certification Path Validation with CRLs

```
public static PkixCertPathValidatorResult validateCertPathWithCrl(
                                X509Certificate taCert, X509Crl taCrl,
                                X509Certificate caCert, X509Crl caCrl,
                                X509Certificate eeCert)
{
    List<X509Certificate> certChain = new List<X509Certificate>();
    certChain.Add(eeCert);
    certChain.Add(caCert);
    PkixCertPath certPath = new PkixCertPath(certChain);

    HashSet<TrustAnchor> trustAnchors = new HashSet<TrustAnchor>();
    trustAnchors.Add(new TrustAnchor(taCert, null));

    List<X509Crl> x509Crls = new List<X509Crl>();
    x509Crls.Add(caCrl);
    x509Crls.Add(taCrl);
    X509CrlCollectionStore x509CrlStore = new X509CrlCollectionStore(x509Crls);

    PkixParameters param = new PkixParameters(trustAnchors);
    param.Date = new DateTimeObject(DateTime.UtcNow);
    param.IsRevocationEnabled = true;
    param.AddCrlStore(x509CrlStore);

    PkixCertPathValidator certPathValidator = new PkixCertPathValidator();
    PkixCertPathValidatorResult result = certPathValidator.Validate(certPath, param);

    return result;
}
```

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

MORE TO FOLLOW