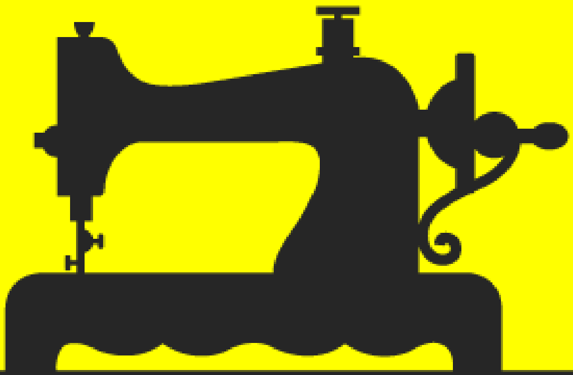


Cryptography in .NET Succinctly

by Stephen Haunts

Syncfusion | Technology Resource Portal

Cryptography in .NET Succinctly

By
Stephen Haunts

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Technical Reviewer: Jim Perry
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

Stephen Haunts has been developing software and applications professionally since 1996 and as a hobby since he was 10 years old. Stephen has worked across many different industries including computer games, online banking, retail finance, and healthcare and pharmaceuticals. Stephen started out programming in BASIC on machines such as the Dragon 32, Vic 20, and the Amiga, and moved onto C and C++ on the IBM PC. Stephen has been developing software in C# and the .NET framework since first being introduced to it in 2003.

In addition to being an accomplished software developer, Stephen is also an experienced development leader and has led, mentored, and coached teams to deliver many high-value, high-impact solutions in finance and healthcare.

Outside of Stephen's day job, he is an experienced tech blogger who runs a blog called Coding in the Trenches, and a training course author for Pluralsight, a provider of online learning services for professional software developers and IT specialists. Stephen also runs several open-source projects including SafePad, Text Shredder, and Block Encryptor, as well as Smoke Tester, a post-deployment testing tool.

Stephen is also an accomplished electronic musician and sound designer.

Introduction

Throughout history, cryptography has been used to confidentially pass messages to different recipients. This has been especially useful in times of war. Up until World War II, with the use of the German enciphering machine Enigma, messages that were encrypted were text-based. With the advent of modern computing, cryptography now tends to operate on raw data and is used to protect our communication channels, email messages, web browsing, financial transactions, and much more. Cryptography is all around us and is, normally, completely transparent to the user. However, just because you can't see it doesn't make it any less important. In the modern day, we face threats from cybercrime, terrorists, and our own government snooping on its citizens. Because of these threats, it is very important to understand and apply cryptography in your software solutions.

This book is aimed at .NET developers who want to understand how to use cryptography in their applications. The user is expected to be reasonably experienced as a .NET developer but no prior knowledge of cryptography and encryption is required.

This book is for developers who want to be pragmatic about encrypting data in their applications by using the .NET framework. The examples in this book target writing applications for Windows desktops and servers, which includes technologies such as console applications, Windows Presentation Foundation (WPF), ASP.NET, Windows Communication Foundation (WCF), Windows Forms (WinForms), and anything else that you might use in the enterprise.

Cryptography is an interesting subject and the math behind how a lot of these algorithms work is fascinating. However, as a developer who has to deliver systems, you don't need that level of knowledge. You just need to know what algorithms are safest to use and how to use them, and that is exactly what this book aims to teach.

The source code examples have been kept as simple as possible to show you how to apply them in your own code. You can treat the code examples in this book (and the source code project that accompanies this book) as a cookbook to which you can refer whenever you need to use any of the cryptography primitives in your own code.

Note: The source code examples from this book can be downloaded from https://bitbucket.org/syncfusiontech/cryptography-in-.net-succinctly.

Chapter 1 What is Cryptography?

Cryptography is the art of protecting information by transforming it (i.e. encrypting it) into an unreadable format called ciphertext. Only those who possess a secret key can decipher (or decrypt) the message into plaintext. Encrypted messages can sometimes be broken by cryptanalysis (also called code breaking) although modern cryptography techniques are virtually unbreakable.

As the Internet and other forms of electronic communication become more prevalent, electronic security is becoming increasingly important. Cryptography is used to protect email messages, credit card information, and corporate data.

There is more to cryptography than just encrypting data, though. There are three main security themes that are covered by cryptography and different cryptography primitives that help you satisfy each concept. These themes are:

- Confidentiality
- Integrity
- Nonrepudiation

Confidentiality

Confidentiality is what you traditionally associate with cryptography. This is where you take a message or some other data and encrypt it to make the original data completely unreadable. There are many different cryptography algorithms that you can use, but this book will cover the main primitives that are in use today, including RSA and Advanced Encryption Standard (AES). We will also cover a couple of primitives (DES and Triple DES) that are not recommended for use in new code but you may have to use them if you are writing code that deals with older legacy systems.

Integrity

In information security, data integrity means maintaining and assuring the accuracy and consistency of data over its entire life cycle. This means that data cannot be modified in an unauthorized or undetected manner. Integrity is violated when a message is actively modified in transit. Systems typically provide message integrity in addition to data confidentiality. We will cover some different cryptography primitives that you can use to help enforce data integrity including hashing algorithms such as MD5, Secure Hash Algorithm (SHA)-1, SHA-256, and SHA-512. We will also cover hash message authentication codes (HMACs) that also use MD5, SHA-1, SHA-256, and SHA-512.

In addition to standard hashing primitives, we will also discuss why it is not a good idea to use hashes to store passwords and discuss an alternative method called a password-based key derivation function.

Nonrepudiation

Nonrepudiation is the assurance that someone cannot deny something. Typically, nonrepudiation refers to the ability to ensure that a party to a contract or a communication cannot deny the authenticity of their signature on a document or the sending of a message that originated with them.

For many years, authorities have sought to make repudiation impossible in some situations. You might send registered mail, for example, so the recipient cannot deny that a letter was delivered. Similarly, a legal document typically requires witnesses to its signing so that the person who signs it cannot deny having done so.

On the Internet, a digital signature is used not only to ensure that a message or document has been electronically signed by the person that purported to sign the document but also, since a digital signature can only be created by one person, to ensure that a person cannot later deny that they furnished the signature.

In this book, we will cover digital signatures that use the RSA cryptographic primitive.

Cryptography in .NET

.NET comes with a rich collection of cryptography objects that you can use to help provide better security in your applications. The cryptography objects in .NET all live within the **`System.Security.Cryptography`** namespace.

Chapter 2 Cryptographic Random Numbers

Typically in .NET, when you need to generate a random number or a pseudorandom number, you would use the **`System.Random`** object to generate the number. For most scenarios, this is fine and will give the appearance of randomness when you apply a different seed each time. When you are dealing with security though, **`System.Random`** is not sufficient as the result of **`System.Random`** is very deterministic and predictable.

Random numbers are important in cryptography as you need them for generating encryption keys for symmetric algorithms such as AES, which we will cover in a later chapter, and also for adding entropy into hashing functions and key derivation functions.

Note: Entropy is the measure of uncertainty associated with a random variable. In cryptography, entropy must be supplied by the cipher for injection into the plaintext of a message so as to neutralize the amount of structure that is present in the unsecure plaintext message.

A better approach is to use the **`RNGCryptoServiceProvider`** object in the **`System.Cryptography`** namespace. **`RNGCryptoServiceProvider`** provides much better randomness than **`System.Random`**, but it does come at a slight cost as the call into **`RNGCryptoServiceProvider`** is much slower. However, this is a necessary trade-off if you require good quality, nondeterministic random numbers for key generation.

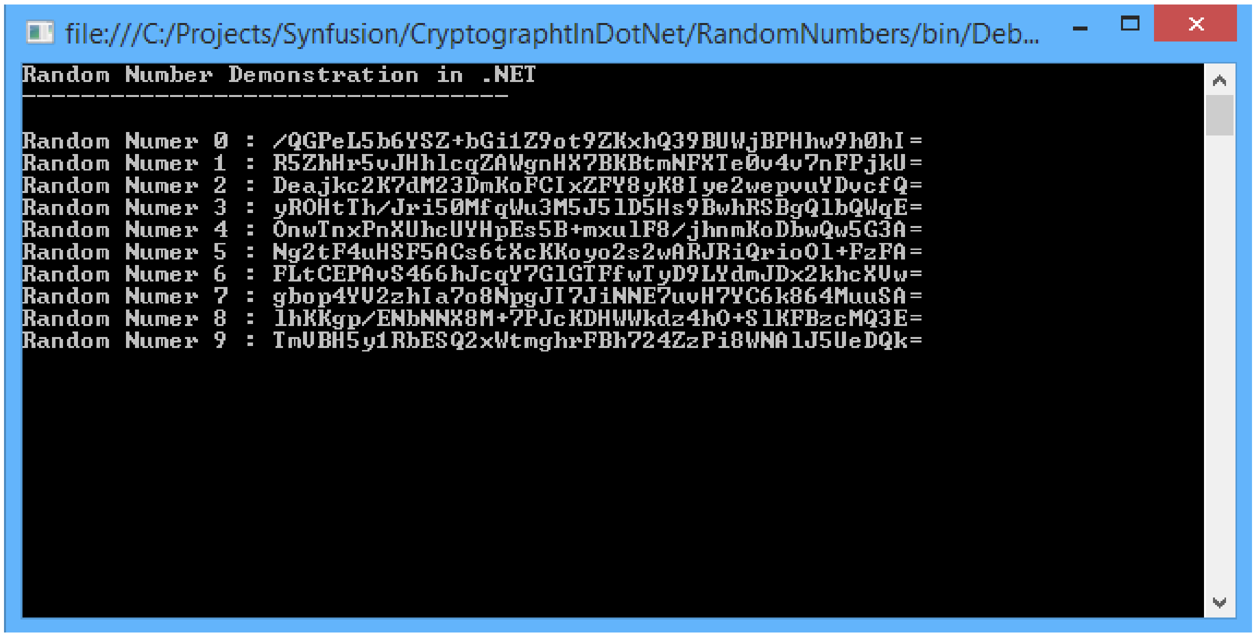
The following code demonstrates how to use **`RNGCryptoServiceProvider`**.

```
public static byte[] GenerateRandomNumber(int length)
{
    using (var randomNumberGenerator = new RNGCryptoServiceProvider())
    {
        var randomNumber = new byte[length];
        randomNumberGenerator.GetBytes(randomNumber);

        return randomNumber;
    }
}
```

Code Listing 1

Once you have constructed the **`RNGCryptoServiceProvider`** object, you make a call to **`GetBytes()`** by passing in a pre-instanced, fixed-length byte array. If, for example, you wanted to generate a random number to use as a 256-bit key for the AES encryption algorithm, you would create an array that was 32 bytes in length, as 32 bytes multiplied by 8 bits in a byte gives you 256 bits total.

Apart from speed of execution and the nondeterministic nature of **`RNGCryptoServiceProvider`**, the other difference between this and **`System.Random`** is that **`RNGCryptoServiceProvider`** is thread safe and **`System.Random`** is not.


file:///C:/Projects/Synfusion/CryptographtInDotNet/RandomNumbers/bin/Deb...

```
Random Number Demonstration in .NET
-----------------------------------

Random Numer 0 : /QGPeL5b6YSZ+bGi1Z9ot9ZKxhQ39BUWjBPHhw9h0hI=
Random Numer 1 : R5ZhHr5vJHhlcqZAWgnHX7BKBtmNFXTe0v4v7nFPjkU=
Random Numer 2 : Deajkc2K7dM23DmKoFCIxZFY8yK8Iye2wepvuYDvcfQ=
Random Numer 3 : yROHtTh/Jri50MfqWu3M5J51D5Hs9BwhRSBgQ1bQWqE=
Random Numer 4 : OnwTnxPnXUhcUYHpEs5B+mxu1F8/jhnmKoDbwQw5G3A=
Random Numer 5 : Ng2tF4uHSF5ACs6tXcKKoyo2s2wARJRiQrioO1+FzFA=
Random Numer 6 : FLtCEPAvS466hJcqY7G1GTFfwTyD9LYdmJDx2khcXVw=
Random Numer 7 : gbop4YV2zhIa7o8NpgJI7JiNNE7uvH7YC6k864MuuSA=
Random Numer 8 : lhKKgp/ENbNNX8M+7PJcKDHWWkdz4hO+S1KFBzcMQ3E=
Random Numer 9 : TmVBH5y1RbESQ2xWtmghrFBh724ZzPi8WNA1J5UeDQk=
```

Figure 1: Randomly generated numbers using RNGCryptoServiceProvider

In the sample code, we generate 10 sets of 32-byte random numbers by using the **`RNGCryptoServiceProvider`** and write the results to the console window. The **`GenerateRandomNumber`** method returns a byte array that is the same size as the parameter that you pass into the method. When we output the random number onto the screen, we first convert it to a Base64 string. This is common practice when displaying the results of a cryptographic operation on the screen.

Note: Base64 is a group of similar binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Random Number " + i + " : "
        + Convert.ToBase64String(Random.GenerateRandomNumber(32)));
}
```

Code Listing 2

If you need to convert from a Base64 encoded string back into a byte array, you call **`Convert.FromBase64String`**.

Chapter 3 Hashing Algorithms

Hashing Functions

A cryptographic hash function is an algorithm that takes an arbitrary block of data and returns a fixed-size string, the (cryptographic) hash value, such that any (accidental or intentional) change to the data will change the hash value. The data to be encoded is often called the “message” and the hash value is sometimes called the “message digest” or, simply, the “digest.”

The ideal cryptographic hash function has four main properties:

- It is easy to compute the hash value for any given message.
- It is infeasible to generate a message that has a given hash.
- It is infeasible to modify a message without changing the hash.
- It is infeasible to find two different messages with the same hash.

Another way of thinking of a hash function is that of creating a unique fingerprint of a piece of data. Generating a hash digest of a block of data is easy to do in .NET. There are various algorithms you can use such as MD5, SHA-1, SHA-256, and SHA-512.

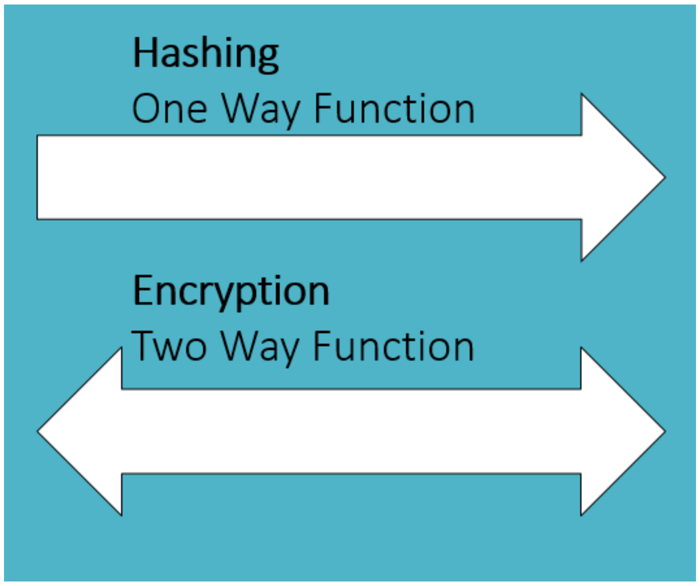


Figure 2: Hashing functions versus encryption functions

As illustrated in the preceding diagram, a hash function is a one-way function. That means once you have hashed some data, you cannot reverse it back to the original data. On the flip side to this, encryption is designed to be a two-way operation. Once you have encrypted some data by using a key, you can reverse the operation and decrypt the data by using the same key.

When you hash some data, the hash will be the same every time you perform the operation unless the original data changes in some way. Even if the data only changes by one bit, the resulting hash code will be completely different. This makes hashing the perfect mechanism for checking the integrity of data.

This is useful when you want to send data across a network to another recipient. Before sending the data, you calculate a hash of the data to get its unique fingerprint. You then send that data and the hash to the recipient. They then calculate the hash of the data they have just received and then compare it to the hash you sent. If the hash codes are the same, then the data has been successfully received without data loss or corruption. If the hash codes do not match, then the data received is not the same as the data originally sent.

The two most common hashing methods used are MD5 and the SHA family of hashes (SHA-1, SHA-256, and SHA-512).

MD5

The MD5 message digest algorithm is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value, which is typically expressed in text format as a 32-digit hexadecimal number or as a Base64-encoded string. MD5 has been used in a wide number of cryptographic applications and is also commonly used to verify file integrity.

MD5 was designed by Ron Rivest in 1991 to replace MD4, an earlier hash function. In 1996, a flaw was found in the design of MD5. While it was not deemed a fatal weakness at the time, cryptographers began to recommend the use of other algorithms such as the SHA family. In 2004, it was shown that MD5 is not collision-resistant, which means that it is possible that generating an MD5 of two sets of data could result in the same hash (although this is quite rare).

You may still need to use MD5 in your applications if you are checking the integrity of data coming from a legacy system that makes use of MD5.

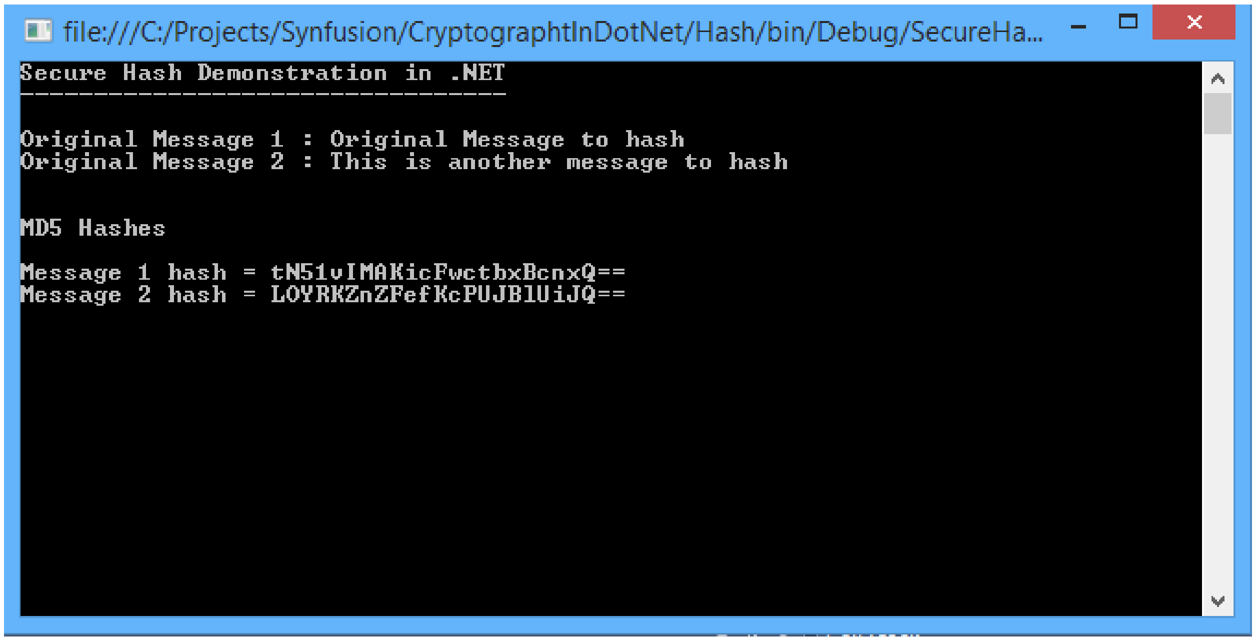
It is easy to calculate an MD5 hash of some data in .NET as shown in the following code.

