Chapter 20. Cryptography

In this chapter, we discuss the major cryptography APIs in .NET:

- Windows Data Protection API (DPAPI)
- Hashing
- Symmetric encryption
- Public key encryption and signing

The types covered in this chapter are defined in the following namespaces:

```
System.Security;
System.Security.Cryptography;
```

Overview

Table 20-1 summarizes the cryptography options in .NET. In the remaining sections, we explore each of these.

Table 20-1. Encryption and hashing options in .NET

Option	Keys to manage	Speed	Strength	Note
`File.Encrypt`	0	Fast	Depends on user's password	Prote trans filesy key i impli logge crede only.
Windows Data Protection	0	Fast	Depends on user's password	Encr byte impli
Hashing	0	Fast	High	One- (irrev trans for st comp check corru
Symmetric Encryption	1	Fast	High	For g encry The s encry Can l mess

Option	Keys to manage	Speed	Strength	Note
Public Key Encryption	2	Slow	High	Encr decry differ for ex symm mess and f signi

.NET also provides more specialized support for creating and validating XML-based signatures in `System.Security.Cryptography.Xml` and types for working with digital certificates in `System.Security.Cryptography.X509Certificates`.

Windows Data Protection

> **NOTE**
>
> Windows Data Protection is available on Windows only, and throws a `PlatformNotSupportedException` on other operating systems.

In the section "File and Directory Operations", we described how you could use `File.Encrypt` to request that the operating system transparently encrypt a file:

```
File.WriteAllText ("myfile.txt", "");
File.Encrypt ("myfile.txt");
File.AppendAllText ("myfile.txt", "sensitive data");
```

The encryption in this case uses a key derived from the logged-in user's password. You can use this same implicitly derived key to encrypt a byte array with the Windows Data Protection API (DPAPI). The DPAPI is exposed through the `ProtectedData` class—a simple type with two static methods:

```
public static byte[] Protect
  (byte[] userData, byte[] optionalEntropy, DataProtectionScope scope);

public static byte[] Unprotect
  (byte[] encryptedData, byte[] optionalEntropy, DataProtectionScope scope);
```

Whatever you include in `optionalEntropy` is added to the key, thereby increasing its security. The `DataProtectionScope` enum argument allows two options: `CurrentUser` or `LocalMachine`. With `CurrentUser`, a key is derived from the logged-in user's credentials; with `LocalMachine`, a machine-wide key is used, common to all users. This means that with the `CurrentUser` scope, data encrypted by one user cannot be decrypted by another. A `LocalMachine` key provides less protection but works under a Windows Service or a program needing to operate under a variety of accounts.

Here's a simple encryption and decryption demonstration:

```
byte[] original = {1, 2, 3, 4, 5};
DataProtectionScope scope = DataProtectionScope.CurrentUser;

byte[] encrypted = ProtectedData.Protect (original, null, scope);
byte[] decrypted = ProtectedData.Unprotect (encrypted, null, scope);
// decrypted is now {1, 2, 3, 4, 5}
```

Windows Data Protection provides moderate security against an attacker with full access to the computer, depending on the strength of the user's password. With `LocalMachine` scope, it's effective only against those with restricted physical and electronic access.

Hashing

A *hashing algorithm* distills a potentially large number of bytes into a small fixed-length *hashcode*. Hashing algorithms are designed such that a single-bit change anywhere in the source data results in a significantly different hashcode. This makes it suitable for comparing files or detecting accidental (or malicious) corruption to a file or data stream.

Hashing also acts as one-way encryption, because it's difficult to impossible to convert a hashcode back into the original data. This makes it ideal for storing passwords in a database, because should your database become compromised, you don't want the attacker to gain access to plain-text passwords. To authenticate, you simply hash what the user types in and compare it to the hash that's stored in the database.

To hash, you call `ComputeHash` on one of the `HashAlgorithm` subclasses, such as `SHA1` or `SHA256`:

```
byte[] hash;
using (Stream fs = File.OpenRead ("checkme.doc"))
  hash = SHA1.Create().ComputeHash (fs);   // SHA1 hash is 20 bytes long
```

The `ComputeHash` method also accepts a byte array, which is convenient for hashing passwords (we describe a more secure technique in "Hashing Passwords"):

```
byte[] data = System.Text.Encoding.UTF8.GetBytes ("stRhong%pword");
byte[] hash = SHA256.Create().ComputeHash (data);
```

> **NOTE**
>
> The `GetBytes` method on an `Encoding` object converts a string to a byte array; the `GetString` method converts it back. An `Encoding` object cannot, however, convert an encrypted or hashed byte array to a string, because scrambled data usually violates text encoding rules. Instead, use `Convert.ToBase64String` and `Convert.FromBase64String`: these convert between any byte array and a legal (and XML- or JSON-friendly) string.

