Basic Crypto w/ the .NET Framework

Overview

The .NET Framework offers basic support for cryptographic operations inside the System.Security.Cryptography namespace in the mscorlib assembly. Out of the box you are provided with implementations of many common symmetric key and public key based algorithms. In addition, the cryptography framework was designed to be extensible so that your implementation of any algorithm can be plugged in quite easily.

This article will show you how to use some of the classes provided to encrypt and decrypt data using symmetric keys, sign messages using public key crypto, and generate hashes of passwords for secure storage. Just remember, cryptographic algorithms are not a silver-bullet that will solve all of your security problems! For a better understanding of why and when you should use these techniques, please see other sources such as Bruce Schneier's Applied Cryptography or Howard & LeBlanc's Writing Secure Code.

All of the examples shown here can be downloaded from my web site.

A Basic Example

Let's start with some code showing how to take the contents of one file and encrypt them into another file. We'll then decrypt the contents back into another file.

```
const string s_plaintext = "plaintext.txt";
const string s_ciphertext = "simple-ciphertext.bin";
const string s_decrypted = "simple-plaintext-decrypted.txt";

void Run()
{
  using(SymmetricAlgorithm algo = SymmetricAlgorithm.Create("Rijndael")) {
    Encrypt(algo);
    Decrypt(algo);
  }

}

void Encrypt(SymmetricAlgorithm algo)
{
  using(Stream cipherText = GetWriteableFileStream(s_ciphertext))
  using(ICryptoTransform enc = algo.CreateEncryptor())
  using(CryptoStream crypt = GetWriteCryptoStream(cipherText, enc))
  using(Stream input = GetReadOnlyFileStream(s_plaintext)) {
    Pump(input,crypt);
    crypt.Close(); //have to call, not called by Dispose
  }
}

void Decrypt()
{
  using(Stream cipherText = GetReadOnlyFileStream(s_ciphertext))
  using(CryptoStream decrypt = GetReadCryptoStream(cipherText, decryptor))
  using(Stream output = GetWriteableFileStream(s_decrypted)) {
    Pump(decrypt,output);
    decrypt.Close(); // have to call, not called by Dispose
  }
}

static void Pump(Stream input, Stream output)
{
  byte[] buffer = new byte[1024];
  int count = 0;

  while((count = input.Read(buffer, 0, 1024)) != 0) {
    output.Write(buffer, 0, count);
  }
}

static FileStream GetReadOnlyFileStream(string path)
{
  return new FileStream(path, FileMode.Open, FileAccess.Read);
}

static FileStream GetWriteableFileStream(string path)
{
  return new FileStream(path, FileMode.OpenOrCreate, FileAccess.Write);
}

static CryptoStream GetWriteCryptoStream(Stream stream, ICryptoTransform transform)
{
  return new CryptoStream(stream, transform, CryptoStreamMode.Write);
}

static CryptoStream GetReadCryptoStream(Stream stream, ICryptoTransform transform)
{
  return new CryptoStream(stream, transform, CryptoStreamMode.Read);
}
```

As you can see, the cryptography namespace was designed using a few different pieces. There’s a lot here, so let’s take it a piece at a time. First, let’s take a look at the Run method:

```
  using(SymmetricAlgorithm algo = SymmetricAlgorithm.Create(“Rijndael”))
```

