Data Security using Encryption

Encryption classes

Beginners will feel awed and even intimidated with the amount of material available on the internet on encryption. Let's turn down the noise by putting down the following tables.

These tables enumerate the classes that can be used to make sure that any data is transferred securely. The reason that we have written these at the outset is to get them out of our system, for now. We will be referring to these tables as we go forward. But our focus will be on the data security mechanisms rather than learning the names of the algorithms and corresponding classes, at every step.

Encryption algorithm classes		
	Symmetric Key Algorithm	**Asymmetric Key Algorithm**
Managed	RijndaelManaged AesManaged	
API Wrappers	DESCryptoServiceProvider TripleDESCryptoServiceProvider RC2CryptoServiceProvider	RSACryptoServiceProvider
Only for digital signature		DSACryptoServiceProvider

Hash algorithm classes		
	Non-Keyed	**Keyed**
Use these	SHA512Managed SHA384Managed SHA256Managed	HMACSHA512 HMACSHA384 HMACSHA256 MACTripleDES
These are no	SHA1Managed (160 bits)	HMACSHA1

longer considered secure	RIPEMD160Managed MD5CryptoServiceProvider	HMACRIPEMD160 HMACMD5

Randomness generator classes	
Pseudorandom bytes generator	RFC2898DeriveBytes
Random bytes generator	RNGCryptoServiceProvider

Managed Classes Vs API Wrappers

Any class suffixed with `CryptoServiceProvider` is just a .Net class with encryption functions that simply call the unmanaged Microsoft Cryptography API encryption functions. Any class suffixed with `Managed` is a .Net class with functions which encrypt data without using the Windows API functions. Needless to say that the `Managed` classes should be preferred to the `CryptoServiceProvider` classes.

Data Encryption

There are 2 types of algorithms used to encrypt data. These are: Symmetric Key Algorithms and Asymmetric Key Algorithms. Here are the steps involved in using them:

Symmetric Key Algorithm

1. One party creates a key.
2. This key is shared with the second party.
3. Each party encrypts data with this key before sending it to the other party
4. Each party decrypts the data received from the other party, using this key.

Asymmetric Key Algorithm

1. One party creates a key. Let's call it *private key*.
2. The same party generates a second key *from the first key*. Let's call it *public key*.
3. The public key is shared with the second party.

4. The second party encrypts data with the public key, before sending it to the first party
5. The first party decrypts the data received from the second party, using the private key.

Insights

1. Data transfer involving 1 asymmetric key works only in 1 direction. This is because:
   (a) Data encrypted using the public key can only be decrypted by the private key, which only 1 party possesses.
   (b) Data is never encrypted by the private key.

   To make 2-way encrypted data transfer possible, 2 set of public and private keys will be required, thus doubling the work for a key administrator.
2. Asymmetric key generally has more number of bits. This makes the data more secure (equivalent to rubbing more color on one's face to hide one's identity). But this also slows the process of encryption/decryption.
3. In the asymmetric key algorithms, data encrypted using one key is decrypted using another key. This is the reason that they are called *asymmetric key* algorithms. How is this done? The answer to this question is out of the scope of this book. But you can vaguely follow the mechanism on Wikipedia.
   A simplified (although not true) explanation goes like this:
   (a) Remember that the public key was created from the private key.
   (b) The algorithm has been devised in a way that at the time of decryption, the public key can be created again and thus decryption can be done.
4. Multiple public keys can be generated from one private key. Continuing the simplified explanation given above, we can say that this delays the process of decryption even more.

Because of these challenges, asymmetric key algorithms are not used to encrypt a large amount of data. Usually they are used to transfer/share the key of a symmetric key algorithm and then the data is exchanged using this symmetric key algorithm.

Symmetric Encryption

The phrase *Data Encryption using a Symmetric Key Algorithm* is usually shortened to *Symmetric Key Encryption* and even to *Symmetric Encryption*. That is the phrase that we will use from here on. There are 4 steps involved in symmetric encryption. Those are:

1. Get the data to be encrypted.
2. Create a cryptor.
3. Use the cryptor to create a `CryptoStream` to which the encrypted data will be written. It is called a `CryptoStream` because any data written to it will be encrypted.
4. Write the data to this `CryptoStream`.

In terms of code, this is how we do it:

Step 1 is a simple file handling mechanism.

Step 2 involves 4 substeps:

a) Create an instance of a `SymmetricAlgorithm` subclass, say `SymAlg`.

b) Make a key, say `ArrBKey` – a byte array with the size equal to the `KeySize` property of the `SymAlg`.

c) Make an IV, say `ArrBIV` – a byte array with the size equal to the `BlockSize` property of the `SymAlg`.

d) Create an `ICryptoTransform` instance, say `Cryptor`, using the `SymAlg`, the `ArrBKey` and the `ArrBIV`.

Step 3 involves 2 substeps:

a) Open a FileStream (or any other stream) for writing, say `fsOut`.

b) Create a `CryptoStream` instance using `fsOut` and `Cryptor`.

Step 4 is a simple file writing mechanism. The only extra step involved is to flush the last block of data using the `FlushFinalBlock` method.