Hash Algorithms in .NET

`SHA1` and `SHA256` are two of the `HashAlgorithm` subtypes provided by .NET. Here are the major algorithms, in ascending order of security:

Class	Algorithm	Hash length in bytes	Strength
`MD5`	MD5	16	Very poor
`SHA1`	SHA-1	20	Poor
`SHA256`	SHA-2	32	Good
`SHA384`	SHA-2	48	Good
`SHA512`	SHA-2	64	Good

All five algorithms execute at roughly similar speeds in their current implementations, with the exception of SHA256, which is 2-3 times faster (this may vary with hardware and operating system). To give a ballpark figure, you can expect at least 500 MB per second on a 2024-era desktop or server with all algorithms. The longer hashes decrease the possibility of *collision* (two distinct files yielding the same hash).

WARNING

Use *at least* `SHA256` when hashing passwords or other security-sensitive data. `MD5` and `SHA1` are considered insecure for this purpose and are suitable to protect only against accidental corruption, not deliberate tampering.

> **NOTE**
>
> .NET 8 and above also support the latest SHA-3 hashing note via the `SHA3_256`, `SHA3_384`, and `SHA3_512` classes. The SHA-3 algorithms are considered even more secure (and slower) than the previously listed algorithms, but require Windows Build 25324+ or Linux with OpenSSL 1.1.1+. You can test whether OS support is available via the static `IsSupported` property on these classes.

Hashing Passwords

The longer SHA algorithms are suitable as a basis for password hashing, if you enforce a strong password policy to mitigate a *dictionary attack*—a strategy whereby an attacker builds a password lookup table by hashing every word in a dictionary.

A standard technique, when hashing passwords, is to incorporate "salt"—a long series of bytes that you initially obtain via a random number generator and then combine with each password before hashing. This frustrates hackers in two ways:

- They must also know the salt bytes.
- They cannot use *rainbow tables* (publicly available *precomputed* databases of passwords and their hashcodes), although a dictionary attack might still be possible with sufficient computing power.

You can further strengthen security by "stretching" your password hashes—repeatedly rehashing to obtain more computationally intensive byte sequences. If you rehash 100 times, a dictionary attack that might otherwise take one month would take eight years. The `KeyDerivation`, `Rfc2898DeriveBytes`, and `PasswordDeriveBytes` classes perform exactly this kind of stretching while also allowing for convenient salting. Of these, `KeyDerivation.Pbkdf2` offers the best hashing:

```
byte[] encrypted = KeyDerivation.Pbkdf2 (
    password: "stRhong%pword",
    salt: Encoding.UTF8.GetBytes ("j78Y#p)/saREN!y3@"),
    prf: KeyDerivationPrf.HMACSHA512,
```

```
    iterationCount: 100,
    numBytesRequested: 64);
```

> **NOTE**
>
> `KeyDerivation.Pbkdf2` requires the NuGet package `Microsoft.AspNetCore.Cryptography.KeyDerivation`. Though it's in the ASP.NET Core namespace, any .NET application can use it.

Symmetric Encryption

Symmetric encryption uses the same key for encryption as for decryption. The .NET BCL provides four symmetric algorithms, of which Rijndael (pronounced "Rhine Dahl" or "Rain Doll") is the premium; the other algorithms are intended mainly for compatibility with older applications. Rijndael is both fast and secure and has two implementations:

- The `Rijndael` class
- The `Aes` class

The two are almost identical, except that `Aes` does not let you weaken the cipher by changing the block size. `Aes` is recommended by the CLR's security team.

`Rijndael` and `Aes` allow symmetric keys of length 16, 24, or 32 bytes: all are currently considered secure. Here's how to encrypt a series of bytes as they're written to a file, using a 16-byte key:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] data = { 1, 2, 3, 4, 5 };   // This is what we're encrypting.