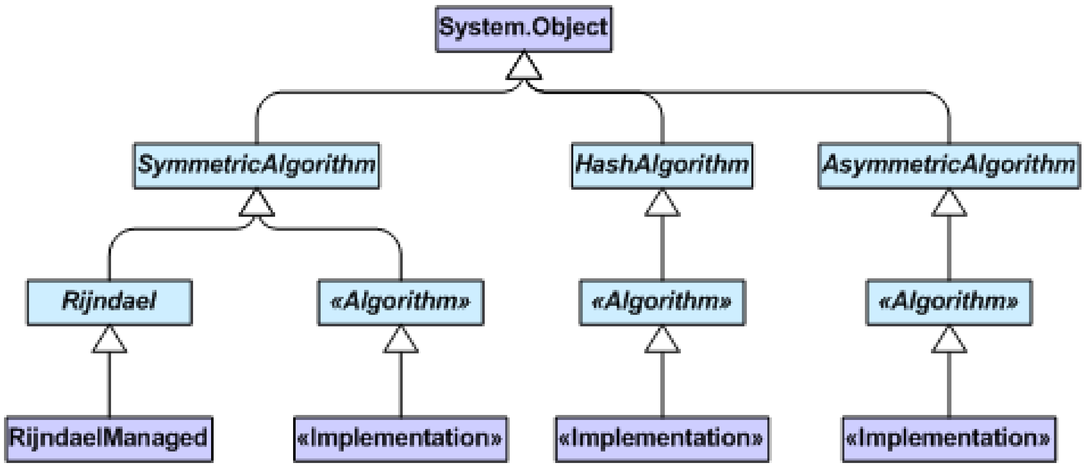
This line creates a SymmetricAlgorithm and sets that algorithm up to be disposed. Algorithms are the heart of the cryptography namespace, so we’ll first look at them. Just as a preview of things to come, let’s quickly dig into the Encrypt method:

```
  using(ICryptoTransform enc = algo.CreateEncryptor())
  using(CryptoStream crypt = GetWriteCryptoStream(cipherText, enc))
```

These two lines setup an ICryptoTransform and a CryptoStream. ICryptoTransform is the thing which actually does the work of transforming blocks from plain text to cipher text and back again. A CryptoStream lets us view the world of symmetric key and one-way hash-based cryptography through the rose-colored glasses of System.Stream. Together, they allow us to put different pieces of the puzzle together in interesting ways. We'll talk about them later in the article.

Algorithm Inheritance Model

Algorithms are the heart of any cryptography library. An algorithm describes how you are going to transform bytes from one format to another. Within the .NET cryptography namespace, different classes of algorithms are designated using an inheritance hierarchy. The hierarchy has three roots: SymmetricAlgorithm, HashAlgorithm and AsymmetricAlgorithm. Symmetric algorithms include common block ciphers such as RC2, 3DES and Rijndael. These are used to quickly encrypt and decrypt information using a shared secret key. Hash algorithms include MD5 and SHA1, as well as some keyed hashes. Hashing is a quick non-reversible transform and is generally used for information verification purposes. Passwords and message digests are often stored in a hashed format. Asymmetric algorithms include the public-key algorithms RSA and DSA. Asymmetric algorithms are generally used to sign messages and for key exchange. They have the benefit of not requiring both parties to share a secret, though they are significantly slower than Symmetric algorithms. Each class of algorithm has its purpose and understanding when to use each will strengthen your application of cryptography. Again, for more information, see Applied Cryptography. Figure 1 shows the inheritance hierarchy.



Each algorithm is separated into a base class that defines the algorithm and an implementation class that performs the work. The base class offers an overloaded static method called Create that will create an instance of an algorithm implementation object. The factory method allows a developer to decouple their assemblies from an actual implementation of any algorithm. Indeed, by using only factory method on the algorithmic category class (such as SymmetricAlgorithm), a developer could isolate their code from the actual algorithm employed. The code below shows how to create a Rijndael algorithm object using three different methods.

```
SymmetricAlgorithm s1 = new RijndaelManaged();
SymmetricAlgorithm s2 = Rijndael.Create();
SymmetricAlgorithm s3 = SymmetricAlgrotihm.Create("Rijndael");
```

Notice that you can create an implementation object directly using new, just like most other objects. I'm not sure why the crypto team chose to implement it this way, other than allowing developers the choice. In my opinion it just complicates things by offering more than one way to create an instance of an implementation object and there's no good reason to expose the RijndaelManaged object to the public. It may be faster, though I have not profiled it.

So, how does the framework know which class to instantiate? The crypto framework takes advantage of the CLR's configuration system to determine what class implements each algorithm. Developers can tap into the configuration settings used to bind algorithm names to implementation types, allowing you to substitute implementations or add new ones. At this point, there is precisely one implementation for each algorithm. There may be a small market here for alternative implementations that take advantage of accelerator cards, or for implementations of algorithms not present in the CLR, such as IDEA or RC5.