```
public class HashData
{
    public static byte[] ComputeHashMD5(byte[] toBeHashed)
    {
        using (var md5 = MD5.Create())
        {
            return md5.ComputeHash(toBeHashed);
        }
    }
}
```

Code Listing 3

You first call **`MD5.Create()`** and then call **`ComputeHash()`** on the object while passing in the data you want to be hashed. The input to **`ComputeHash()`** has to be a byte array, so if you are calculating a hash of a file, you will need to load it up into a byte array first.

The following figure shows two pieces of text that have been hashed and displayed in the console window as a Base64-encoded string.


file:///C:/Projects/Synfusion/CryptographtInDotNet/Hash/bin/Debug/SecureHa...

```
Secure Hash Demonstration in .NET
---------------------------------

Original Message 1 : Original Message to hash
Original Message 2 : This is another message to hash


MD5 Hashes

Message 1 hash = tN51vIMAKicFwctbxBcnxQ==
Message 2 hash = LOYRKZnZFefKcPUJB1UiJQ==
```

Figure 3: Text hashed with MD5

The code for the preceding example is as follows.

```
class Program
{
static void Main(string[] args)
{
    var originalMessage = "Original Message to hash";
    var originalMessage2 = "This is another message to hash";

    Console.WriteLine("Secure Hash Demonstration in .NET");
    Console.WriteLine("---------------------------------");
    Console.WriteLine();
    Console.WriteLine("Original Message 1 : " + originalMessage);
    Console.WriteLine("Original Message 2 : " + originalMessage2);
    Console.WriteLine();

    var md5hashedMessage =
HashData.ComputeHashMD5(Encoding.UTF8.GetBytes(originalMessage));
    var md5hashedMessage2 =
HashData.ComputeHashMD5(Encoding.UTF8.GetBytes(originalMessage2));

    Console.WriteLine();
    Console.WriteLine("MD5 Hashes");
    Console.WriteLine();
    Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(md5hashedMessage));
```

```
    Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(md5hashedMessage2));

    Console.ReadLine();
}
```

Code Listing 4

Secure Hash Algorithm (SHA) Family

The SHA family is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).

The SHA family covers some different variants, including:

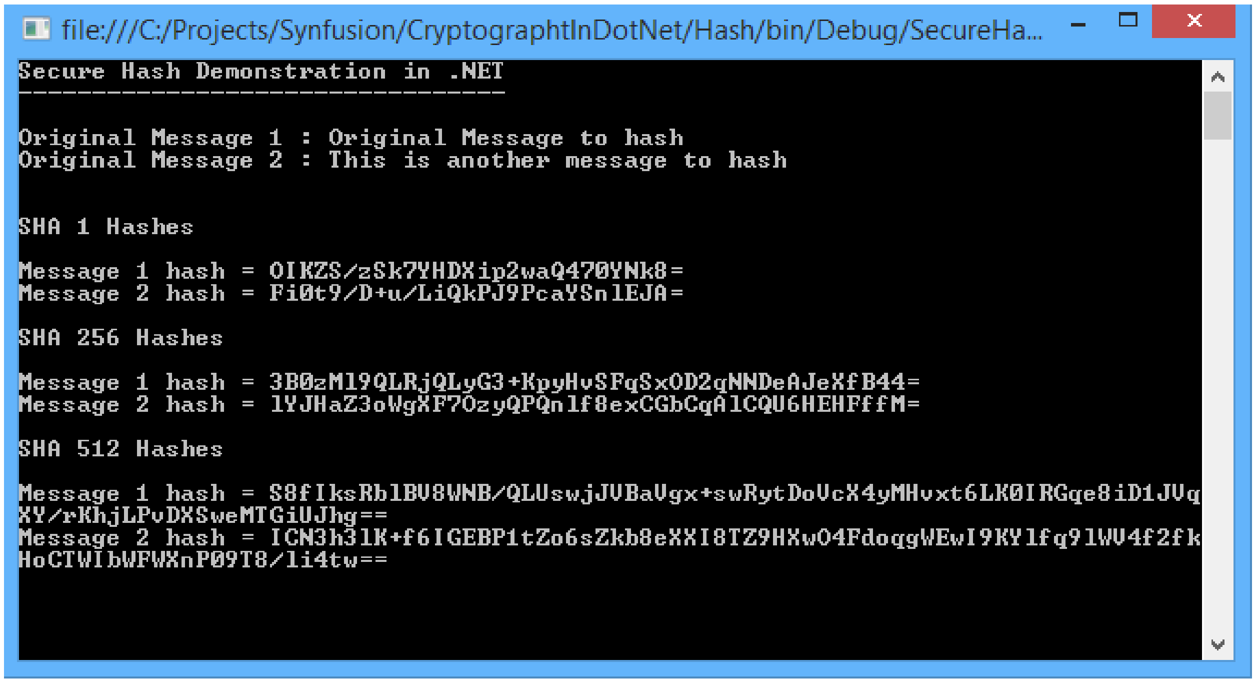
- **SHA-1**: A 160-bit hash function which resembles the earlier MD5 algorithm. This was designed by the National Security Agency to be part of the Digital Signature Algorithm. Cryptographic weaknesses were discovered in SHA-1 and the standard was no longer approved for most cryptographic uses after 2010.
- **SHA-2**: A family of two similar hash functions with different block sizes known as SHA-256 and SHA-512. They differ in the word size: SHA-256 uses 32-bit words whereas SHA-512 uses 64-bit words. These versions of the SHA algorithm were also designed by the NSA.
- **SHA-3**: A hash function formerly called Keccak, chosen in 2012 after a public competition among non-NSA designers. It supports the same hash lengths as SHA-2 and its internal structure differs significantly from the rest of the SHA family. SHA-3 is not currently supported in the .NET framework directly, although third-party implementations are available.

Implementing SHA in your applications is a very straightforward process as the signatures of the SHA objects are identical to those of the MD5 objects.

```
public class HashData
{
    public static byte[] ComputeHashSHA1(byte[] toBeHashed)
    {
        using (var sha1 = SHA1.Create())
        {
            return sha1.ComputeHash(toBeHashed);
        }
    }

    public static byte[] ComputeHashSHA256(byte[] toBeHashed)
    {
        using (var sha256 = SHA256.Create())
        {
            return sha256.ComputeHash(toBeHashed);
        }
    }
    public static byte[] ComputeHashSHA512(byte[] toBeHashed)
    {
        using (var sha512 = SHA512.Create())
        {
            return sha512.ComputeHash(toBeHashed);
        }
    }
}
```

Code Listing 5

Like the **`MD5`** class, you call **`Create`** on the **`SHA1`**, **`SHA256`**, and **`SHA512`** objects. Next, you call **`ComputeHash`**, which returns the hash as a byte array. The following screenshot shows the results of the different SHA variants (SHA-1, SHA-256, and SHA-512) being run against two different messages.


file:///C:/Projects/Synfusion/CryptographtInDotNet/Hash/bin/Debug/SecureHa...

```
Secure Hash Demonstration in .NET
---------------------------------

Original Message 1 : Original Message to hash
Original Message 2 : This is another message to hash


SHA 1 Hashes

Message 1 hash = OIKZS/zSk7YHDXip2waQ470YNk8=
Message 2 hash = Fi0t9/D+u/LiQkPJ9PcaYSnlEJA=

SHA 256 Hashes

Message 1 hash = 3B0zMl9QLRjQLyG3+KpyHvSFqSxOD2qNNDeAJeXfB44=
Message 2 hash = lYJHaZ3oWgXF7OzyQPQnlf8exCGbCqAlCQU6HEHFffM=

SHA 512 Hashes

Message 1 hash = S8fIksRblBV8WNB/QLUswjJVBaVgx+swRytDoVcX4yMHvxt6LK0IRGqe8iD1JVq
XY/rKhjLPvDXSweMTGiUJhg==
Message 2 hash = ICN3h3lK+f6IGEBP1tZo6sZkb8eXXI8TZ9HXwO4FdoqgWEwI9KYlfq9lWV4f2fk
HoCTWIbWFWXnP09T8/li4tw==
```

Figure 4: SHA-1, SHA-256, and SHA-512 example

The code for generating the hashes in the preceding example is as follows.

```
class Program
{
    static void Main(string[] args)
    {
        var originalMessage = "Original Message to hash";
        var originalMessage2 = "This is another message to hash";

        Console.WriteLine("Secure Hash Demonstration in .NET");
        Console.WriteLine("--------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Message 1 : " + originalMessage);
        Console.WriteLine("Original Message 2 : " + originalMessage2);
        Console.WriteLine();

        var sha1hashedMessage =
HashData.ComputeHashSHA1(Encoding.UTF8.GetBytes(originalMessage));
        var sha1hashedMessage2 =
HashData.ComputeHashSHA1(Encoding.UTF8.GetBytes(originalMessage2));

        var sha256hashedMessage =
HashData.ComputeHashSHA256(Encoding.UTF8.GetBytes(originalMessage));
        var sha256hashedMessage2 =
HashData.ComputeHashSHA256(Encoding.UTF8.GetBytes(originalMessage2));

        var sha512hashedMessage =
HashData.ComputeHashSHA512(Encoding.UTF8.GetBytes(originalMessage));
        var sha512hashedMessage2 =
HashData.ComputeHashSHA512(Encoding.UTF8.GetBytes(originalMessage2));

        Console.WriteLine();
        Console.WriteLine("SHA 1 Hashes");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(sha1hashedMessage));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(sha1hashedMessage2));
        Console.WriteLine();

        Console.WriteLine("SHA 256 Hashes");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(sha256hashedMessage));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(sha256hashedMessage2));
        Console.WriteLine();
        Console.WriteLine("SHA 512 Hashes");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(sha512hashedMessage));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(sha512hashedMessage2));
        Console.WriteLine();
        Console.ReadLine();
    }
```

Code Listing 6

Message Authentication Codes

If you combine a one-way hash function with a secret cryptographic key, you get what is called a hash message authentication code (HMAC).

Like a hash code, an HMAC is used to verify the integrity of a message. An HMAC also allows you to verify the authentication of that message because only a person who knows the key can calculate the same hash of the message. An HMAC can be used with different hashing functions like MD5 or the SHA family of algorithms.

The cryptographic strength of an HMAC depends on the size of the key that is used. The most common attack against an HMAC is a brute force attack to uncover the key. HMACs are substantially less affected by collisions than their underlying hashing algorithms alone.

HMACs are used when you need to check both integrity and authenticity. For example, consider a scenario in which you are sent a piece of data along with its hash. You can verify the integrity of the message by re-computing the hash of the message and comparing it with the hash that you received. However, you don't know for sure if the message and the hash was sent by someone you know or trust. If you used an HMAC, you could re-compute the HMAC by using a secret key that only you and a trusted party know, and compare it with the HMAC you just received. This serves the purpose of authenticity.

In the following example, I will show you how to use an HMAC that uses a 32-byte (256-bit) key and an HMAC based on the SHA-256 hashing algorithm. I used a 256-bit key here as it is the same size key as you would use for AES encryption but you can go higher. In addition to **HMACSHA256**, you can use **HMACSHA1**, **HMACSHA512**, and **HMACMD5**. The interfaces are all the same.

Let’s look at the following code.