Sample Code

Namespaces required:

```
// Namespace containing Encoding
using System.Text;
// Namespace containing the SymmetricAlgorithm, RijndaelManaged,
//    ICryptoTransform, CryptoStream
using System.Security.Cryptography;
// Namespace for the file handling mechanisms
using System.IO;
```

Code:

```
private void EncryptFile(string sFileIn, string sFileOut) {
    // Step 1
    FileStream fsIn;
    try {
        fsIn = new FileStream(sFileIn, FileMode.Open, FileAccess.Read);
    }
    catch { return; }
    // Step 2a
    SymmetricAlgorithm SymAlg = new RijndaelManaged();
    // Step 2b
    char[] sKeyAndIV = "Some Key and IV. For key after..".ToCharArray();
    byte[] ArrBKey =
        Encoding.ASCII.GetBytes(sKeyAndIV, 0, SymAlg.KeySize / 8);
    // Step 2c
    byte[] ArrBIV =
        Encoding.ASCII.GetBytes(sKeyAndIV, 0, SymAlg.BlockSize / 8);
    // Step 2d
    ICryptoTransform Cryptor = SymAlg.CreateEncryptor(ArrBKey, ArrBIV);
    // Step 3a
    FileStream fsOut;
    try {
        fsOut = new FileStream(sFileOut, FileMode.OpenOrCreate);
    }
    catch { fsIn.Close(); return; }
    try {
        // Step 3b
        CryptoStream cs =
            new CryptoStream(fsOut, Cryptor, CryptoStreamMode.Write);
        // Step 4
        byte[] ArrBInput = new byte[1024];
        int iBytes;
        while ((iBytes = fsIn.Read(ArrBInput, 0, 1024)) > 0) {
            cs.Write(ArrBInput, 0, iBytes);
        }
        // Extra step - only for encryption, not for decryption
        cs.FlushFinalBlock();
    }
    finally { fsIn.Close(); fsOut.Close(); }
}
```

Tip: The `CreateEncryptor`/`CreateDecryptor` methods of a Managed class instance create a `RijndaelManagedTransform` instance and of a CSP instance create a `CryptoAPITransform` instance.

How to decrypt the data?

The only way to test our encryption code is to perform decryption. If we get back what we sent, then the code is working fine. The basic 4 steps will remain the same. Just keep in mind that now, we are reading the encrypted file, rather than creating it. Here is how this code will look like.

```
private void DecryptFile(string sFileIn, string sFileOut) {
    // Step 1 – replaced by Step 3a of EncryptFile
    FileStream fsOut;
    try {
        fsOut = new FileStream(sFileOut, FileMode.OpenOrCreate);
    }
    catch { return; }
    // Step 2a
    SymmetricAlgorithm SymAlg = new RijndaelManaged();
    // Step 2b
    char[] sKeyAndIV = "Some Key and IV. For key after..".ToCharArray();
    byte[] ArrBKey =
        Encoding.ASCII.GetBytes(sKeyAndIV, 0, SymAlg.KeySize / 8);
    // Step 2c
    byte[] ArrBIV =
        Encoding.ASCII.GetBytes(sKeyAndIV, 0, SymAlg.BlockSize / 8);
    // Step 2d – CreateDecryptor instead of CreateEncryptor
    ICryptoTransform Cryptor = SymAlg.CreateDecryptor(ArrBKey, ArrBIV);
    // Step 3a – replaced by Step 1 of EncryptFile
    FileStream fsIn;
    try {
        fsIn = new FileStream(sFileIn, FileMode.Open, FileAccess.Read);
    }
    catch { fsOut.Close();  return; }
    try {
        // Step 3b – Read rather than Write
        CryptoStream cs =
            new CryptoStream(fsIn, Cryptor, CryptoStreamMode.Read);
        // Step 4 – Read rather than write
        byte[] ArrBInput = new byte[1024];
        int iBytes;
        while ((iBytes = cs.Read(ArrBInput, 0, 1024)) > 0) {
            fsOut.Write(ArrBInput, 0, iBytes);
        }
        // No need to flush
    }
    finally { fsIn.Close(); fsOut.Close(); }
}
```

Advanced Analyses

What is an IV?

IV stands for Initialization Vector. Paraphrasing from Wikipedia: an IV is a byte array that is required to allow a block of data to be encrypted to produce a unique stream independent from other streams produced by the same encryption key, without having to go through a (usually lengthy) re-keying process.

How the algorithm does this is beyond the scope of this book.

Note: The introduction of IV calls for updating the *Symmetric Key Algorithm* subsection of the *Data Encryption* section. Just replace the word *key* with the

phrase *key and IV*, and we will be all set.

Creating truly random Key and IV

In the encryption code, we just took a string and converted it into byte arrays which we assigned to the Key and the IV. Then, we used the same method for decryption. Microsoft has provided 2 better methods to create the Key and the IV. These are discussed below. The key creation and storage should be a separate step, followed by repeated use of the key during both encryption and decryption. For the sake of simplicity, we are keeping the key-creation a part of the encryption process.

Method 1:

1. At the time of encryption, call the `GenerateKey` method of the `SymmetricAlgorithm` instance, say `SymAlg`. This creates a set of random `Key` and `IV` for the `SymAlg`.
2. Use the overloaded `CreateEncryptor` method, in which the `Key` and the `IV` are not specified. Internally, the `Cryptor` is created using the `Key` and the `IV` of the `SymAlg`.
3. Make sure that the `Key` and the `IV` are stored to a file before encrypting. Remember, the decrypting party needs them. Since they are random byte arrays, we cannot regenerate them without storing them somewhere.