Using an Algorithm

Now that we have an algorithm, we need to use it to do something. Another concept used by the crypto framework comes from a desire to make working with symmetric and hashing algorithms easier. Generally, when a developer wants to encrypt or decrypt something, they're going to use a symmetric algorithm or a hashing algorithm and they're going to be processing data coming from a System.IO.Stream-based class. Asymmetric algorithms, which are significantly slower than symmetric algorithms or hashing, are not included in the model as you would rarely use one to encrypt large amounts of data.

To make working with stream-based data easier, the crypto framework team designed the CryptoStream class and the ICryptoTransform interface. CryptoStream derives from System.IO.Stream and enables developers to chain multiple operations together into one stream. The operation to perform is specified by the ICryptoTransform supplied to the CryptoStream constructor. From the code above:

```
void Encrypt(SymmetricAlgorithm algo)
{
  using(Stream cipherText = GetWriteableFileStream(s_ciphertext))
  using(ICryptoTransform enc = algo.CreateEncryptor())
  using(CryptoStream crypt = GetWriteCryptoStream(cipherText, enc))
  using(Stream input = GetReadOnlyFileStream(s_plaintext)) {
    Pump(input,crypt);
    crypt.Close(); //have to call, not called by Dispose
  }
}

static CryptoStream GetWriteCryptoStream(Stream stream, ICryptoTransform transform)
{
  return new CryptoStream(stream, transform, CryptoStreamMode.Write);
}

static CryptoStream GetReadCryptoStream(Stream stream, ICryptoTransform transform)
{
  return new CryptoStream(stream, transform, CryptoStreamMode.Read);
}
```

Okay, so what's going on here? First, let me explain that big stack of using(..) statements. This wasn't perfectly clear the first time I saw it, but the construct is rather handy. Remember that the scope of a block statement like using or if is either the next line, or the statements between { and }. Well, you can nest the "next-line" rule, so that the end result of stacking using statements like this to avoid having to use all the braces. The Dispose() methods will be called from the inside out, so this is really a handy way to use a bunch of disposable objects together. Pretty much all the objects in the crypto namespace implement IDisposable, so you'll see lots of code with using blocks. Now that we understand that, let's go through the code bit by bit.

```
  using(Stream cipherText = GetWriteableFileStream(s_ciphertext))
```

The first statement sets up the file stream we're going to write our encrypted bytes to.

```
using(ICryptoTransform enc = algo.CreateEncryptor())
```

The next statement uses the factory method on the algorithm class to create an encrypting ICryptoTransform. This transform knows how to take a chunk of bytes and encrypt them using the algorithm.

```
using(CryptoStream crypt = GetWriteCryptoStream(cipherText, enc))
```

Here we create the CryptoStream. Notice that we're passing in another Stream, the one we wish to write to, and the transform. I wrap up the process in a little helper method to save on some typing.

```
using(Stream input = GetReadOnlyFileStream(s_plaintext)) {
```

Now we're grabbing a Stream that represents the input file, which we're going to send through the CryptoStream and down into the cipherText stream.

```
Pump(input,crypt);
crypt.Close(); //have to call, not called by Dispose
```

Okay, last step in the process. Here, we're pumping all of the data from the input stream into the output stream. What happening is we're reading from the plaintext file and then writing the result into the CryptoStream. The CryptoStream takes the bytes, transforms them, and then writes them into the underlying Stream (in this case, a FileStream). Pretty simple really. That little call to crypt.Close() is of particular interest though. Remember that we're inside a using block for the CryptoStream and that Dispose() is going to be called on that CryptoStream when it leaves the current scope.

Generally, objects derived from System.IO.Stream do not implement IDisposable on their own, but instead override Stream.Close() and do any necessary cleanup work there. Stream implements IDisposable as a virtual call to Close(), so in general subclasses don't have to do anything other than override Close(). Dispose() is generally equivalent to Close() for Stream-derived classes.