```
public class HMAC
{
    private const int KEY_SIZE = 32;

    public static byte[] GenerateKey()
    {
        using (var randomNumberGenerator = new RNGCryptoServiceProvider())
        {
            var randomNumber = new byte[KEY_SIZE];
            randomNumberGenerator.GetBytes(randomNumber);

            return randomNumber;
        }
    }

    public static byte[] ComputeHMACSHA256(byte[] toBeHashed, byte[] key)
    {
        using (var hmac = new HMACSHA256(key))
        {
            return hmac.ComputeHash(toBeHashed);
        }
    }

    public static byte[] ComputeHMACSHA1(byte[] toBeHashed, byte[] key)
    {
        using (var hmac = new HMACSHA1(key))
        {
            return hmac.ComputeHash(toBeHashed);
        }
    }

    public static byte[] ComputeHMACSHA512(byte[] toBeHashed, byte[] key)
    {
        using (var hmac = new HMACSHA512(key))
        {
            return hmac.ComputeHash(toBeHashed);
        }
    }

    public static byte[] ComputeHMACMD5(byte[] toBeHashed, byte[] key)
    {
        using (var hmac = new HMACMD5(key))
        {
            return hmac.ComputeHash(toBeHashed);
        }
    }
}
```

Code Listing 7

The hash class contains a method to generate the key by using the **`RNGCryptoServiceProvider`** that we have previously discussed. Then, there is the **`ComputeHmacSHA256`** method that takes a byte array of the code that you want to hash and a byte array for the encryption key.

To use the code, you do the following.

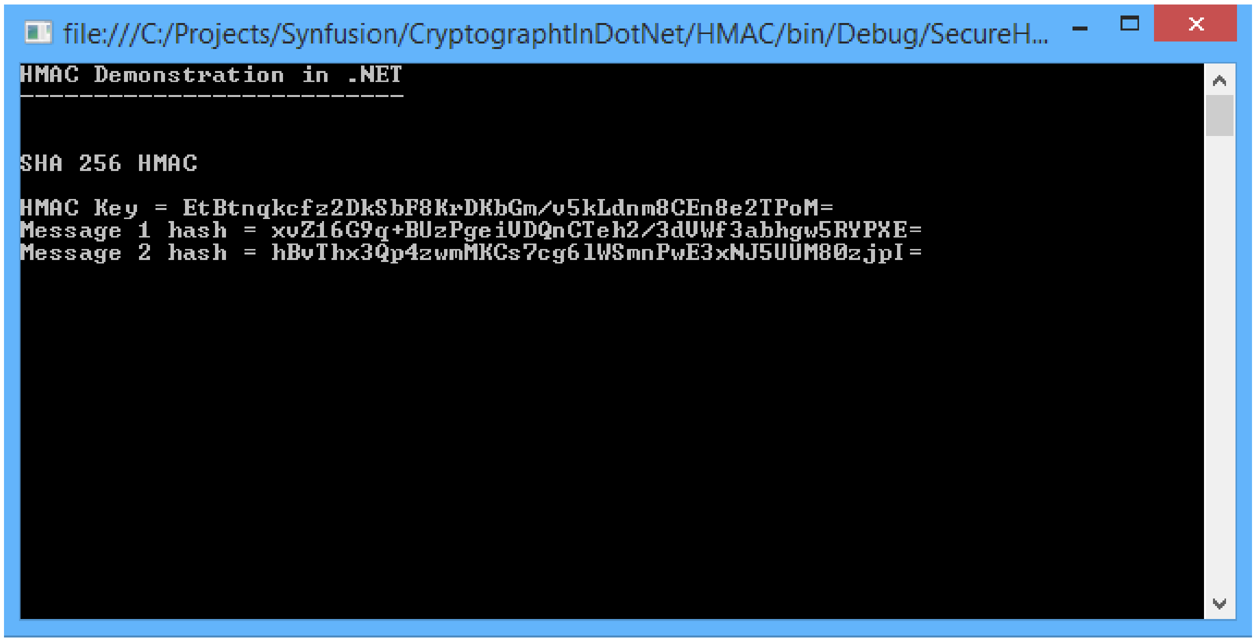
```
class Program
{
    static void Main(string[] args)
    {
        var originalMessage = "Original Message to hash";
        var originalMessage2 = "This is another message to hash";

        Console.WriteLine("HMAC Demonstration in .NET");
        Console.WriteLine("---------------------------");
        Console.WriteLine();

        var key = HMAC.GenerateKey();

        var hmacMd5Message = 
HMAC.ComputeHMACMD5(Encoding.UTF8.GetBytes(originalMessage), key);
```

```
        var hmacMd5Message2 =
HMAC.ComputeHMACMD5(Encoding.UTF8.GetBytes(originalMessage2), key);

        var hmacSha1Message =
HMAC.ComputeHMACSHA1(Encoding.UTF8.GetBytes(originalMessage), key);
        var hmacSha1Message2 =
HMAC.ComputeHMACSHA1(Encoding.UTF8.GetBytes(originalMessage2), key);

        var hmacSha256Message =
HMAC.ComputeHMACSHA256(Encoding.UTF8.GetBytes(originalMessage), key);
        var hmacSha256Message2 =
HMAC.ComputeHMACSHA256(Encoding.UTF8.GetBytes(originalMessage2), key);

        var hmacSha512Message =
HMAC.ComputeHMACSHA512(Encoding.UTF8.GetBytes(originalMessage), key);
        var hmacSha512Message2 =
HMAC.ComputeHMACSHA512(Encoding.UTF8.GetBytes(originalMessage2), key);

        Console.WriteLine();
        Console.WriteLine("MD5 HMAC");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(hmacMd5Message));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(hmacMd5Message2));

        Console.WriteLine();
        Console.WriteLine("SHA 1 HMAC");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(hmacSha1Message));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(hmacSha1Message2));

        Console.WriteLine();
        Console.WriteLine("SHA 256 HMAC");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(hmacSha256Message));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(hmacSha256Message2));

        Console.WriteLine();
        Console.WriteLine("SHA 512 HMAC");
        Console.WriteLine();
        Console.WriteLine("Message 1 hash = " +
Convert.ToBase64String(hmacSha512Message));
        Console.WriteLine("Message 2 hash = " +
Convert.ToBase64String(hmacSha512Message2));
        Console.WriteLine();
```

```
            Console.ReadLine();
        }
    }
```

Code Listing 8

Running this code gives you the following result where you can see the key that was generated and the result of hashing two strings with **`HMACSHA256`**.


file:///C:/Projects/Synfusion/CryptographtInDotNet/HMAC/bin/Debug/SecureH...

```
HMAC Demonstration in .NET
--------------------------


SHA 256 HMAC

HMAC Key = EtBtnqkcfz2DkSbF8KrDKbGm/v5kLdnm8CEn8e2TPoM=
Message 1 hash = xvZ16G9q+BUzPgeiVDQnCTeh2/3dVWf3abhgw5RYPXE=
Message 2 hash = hBvThx3Qp4zwmMKCs7cg6lWSmnPwE3xNJ5UUM80zjpI=
```

Figure 5: HMAC example using a secret key and HMACSHA256

Chapter 4 Password Storage

Using Hashes to Store Passwords

A common usage scenario for hashes is to encode passwords for storing in a database. With the advent of modern processors and graphical processing units (GPUs), it is not recommended you take this approach as hashes can be brute force attacked or attacked by using rainbow tables.

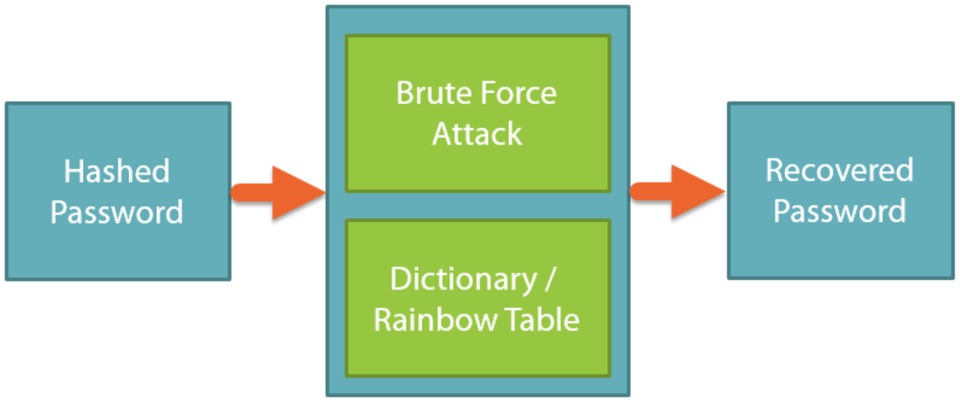


Figure 6: Hashed passwords can be brute force attacked or attacked using rainbow tables

A better approach is to increase the entropy of the password being attacked by making the password harder to recover. This can be done by adding a salt onto the password before hashing.

Note: A rainbow table contains precomputed hashes for different combinations of messages. These tables are used to determine the original plaintext from an already computed hash value. Rainbow tables can be many gigabytes in size and they greatly speed up attacks to recover the original value of a hash.

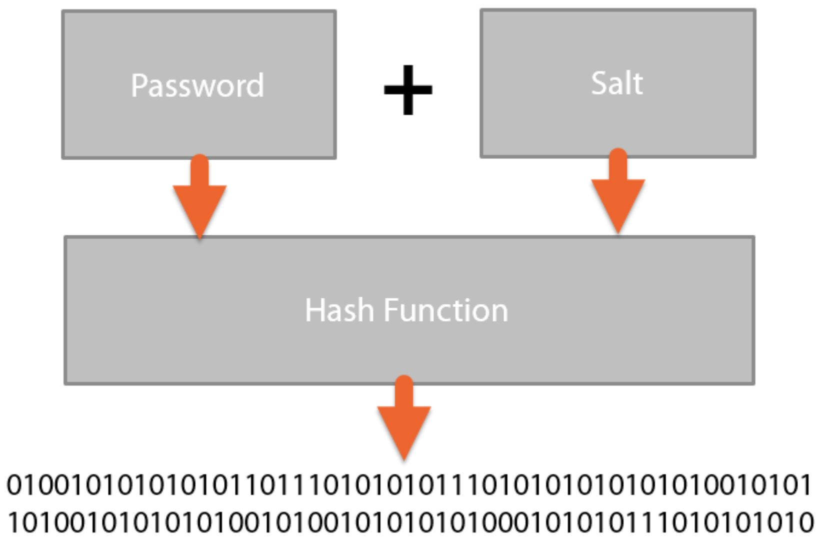


Figure 7: Add a salt to the password before hashing to increase entropy

The salt can be generated as a random number and then appended on the password before hashing. In the following code, we have a hash class containing everything we need to do this. This has the hash function itself, the salt generator, and a method to combine two byte arrays together.

```
public class Hash
{
    public static byte[] HashPasswordWithSalt(byte[] toBeHashed, byte[] salt)
    {
        using (var sha256 = SHA256.Create())
        {
            return sha256.ComputeHash(Combine(toBeHashed, salt));
        }
    }

    private static byte[] Combine(byte[] first, byte[] second)
    {
        var ret = new byte[first.Length + second.Length];

        Buffer.BlockCopy(first, 0, ret, 0, first.Length);
        Buffer.BlockCopy(second, 0, ret, first.Length, second.Length);

        return ret;
    }

    public static byte[] GenerateSalt()
    {
        const int SALT_LENGTH = 32;

        using (var randomNumberGenerator = new RNGCryptoServiceProvider())
        {
            var randomNumber = new byte[SALT_LENGTH];
            randomNumberGenerator.GetBytes(randomNumber);

            return randomNumber;
        }
    }
}
```

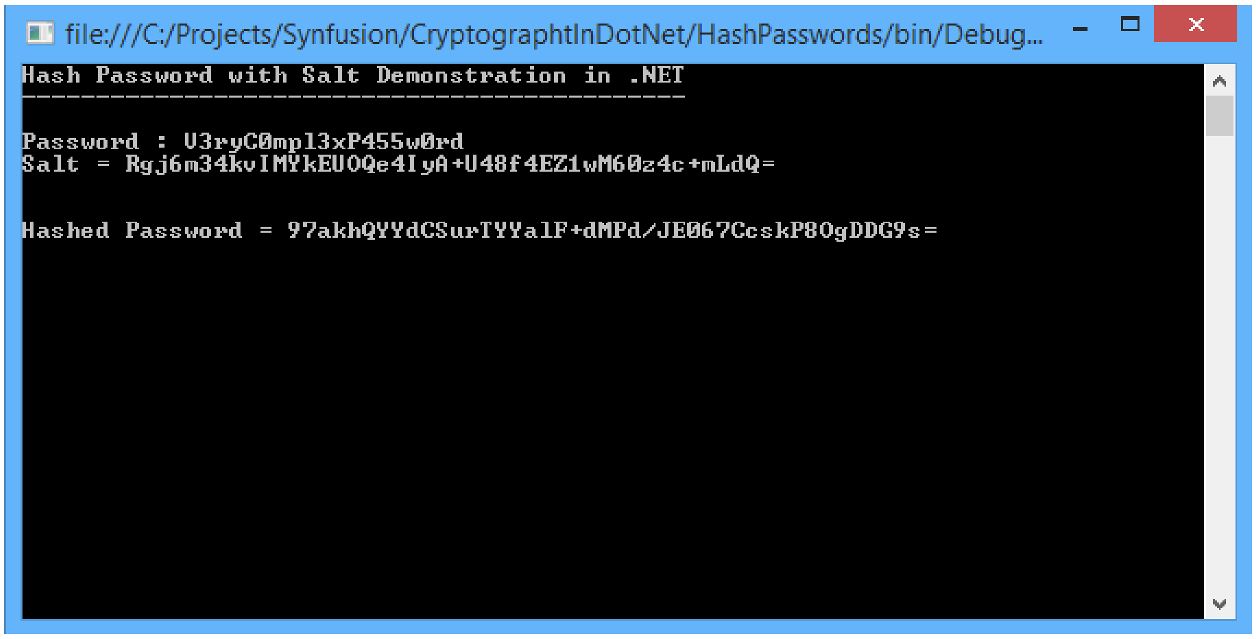
Code Listing 9

To use the preceding code example to salt and hash a password, we use the following code.

```
class Program
{
    static void Main(string[] args)
    {
        var password = "V3ryC0mpl3xP455w0rd";
        byte[] salt = Hash.GenerateSalt();

        Console.WriteLine("Hash Password with Salt Demonstration in .NET");
        Console.WriteLine("---------------------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Message 1 : " + password);
        Console.WriteLine("Salt = " + Convert.ToBase64String(salt));
        Console.WriteLine();

        var hashedPassword1 = Hash.HashPasswordWithSalt(
            Encoding.UTF8.GetBytes(password),
            salt);

        Console.WriteLine();
        Console.WriteLine("Hashed Password = " +
Convert.ToBase64String(hashedPassword1));
        Console.WriteLine();

        Console.ReadLine();
    }
}
```

Code Listing 10

When you run the sample program, you get the following output.


file:///C:/Projects/Synfusion/CryptographtInDotNet/HashPasswords/bin/Debug...

```
Hash Password with Salt Demonstration in .NET
---------------------------------------------

Password : U3ryC0mpl3xP455w0rd
Salt = Rgj6m34kvIMYkEUOQe4IyA+U48f4EZ1wM60z4c+mLdQ=


Hashed Password = 97akhQYYdCSurTYYalF+dMPd/JE067CcskP8OgDDG9s=
```

Figure 8: Example of a salted and hashed password

In this example, the salt value is combined with the password and then hashed by using SHA-256. When you store the password in the database, you also store the salt value. The salt does not need to be secret; it is there to add entropy to the password to make it harder to brute force attack or attack by using a rainbow table.

This solution for storing a password is much better than just storing a hashed password but it is still susceptible to brute force attacks as processor speeds scale with Moore's law. A stored hashed and salted password may be safe today but, in five years, it might be a trivial job to crack it.

A better solution is to use a password-based key derivation function to hash and store your password.

Password-based Key Derivation Functions

As we just discussed, the problem with hashing and storing a password (even with a salt) is that as processors get faster over the years, we run the risk of what we currently think are secure passwords being compromised because processors can use brute force attacks and rainbow table attacks faster. What we need is a solution that allows us to still hash our passwords but helps us guard against advancements in Moore's law.

Note: Moore's law, named after Gordon E. Moore, the co-founder of Intel, states that over time the number of transistors in a circuit will double approximately every 2 years.

A password-based key derivation function, or PBKDF2 as it is also known, is part of the RSA Public Key Cryptographic Standards series (PKCS #5 version 2.0). PBKDF2 is also part of the Internet Engineering Task Force's RFC2898 specification. A password-based key derivation function takes a password, a salt to add additional entropy to the password, and a number of iterations value. The number of iterations value repeats a hash or encryption cipher over the password multiple times to produce a derived key for the password that can be stored in a database or used as an encryption key.

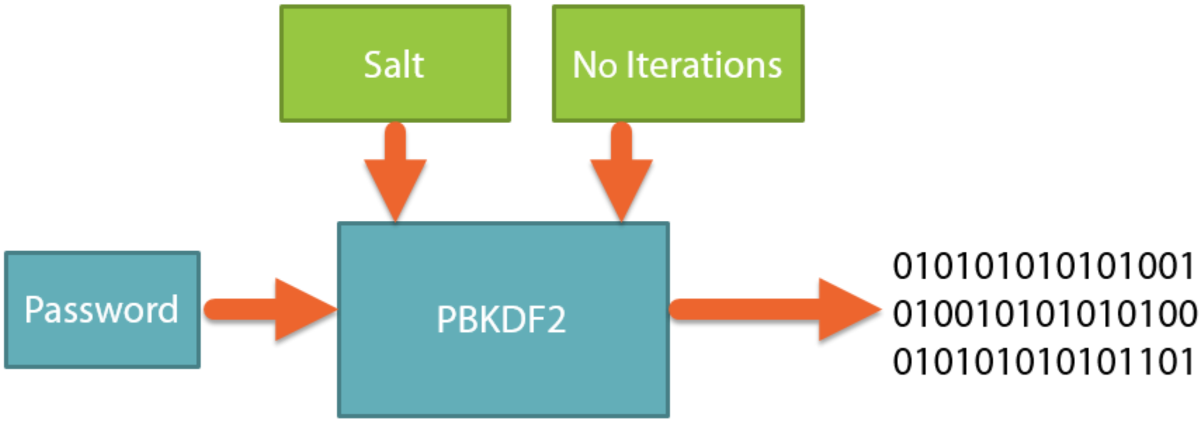


Figure 9: Password-based key derivation function

By repeating a hash or encryption process over the password multiple times, you are algorithmically slowing down the hashing process which makes brute force attacks against the password much harder. This means that fewer passwords can be tested at once. As processors get faster over time, you can increase the number of iterations used to create the new password, which means a password-based key derivation function can scale with Moore's law. A good default to start with for the number of iterations is around 50,000.

By adding a salt to the derivation function, you further reduce the ability of a rainbow table to be used to recover the original password. A good minimum length for your salt is at least 64 bits (8 bytes).

The .NET object we will use to perform the key derivation is **`Rfc2898DeriveBytes`**. When constructing this object, you pass in the data to be hashed as a byte array, the salt as a byte array, and the number of rounds you want the algorithm to perform.

The salt is generated by using the **`RNGCryptoServiceProvider`** object. The salt does not have to be secret and can be stored alongside the hashed password. The salt is designed to give added entropy to the password being hashed.

The following code demonstrates how to hash a password with **`Rfc2898DeriveBytes`**.