The following sample code snippets replace the Step 2 of the full Sample Codes discussed earlier.

Sample code snippet for encryption:

```
// Step 2a
SymmetricAlgorithm SymAlg = new RijndaelManaged();
// Steps 2b and 2c - replace
SymAlg.GenerateKey();
// Extra step – Store the key and the IV by writing to a file
//    Implement this method as you like
this.StoreKeyIVToFile(SymAlg, "C:/Temp/KeyAndIV.bin");
// Step 2d – Remove the arguments
ICryptoTransform Cryptor = SymAlg.CreateEncryptor();
```

Sample code snippet for decryption:

```
// Step 2a
SymmetricAlgorithm SymAlg = new RijndaelManaged();
// Steps 2b and 2c - read the key and the IV from the file
//        provided by the encrypting party
//    Implement this method as you like
```

```
this.ReadKeyIVFromFile(SymAlg, "C:/Temp/KeyAndIV.bin");
// Step 2d – Remove the arguments
ICryptoTransform Cryptor = SymAlg.CreateDecryptor();
```

Helper Code:

```
private void StoreKeyIVToFile(SymmetricAlgorithm SymAlg, string sFileKeyAndIV) {
    FileStream fsOut;
    try { fsOut = new FileStream(sFileKeyAndIV, FileMode.OpenOrCreate); }
    catch { return; }
    try {
        fsOut.Write(SymAlg.Key, 0, SymAlg.KeySize / 8);
        fsOut.Write(SymAlg.IV, 0, SymAlg.BlockSize / 8);
    }
    finally { fsOut.Close(); }
}

private void ReadKeyIVFromFile(SymmetricAlgorithm SymAlg, string sFileKeyAndIV)
{
    FileStream fsIn;
    try { fsIn = new FileStream(sFileKeyAndIV, FileMode.Open, FileAccess.Read); }
    catch { return; }
    try {
        byte[] ArrBKey = new byte[SymAlg.KeySize / 8];
        fsIn.Read(ArrBKey, 0, SymAlg.KeySize / 8);
        SymAlg.Key = ArrBKey;
        byte[] ArrBIV = new byte[SymAlg.BlockSize / 8];
        fsIn.Read(ArrBIV, 0, SymAlg.BlockSize / 8);
        SymAlg.IV = ArrBIV;
    }
    finally { fsIn.Close(); }
}
```

Method 2:

1. Create the following 4 variables:
   (a) A string, say `sPasswordShare`.
   (b) A positive integer, say `iIterationsShare`.
   (c) A string, say `sDummyShare`, of more than 8 characters.
   (d) A byte array derived from `sDummyShare`, say `ArrBSalt`.
2. Share the first 3 variables between the parties.
3. At both the time of encryption and decryption, create an instance of `RFC2898DeriveBytes`, say `rdb`, using `sPasswordShare`, `iIterationsShare` and `ArrBSalt`.
4. Use the `GetBytes` method of `rdb` to create two pseudorandom arrays of bytes – `ArrBKey` and `ArrBIV`. They are called pseudorandom arrays because they can be created with complete fidelity by providing a unique set of `sPasswordShare`, `iIterationsShare` and `ArrBSalt` to an `RFC2898DeriveBytes` instance. Nevertheless, `RFC2898DeriveBytes` assures that the `sPasswordShare` and the `ArrBSalt` cannot be derived from these random arrays. The higher the value of `iIterationsShare`, the more difficult it is to derive these two variables.

5. Use the `ArrBKey` and `ArrBIV` to create the `Cryptor` as done earlier.

The following sample code snippets replace the Step 2 of the full Sample Code discussed earlier.

Sample code snippet for encryption:

```
// Step 2a
SymmetricAlgorithm SymAlg = new RijndaelManaged();
// Step 2b
string sPasswordShare = "Password shared between the parties.";
int iIterationsShare = 2000;
string sDummyShare = "Another string to be shared between the parties.";
byte[] ArrBSalt = Encoding.ASCII.GetBytes(sDummyShare);
Rfc2898DeriveBytes rdb = new Rfc2898DeriveBytes(sPasswordShare, ArrBSalt,
iIterationsShare);
byte[] ArrBKey = rdb.GetBytes(SymAlg.KeySize / 8);
// Step 2c
byte[] ArrBIV = rdb.GetBytes(SymAlg.BlockSize / 8);
// Step 2d
ICryptoTransform Cryptor = SymAlg.CreateEncryptor(ArrBKey, ArrBIV);
```

Sample code snippet for decryption:

```
// Step 2a
SymmetricAlgorithm SymAlg = new RijndaelManaged();
// Step 2b
string sPasswordShare = "Password shared between the parties.";
int iIterationsShare = 2000;
string sDummyShare = "Another string to be shared between the parties.";
byte[] ArrBSalt = Encoding.ASCII.GetBytes(sDummyShare);
Rfc2898DeriveBytes rdb = new Rfc2898DeriveBytes(sPasswordShare, ArrBSalt,
iIterationsShare);
byte[] ArrBKey = rdb.GetBytes(SymAlg.KeySize / 8);
// Step 2c
byte[] ArrBIV = rdb.GetBytes(SymAlg.BlockSize / 8);
// Step 2d
ICryptoTransform Cryptor = SymAlg.CreateDecryptor(ArrBKey, ArrBIV);
```

Note:

1. Instead of sharing the password, the iteration count and the dummy string, you could have shared the key and the IV by first storing them to a file, as done in Method 1.
2. Another class called `PasswordDeriveBytes` also does this for us. Its constructor does not have a parameter for the iteration count.