Unless, of course, you're CryptoStream! CryptoStream implements IDisposable on it's own, which does **not** call Close()! CryptoStream.Dispose() instead clears out its internal buffers and does nothing more. It does not flush the remaining bytes in the buffer to the underlying Stream or do quite what you would expect. This means that you have to call either FlushFinalBlock() or Close() on the CyptoStream to get a valid encrypted or decrypted stream. The difference is that CryptoStream.FlushFinalBlock() will not call Close() on the underlying stream while CryptoStream.Close() will. Which one you use depends on how you'd like to write your code.

Also, be aware that the code below will not call CryptoStream.Close (or Stream.Dispose for that matter):

```
using(Stream s = new CryptoStream(...)) { ... }
```

When the using asks for the IDisposable interface, the CryptoStream hands back it's implementation, not the Stream's implementation. Remember that just like in COM, when you query an object for an interface the object must always return the same instance. It's just not always obvious on reading the code that you're going to get CryptoStream's implementation of IDisposable and that there is in fact no way to call Stream's implementation directly.

Is this a bug? I'm not sure. CryptoStream's Close() calls Close() on the underlying Stream as well, which changes the semantics of Stream.Close() a bit. It's a tough call. I think that CryptoStream.Dispose() should call FlushFinalBlock() at the very least though. Having to remember to call either Close() or FlushFinalBlock() is a bit a pain and has led to a number of hours spent banging my against a wall in frustration.

In the example above, if you want to change the algorithm being used, you can simply change the string passed to SymmetricAlgorithm.Create(). Everything else will just work itself out. By default, the SymmetricAlgorithm that’s created is configured using secure defaults (PKCS7 padding and it operates in CBC mode) with a randomly generated key and initialization vector. If you need to specify a key or IV, or want to change the padding or chaining mode, you can access all of those things via properties on the algorithm. Just check the LegalKeySizes property to determine if the key size you wish to use is valid. Also, be aware that PaddingMode.Zeros is broken in the v1 implementation of the Framework. If you use this padding mode for backwards compatibility purposes, you’ll have to find some way to determine the number of bytes you’re going to pull from the stream before you start. Prefixing the stream with the number of bytes contained within is always a good way.

Last, figuring out the Decrypt function is left as an exercise to the reader.

Chaining Algorithms

Now we know how to take a stream and either encrypt or decrypt the information contained within using a symmetric algorithm. That’s pretty handy, but there’s a problem. The output from the encryption process is binary and not printable. What if we need to save the encrypted bytes in some kind of a text medium, like an xml file? An easy way to do this is the Base64 encode the data.

Luckily, the crypto framework provides a pair of ICryptoTransform implementations that will transform to and from aBase64 encoding. The classes are ToBase64Transform and FromBase64Transform. To use them, we will take advantage CryptoStream chaining. Here’s the replacement methods for the code above:

```
private void Encrypt(SymmetricAlgorithm algo) {
  using(Stream cipherText = GetWriteableFileStream(s_ciphertext))
  using(ICryptoTransform b64 = new ToBase64Transform())
  using(ICryptoTransform enc = algo.CreateEncryptor())
  using(CryptoStream toBase64 = GetWriteCryptoStream(cipherText, b64))
  using(CryptoStream crypt = GetWriteCryptoStream(toBase64,enc))
  using(Stream input = GetReadOnlyFileStream(s_plaintext)) {
    Pump(input,crypt);
    crypt.Close(); //have to call, not called by Dispose
  }
}

private void Decrypt(SymmetricAlgorithm algo) {
  using(Stream cipherText = GetReadOnlyFileStream(s_ciphertext))
  using(ICryptoTransform b64 = new FromBase64Transform())
  using(ICryptoTransform dec = algo.CreateDecryptor())
  using(CryptoStream frBase64 = GetReadCryptoStream(cipherText, b64))
  using(CryptoStream decrypt = GetReadCryptoStream(frBase64,dec))
  using(Stream output = GetWriteableFileStream(s_decrypted)) {
    Pump(decrypt,output);
    decrypt.Close(); // have to call, not called by Dispose
  }
}
```