```
public class PBKDF2
{
    public static byte[] GenerateSalt()
    {
        using (var randomNumberGenerator = new RNGCryptoServiceProvider())
        {
            var randomNumber = new byte[32];
            randomNumberGenerator.GetBytes(randomNumber);

            return randomNumber;
        }
    }

    public static byte[] HashPassword(byte[] toBeHashed, byte[] salt, int
numberOfRounds)
    {
        using (var rfc2898 = new Rfc2898DeriveBytes(toBeHashed, salt,
numberOfRounds))
        {
            return rfc2898.GetBytes(32);
        }
    }
}
```

Code Listing 11

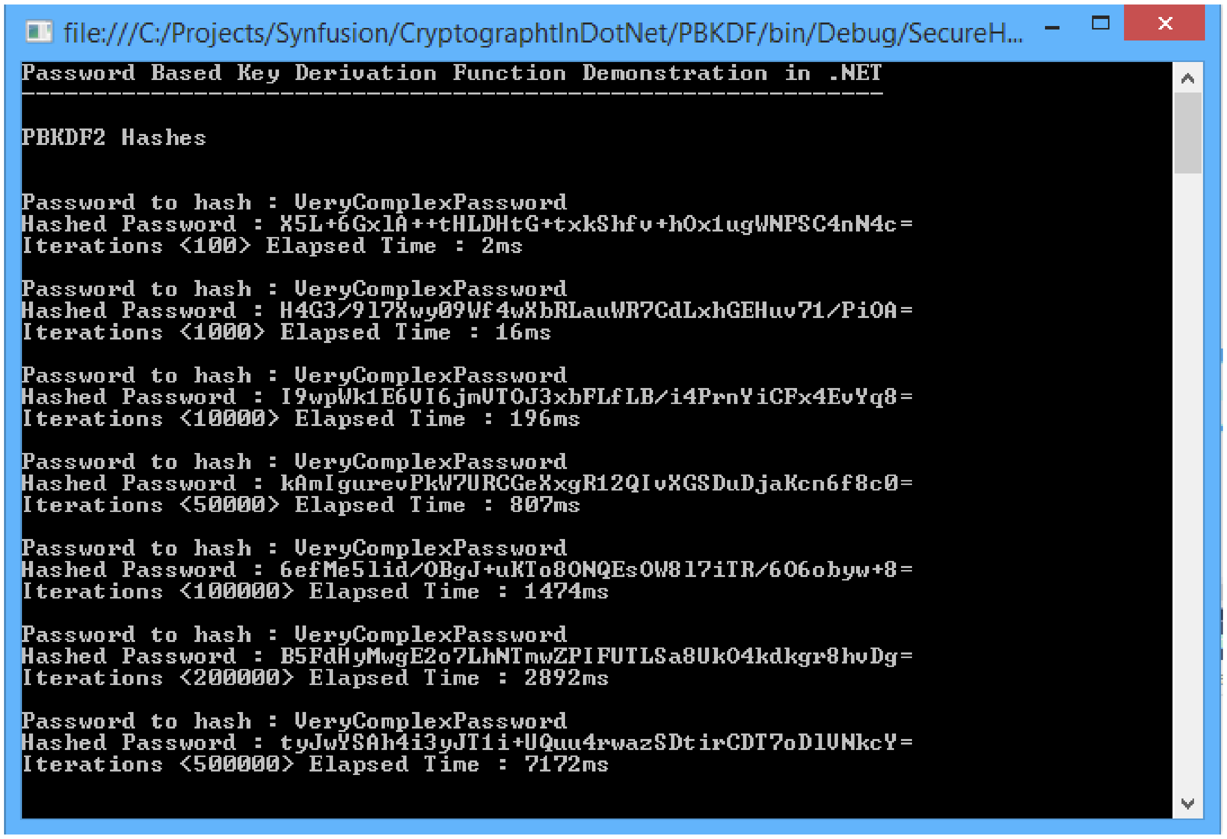
Once **`Rfc2898DeriveBytes`** has been instantiated, you then call **`GetBytes()`** to retrieve the number of bytes you want to represent the hashed password. In the previous example, we get 32 bytes (256 bits) of data for our password. I don't recommend using less, but you can use more. This depends on how many passwords you have to store and the amount of storage you want to use.

In the following code, I demonstrate the usage of the **`PBKDF2`** class created in the preceding example. The example calculates multiple hashes, with varying numbers of rounds being used. Each hashed password is timed so that you can see the impact of increasing the number of iterations.

```
class Program
{
    static void Main(string[] args)
    {
        var passwordToHash = "VeryComplexPassword";

        Console.WriteLine("Password Based Key Derivation Function Demonstration in
.NET");
        Console.WriteLine("-------------------------------------------------------
-----");
        Console.WriteLine();
        Console.WriteLine("PBKDF2 Hashes");
        Console.WriteLine();

        HashPassword(passwordToHash, 100);
        HashPassword(passwordToHash, 1000);
        HashPassword(passwordToHash, 10000);
        HashPassword(passwordToHash, 50000);
        HashPassword(passwordToHash, 100000);
        HashPassword(passwordToHash, 200000);
        HashPassword(passwordToHash, 500000);

        Console.ReadLine();
    }

    private static void HashPassword(string passwordToHash, int numberOfRounds)
    {
        var sw = new Stopwatch();

        sw.Start();
        var hashedPassword = 
PBKDF2.HashPassword(Encoding.UTF8.GetBytes(passwordToHash),
                                                    PBKDF2.GenerateSalt(),
                                                    numberOfRounds);
        sw.Stop();


        Console.WriteLine();
        Console.WriteLine("Password to hash : " + passwordToHash);
        Console.WriteLine("Hashed Password : " + 
Convert.ToBase64String(hashedPassword));
        Console.WriteLine("Iterations <" + numberOfRounds + "> Elapsed Time : " + 
sw.ElapsedMilliseconds + "ms") ;
    }
}
```

Code Listing 12

After running the example, you get the following output in the console window.

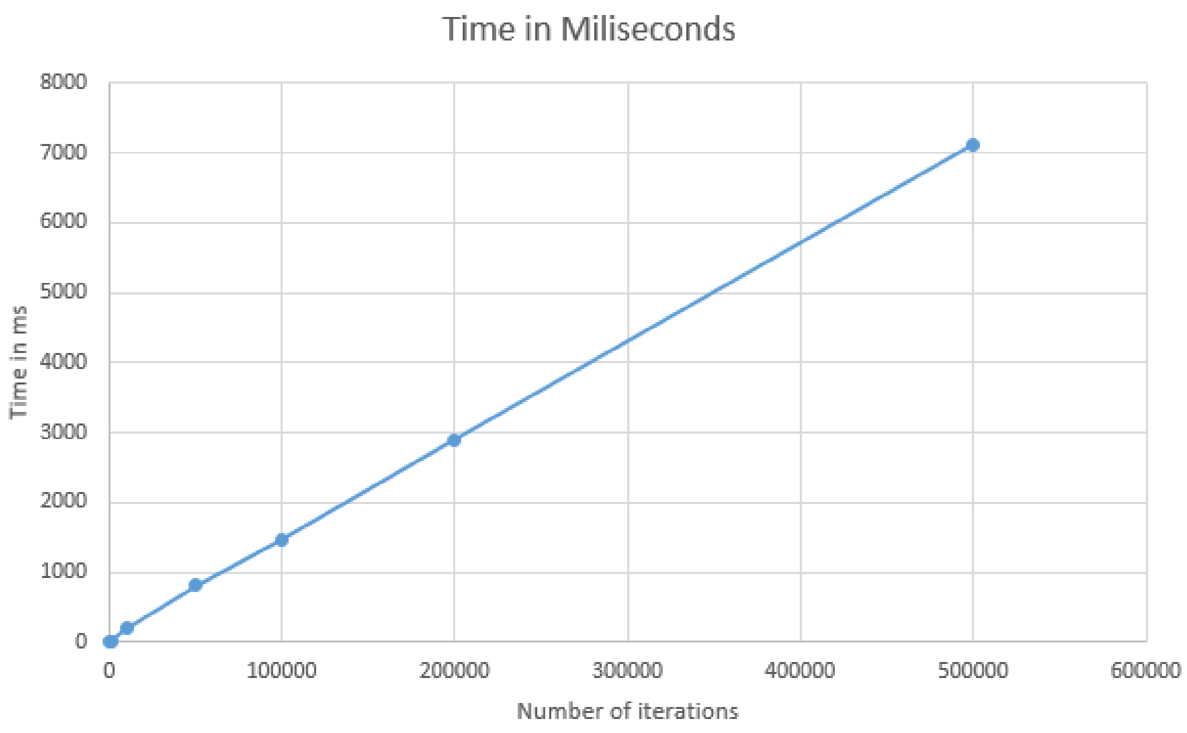


```
file:///C:/Projects/Synfusion/CryptographtInDotNet/PBKDF/bin/Debug/SecureH...
Password Based Key Derivation Function Demonstration in .NET
-----------------------------------------------------------

PBKDF2 Hashes


Password to hash : VeryComplexPassword
Hashed Password : X5L+6GxlA++tHLDHtG+txkShfv+hOx1ugWNPSC4nN4c=
Iterations <100> Elapsed Time : 2ms

Password to hash : VeryComplexPassword
Hashed Password : H4G3/9l7Xwy09Wf4wXbRLauWR7CdLxhGEHuv71/PiOA=
Iterations <1000> Elapsed Time : 16ms

Password to hash : VeryComplexPassword
Hashed Password : I9wpWk1E6VI6jmVTOJ3xbFLfLB/i4PrnYiCFx4EvYq8=
Iterations <10000> Elapsed Time : 196ms

Password to hash : VeryComplexPassword
Hashed Password : kAmIgurevPkW7URCGeXxgR12QIvXGSDuDjaKcn6f8c0=
Iterations <50000> Elapsed Time : 807ms

Password to hash : VeryComplexPassword
Hashed Password : 6efMe5lid/OBgJ+uKTo8ONQEsOW8l7iTR/6O6obyw+8=
Iterations <100000> Elapsed Time : 1474ms

Password to hash : VeryComplexPassword
Hashed Password : B5FdHyMwgE2o7LhNTmwZPIFUTLSa8UkO4kdkgr8hvDg=
Iterations <200000> Elapsed Time : 2892ms