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
using (Stream f = File.Create ("encrypted.bin"))
```

```
using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  c.Write (data, 0, data.Length);
```

The following code decrypts the file:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};

byte[] decrypted = new byte[5];

using (SymmetricAlgorithm algorithm = Aes.Create())
using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
using (Stream f = File.OpenRead ("encrypted.bin"))
using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte()) > -1;)
    Console.Write (b + " ");                          // 1 2 3 4 5
```

In this example, we made up a key of 16 randomly chosen bytes. If the wrong key was used in decryption, `CryptoStream` would throw a `CryptographicException`. Catching this exception is the only way to test whether a key is correct.

As well as a key, we made up an IV, or *Initialization Vector*. This 16-byte sequence forms part of the cipher—much like the key—but is not considered *secret*. If you're transmitting an encrypted message, you would send the IV in plain text (perhaps in a message header) and then *change it with every message*. This would render each encrypted message unrecognizable from any previous one—even if their unencrypted versions were similar or identical.

> **NOTE**
>
> If you don't need—or want—the protection of an IV, you can defeat it by using the same 16-byte value for both the key and the IV. Sending multiple messages with the same IV, though, weakens the cipher and might even make it possible to crack.

The cryptography work is divided among the classes. `Aes` is the mathematician; it applies the cipher algorithm, along with its `encryptor`

and `decryptor` transforms. `CryptoStream` is the plumber; it takes care of stream plumbing. You can replace `Aes` with a different symmetric algorithm yet still use `CryptoStream`.

`CryptoStream` is *bidirectional*, meaning you can read or write to the stream depending on whether you choose `CryptoStreamMode.Read` or `CryptoStreamMode.Write`. Both encryptors and decryptors are read *and* write savvy, yielding four combinations—the choice can have you staring at a blank screen for a while! It can be helpful to model reading as "pulling" and writing as "pushing." If in doubt, start with `Write` for encryption and `Read` for decryption; this is often the most natural.

To generate a random key or IV, use `RandomNumberGenerator` in `System.Cryptography`. The numbers it produces are genuinely unpredictable, or *cryptographically strong* (the `System.Random` class does not offer the same guarantee). Here's an example:

```
byte[] key = new byte [16];
byte[] iv  = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes (key);
rand.GetBytes (iv);
```

Or, from .NET 6:

```
byte[] key = RandomNumberGenerator.GetBytes (16);
byte[] iv = RandomNumberGenerator.GetBytes (16);
```

If you don't specify a key and IV, cryptographically strong random values are generated automatically. You can query these through the `Aes` object's `Key` and `IV` properties.

Encrypting in Memory

From .NET 6, you can utilize the `EncryptCbc` and `DecryptCbc` methods to shortcut the process of encrypting and decrypting byte arrays:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using Aes algorithm = Aes.Create();
  algorithm.Key = key;
  return algorithm.EncryptCbc (data, iv);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using Aes algorithm = Aes.Create();
  algorithm.Key = key;
  return algorithm.DecryptCbc (data, iv);
}
```

Here's an equivalent that works in all.NET versions:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
    return Crypt (data, encryptor);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Aes algorithm = Aes.Create())
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
    return Crypt (data, decryptor);
}

static byte[] Crypt (byte[] data, ICryptoTransform cryptor)
{
  MemoryStream m = new MemoryStream();
  using (Stream c = new CryptoStream (m, cryptor, CryptoStreamMode.Write))
    c.Write (data, 0, data.Length);
  return m.ToArray();
}
```

Here, `CryptoStreamMode.Write` works best for both encryption and decryption, since in both cases we're "pushing" into a fresh memory stream.

Here are overloads that accept and return strings:

```
public static string Encrypt (string data, byte[] key, byte[] iv)
{
  return Convert.ToBase64String (
    Encrypt (Encoding.UTF8.GetBytes (data), key, iv));
}

public static string Decrypt (string data, byte[] key, byte[] iv)
{
  return Encoding.UTF8.GetString (
    Decrypt (Convert.FromBase64String (data), key, iv));
}
```

The following demonstrates their use:

```
byte[] key = new byte[16];
byte[] iv = new byte[16];

var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);

string encrypted = Encrypt ("Yeah!", key, iv);
Console.WriteLine (encrypted);                // R1/5gYvcxyR2vzPjnT7yaQ==

string decrypted = Decrypt (encrypted, key, iv);
Console.WriteLine (decrypted);                // Yeah!
```