Asymmetric Encryption

Sadly, the technique used for asymmetric encryption is different from that for the symmetric encryption. But the good part is that the technique is very easy – as easy as calling the `Encrypt` method. Since

Microsoft is cutting us a slack here, let's use that in creating an almost complete scenario for asymmetric encryption.

Here is how it goes:

1. Party 1 creates a key and stores both its public and the private parameters in one file, say `AsymKey.xml` and only its public parameters in another file, say `AsymKey.xml.public`.
2. Party 1 provides the `AsymKey.xml.public` to Party 2, usually through a channel different from the one through which the data is to be transferred. We do not have to code for this.
3. Party 2 encrypts the data using the `AsymKey.xml.public` file.
4. Party 1 decrypts the data using the `AsymKey.xml` file.

Action 1 consists of 3 steps:

1. Create an instance of `AsymmetricAlgorithm` subclass, say `AsymAlg`.
2. Call the `ToXmlString` method of `AsymAlg` with the argument `includePrivateParameters` as true. This will create an XML string containing both the public and the private parameters of the key. This XML should be stored in a file.
3. Call the `ToXmlString` method of `AsymAlg` with the argument `includePrivateParameters` as false. This will create an XML string containing only the public parameters of the key. This XML should be stored in a separate file.

Action 3 consists of 5 steps:

1. Get the data to be encrypted.
2. Create an instance of `AsymmetricAlgorithm` class, say `AsymAlg`.
3. Read the file containing only the public parameters into a string. Call the `FromXmlString` method of `AsymAlg` by passing this string. This will configure `AsymAlg` with the public key.
4. Create the output stream.
5. Encrypt the data and write to the appropriate output stream.

Step 5 needs the following explanation:

a) Asymmetric encryption can only work on a data which has size equal to or less than the size of the key minus the padding that the algorithm is going to put on the data. As a thumb rule, the padding takes up 64 bytes of space.

b) There are 2 types of padding: PKCS (default) and OAEP (for Microsoft Windows XP or higher). The PKCS padding can be used by passing `false` as the second argument of the `Encrypt` method and the OAEP padding can be used by passing `true` as the second argument of the Encrypt method.

Action 4 is just a mirror image of Action 3.

Sample code:

Namespaces:

```
using System.Text;
using System.Security.Cryptography;
using System.IO;
```

Action code:

Action 1

```
// Includes Step 1
StoreAsymKeyToFile(new RSACryptoServiceProvider(), C:\Temp\AsymKey.xml");
```

Action 3

```
EncryptFileAsym("C:\Temp\FileToEncode.txt", "C:\Temp\EncodedFile.enc",
"C:\Temp\AsymKey.xml.public");
```

Action 4

```
EncryptFileAsym("C:\Temp\EncodedFile.enc", "C:\Temp\SameAsOriginal.txt",
"C:\Temp\AsymKey.xml");
```

Step code:

```
private void StoreAsymKeyToFile(
    AsymmetricAlgorithm AsymAlg, string sFileAsymKey
) {
    StreamWriter swKey;
    try { swKey = new StreamWriter(sFileAsymKey); }
    catch { return; }
    try
    {
        // Step 2
        swKey.Write(AsymAlg.ToXmlString(true));
        swKey.Close();
        // Step 3
        swKey = new StreamWriter(sFileAsymKey + ".public");
        swKey.Write(AsymAlg.ToXmlString(false));
    }
    finally { swKey.Close(); }
}
```

```
private void ReadAsymKeyFromFile(
    AsymmetricAlgorithm AsymAlg, string sFileKey
) {
    StreamReader srKey;
    try { srKey = new StreamReader(sFileKey); }
    catch { return; }
    try {
        AsymAlg.FromXmlString(srKey.ReadToEnd());
    }
    finally { srKey.Close(); }
}

private void EncryptFileAsym(
    string sFileIn, string sFileOut, string sFilePublicKey
) {
    // Step 1
    FileStream fsIn;
    try {
        fsIn = new FileStream(sFileIn, FileMode.Open, FileAccess.Read);
    }
    catch { return; }
    // Step 2 - Encrypt function is not available with
    //     AsymmetricAlgorithm class, so use RSACryptoServiceProvider
    RSACryptoServiceProvider AsymAlg = new RSACryptoServiceProvider();
    // Step 3 - uses FromXmlString
    this.ReadAsymKeyFromFile(AsymAlg, sFilePublicKey);
    // Step 4
    FileStream fsOut;
    try {
        fsOut = new FileStream(sFileOut, FileMode.OpenOrCreate);
    }
    catch { fsIn.Close(); return; }
    try {
        int iBytes = AsymAlg.KeySize / 8 - 64;
        byte[] ArrBInput = new byte[iBytes];
        while ((iBytes = fsIn.Read(ArrBInput, 0, iBytes)) > 0) {
            ArrBInput = this.GetPortion(ArrBInput, 0, iBytes);
            // Step 5
            byte[] ArrBTemp = AsymAlg.Encrypt(ArrBInput, true);
            fsOut.Write(ArrBTemp, 0, ArrBTemp.Length);
        }
    }
    finally { fsIn.Close(); fsOut.Close(); }
}

private void DecryptFileAsym(
    string sFileIn, string sFileOut, string sFileFullKey
) {
    // Steps 1, 2, 3, 4 same as in EncryptFileAsym
    try
    {
        byte[] ArrBInput = new byte[AsymAlg.KeySize / 8 ];
        while (fsIn.Read(ArrBInput, 0, AsymAlg.KeySize / 8) > 0)
        {
            // Step 5
```

```
            byte[] ArrBTemp = AsymAlg.Decrypt(ArrBInput, true);
            fsOut.Write(ArrBTemp, 0, ArrBTemp.Length);
        }
    }
    finally { fsIn.Close(); fsOut.Close(); }
}
```