Password to hash : VeryComplexPassword
Hashed Password : tyJwYSAh4i3yJT1i+UQuu4rwazSDtirCDT7oDlVNkcY=
Iterations <500000> Elapsed Time : 7172ms
```

Figure 10: Example output from the PBKDF2 sample showing different timings depending on the number of iterations

As you can see in the following chart, the more iterations you perform, the longer the hash takes to calculate. In the following chart, you can see the relationship between the number of iterations and the time in milliseconds.

If you are storing passwords, then you will typically be using them for authentication into your system. You need to trade off the amount of time it takes to calculate the hash by using the PBKDF2 against the responsiveness of someone logging into your system, as you will need to recalculate the hash on your server to make sure it matches the password hash sent to you. The more iterations you use, the slower this will be.



.

Figure 11: Relationship between the number of iterations and the time in milliseconds

Chapter 5 Symmetric Encryption

A symmetric encryption algorithm is a two-way encryption process that uses the same key for both encryption and decryption of your message. In theory, this sounds straightforward as both the sender and receiver of the message should know the key, but in practice securely sharing a key is very hard to do.

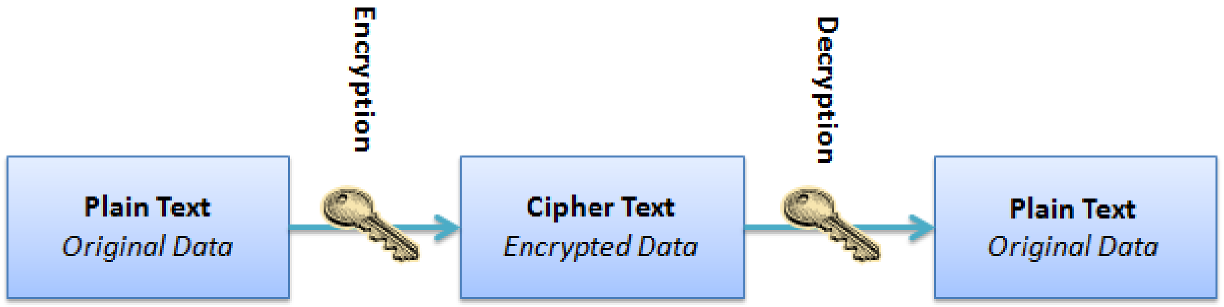


Figure 12: Symmetric encryption process

Symmetric-key algorithms can fall into 2 categories. They are:

- Stream ciphers, where each byte is encrypted one at a time.
- Block ciphers, where data is encrypted in blocks. These blocks can be different sizes based on the algorithms being used. The Advanced Encryption Standard uses a block size of 128 bits.

In this chapter, we will focus on block ciphers. We will cover the Data Encryption Standard (DES), Triple DES, and the Advanced Encryption Standard (AES).

Similarities in the .NET Libraries

The AES, DES, and Triple DES algorithms in .NET all share a common abstract base class, **`SymmetricAlgorithm`**, which gives them a lot of common functionality. Before we look at the individual algorithms and examples of their use, let's look at some of the common properties associated with them that you will need to use. The rest of the class is documented on MSDN at http://msdn.microsoft.com/en-us/library/system.security.cryptography.symmetricalgorithm(v=vs.110).aspx.

Encryption Mode

The first property we will look at is the **`Mode`** property on the **`SymmetricAlgorithm`** object. Block cipher algorithms such as AES, DES, and Triple DES encrypt data in block units rather than in single bytes at a time. The most common block size is eight bytes. Block ciphers use the same encryption algorithm for each block. Because of this, a block of plaintext will always return the same ciphertext when encrypted with the same key and algorithm. Because this behavior can be used to crack a cipher, cipher modes are introduced that modify the encryption process based on feedback from earlier block encryptions. The modes are listed in the following table.

Table 1: Symmetric Algorithm Modes in .NET

Mode Name	Description
CBC	The cipher block chaining (CBC) mode introduces feedback. Before each plaintext block is encrypted, it is combined with the ciphertext of the previous block with a bitwise exclusive OR operation. This ensures that, even if the plaintext contains many identical blocks, they will each encrypt to a different ciphertext block. The initialization vector (IV) is combined with the first plaintext block by a bitwise exclusive OR operation before the block is encrypted.
CFB	Ciphertext feedback (CFB) is a mode of operation for a block cipher. In contrast to the CBC mode, which encrypts a set number of bits of plaintext at a time, it is at times desirable to encrypt and transfer some plaintext values instantly, one at a time. Like CBC, CFB also makes use of an IV. CFB uses a block cipher as a component of a random number generator. In CFB mode, the previous ciphertext block is encrypted and the output is exclusively ORed with the current plaintext block to create the current ciphertext block. The exclusive OR operation conceals plaintext patterns. Plaintext cannot be directly worked on unless there is a retrieval of blocks from either the beginning or end of the ciphertext.
CTS	The ciphertext stealing (CTS) mode handles any length of plaintext and produces ciphertext whose length matches the plaintext length. This mode behaves like the CBC mode for all but the last two blocks of the plaintext.
ECB	The electronic codebook (ECB) mode encrypts each block individually. Any blocks that are identical and in the same message, or that are in a different message encrypted with the same key, will be transformed into identical ciphertext blocks. The disadvantage of ECB mode is that identical plaintext blocks are encrypted to identical ciphertext blocks (thus, it does not hide data patterns well). In some senses, it doesn't provide message confidentiality at all and so it is not recommended for cryptographic protocols.

Mode Name	Description
OFB	The output feedback (OFB) mode processes small increments of plaintext into ciphertext instead of processing an entire block at a time. This mode is similar to CFB, but it differs in the way the shift register is filled. If a bit in the ciphertext is mangled, the corresponding bit of plaintext will be mangled. However, if there are extra or missing bits from the ciphertext, the plaintext will be mangled from that point on.

The default mode in .NET for AES, DES, and Triple DES is CBC mode, and unless you have a good need to change that default mode, you should just go with the default.

Padding

The next property we will look at is `Padding`. Padding specifies what padding to apply when the message block being encrypted is shorter than the full number of bytes needed for a cryptographic operation.

Table 2: Symmetric Algorithm Padding Types

Padding Name	Description
ANSIX923	The ANSIX923 padding string consists of a sequence of bytes filled with zeros before the length.
ISO10126	The ISO10126 padding string consists of random data before the length.
None	No padding is done.
PKCS7	The PKCS #7 padding string consists of a sequence of bytes, each of which is equal to the total number of padding bytes added.
Zeros	The padding string consists of bytes set to zero.

The default padding in .NET for AES, DES, and Triple DES is PKCS7 mode, and unless you have a good need to change it, you should just go with the default.

Key

The **`Key`** property is a byte array that is used to store the encryption key prior to running encrypt and decrypt operations. The data in this property can literally be anything, but you should make sure you generate a secure key. There are two ways you can do this. You can use the **`RNGCryptoServiceProvider`** object and generate a byte array to the desired key length, or you can call the **`GenerateKey()`** method. Either way is fine. The examples used in this chapter all use the **`RNGCryptoServiceProvider`** object to generate the random number.

Initialization Vector (IV)

The initialization vector (**`IV`**) property is a byte array that is used to store an IV. An IV is an arbitrary number that can be used along with a secret key for data encryption. This number, also called a nonce, is employed only one time in any session.

The use of an IV prevents repetition in data encryption, making it more difficult for a hacker who is using a dictionary attack to find patterns to break a cipher. For example, a sequence might appear twice or more within the body of a message. If there are repeated sequences in encrypted data, an attacker could assume that the corresponding sequences in the plaintext message were also identical. The IV prevents the appearance of corresponding duplicate character sequences in the ciphertext.

The ideal IV is a random number that is made known to the destination computer to facilitate decryption of the data when it is received. The IV can be agreed on in advance, transmitted independently, or included as part of the session setup prior to the exchange of the message data. The length of the IV (the number of bits or bytes it contains) depends on the method of encryption. The IV does not have to be kept secret and can be transmitted or stored along with the message in the clear.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) used to be the default standard for symmetric encryption of data. DES was based on an earlier design by Horst Feistel and developed by IBM in the early 1970s. DES was submitted to the National Bureau of Standards as part of a drive to propose a new standard for protecting government data.

The National Bureau of Standards eventually selected a modified version of DES after consultation with the National Security Agency, and it was released as a standard in 1997.

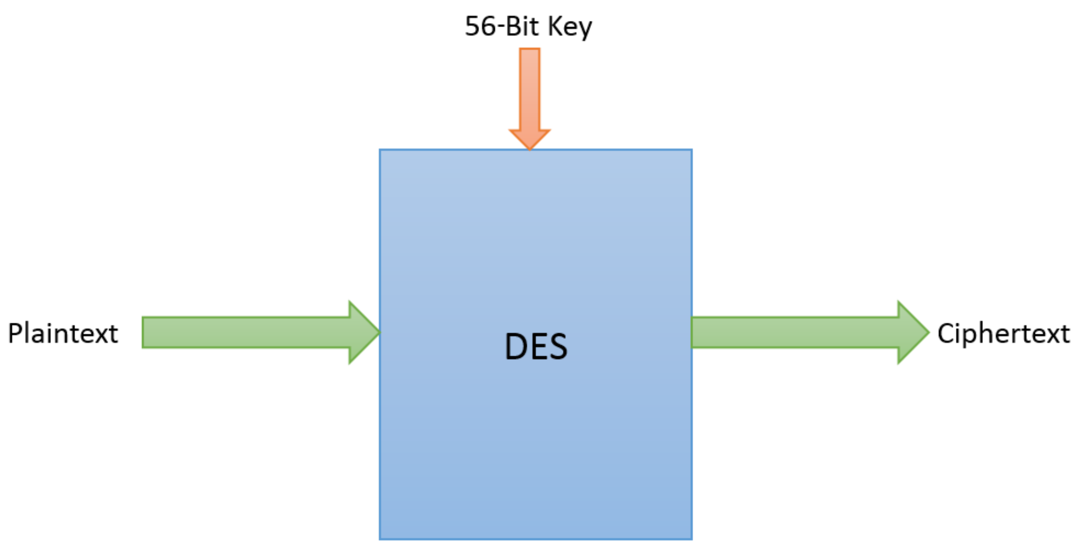


Figure 13: DES encryption algorithm

If you are developing a new system that requires the encryption of data, then I do not recommend using DES as it is considered a weak algorithm by today's standards due to the 56-bit key size.

DES is being included in this book as it is feasible that you may have to integrate with an older legacy system that still uses DES as its encryption standard.

In the following example, you will see a class called **`DesEncryption`** that contains two public methods, **`Encrypt`** and **`Decrypt`**.

```
public class DesEncryption
{
    public byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] iv)
    {
        using (var des = new DESCryptoServiceProvider())
        {
            des.Mode = CipherMode.CBC;
            des.Padding = PaddingMode.PKCS7;

            des.Key = key;
            des.IV = iv;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
des.CreateEncryptor(), CryptoStreamMode.Write);
                cryptoStream.Write(dataToEncrypt, 0, dataToEncrypt.Length);
                cryptoStream.FlushFinalBlock();

                return memoryStream.ToArray();
            }
        }
    }

    public byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] IV)
    {
        using (var des = new DESCryptoServiceProvider())
        {
            des.Mode = CipherMode.CBC;
            des.Padding = PaddingMode.PKCS7;

            des.Key = key;
            des.IV = IV;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
des.CreateDecryptor(), CryptoStreamMode.Write);

                cryptoStream.Write(dataToDecrypt, 0, dataToDecrypt.Length);
                cryptoStream.FlushFinalBlock();

                return memoryStream.ToArray(); ;
            }
        }
    }
}
```

Code Listing 13

To encrypt some data, you first construct the **`DESCryptoServiceProvider`** object and wrap it in a **`using`** statement so that it is properly disposed of. Then, the **`Mode`** and **`Padding`** is explicitly set. They are set to the defaults in this example, but it doesn't hurt to do this explicitly to state your intention when configuring the algorithm.

Next, the **`Key`** and **`IV`**s are set. These are passed into the **`Encrypt`** and **`Decrypt`** methods. The key and IVs have to be the same for both the encrypt and decrypt operations. The key has to be secret, but the IV doesn't have to be.

After that, a new **`MemoryStream`** is constructed and passed into a new instance of the **`CryptoStream`** object, along with the result of the **`CreateEncryptor()`** or **`CreateDecryptor()`** method and the **`CryptoStreamMode.Write`** enumeration. **`des.CreateEncryptor()`** or **`des.CreateDecryptor()`** creates a symmetric encryptor or decryptor object with the current **`Key`** property and **`IV`**.

Note: .NET uses a stream-oriented design for cryptography. The core of this design is CryptoStream. Any cryptographic objects that implement CryptoStream can be chained together with any objects that implement Stream, so the streamed output from one object can be fed into the input of another object.

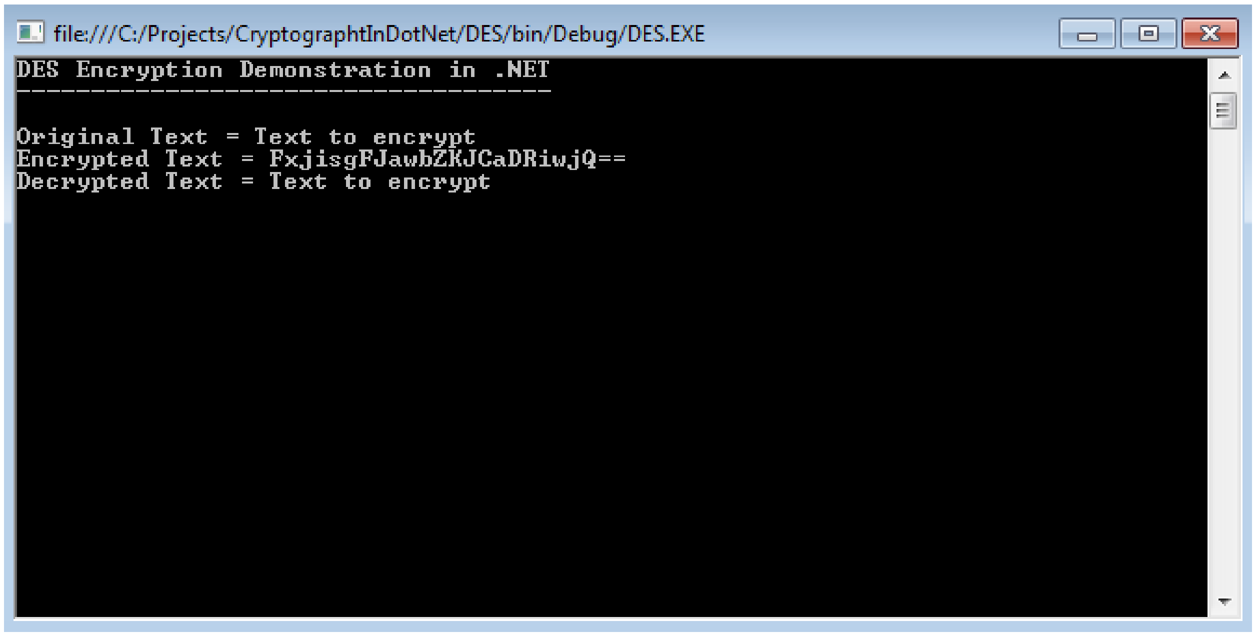
Once the **CryptoStream** object has been created, you then call the **Write()** method by passing in the data to encrypt or decrypt and the data length. Then, a call to **FlushFinalBlock()** is made to update the underlying data with the current state of the buffer and then clear the buffer. Next, you call **ToArray()** on the initial **MemoryStream** to convert the final result into a byte array to pass back to the calling object.

In the following code sample, you can see an example in which the **DesEncryption** class is used to encrypt and decrypt some data.

```
class Program
{
    static void Main(string[] args)
    {
        var des = new DesEncryption();
        var key = Random.GenerateRandomNumber(8);
        var iv = Random.GenerateRandomNumber(8);
        var original = "Text to encrypt";

        var encrypted = des.Encrypt(Encoding.UTF8.GetBytes(original), key, iv);
        var decrypted = des.Decrypt(encrypted, key, iv);

        var decryptedMessage = Encoding.UTF8.GetString(decrypted);

        Console.WriteLine("DES Encryption Demonstration in .NET");
        Console.WriteLine("------------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Text = " + original);
        Console.WriteLine("Encrypted Text = " +
Convert.ToBase64String(encrypted));
        Console.WriteLine("Decrypted Text = " + decryptedMessage);

        Console.ReadLine();
    }
}
```

Code Listing 14

DES internally uses a 56-bit key, but you will notice we are passing in eight bytes, which is 64 bits. Out of these 64 bits, only 56 are actually used by the DES algorithm. Eight bits are used for checking parity and are discarded thereafter.



```
DES Encryption Demonstration in .NET
------------------------------------

Original Text = Text to encrypt
Encrypted Text = FxjisgFJawbZKJCaDRiwjQ==
Decrypted Text = Text to encrypt
```

Figure 14: Output from the DES sample application

Triple DES

Triple DES is a variant of the Data Encryption Standard (DES) algorithm where DES is applied to a message three times. As computer hardware increased in processing power, the original DES algorithm was subjected to many brute force attacks. Triple DES was a response to these attacks without the need to develop a new block cipher.

As with DES, if you are developing a new system, you should avoid using 3DES to encrypt your data. You should only use 3DES if you have to integrate with a legacy system that still makes use of it.

In the following example, you will see a class called **`TripleDESEncryption`** that contains two public methods, **`Encrypt`** and **`Decrypt`**.

```
public class TripleDESEncryption
{
    public byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] iv)
    {
        using (var des = new TripleDESCryptoServiceProvider())
        {
            des.Mode = CipherMode.CBC;
            des.Padding = PaddingMode.PKCS7;

            des.Key = key;
            des.IV = iv;
            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
des.CreateEncryptor(), CryptoStreamMode.Write);
                cryptoStream.Write(dataToEncrypt, 0, dataToEncrypt.Length);
                cryptoStream.FlushFinalBlock();

                return memoryStream.ToArray();
            }
        }
    }

    public byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] IV)
    {
        using (var des = new TripleDESCryptoServiceProvider())
        {
            des.Mode = CipherMode.CBC;
            des.Padding = PaddingMode.PKCS7;

            des.Key = key;
            des.IV = IV;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
des.CreateDecryptor(), CryptoStreamMode.Write);

                cryptoStream.Write(dataToDecrypt, 0, dataToDecrypt.Length);
                cryptoStream.FlushFinalBlock();

                var decryptBytes = memoryStream.ToArray();

                return decryptBytes;
            }
        }
    }
}
```

Code Listing 15

To encrypt some data, you first construct the **`TripleDESCryptoServiceProvider`** object and wrap it in a **`using`** statement so that it is properly disposed of. Next, the **`Mode`** and **`Padding`** are explicitly set. They are set to the defaults in this example, but it doesn't hurt to do this explicitly to state your intention when configuring the algorithm.

After that, the key and IVs are set. These are passed into the **`Encrypt`** and **`Decrypt`** methods. They key and IVs have to be the same for both the encrypt and decrypt operations. The key has to be secret but the IV doesn't have to be.

Next, a new **MemoryStream** is constructed and passed into a new instance of the **CryptoStream** object along with the result of the **CreateEncryptor()** or **CreateDecryptor()** method and the **CryptoStreamMode.Write** enumeration. **TripleDesEncryption.CreateEncryptor()** or **TripleDesEncryption**. **CreateDecryptor()** creates a symmetric encryptor or decryptor object with the current **Key** property and **IV**.

Once the **CryptoStream** object has been created, you then call the **Write()** method by passing in the data to encrypt or decrypt and the data length. Then, a call to **FlushFinalBlock()** is made to update the underlying data with the current state of the buffer, and then clear the buffer. Next, you call **ToArray()** on the initial **MemoryStream** to convert the final result into a byte array to pass back to the calling object.

In the following code sample, you can see an example in which the **TripleDesEncryption** class is used to encrypt and decrypt some data.