Chaining Encryption Streams

`CryptoStream` is a decorator, meaning that you can chain it with other streams. In the following example, we write compressed encrypted text to a file and then read it back:

```
byte[] key = new byte [16];
byte[] iv = new byte [16];

var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);

using (Aes algorithm = Aes.Create())
{
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
  using (Stream f = File.Create ("serious.bin"))
```

```
  using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
  using (Stream d = new DeflateStream (c, CompressionMode.Compress))
  using (StreamWriter w = new StreamWriter (d))
    await w.WriteLineAsync ("Small and secure!");

  using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
  using (Stream f = File.OpenRead ("serious.bin"))
  using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
  using (Stream d = new DeflateStream (c, CompressionMode.Decompress))
  using (StreamReader r = new StreamReader (d))
    Console.WriteLine (await r.ReadLineAsync());     // Small and secure!
}
```

(As a final touch, we make our program asynchronous by calling `WriteLineAsync` and `ReadLineAsync` and awaiting the result.)

In this example, all one-letter variables form part of a chain. The mathematicians—`algorithm`, `encryptor`, and `decryptor`—are there to assist `CryptoStream` in the cipher work, as illustrated in Figure 20-1.

Chaining streams in this manner demands little memory, regardless of the ultimate stream sizes.

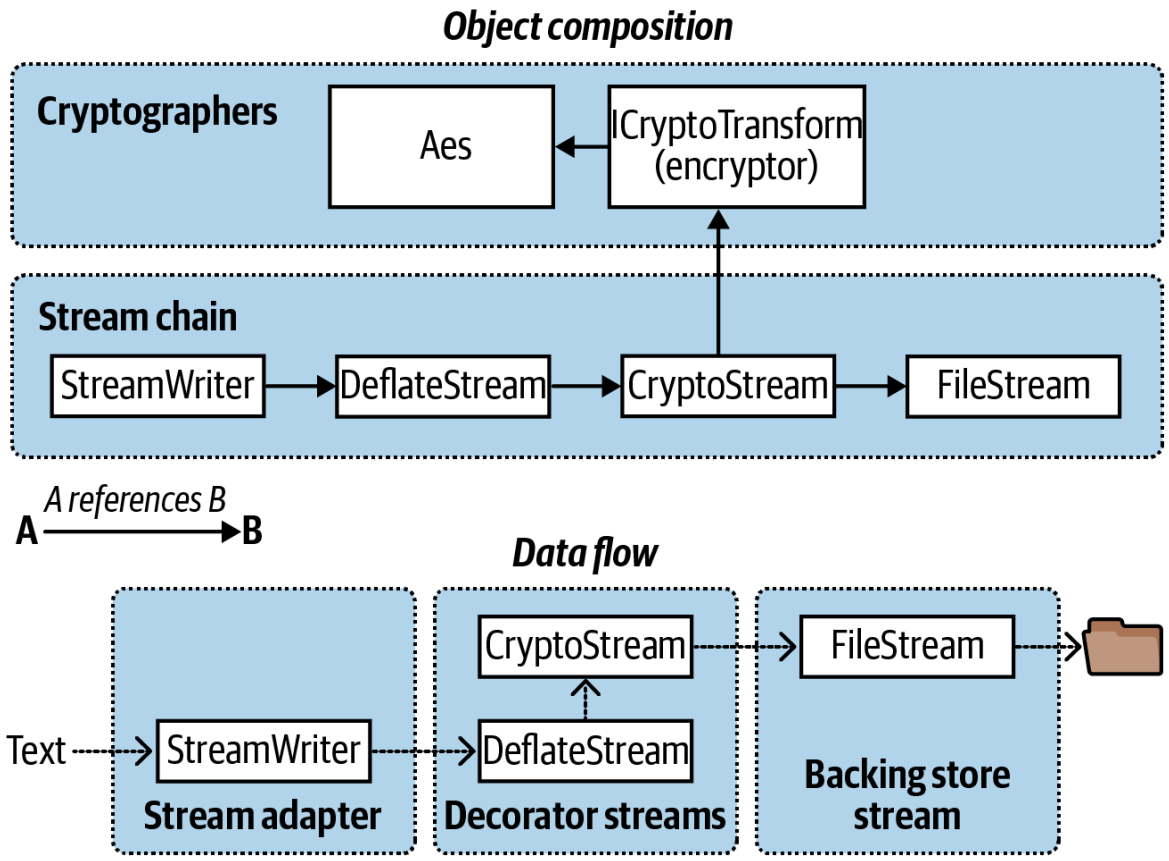

Figure 20-1. Chaining encryption and compression streams

Disposing Encryption Objects

Disposing a `CryptoStream` ensures that its internal cache of data is flushed to the underlying stream. Internal caching is necessary for encryption algorithms because they process data in blocks, rather than one byte at a time.