Helper Code:

```
private byte[] GetPortion(byte[] ArrObj, int iOffset, int iLength) {
    if (ArrObj == null) return ArrObj;
    if (iLength < 0 || ArrObj.Length <= iOffset + iLength) {
        if (iOffset == 0) return ArrObj;
        iLength = ArrObj.Length - iOffset;
    }
    MemoryStream memstrm = new MemoryStream();
    memstrm.Write(ArrObj, iOffset, iLength);
    return memstrm.ToArray();
}
```

Advanced Analyses

Parameters of Asymmetric Algorithm

The only difficult part compared to the code for symmetric encryption is the storage of the key. The key is not even a property of `AsymAlg`. As a matter of fact, the key has so many parts that it is called *Parameters,* as in `RSAParameters` (a `struct`) or `CspParameters` (a `class`). Even these parameters cannot be accessed by some simple property. The only way to get or set them in code is to use the `ExportParameters` or the `ImportParameters` methods of an `AsymmetricAlgorithm` instance. The other way is of course by using the `ToXmlString` and the `FromXmlString` methods.

Let's take a quick look at a set of asymmetric parameters. Here is a fake though representative one:

```
<RSAKeyValue>
  <Modulus>
xlfEkovHbtxcKDP/THmqMeKMMDYmR4I+ZW2DXoGe+UD3QQZ8VAUShrKJ9X3TFDI2X0XCs+VTJ
DhyK49bFfVOcKMp4ZzXaVBZRnI2Q920EcyVFtuR95yVQdU8EZfyNSvDyavEebnA1heoMWoPhv
MRNwYp0hhDnYvopAQVERdt8Nc=</Modulus>
  <Exponent>ABCD</Exponent>
  <P>
7BH2OkjwTIw0Im1XxLnJxq8HkFSBvj7J8dDMqD53crvdw9iu5sDOSW9oXR0Da22qml8/Z8WkO
INGbSaNskdOTQ==</P>
  <Q>
2mHnlUWXUcOEQJfLXRla+THxK44pvTML0vrVrdjvwAC5D5tdHwxWvt0T+SPneOsQGcLwXXdam
yuQUYs1im11sw==</Q>
  <DP>
6YueudgbmBu2FbeB8Ryz8Mp/+Msd6kfBCGc3h9MB44LfwDA8EBL1+L0xxahFEaXP8hH0bXiZD
y0KJDzWO1k8oQ==</DP>
  <DQ>
m5h90WF3gnr2X1K3nKrcWNDHEJyjyUV6fAOh7CA2URhxxKaBb/4irS7Z+9g2y5u1M/JUVFOc/
5MLp44AG1YR3w==</DQ>
```

```
  <InverseQ>
OwrEZICTxPyqql0U4KuyE6DCypkyxfcLHHQtxWIzF++USnT0M15L8RgtpP+mvwhvToBUZO+lO
P4gMINl8GTFIw==</InverseQ>
  <D>
J1IeSOAcq/piyW/B251VIGHjRrPguyhhvL9tm9SFFxoGlgvetsqBNnfmePaYUVha52/fViC8v
p1/qokQbeo6ig8JQaGiuFpKQEZkkGV1GMAxUDREANsY+//031Oht8Ol2kjUPUvyY2C+MyfZai
WGxBmIh6j9UAV9g+AzjyfMdek=</D>
</RSAKeyValue>
```

The Modulus and the Exponent constitute the *public key*. The rest constitute the *private key*.

KeySize

1. The main difference between the algorithms listed in the tables of classes is the size of their keys. The bigger the `KeySize`, the more secure is the encryption.
2. `KeySize` is always in bits.
3. The `KeySize` can be configured for some algorithms like `RijndaelManaged` and `RSACryptoServiceProvider`.

Hierarchy

Let's have a look at the table of encryption classes. Each encryption class inherits from a class, the name of which can be arrived at by removing the *Managed* or the *CryptoServiceProvider* suffix. Those parent classes themselves inherit from the `SymmetricAlgorithm` or the `AsymmetricAlgorithm` abstract class, as appropriate.