```
class Program
{
    static void Main(string[] args)
    {
        var tripleDes = new TripleDESEncryption();
        var key = Random.GenerateRandomNumber(24);
        var iv = Random.GenerateRandomNumber(8);
        var original = "Text to encrypt";

        var encrypted = tripleDes.Encrypt(Encoding.UTF8.GetBytes(original), key,
iv);
        var decrypted = tripleDes.Decrypt(encrypted, key, iv);

        var decryptedMessage = Encoding.UTF8.GetString(decrypted);

        Console.WriteLine("Triple DES Encryption Demonstration in .NET");
        Console.WriteLine("---------------------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Text = " + original);
        Console.WriteLine("Encrypted Text = " +
Convert.ToBase64String(encrypted));
        Console.WriteLine("Decrypted Text = " + decryptedMessage);

        Console.ReadLine();
    }
}
```

Code Listing 16

DES internally uses a 56-bit key. 3DES works by running DES three times in a series. You have a choice of two key configurations with **TripleDESCryptoServiceProvider**. In the preceding example, when the key is generated, 24 bytes are created. This is three times eight-byte keys which is 192 bits in total, or three times 56-bit DES keys including eight bits of parity per key.

When you use the keys in this configuration, 3DES looks like the following diagram. Here, some plaintext is passed into the first instance of DES and encrypted with key 1. Next, the results of that are passed into another instance of DES and encrypted with key 2. Finally, the output of this is fed into another instance of DES and encrypted with key 3. The result of this is the encrypted ciphertext.

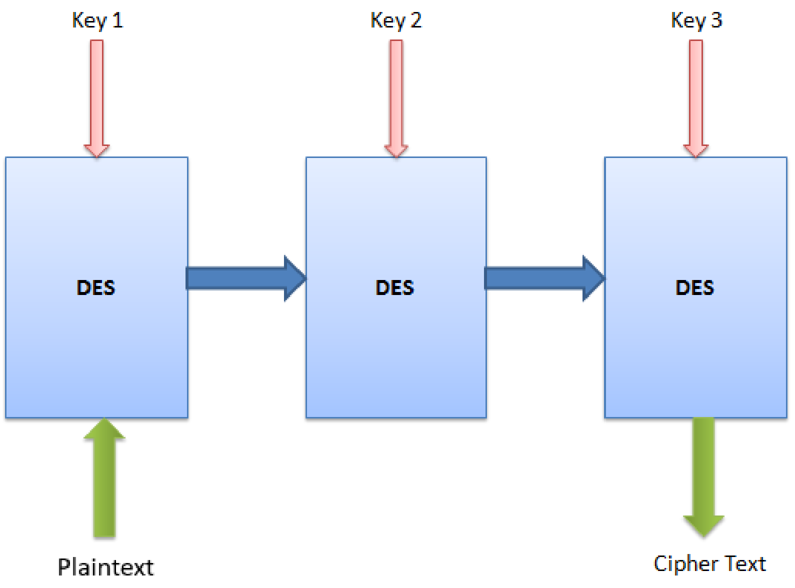


Figure 15: 3DES using three independent keys

Instead of generating a key 24 bytes in length, you can generate a key that is 16 bytes in length. This equates to two times 56-bit DES keys including eight bits of parity per key. This will make 3DES operate as in the following diagram.

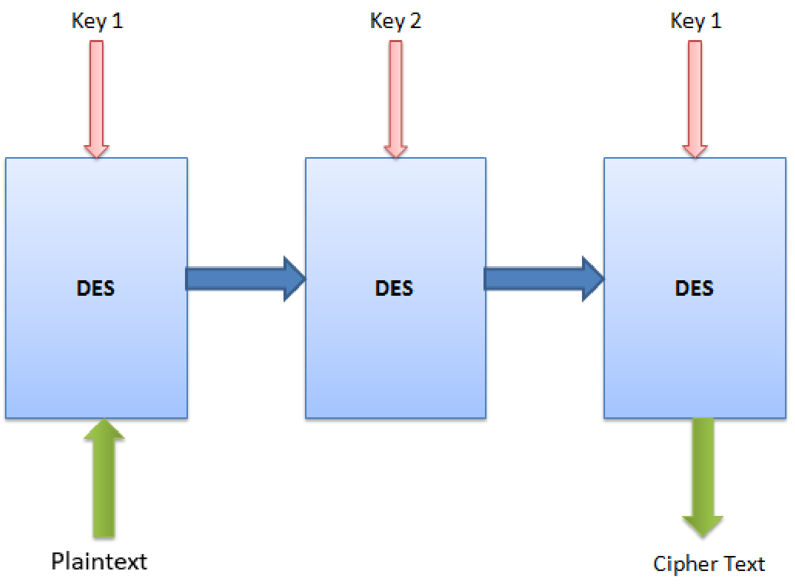
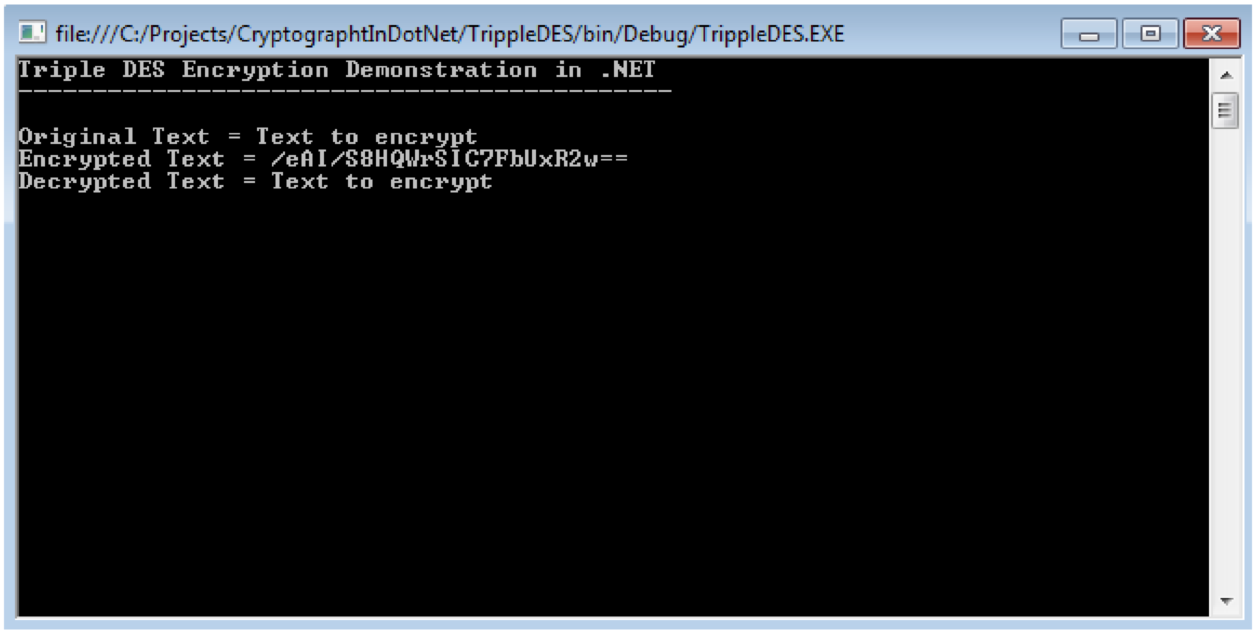


Figure 16: 3DES using two independent keys

Here, some plaintext is passed into the first instance of DES and is encrypted with key 1. Then, the results of that are passed into another instance of DES and encrypted with key 2. Finally, the output of this is fed into another instance of DES and encrypted again with key 1. The result of this is the encrypted ciphertext.


file:///C:/Projects/CryptographtInDotNet/TrippleDES/bin/Debug/TrippleDES.EXE

```
Triple DES Encryption Demonstration in .NET
--------------------------------------------

Original Text = Text to encrypt
Encrypted Text = /eAI/S8HQWrSIC7FbUxR2w==
Decrypted Text = Text to encrypt
```

Figure 17: Output from the 3DES sample application

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is the latest encryption standard adopted by NIST in 2001 for the symmetric encryption of messages. The AES algorithm was selected as part of a contest to find a replacement for the Data Encryption Standard (DES). This algorithm was based on the Rijndael cipher developed by two Belgian mathematicians, Joan Daemen and Vincent Rijmen.

As part of the NIST standard, AES uses a block size of 128 bits and supports three key lengths: 128, 192, and 256 bits.

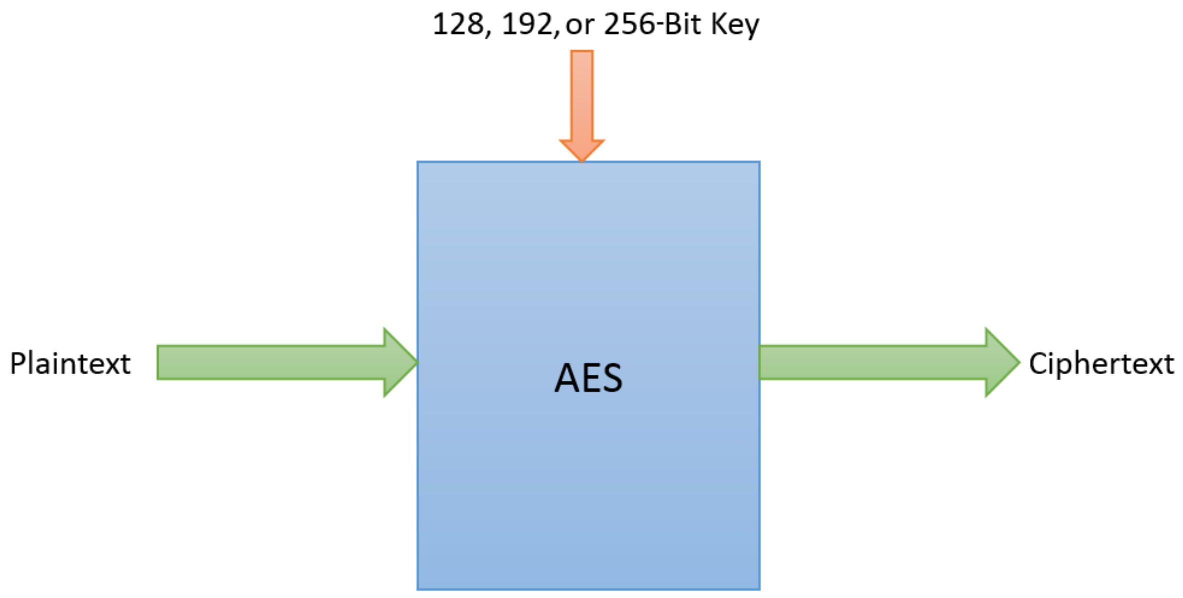


Figure 18: AES encryption algorithm

If you are developing a new system and you are required to encrypt data, then you should definitely use the AES as your encryption algorithm. At the time of writing, AES is still considered the recommended standard. You should use a key size of 256 bits (32 bytes).

AesManaged or AesCryptoServiceProvider

.NET provides two implementations of the AES encryption algorithm, **`AesManaged`** and **`AesCryptoServiceProvider`**, but which one should you use? They both provide the same functionality in that they both implement the AES encryption specification. But the main difference is that **`AesManaged`** is a .NET-specific implementation whereas **`AesCryptoServiceProvider`** uses the underlying cryptography libraries in Windows, which are FIPS-certified.

Note: The NIST issued the FIPS 140-2 Publication Series to coordinate the requirements and standards for cryptography modules that include both hardware and software components. Protection of a cryptographic module within a security system is necessary to maintain the confidentiality and integrity of the information protected by the module. This standard specifies the security requirements that will be satisfied by a cryptographic module.

If you want to ensure you are encrypting data with AES by using a compliant implementation, then **`AesCryptoServiceProvider`** is the implementation that you will want to use, especially if you need to interoperate with systems that are also compliant with FIPS 140-2. The following examples are all based around **`AesCryptoServiceProvider`**.

```
class Program
{
    static void Main(string[] args)
    {
        var aes = new AesEncryption();
        var key = Random.GenerateRandomNumber(32);
        var iv = Random.GenerateRandomNumber(16);
        var original = "Text to encrypt";

        var encrypted = aes.Encrypt(Encoding.UTF8.GetBytes(original), key, iv);
        var decrypted = aes.Decrypt(encrypted, key, iv);

        var decryptedMessage = Encoding.UTF8.GetString(decrypted);

        Console.WriteLine("AES Encryption Demonstration in .NET");
        Console.WriteLine("------------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Text = " + original);
        Console.WriteLine("Encrypted Text = " +
Convert.ToBase64String(encrypted));
        Console.WriteLine("Decrypted Text = " + decryptedMessage);

        Console.ReadLine();
    }
}
```

Code Listing 17

To encrypt some data, you first construct the **`AesCryptoServiceProvider`** object and wrap it in a **`using`** statement so that it is properly disposed of. Then, the **`Mode`** and **`Padding`** is explicitly set. They are set to the defaults in this example, but it doesn't hurt to do this explicitly to state your intention when configuring the algorithm.

Next, the **`Key`** and **`IV`**s are set. These are passed into the **`Encrypt`** and **`Decrypt`** methods. The key and IVs have to be the same for both the encrypt and decrypt operations. The key has to be secret but the IV doesn't have to be.

After that, a new **`MemoryStream`** is constructed and passed into a new instance of the **`CryptoStream`** object along with the result of the **`CreateEncryptor()`** or **`CreateDecryptor()`** method and the **`CryptoStreamMode.Write`** enumeration. **`AesEncryption.CreateEncryptor()`** or **`AesEncryption.CreateDecryptor()`** creates a symmetric encryptor or decryptor object with the current **`Key`** property and **`IV`**.

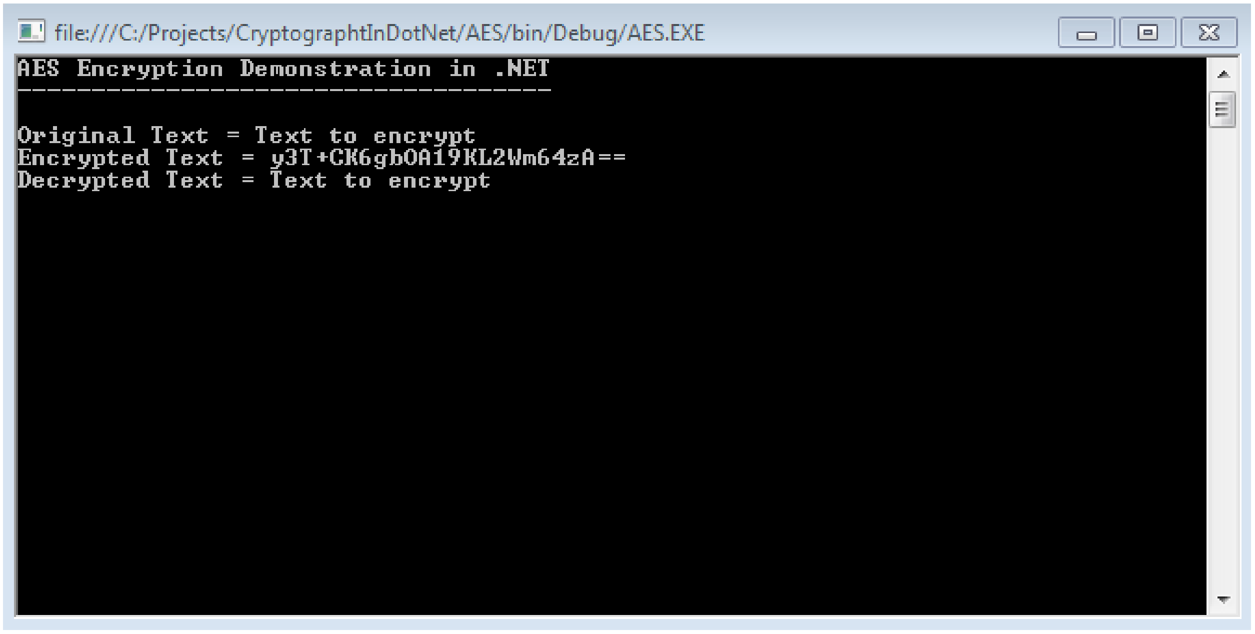
Once the **`CryptoStream`** object has been created, you then call the **`Write()`** method by passing in the data to encrypt or decrypt and the data length. Then, a call to **`FlushFinalBlock()`** is made to update the underlying data with the current state of the buffer and then clear the buffer. Next, you call **`ToArray()`** on the initial **`MemoryStream`** to convert the final result into a byte array to pass back to the calling object.

In the following code sample, you can see an example in which the **`AesEncryption`** class is used to encrypt and decrypt some data.