`CryptoStream` is unusual in that its `Flush` method does nothing. To flush a stream (without disposing it) you must call `FlushFinalBlock`. In contrast to `Flush`, you can call `FlushFinalBlock` only once, and then no further data can be written.

We also disposed the mathematicians—the `Aes` algorithm and `ICryptoTransform` objects (`encryptor` and `decryptor`). When the Rijndael transforms are disposed, they wipe the symmetric key and related data from memory, preventing subsequent discovery by other software

running on the computer (we're talking malware). You can't rely on the garbage collector for this job, because it merely flags sections of memory as available; it doesn't write zeros over every byte.

The easiest way to dispose an `Aes` object outside of a `using` statement is to call `Clear`. Its `Dispose` method is hidden via explicit implementation (to signal its unusual disposal semantics, whereby it clears memory rather than releasing unmanaged resources).

> **NOTE**
>
> You can further reduce your application's vulnerability to leaking secrets via released memory by doing the following:
>
> - Avoiding strings for security information (being immutable, a string's value can never be cleared once created)
> - Overwriting buffers as soon as they're no longer needed (for instance, by calling `Array.Clear` on a byte array)

Key Management

Key management is a critical element of security: if your keys are exposed, so is your data. You need to consider who should have access to keys and how to back them up in case of hardware failure while storing them in a manner that prevents unauthorized access.

It is inadvisable to hardcode encryption keys because popular tools exist to decompile assemblies with little expertise required. A better option (on Windows) is to manufacture a random key for each installation, storing it securely with Windows Data Protection.

For applications deployed to the cloud, Microsoft Azure and Amazon Web Services (AWS) offer key-management systems with additional features that can be useful in an enterprise environment, such as audit trails. If you're encrypting a message stream, public-key encryption still provides the best option.

Public-Key Encryption and Signing

Public-key cryptography is *asymmetric*, meaning that encryption and decryption use different keys.

Unlike symmetric encryption, for which any arbitrary series of bytes of appropriate length can serve as a key, asymmetric cryptography requires specially crafted key pairs. A key pair contains a *public key* and *private key* component that work together as follows:

- The public key encrypts messages.
- The private key decrypts messages.

The party "crafting" a key pair keeps the private key secret while distributing the public key freely. A special feature of this type of cryptography is that you cannot calculate a private key from a public key. So, if the private key is lost, encrypted data cannot be recovered; conversely, if a private key is leaked, the encryption system becomes useless.

A public key handshake allows two computers to communicate securely over a public network, with no prior contact and no existing shared secret. To see how this works, suppose that computer *Origin* wants to send a confidential message to computer *Target*:

1. *Target* generates a public/private key pair and then sends its public key to *Origin*.
2. *Origin* encrypts the confidential message using *Target*'s public key and then sends it to *Target*.
3. *Target* decrypts the confidential message using its private key.

An eavesdropper will see the following:

- *Target*'s public key
- The secret message, encrypted with *Target*'s public key

But without *Target*'s private key, the message cannot be decrypted.

> **NOTE**
>
> This doesn't prevent against a man-in-the-middle attack: in other words, *Origin* cannot know that *Target* isn't some malicious party. To authenticate the recipient, the originator needs to already know the recipient's public key or be able to validate its key through a *digital site certificate*.

Because public key encryption is relatively slow and its message size limited, the secret message sent from *Origin* to *Target* typically contains a fresh key for subsequent *symmetric* encryption. This allows public key encryption to be abandoned for the remainder of the session, in favor of a symmetric algorithm capable of handling larger messages. This protocol is particularly secure if a fresh public/private key pair is generated for each session because no keys then need to be stored on either computer.

> **NOTE**
>
> The public key encryption algorithms rely on the message being smaller than the key. This makes them suitable for encrypting only small amounts of data, such as a key for subsequent symmetric encryption. If you try to encrypt a message much larger than half the key size, the provider will throw an exception.