Hashing

Hashing is a quick way of testing if a piece of data sent by a party was tampered on the way to the second party. This is done by computing out a very small piece of data from a larger data, using a complex algorithm. Every time this algorithm is applied to a particular data, the result will be the same small piece of data. This small piece of data is called the *hash* of the larger data.

Logical Steps

1. The 2 parties agree on the hash algorithm that will be used.
2. Party 1 computes the hash of the data to be sent, using this algorithm.

3. Party 1 sends the data to Party 2.
4. Party 1 sends the hash to Party 2. This could also have been done in a recent or distant past, or in the case of a password, Party 2 might have calculated and stored the hash in the past.
5. Party 2 computes the hash of the received data, using the same algorithm.
6. Party 2 compares the computed and the received hashes. In case they are identical, Party 2 processes the data. In case they are different, Party 2 takes an appropriate action like alerting Party 1 or at least not processing the data.

Why do we need Hashing?

Data integrity

Let's say some data is sent across as an email or as a web page, followed by the hash of the same data. With the use of the above mentioned process, the data integrity over the internet can be checked. The small size of the hash is usually an assurance that the hash itself will be transferred fine over the internet. Even if the hash is broken, and thus the data integrity is believed to be broken, Party 2 can ask Party 1 to send the data once again.

Password security

Party 1 can have some password to access some resource of Party 2. At the time of creation of the password, Party 2 can create its hash and store this hash. At the time of the use of the password, Party 2 can hash the sent password and compare it against the stored hash. If they are identical, Party 1 can be allowed the access to the resource. Thus, even the password administrator will never know any password.

Security

Hashing can be used to check if a data has been tampered by a hacker. This will be explained in the *Keyed Hashing* section.

What is there to program then?

The only program that is to be written is to compute the hash, which is actually easier done than said! That's all that will be executed by the two parties, followed by a byte-array-comparison by Party 2.

Sample code

Namespaces:

```
using System.Text;
using System.Security.Cryptography;
using System.IO;
```

Code:

```
private void HashFile(string sFileIn, string sFileOut) {
    // Step 1
    FileStream fsIn;
    try {fsIn = new FileStream(sFileIn, FileMode.Open, FileAccess.Read);}
    catch { return; }
    // Step 2
    HashAlgorithm HshAlg = new SHA512Managed();
    // Step 3
    FileStream fsOut;
    try { fsOut = new FileStream(sFileOut, FileMode.OpenOrCreate); }
    catch { fsIn.Close(); return; }
    try {
        // Step 4
        byte[] ArrBHsh = HshAlg.ComputeHash(fsIn);
        fsOut.Write(ArrBHsh, 0, ArrBHsh.Length);
    }
    finally { fsIn.Close(); fsOut.Close(); }
}
```

A Sample Hash

＃旱僻□躭乆打 穩便劧□剳 + 牵□ᄉᄀ連譁ª□踖藾豊□□晴瞜欝螅

Code analyses

1. `ComputeHash` has a few overloads. We can either read the entire file into a byte array and then pass it as an argument, or we can pass a FileStream as an argument, as in the Sample Code.
2. If you run the code with the same input but a different output file, the hash will remain the same as the above.

3. The output file will be of the same size as the hashing algorithm name. For example, in case of the above code, the hash file is 512 bits (64 bytes) in size.

Tip: The last computed hash is stored in the `Hash` property of a `HashAlgorithm` instance.

Keyed Hashing

The hashing process that we described above is used mainly to check the data integrity. It does not protect us from a hacker. The process can be easily replicated by a hacker. He/she can then send a tampered data and its hash. The hash calculated by Party 2 will match the hash for the tampered data (GIGO).

The solution is that the two parties exchange a secret key and use that to first hash the data followed by encrypting the hash. Simply replacing the `HashAlgorithm` with the `KeyedHashAlgorithm` and providing it with a key does this for us.

Code snippet

```
private void KeyedHashFile(string sFileIn, string sFileOut) {
    // Step 1 – same as in Hashing
    // Step 2a – creation of a key
    string sPasswordShare = "Password shared between the parties.";
    int iIterationsShare = 2000;
    string sDummyShare =
        "Another string to be shared between the parties.";
    byte[] ArrBSalt = Encoding.ASCII.GetBytes(sDummyShare);
    Rfc2898DeriveBytes rdb =
        new Rfc2898DeriveBytes(sPasswordShare,ArrBSalt,iIterationsShare);
    byte[] ArrBKey = rdb.GetBytes(512);
    // Step 2b – KeyedHashAlgorithm instead of HashAlgorithm
    KeyedHashAlgorithm HshAlg = new HMACSHA512(ArrBKey);
    // Steps 3 and 4 – same as in Hashing
}
```

Code analysis

All that we have done is added the code to create a key and provided it to a `KeyedHashAlgorithm` constructor. We could also have created a `KeyedHashAlgorithm` instance and then assigned the key to its `Key` property.

We can use any of the methods we explored in the *Symmetric Encryption* section, to create a key.

Hierarchy

Let's have a look at the table of classes. Each hash algorithm class inherits from a class, the name of which can be arrived at by removing the *Managed* or the *CryptoServiceProvider* suffix. Those parent classes themselves inherit from the `HashAlgorithm` or the `KeyedHashAlgorithm` abstract class, as appropriate. The `KeyedHashAlgorithm` class inherits from the `HashAlgorithm` class.