Notice that we’re creating two CryptoStreams in each method instead of one. Also, notice the addition of the

FromBase64Transform and the ToBase64Transform. The really interesting bits are these lines:

```
  using(CryptoStream toBase64 = GetWriteCryptoStream(cipherText,b64))
  using(CryptoStream crypt = GetWriteCryptoStream(toBase64, enc))
```

You pass the first CryptoStream into the constructor of the second. When you write into the outer CryptoStream (crypt), the write will be passed onto the underlying stream, toBase64. toBase64 then does its work and passes the results onto the underlying FileStream. Pretty nifty! You can continue to chain algorithms (or any other action really) as much as you see fit. This model is incredibly useful and extensible and was one of the better decisions made by the crypto framework team.

Hashing

Hashing uses a non-reversible algorithm, such as MD5 or SHA1, to create a unique short sequence of bytes from a larger sequence of bytes. Hashing is useful for verifying that the contents of a message or file has not changed since creation.

The primary interface to HashAlgorithm is the ComputeHash method. It’s overloaded to take either a Stream or a byte[]. Here’s a sample:

```
private void HashStream() {
  using(Stream input = GetReadOnlyFileStream(s_plaintext))
  using(HashAlgorithm hashAlg = HashAlgorithm.Create("MD5")) {
    byte[] hash = hashAlg.ComputeHash(input);
    PrintHash(s_plaintext, hash);
  }
}

private void HashBytes() {
  string password ="my-password";
  using(HashAlgorithm hash = HashAlgorithm.Create("MD5")) {
    byte[] hashed = hash.ComputeHash(Encoding.ASCII.GetBytes(password));
    PrintHash(password, hashed);
  }
}

private void PrintHash(string what, byte[] hash) {
  Console.WriteLine("Hash of {0}: {1}", what,
  Convert.ToBase64String(hash));
}
```

The code is quite a bit simpler than the encryption / decryption code. Also notice the alternate way to create a base64-encoded string from a byte[], Convert.ToBase64String(). Alternatively, instead of base64 encoding the hash code, you could easily put the byte[] into a SQL Server field of type binary. To use other algorithms, just change the string passed to HashAlgorithm.Create(). If you’re going to use a KeyedHashAlgorithm, you can just use new on the actual implementation object. That way, you can pass the key into the constructor instead of having to set it after creation by a factory method.

Finally, if you use hashing to store passwords, be sure to use a salt. A salt is a small sequence of bytes prefixed or appended to the password before hashing. For example, instead of just hashing the password "my-password", you would hash "12345" + "my-password". Having a salt helps to prevent dictionary attacks against your password database, should anyone obtain a copy of it. There’s a class called PasswordDeriveBytes in the crypto framework that looks interesting, but unfortunately it appears to only be for generating keys for the symmetric algorithms.

Public Key Crypto

So far, we've covered private key encryption and hashing. The last area to cover is public key based cryptography, which is primarily used for key exchange and message signing. Public key encryption uses two keys. Any information encrypted with one key can be decrypted with the other. The two keys are known as the public key and the private key. The private key must be kept safe and kept secret while the public key can be distributed to anyone who wants it, through a number of mediums. This makes public key encryption ideal for encrypting traffic between two parties without having to set up a secret key beforehand. The downside is that is very slow.