The RSA Class

.NET provides a number of asymmetric algorithms, of which RSA is the most popular. Here's how to encrypt and decrypt with RSA:

```
byte[] data = { 1, 2, 3, 4, 5 };   // This is what we're encrypting.

using (var rsa = new RSACryptoServiceProvider())
{
  byte[] encrypted = rsa.Encrypt (data, true);
  byte[] decrypted = rsa.Decrypt (encrypted, true);
}
```

Because we didn't specify a public or private key, the cryptographic provider automatically generated a key pair, using the default length of 1,024 bits; you can request longer keys in increments of eight bytes, through the constructor. For security-critical applications, it's prudent to request 2,048 bits:

```
var rsa = new RSACryptoServiceProvider (2048);
```

Generating a key pair is computationally intensive—taking perhaps 10 ms. For this reason, the RSA implementation delays this until a key is actually needed, such as when calling `Encrypt`. This gives you the chance to load in an existing key—or key pair, should it exist.

The methods `ImportCspBlob` and `ExportCspBlob` load and save keys in byte array format. `FromXmlString` and `ToXmlString` do the same job in a string format, the string containing an XML fragment. A `bool` flag lets you indicate whether to include the private key when saving. Here's how to manufacture a key pair and save it to disk:

```
using (var rsa = new RSACryptoServiceProvider())
{
  File.WriteAllText ("PublicKeyOnly.xml", rsa.ToXmlString (false));
  File.WriteAllText ("PublicPrivate.xml", rsa.ToXmlString (true));
}
```

Because we didn't provide existing keys, `ToXmlString` forced the manufacture of a fresh key pair (on the first call). In the next example, we read back these keys and use them to encrypt and decrypt a message:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to encrypt");

string publicKeyOnly = File.ReadAllText ("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText ("PublicPrivate.xml");

byte[] encrypted, decrypted;

using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
  rsaPublicOnly.FromXmlString (publicKeyOnly);
```

```
  encrypted = rsaPublicOnly.Encrypt (data, true);


  // The next line would throw an exception because you need the private
  // key in order to decrypt:
  // decrypted = rsaPublicOnly.Decrypt (encrypted, true);
}

using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
  // With the private key we can successfully decrypt:
  rsaPublicPrivate.FromXmlString (publicPrivate);
  decrypted = rsaPublicPrivate.Decrypt (encrypted, true);
}
```

Digital Signing

You also can use public key algorithms to digitally sign messages or documents. A signature is like a hash, except that its production requires a private key and so cannot be forged. The public key is used to verify the signature. Here's an example:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create();         // Our chosen hashing algorithm.

// Generate a new key pair, then sign the data with it:
using (var publicPrivate = new RSACryptoServiceProvider())
{
  signature = publicPrivate.SignData (data, hasher);
  publicKey = publicPrivate.ExportCspBlob (false);    // get public key
}

// Create a fresh RSA using just the public key, then test the signature.
using (var publicOnly = new RSACryptoServiceProvider())
{
  publicOnly.ImportCspBlob (publicKey);
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // True

  // Let's now tamper with the data and recheck the signature:
  data[0] = 0;
  Console.Write (publicOnly.VerifyData (data, hasher, signature)); // False

  // The following throws an exception as we're lacking a private key:
```

```
  signature = publicOnly.SignData (data, hasher);
}
```

Signing works by first hashing the data and then applying the asymmetric algorithm to the resultant hash. Because hashes are of a small fixed size, large documents can be signed relatively quickly (public key encryption is much more CPU-intensive than hashing). If you want, you can do the hashing yourself and then call `SignHash` instead of `SignData`:

```
using (var rsa = new RSACryptoServiceProvider())
{
  byte[] hash = SHA1.Create().ComputeHash (data);
  signature = rsa.SignHash (hash, CryptoConfig.MapNameToOID ("SHA1"));
  ...
}
```

`SignHash` still needs to know what hash algorithm you used; `CryptoConfig.MapNameToOID` provides this information in the correct format from a friendly name such as “SHA1”.

`RSACryptoServiceProvider` produces signatures whose size matches that of the key. Currently, no mainstream algorithm produces secure signatures significantly smaller than 128 bytes (suitable for product activation codes, for instance).

NOTE

For signing to be effective, the recipient must know, and trust, the sender’s public key. This can happen via prior communication, preconfiguration, or a site certificate. A site certificate is an electronic record of the originator’s public key and name—itself signed by an independent trusted authority. The namespace `System.Security.Cryptography.X509Certificates` defines the types for working with certificates.