Digital Signature

What is a digital signature?

A digital signature is a byte array generated for some data, using this data and a key.

That sounds like a keyed hash. Is it different?

There is 1 *Yes* and 2 *No*'s to this question.

1. Yes: the data is hashed and the hash is encrypted as in keyed hashing.
2. No: the difference is the same as that between symmetric and asymmetric key algorithms.
   In a keyed hash, a private key is used to create and recreate (for verification) the hash. In a digital signature, the private key is used to create the signature and a public key is used to verify the signature. As a matter of fact, digital signatures are actually created and verified using asymmetric key algorithms.
3. No: A keyed hash for a piece of data will always be the same. On the other hand, a digital signature created for the same piece of data will be different, every time it is created. Nevertheless, all these signatures will verify correctly.

Logical steps

1. The 2 parties agree on the asymmetric algorithm that will be used.
2. Party 1 creates a private and a public key. It shares the public key with Party 2.
3. Party 1 creates a digital signature of the data to be sent, using the algorithm agreed upon, which itself is created using the *private* key.

4. Party 1 sends the data and the digital signature to Party 2.
5. Party 2 verifies the data with the digital signature using the algorithm agreed upon, which itself is created using the *public* key.

As you can see, these steps are simply a combination of keyed hashing and the asymmetric encryption/decryption processes.

Sample code:

Namespaces:

```
using System.Text;
using System.Security.Cryptography;
using System.IO;
```

Action code:

```
StoreAsymKeyToFile(new DSACryptoServiceProvider(),
“C:\Temp\DigitalSignatureAsymKey.xml”);

SignFile("C:\Temp\FileToSign.txt", "C:\Temp\SignatureFile.txt",
“C:\Temp\DigitalSignatureAsymKey.xml”);

VerifySignFile("C:\Temp\FileToSign.txt", "C:\Temp\SignatureFile.txt",
“C:\Temp\DigitalSignatureAsymKey.xml.public”);
```

Step code:

```
// Implementation of the StoreAsymKeyToFile and the ReadAsymKeyFromFile
methods is the same as in the Asymmetric Encryption section.

private void SignFile(
    string sFileIn, string sFileSignature, string sFilePrivateKey
) {
    // Step 1
    byte[] ArrBInput;
    try { ArrBInput = File.ReadAllBytes(sFileIn); }
    catch { return; }
    // Step 2 – SignData function is not available with
    //     AsymmetricAlgorithm class, so use DSACryptoServiceProvider
    DSACryptoServiceProvider AsymAlg = new DSACryptoServiceProvider();
    // Step 3 - uses FromXmlString
    this.ReadAsymKeyFromFile(AsymAlg, sFilePrivateKey);
    //Step 4
    byte[] ArrBSign = AsymAlg.SignData(ArrBInput);
    try {
        //Step 5
        File.WriteAllBytes(sFileSignature, ArrBSign);
    }
    finally { }
}
```

```
private bool VerifySignFile(
    string sFileIn, string sFileSignature, string sFilePubOrPvtKey
) {
    // Steps 1 & 2 same as in SignFile
    // Step 3 - uses FromXmlString
    this.ReadAsymKeyFromFile(AsymAlg, sFilePubOrPvtKey);
    //Step 4
    byte[] ArrBSign;
    try { ArrBSign = File.ReadAllBytes(sFileSignature); }
    catch { return false; }
    //Step 5
    return AsymAlg.VerifyData(ArrBInput, ArrBSign);
}
```

Advanced Analyses

Comparing with Asymmetric Encryption

Although you should not confuse yourselves by comparing the asymmetric encryption with the digital signature, you may be inclined to do so. Just remember that in the asymmetric encryption, output (encrypted output) is created using the public key but in the digital signature, the output (digital signature) is created using the private key. Similarly, the encrypted output can only be decrypted by using the private key but the digital signature can be verified by either the public or the private key.

Split Hashing and Signing

Remember, `SignData` creates a hash and then creates a digital signature for this hash. We can split this functionality. That is, we can create a hash with an algorithm of our choice and then use the `SignHash` method of an `AsymmetricAlgorithm` instance, say `AsymAlg`, to create the signature. Similarly, we can verify the signature of a hash, by using the `VerifyHash` method of the `AsymAlg`. The first argument in both the cases is of course the hash and the second argument is the string representation of the algorithm used to create that hash.

`RSACryptoServiceProvider` takes it a step further. It forces a programmer to provide his/her own hash algorithm as an argument to the `SignData` and the `VerifyData` methods. This algorithm has to be passed as the last argument of these methods.

Tip: Provide a non-keyed hash algorithm for a digital signature. Otherwise we will have to manage the keys of the hash algorithm in addition to the key for the asymmetric algorithm.

A `SignData` Overload

An overload of `SignData` takes a `FileStream` as its argument. We can pass the `FileStream` for the file for which the digital signature is to be created to this method.

RandomNumberGenerator

We go to lengths to generate randomness in our keys. For a `SymmetricAlgorithm`, we use the `GenerateKey` method. Or we use the `RFC2898DeriveBytes` to create a pseudorandom key. Microsoft even does this for us at the time of instantiation of any encryption algorithm. Randomness is something which is difficult to analyze. This thought gives strength to the philosophy of encryption. In order to provide a programmer with the strength of this thought, Microsoft has created a class called `RNGCryptoServiceProvider`, which inherits from the `RandomNumberGenerator` abstract class.