```
public class AesEncryption
{
    public byte[] GenerateRandomNumber(int length)
    {
        using (var randomNumberGenerator = new RNGCryptoServiceProvider())
        {
            var randomNumber = new byte[length];
            randomNumberGenerator.GetBytes(randomNumber);

            return randomNumber;
        }
    }

    public byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] iv)
    {
        using (var aes = new AesCryptoServiceProvider())
        {
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            aes.Key = key;
            aes.IV = iv;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
aes.CreateEncryptor(), CryptoStreamMode.Write);
                cryptoStream.Write(dataToEncrypt, 0, dataToEncrypt.Length);
                cryptoStream.FlushFinalBlock();

                return memoryStream.ToArray();
            }
        }
    }

    public byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] IV)
    {
        using (var aes = new AesCryptoServiceProvider())
        {
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            aes.Key = key;
            aes.IV = IV;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
aes.CreateDecryptor(), CryptoStreamMode.Write);

                cryptoStream.Write(dataToDecrypt, 0, dataToDecrypt.Length);
                cryptoStream.FlushFinalBlock();

                var decryptBytes = memoryStream.ToArray();
                return decryptBytes;
            }
        }
    }
}
```

Code Listing 18

AES internally uses a 128, 192, or 256-bit key. In the previous example, we are using a 256-bit (32 bytes) key. The IV is set to 16 bytes (128 bits). This is used to initialize the first block in the encryption process. As previously stated, the IV does not have to be kept secret and can be sent in the clear along with the encrypted message. You should not reuse the same IV for subsequent messages. You should generate a new IV either by using the **`RNGCryptoServiceProvider`** or the **`GenerateIV()`** method in the **`AesCryptoServiceProvider`** class.


file:///C:/Projects/CryptographtInDotNet/AES/bin/Debug/AES.EXE

```
AES Encryption Demonstration in .NET
------------------------------------

Original Text = Text to encrypt
Encrypted Text = y3T+CK6gbOA19KL2Wm64zA==
Decrypted Text = Text to encrypt
```

Figure 19: Example of the AES encryption algorithm

Chapter 6 Asymmetric Encryption

The main problem with symmetric encryption is that of securely sharing keys. For a recipient to decrypt a message, they need the same key as the sender, and this exchange of keys can be very difficult to do securely.

A good solution to this problem is to use asymmetric cryptography, which is also referred to as public key cryptography. With public key cryptography you have two keys. A public key which anyone can know, and a private key which only the recipient of a message knows. These keys are mathematically linked.

A sender uses the public key to encrypt a message and the recipient uses their private key to decrypt the message. The term "asymmetric" is used because this method uses two different linked keys that perform inverse operations from each other whereas symmetric cryptography uses the same key to perform both operations.

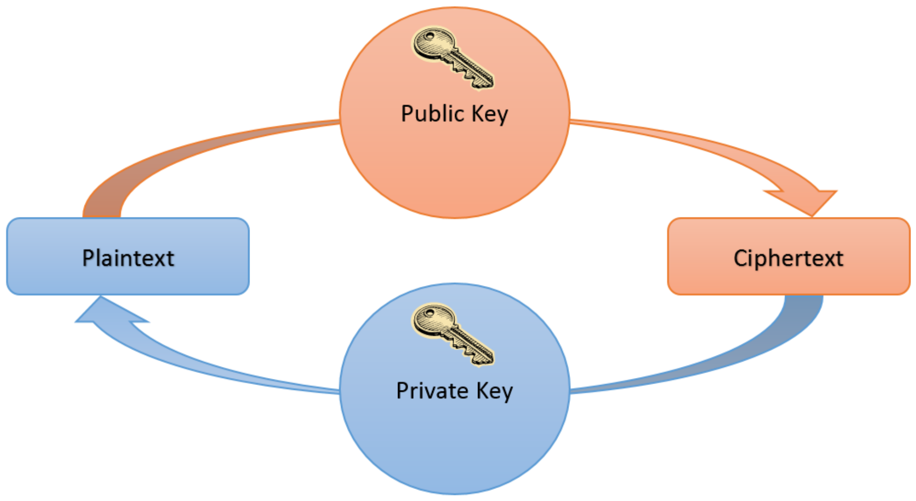


Figure 20: RSA encryption algorithm

It is very easy to generate both the public and private key pair, but the power of asymmetric cryptography comes from the fact it is impossible for a private key to be determined from its corresponding public key. It is only the private key that needs to be kept secret in the key pair.

RSA

RSA is public key encryption technology developed by RSA Security LLC (formerly RSA Security, Inc.). The acronym stands for Rivest, Shamir, and Adelman, the inventors of the technique. The RSA algorithm is based on the fact that there is no efficient way to factor very large numbers. Deducing an RSA key, therefore, requires an extraordinary amount of computer processing power and time.

The RSA algorithm has become the de facto standard for industrial-strength encryption, especially for data sent over the Internet. RSA is built into many software products.

There is a drawback to the RSA algorithm, though. You can only encrypt data that is smaller than the size of the key, so this makes RSA quite limited. It is more common to use RSA to encrypt a randomly generated, symmetric AES key. This means you can send the RSA-encrypted AES key safely to a recipient and then use AES with that key to encrypt your data. We will explore this option in the next chapter on hybrid encryption.

In the following examples, we will look at three example implementations of RSA in .NET. The first uses an in-memory set of keys, the second uses keys that are stored in XML files, and the final example stores its keys in the Crypto Service Provider key store.

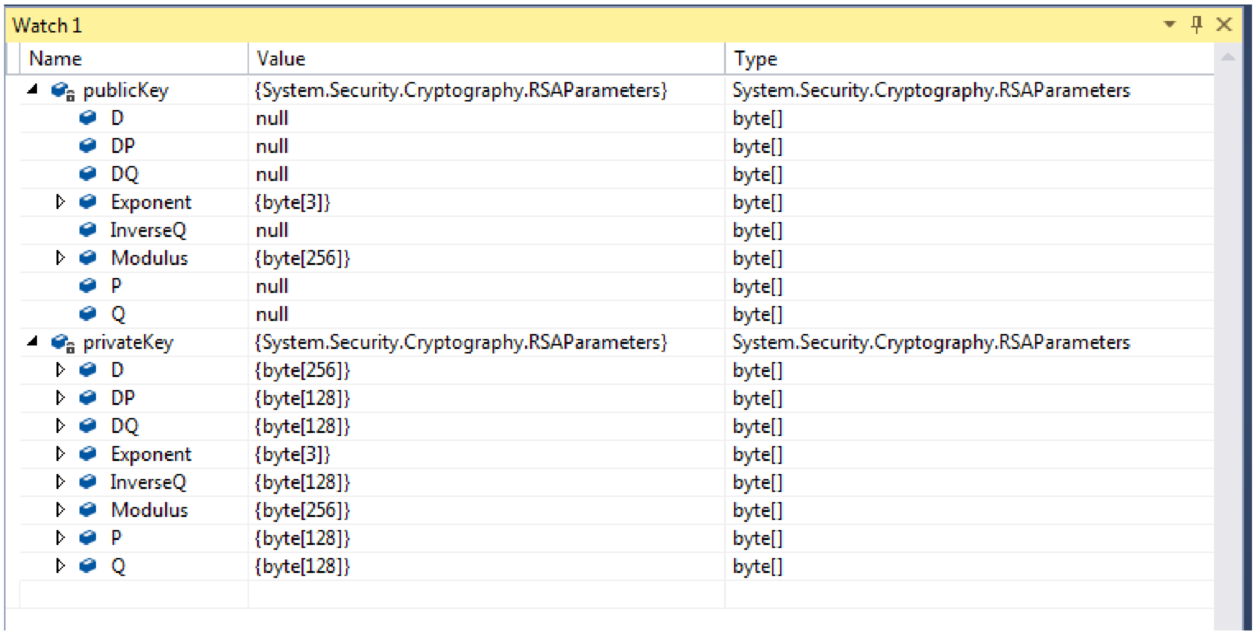
Example with In-memory Keys

In the following example we have a class called **`RSAWithRSAParameterKey`**. This class is a self-contained implementation that will allow you to generate a public/private key pair, and use those keys to encrypt and decrypt some data.

```
public class RSAWithRSAParameterKey
{
    private RSAParameters publicKey;
    private RSAParameters privateKey;

    public void AssignNewKey()
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            publicKey = rsa.ExportParameters(false);
            privateKey = rsa.ExportParameters(true);
        }
    }

    public byte[] EncryptData(byte[] dataToEncrypt)
    {
        byte[] cipherbytes;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            rsa.ImportParameters(publicKey);

            cipherbytes = rsa.Encrypt(dataToEncrypt, true);
        }

        return cipherbytes;
    }

    public byte[] DecryptData(byte[] dataToEncrypt)
    {
        byte[] plain;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;

            rsa.ImportParameters(privateKey);
            plain = rsa.Decrypt(dataToEncrypt, true);
        }

        return plain;
    }
}
```

Code Listing 19

The first method is **`AssignNewKey()`**. This method will generate both a public and a private key and store them in two private properties, **`publicKey`** and **`privateKey`**. These two properties represent the raw key data expressed as byte arrays.


Watch 1

Name	Value	Type
publicKey	{System.Security.Cryptography.RSAParameters}	System.Security.Cryptography.RSAParameters
D	null	byte[]
DP	null	byte[]
DQ	null	byte[]
Exponent	{byte[3]}	byte[]
InverseQ	null	byte[]
Modulus	{byte[256]}	byte[]
P	null	byte[]
Q	null	byte[]
privateKey	{System.Security.Cryptography.RSAParameters}	System.Security.Cryptography.RSAParameters
D	{byte[256]}	byte[]
DP	{byte[128]}	byte[]
DQ	{byte[128]}	byte[]
Exponent	{byte[3]}	byte[]
InverseQ	{byte[128]}	byte[]
Modulus	{byte[256]}	byte[]
P	{byte[128]}	byte[]
Q	{byte[128]}	byte[]

Figure 21: RSA public and private keys stored in an RSAParameters object

In the previous figure, you can see the contents of the public key and the private key. The public key contains a subset of the data that is in the private key. Each component of the key is described in the following table, which is quoted from this [MSDN article]() on the **`RSAParameters`** object.

Table 3: RSAParameters Fields

RSAParameters field	Contains	Corresponding PKCS #1 field
D	d, the private exponent	privateExponent
DP	d mod (p - 1)	exponent1
DQ	d mod (q - 1)	exponent2
Exponent	e, the public exponent	publicExponent
InverseQ	(InverseQ)(q) = 1 mod p	coefficient
Modulus	n	modulus
P	p	prime1
Q	q	prime2

With the key data stored in memory, you can save them to a binary file or do with them as you see fit. Remember that the private key is the key you need to keep secret and the public key is the key you give out to people who want to send encrypted data to you with the RSA algorithm.

The second method is the **`EncryptData`** method. This method takes a byte array of the data you wish to encrypt. The **`RSACryptoServiceProvider`** object is then created and the size of the key is passed in. In this case, we are using a 2,048-bit key which is the recommended minimum key length to use at the moment.

Once this object has been created, the **PersistKeyInCsp** flag is set to false. This flag is set to true when you want to persist your key in a key container. The **PersistKeyInCsp** property is automatically set to true when you specify a key container name in a **CspParameters** object and use it to initialize an **RSACryptoServiceProvider** object. You can also specify a container name by using the **KeyContainerName** field. If you set the **PersistKeyInCsp** property to true without initializing the **RSACryptoServiceProvider** object with a **CspParameters** object, a random key container name prepended with "CLR" is created. In this example, we are not using a key container.

Next, the public key is imported into the **RSACryptoServiceProvider** instance. At this point, all you need to do to encrypt some data is pass in your raw byte array of data. The second parameter is a Boolean to specify Optimal Asymmetric Encryption Padding (OAEP). OAEP is a scheme that improves the resistance against chosen ciphertext attacks.

Note: A chosen ciphertext attack is a type of attack where a known ciphertext and its decrypted form can be analyzed to try and determine the secret key used for the original encryption. The more ciphertexts and their decrypted plaintexts the attacker has, the greater chance they have of determining the original key used.

Once the **Encrypt** method has finished, a byte array containing your RSA-encrypted data will be returned.

The **Decrypt** method is pretty much the same as the **Encrypt** method. The only difference between the two is that you call **Decrypt** instead of **Encrypt** and the resulting byte array will contain your decrypted data.

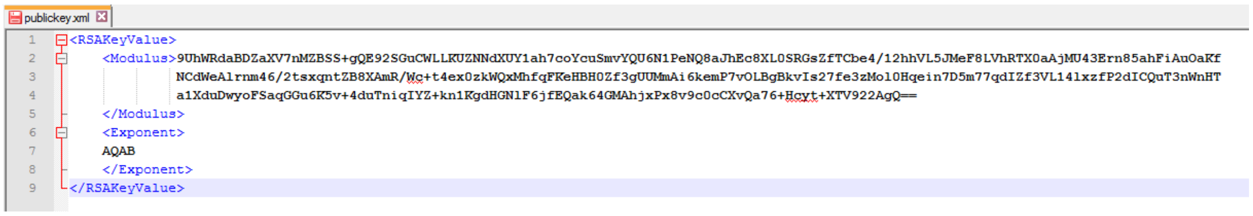
Example with XML-based Keys

In the following example, we have a class called **RSAWithXMLKey**. This class is a self-contained implementation that will allow you to generate a public/private key pair and use those keys to encrypt and decrypt some data.

```
public class RsaWithXMLKey
{
    public void AssignNewKey(string publicKeyPath, string privateKeyPath)
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;

            if (File.Exists(privateKeyPath))
            {
                File.Delete(privateKeyPath);
            }

            if (File.Exists(publicKeyPath))
            {
                File.Delete(publicKeyPath);
            }

            var publicKeyfolder = Path.GetDirectoryName(publicKeyPath);
            var privateKeyfolder = Path.GetDirectoryName(privateKeyPath);

            if (!Directory.Exists(publicKeyfolder))
            {
                Directory.CreateDirectory(publicKeyfolder);
            }

            if (!Directory.Exists(privateKeyfolder))
            {
                Directory.CreateDirectory(privateKeyfolder);
            }

            var publicKey = rsa.ToXmlString(false);
            File.WriteAllText(publicKeyPath, publicKey);
            File.WriteAllText(privateKeyPath, rsa.ToXmlString(true));
        }
    }

    public byte[] EncryptData(string publicKeyPath, byte[] dataToEncrypt)
    {
        byte[] cipherbytes;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            rsa.FromXmlString(File.ReadAllText(publicKeyPath));

            cipherbytes = rsa.Encrypt(dataToEncrypt, false);
        }

        return cipherbytes;
    }

    public byte[] DecryptData(string privateKeyPath, byte[] dataToEncrypt)
    {
        byte[] plain;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            rsa.FromXmlString(File.ReadAllText(privateKeyPath));
            plain = rsa.Decrypt(dataToEncrypt, false);
        }

        return plain;
    }
}
```

Code Listing 20

The first method is again **`AssignNewKey()`**. This method will generate both a public and a private key and store them in two XML files, **c:\temp\publicKey.xml** and **c:\temp\privateKey.xml**. These two files represent the raw key data expressed as byte arrays.