The framework supports two algorithms, RSA and DSA. RSA can be used for key exchange or to sign messages, while DSA can only be used to sign. First, let's check out the code to sign a message:

```
public void Run() {
  using(RSA rsa = RSA.Create()) {
    VerifySignature(rsa, CreateSignature(rsa));
  }
}

private byte[] CreateSignature(AsymmetricAlgorithm alg) {
  AsymmetricSignatureFormatter format = new RSAPKCS1SignatureFormatter(alg);
  format.SetHashAlgorithm("SHA1");
  byte[] sig = format.CreateSignature(GetHash(s_plaintext));
  PrintSig(s_plaintext, sig);
  return sig;
}

private void VerifySignature(AsymmetricAlgorithm alg, byte[] sig) {
  AsymmetricSignatureDeformatter format = new RSAPKCS1SignatureDeformatter(alg);
  format.SetHashAlgorithm("SHA1");
  Console.WriteLine(
    "The signature of {0} is {1}",
    s_plaintext,
    format.VerifySignature(GetHash(s_plaintext), sig) ? "valid" :"invalid");
}

private byte[] GetHash(string path) {
  using(Stream input = GetReadOnlyFileStream(path))
  using(HashAlgorithm hashAlg = HashAlgorithm.Create("SHA1")) {
    return hashAlg.ComputeHash(input);
  }
}

private void PrintSig(string what, byte[] hash) {
  Console.WriteLine("Signature of {0}: {1}", what, Convert.ToBase64String(hash));
}
```

Again, this is pretty quick code for something that's really quite complex. The crypto team did a great job wrapping up support for signature generation. Basically, we create an AsymmetricSignatureFormatter to create a signature and an AsymmetricSignatureDeformatter to verify a signature. One very important thing, we must call SetHashAlgorithm() before calling CreateSignature or VerifySignature, and you must set the hash algorithm to the same type used to create the hash.

Notice that I'm using SHA1 both to create the hash and for the signature. Again, to switch algorithms, just instantiate a different formatter.

The second interesting thing you can do with public key crypto is key exchange. With key exchange, I can create a key, sign it with my private key, encrypt it with my partners public key and then send it along. My partner can decrypt it with their private key, verify the signature with my public key. After that's done, we've agreed on a secret key. This concept is codified with the AsymmetricKeyExchangeFormatter and AsymmetricKeyExchangeDeformatter. As of this writing, only RSA-based key exchanges are implemented. Here's a sample that exchanges a key between two parties:

```
  const string s_secret = "too many secrets";
  const string s_publicKeyFile = "blowery.pubkey";

  public override void Run() {

    byte[] exchange;
    string pubKeyStr;

    using(RSA privKey = RSA.Create()) {
      // make public key
      pubKeyStr = privKey.ToXmlString(false);
      Console.WriteLine("My public key is {0}", pubKeyStr);

      // here we're Alice
      using(RSA pubKey = RSA.Create())
        pubKey.FromXmlString(pubKeyStr);
        exchange = CreateKeyExchange(pubKey);
      }

      // here we're Bob
      CrackKeyExchange(privKey, exchange);
    }
  }

  private byte[] CreateKeyExchange(RSA rsa) {
    AsymmetricKeyExchangeFormatter format = new RSAPKCS1KeyExchangeFormatter(rsa);
    return format.CreateKeyExchange(CreateSecret());
  }

  private void CrackKeyExchange(RSA rsa, byte[] keyExchange) {
    AsymmetricKeyExchangeDeformatter format = new
RSAPKCS1KeyExchangeDeformatter(rsa);
    byte[] secret = format.DecryptKeyExchange(keyExchange);
    Console.WriteLine("Ah ha! The secret is '{0}'",
Encoding.ASCII.GetString(secret));
  }

  private byte[] CreateSecret() {
    Console.WriteLine("Shhhh! The secret is '{0}'", s_secret);
    return Encoding.ASCII.GetBytes(s_secret);
  }
```

This example can be a bit confusing at first. Remember that during a key exchange, we have two parties, Alice and Bob. If

Alice wants to send Bob something secret, she has to encrypt the information with Bob’s public key. In the sample code, we’re pretending first to be Alice, then Bob. Alice only has access to Bob’s public key, so she loads it into an RSA algorithm object and creates the key exchange. Bob, who has both the public and private keys, can then crack the key exchange and extract the secret. It amazes me how little code this ends up being to implement. Obviously, you’ll want a slightly stronger implementation of CreateSecret().

That's It!

Well, that’s really it for System.Security.Cryptography. In this rather lengthy article, we’ve covered symmetric, asymmetric, and one-way algorithms. We’ve also covered how to use them together and exposed a couple quirks. I hope this article has been as enlightening for you to read as it was for me to research and write.