You can use the `GetBytes` or the `GetNonZeroBytes` method of an `RNGCryptoServiceProvider` instance to get a randomized byte array. You can use this array to generate a truly random set of large, small or normal numbers, as required.

Sample code

```
private void GetRandomNumbers(
    ref System.Collections.Generic.List<int> LstSmallRandomNums,
    ref System.Collections.Generic.List<int> LstNormalRandomNums,
    ref System.Collections.Generic.List<int> LstLargeRandomNums, int iLen
) {
    RNGCryptoServiceProvider rc = new RNGCryptoServiceProvider();
    byte[] ArrBRandom = new byte[iLen];
    rc.GetNonZeroBytes(ArrBRandom);
    for (int iTmp = 0; iTmp < ArrBRandom.Length - 1; iTmp++) {
        LstSmallRandomNums.Add(ArrBRandom[iTmp] % 5);
        LstNormalRandomNums.Add(ArrBRandom[iTmp]);
        LstLargeRandomNums.Add(ArrBRandom[iTmp] * ArrBRandom[iTmp + 1]);
    }
}
```

ProtectedData and ProtectedMemory

You can use the `ProtectedData` class to encrypt some data, which you can then store in a file. Its methods `Protect` (to encrypt) and `Unprotect` (to decrypt) take a byte array to encrypt/decrypt, an additional byte array to

make the encryption stronger and a `DataProtectionScope enum` of `CurrentUser` or `LocalMachine`.

Similarly, you can use the `ProtectedMemory` class to encrypt some variable in the memory. Its methods `Protect` and `Unprotect` take a byte array to encrypt/decrypt and a `MemoryProtectionScope enum` of `SameLogon` or `SameProcess` or `CrossProcess`.

Both these classes are in the `System.Security.dll`.

The differences between these classes are:

1. `ProtectedData` methods work on a copy of the byte array to be encrypted/decrypted, whereas `ProtectedMemory` works on the input byte array itself.
2. `ProtectedMemory` only works on the byte array the length of which is a multiple of 16.

CryptoConfig

`CryptoConfig` is a factory class used to create a cryptography class from a string. Here is a good example.

```
DSACryptoServiceProvider DsaAlg =
   (DSACryptoServiceProvider)( (new CryptoConfig).CreateFromName("DSA") )
```

Memory Sheet

Symmetric Encryption

```
Cryptor  =
  SymAlg.CreateEncryptor(
    ArrBKey, ArrBIV)
cs = new CryptoStream(
  fStrmOut, Cryptor,
  CryptoStreamMode.Write)
while(..){ cs.Write(..) }
cs.FlushFinalBlock
```

Symmetric Decryption

```
Cryptor  =
  SymAlg.CreateDecryptor(
    ArrBKey, ArrBIV)
cs = new CryptoStream(
  fStrmIn, Cryptor,
  CryptoStreamMode.Read)
while( cs.Read(..) ){..}
```

Asymmetric Encryption

```
FromXmlString or ImportParameters
for Key

Encrypt method
```

Asymmetric Decryption

```
Decrypt method
```

Hashing

```
ArrBHash =
  HashAlgorithm.ComputeHash(
    fStrmIn)

KydHashAlg =new HMACSHA512(ArrBKey)
ArrBHsh =
  KydHashAlg.ComputeHash(fStrmIn)
```

Digital Signature

```
ArrBSign =
  AsymmetricAlgorithm.SignData(
    ArrBInput)

bMatch =
  AsymmetricAlgorithm.VerifyData(
    ArrBInput, ArrBSign)

ArrBSign =
  AsymmetricAlgorithm.SignHash(
    ArrBHash, "SHA512")

bMatch =
  AsymmetricAlgorithm.VerifyHash(
    ArrBHash, "SHA512", ArrBSign)
```

Random Bytes

```
Instantiation of symmetric or
asymmetric algorithms

SymmetricAlgorithm.GenerateKey()

(new Rfc2898DeriveBytes(
  sPasswordShare, ArrBSalt,
  iIterationsShare)
).GetBytes(iBytes)

(new PasswordDeriveBytes(
  sPasswordShare, ArrBSalt)
).GetBytes(iBytes)

(new RNGCryptoServiceProvider()
).GetBytes(ArrB)
or .GetNonZeroBytes(ArrB)
```

ProtectedData & ProtectedMemory

```
ProtectedMempry takes multiples of
16 bytes and does not make a copy
```

CryptoConfig

```
(new CryptoConfig
).CreateFromName("TripleDES")
```

References

http://msdn.microsoft.com/en-us/library/92f9ye3s(VS.71).aspx

http://en.wikipedia.org/wiki/Initialization_vector

http://msdn.microsoft.com/en-us/library/system.security.cryptography.rsacryptoserviceprovider_members(VS.71).aspx

http://msdn.microsoft.com/en-us/library/system.security.cryptography.cryptoconfig_members.aspx

http://msdn.microsoft.com/en-us/library/system.security.cryptography.protecteddata.aspx

http://msdn.microsoft.com/en-us/library/system.security.cryptography.protectedmemory.aspx