```xml
<RSAKeyValue>
    <Modulus>9UhWRdaBDZaXV7nMZBSS+gQE92SGuCWLLKUZNNdXUY1ah7coYcuSmvYQU6N1PeNQ8aJhEc8XL0SRGsZfTCbe4/12hhVL5JMeF8LVhRTX0aAjMU43Ern85ahFiAuOaKf
            NCdWeAlrnm46/2tsxqntZB8XAmR/Wc+t4ex0zkWQxMhfqFKeHBH0Zf3gUUMmAi6kemP7vOLBgBkvIs27fe3zMol0Hqein7D5m77qdIZf3VL14lxzfP2dICQuT3nWnHT
            a1XduDwyoFSaqGGu6K5v+4duTniqIYZ+kn1KgdHGNlF6jfEQak64GMAhjxPx8v9c0cCXvQa76+Hcyt+XTV922AgQ==
    </Modulus>
    <Exponent>
    AQAB
    </Exponent>
</RSAKeyValue>
```

Figure 22: XML representation of the RSA public key

The second method is the **`EncryptData`** method. This method takes a byte array containing the data you wish to encrypt. Then, the **`RSACryptoServiceProvider`** object is created and the size of the key is passed in. In this case, we are using a 2,048-bit key which is the recommended minimum key length to use currently.

Once this object has been created, the **`PersistKeyInCsp`** flag is set to false. This flag is set to true when you want to persist your key in a key container. The **`PersistKeyInCsp`** property is automatically set to true when you specify a key container name in a **`CspParameters`** object and use it to initialize an **`RSACryptoServiceProvider`** object. You can also specify a container name by using the **`KeyContainerName`** field. If you set the **`PersistKeyInCsp`** property to true without initializing the **`RSACryptoServiceProvider`** object with a **`CspParameters`** object, a random key container name prepended with "CLR" is created. In this example, we are not using a key container.

Next, the public key is imported into the **`RSACryptoServiceProvider`** instance. Then, all you need to do to encrypt some data is pass in your raw byte array of data. The second parameter is a Boolean to specify OAEP padding.

Once the **`Encrypt`** method has finished, you will be returned a byte array containing your RSA-encrypted data.

Example with Key Container-based Keys

In the following example, we have a class called **`RSAWithCSPKey`**. This class is a self-contained implementation that will allow you to generate a public/private key pair and use those keys to encrypt and decrypt some data.

```csharp
public class RSAWithCSPKey
{
    const string CONTAINER_NAME = "MyContainer";

    public void AssignNewKey()
    {
        const int PROVIDER_RSA_FULL = 1;

        CspParameters cspParams = new CspParameters(PROVIDER_RSA_FULL);
        cspParams.KeyContainerName = CONTAINER_NAME;
        cspParams.Flags = CspProviderFlags.UseMachineKeyStore;
        cspParams.ProviderName = "Microsoft Strong Cryptographic Provider";
        var rsa = new RSACryptoServiceProvider(cspParams);
        rsa.PersistKeyInCsp = true;
    }

    public void DeleteKeyInCSP()
    {
        var cspParams = new CspParameters();
        cspParams.KeyContainerName = CONTAINER_NAME;
        var rsa = new RSACryptoServiceProvider(cspParams);
        rsa.PersistKeyInCsp = false;
        rsa.Clear();
    }

    public byte[] EncryptData(byte[] dataToEncrypt)
    {
        byte[] cipherbytes;

        var cspParams = new CspParameters();
        cspParams.KeyContainerName = CONTAINER_NAME;

        using (var rsa = new RSACryptoServiceProvider(2048, cspParams))
        {
            cipherbytes = rsa.Encrypt(dataToEncrypt, false);
        }

        return cipherbytes;
    }

    public byte[] DecryptData(byte[] dataToDecrypt)
    {
        byte[] plain;

        var cspParams = new CspParameters();
        cspParams.KeyContainerName = CONTAINER_NAME;

        using (var rsa = new RSACryptoServiceProvider(2048, cspParams))
        {
            plain = rsa.Decrypt(dataToDecrypt, false);
        }

        return plain;
    }
}
```

Code Listing 21

The **`AssignNewKey`** method will create a new set of keys and store them in a key container provided by the cryptographic service provider. A Cryptographic Service Provider (CSP) is a software library that implements the Microsoft CryptoAPI. CSPs implement encoding and decoding functions.

CSPs are independent modules that can be used by different applications. A program calls CryptoAPI functions and these are redirected to the CSPs functions. Since CSPs are responsible for implementing cryptographic algorithms and standards, applications do not need to be concerned about security details. All cryptographic activity is implemented in CSPs; the CryptoAPI only works as a bridge between the application and the CSP.

When setting up the **`CspParameters`** object, you must provide the following pieces of information.

Table 4: Required CspParameters Properties

Property	Description
Key Container Name	Specify a name for your key container. You can use the container name to retrieve the persisted key within that container.
Flags	Represents the flags for **`CspParameters`** that modify the behavior of the CSP. See Table 5.
Provider Name	Represents the provider name for **`CspParameters`**. The “Microsoft Strong Cryptographic Provider” is used as the default RSA Full CSP. It supports all of the algorithms of the Microsoft Enhanced Cryptographic Provider and all of the same key lengths.

The flags that are set in this example modify the behavior of the CSP. The flag set in this example, **`UseMachineKeyStore`**, instructs the CSP to store the flags in the computer’s key store. The other flags you can use are as follows.

Table 5: CSP Flags

Flag	Description
CreateEphemeralKey	Create a temporary key that is released when the associated RSA object is closed. Do not use this flag if you want your key to be independent of the RSA object.
NoFlags	Do not specify any settings.
NoPrompt	Prevent the CSP from displaying any user interface for this context.

Flag	Description
UseArchivableKey	Allow a key to be exported for archival or recovery.
UseDefaultKeyContainer	Use key information from the default key container.
UseExistingKey	Use key information from the current key.
UseMachineKeyStore	Use key information from the computer's key store.
UseNonExportableKey	Use key information that cannot be exported.
UseUserProtectedKey	Notify the user through a dialog box or another method when certain actions are attempting to use a key. This flag is not compatible with the NoPrompt flag.

Once all the necessary parameters are set on the **`CspParameters`** object, an instance of the **`RSACryptoServiceProvider`** is created and the **`PersistKeyInCsp`** flag is set to true to persist the key.

The next method in this example is **`DeleteKeyInCSP()`**. This method calls up the container by name and passes it into the constructor for the **`RSACryptoServiceProvider`** class. Once that object has been instanced, the **`PersistKeyInCsp`** flag is then set to false and the **`Clear()`** method is called. This will remove the key.

The **`Encrypt`** and **`Decrypt`** methods are similar to previous versions except that the key is first retrieved and put into a **`CspParameters`** object before being passed into the **`RSACryptoServiceParameters`** object.

Example Usage

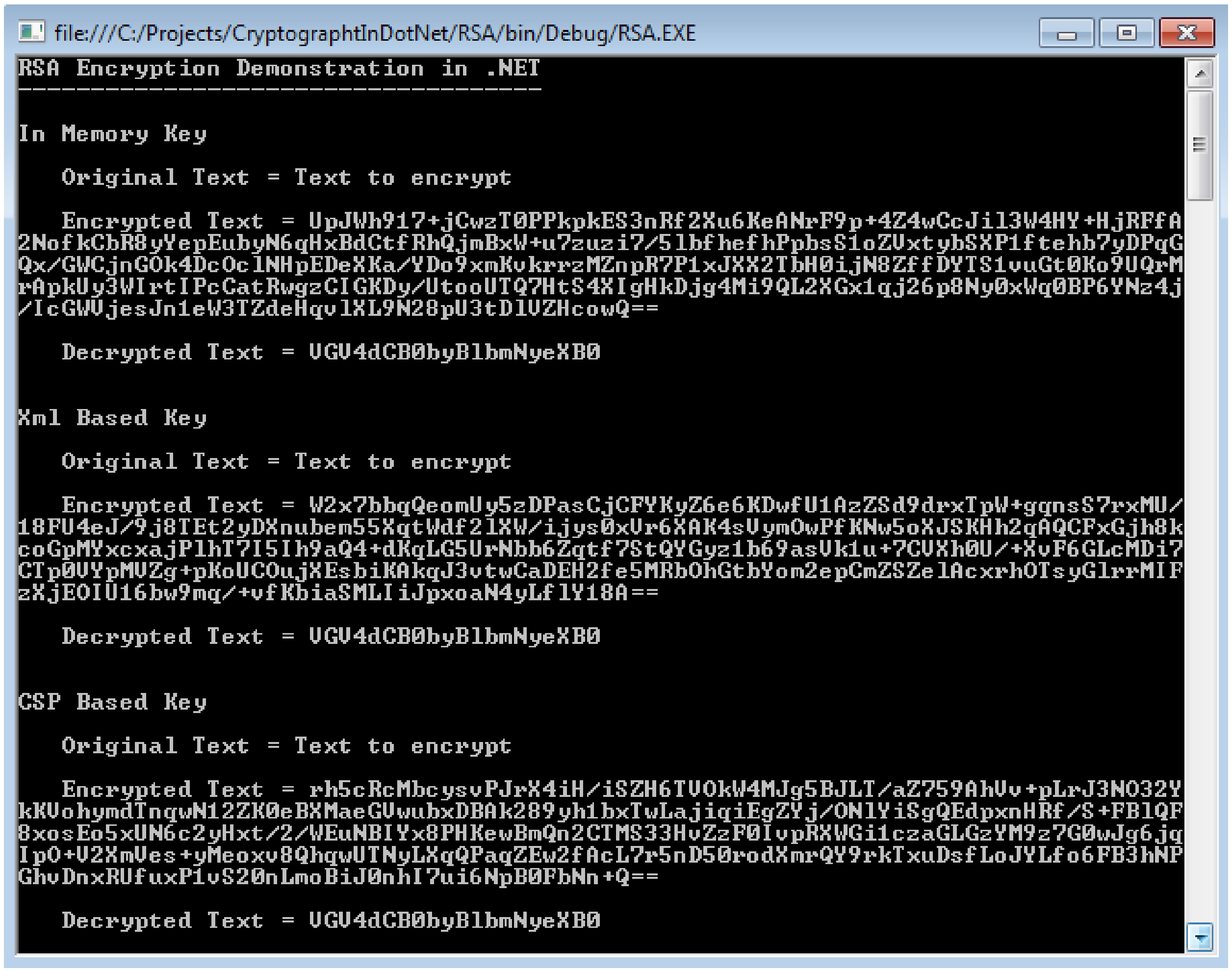
Now that we have looked at three different configurations for using the **`RSACryptoServiceProvider`**, the following code shows an example of these sample classes being used.

```
class Program
{
    static void Main(string[] args)
    {
        var rsa = new RsaWithXMLKey();
        var rsaCsp = new RSAWithCSPKey();
        var rsaParams = new RSAWithRSAParameterKey();

        const string original = "Text to encrypt";
        const string publicKeyPath = "c:\\temp\\publickey.xml";
        const string privateKeyPath = "c:\\temp\\privatekey.xml";

        rsa.AssignNewKey(publicKeyPath, privateKeyPath);
        var encrypted = rsa.EncryptData(publicKeyPath,
Encoding.UTF8.GetBytes(original));
        var decrypted = rsa.DecryptData(privateKeyPath, encrypted);

        rsaCsp.AssignNewKey();
        var encryptedCSP = rsaCsp.EncryptData(Encoding.UTF8.GetBytes(original));
        var decryptedCSP = rsaCsp.DecryptData(encryptedCSP);
        rsaCsp.DeleteKeyInCSP();

        rsaParams.AssignNewKey();
        var encryptedRsaParams =
rsaParams.EncryptData(Encoding.UTF8.GetBytes(original));
        var decryptedRsaParams = rsaParams.DecryptData(encryptedRsaParams);


        Console.WriteLine("RSA Encryption Demonstration in .NET");
        Console.WriteLine("------------------------------------");
        Console.WriteLine();
        Console.WriteLine("In Memory Key");
        Console.WriteLine();
        Console.WriteLine("   Original Text = " + original);
        Console.WriteLine();
        Console.WriteLine("   Encrypted Text = " +
Convert.ToBase64String(encryptedRsaParams));
        Console.WriteLine();
        Console.WriteLine("   Decrypted Text = " +
Convert.ToBase64String(decryptedRsaParams));
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("Xml Based Key");
        Console.WriteLine();
        Console.WriteLine("   Original Text = " + original);
        Console.WriteLine();
        Console.WriteLine("   Encrypted Text = " +
Convert.ToBase64String(encrypted));
        Console.WriteLine();
        Console.WriteLine("   Decrypted Text = " +
Convert.ToBase64String(decrypted));
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("CSP Based Key");
        Console.WriteLine();
        Console.WriteLine("   Original Text = " + original);
        Console.WriteLine();
        Console.WriteLine("   Encrypted Text = " +
Convert.ToBase64String(encryptedCSP));
```

```
        Console.WriteLine();
        Console.WriteLine("   Decrypted Text = " +
Convert.ToBase64String(decryptedCSP));
        Console.ReadLine();
    }
}
```

Code Listing 22

The examples are each self-contained and share the same structure. The RSA parameter and CSP keys examples store their keys internally, and the XML key example stores its keys in XML files. The examples are easy to adapt to whichever key storage you need, whether that is a hardware security module or a database.



file:///C:/Projects/CryptographtInDotNet/RSA/bin/Debug/RSA.EXE

```
RSA Encryption Demonstration in .NET
------------------------------------

In Memory Key

   Original Text = Text to encrypt

   Encrypted Text = UpJWh917+jCwzT0PPkpkES3nRf2Xu6KeANrF9p+4Z4wCcJil3W4HY+HjRFfA
2NofkCbR8yYepEubyN6qHxBdCtfRhQjmBxW+u7zuzi7/5lbfhefhPpbsS1oZUxtybSXP1ftehb7yDPqG
Qx/GWCjnGOk4DcOclNHpEDeXKa/YDo9xmKvkrrzMZnpR7P1xJXX2TbH0ijN8ZffDYTS1vuGt0Ko9UQrM
rApkUy3WIrtIPcCatRwgzCIGKDy/UtooUTQ7HtS4XIgHkDjg4Mi9QL2XGx1qj26p8Ny0xWq0BP6YNz4j
/IcGWUjesJn1eW3TZdeHqvlXL9N28pU3tDlUZHcowQ==

   Decrypted Text = VGV4dCB0byBlbmNyeXB0


Xml Based Key

   Original Text = Text to encrypt

   Encrypted Text = W2x7bbqQeomUy5zDPasCjCFYKyZ6e6KDwfU1AzZSd9drxTpW+gqnsS7rxMU/
18FU4eJ/9j8TEt2yDXnubem55XqtWdf2lXW/ijys0xVr6XAK4sVymOwPfKNw5oXJSKHh2qAQCFxGjh8k
coGpMYxcxajPlhT7I5Ih9aQ4+dKqLG5UrNbb6Zqtf7StQYGyz1b69asVk1u+7CVXh0U/+XvF6GLcMDi7
CTp0VYpMVZg+pKoUCOujXEsbiKAkqJ3vtwCaDEH2fe5MRbOhGtbYom2epCmZSZelAcxrhOTsyGlrrMIF
zXjEOIU16bw9mq/+vfKbiaSMLIiJpxoaN4yLflY18A==

   Decrypted Text = VGV4dCB0byBlbmNyeXB0


CSP Based Key

   Original Text = Text to encrypt

   Encrypted Text = rh5cRcMbcysvPJrX4iH/iSZH6TVOkW4MJg5BJLT/aZ759AhVv+pLrJ3NO32Y
kKVohymdTnqwN12ZK0eBXMaeGVwubxDBAk289yh1bxTwLajiqiEgZYj/ONlYiSgQEdpxnHRf/S+FBlQF
8xosEo5xUN6c2yHxt/2/WEuNBIYx8PHKewBmQn2CTMS33HvZzF0IvpRXWGi1czaGLGzYM9z7G0wJg6jq
IpO+V2XmVes+yMeoxv8QhqwUTNyLXqQPaqZEw2fAcL7r5nD50rodXmrQY9rkTxuDsfLoJYLfo6FB3hNP
GhvDnxRUfuxP1vS20nLmoBiJ0nhI7ui6NpB0FbNn+Q==

   Decrypted Text = VGV4dCB0byBlbmNyeXB0
```

Figure 23: Example of RSA usage

Chapter 7 Hybrid Approach (RSA+AES)

In the previous chapter, it was stated that with RSA you cannot encrypt data that is larger than the length of the asymmetric key. So what do you do if you need to encrypt a larger block of data? Ideally, you would want to use a symmetric encryption algorithm such as AES, but the problem you face is that of secure and reliable key exchange.

With AES, each recipient has to have the same key. This all sounds easy in principle but how do you get that same key securely to each person? You can't just easily send the key to them.

So what you want to do is employ what is called a hybrid encryption system. This is where you use a combination of both RSA and AES.

Let's assume we have two people involved: the sender, Alice, and the receiver, Bob.

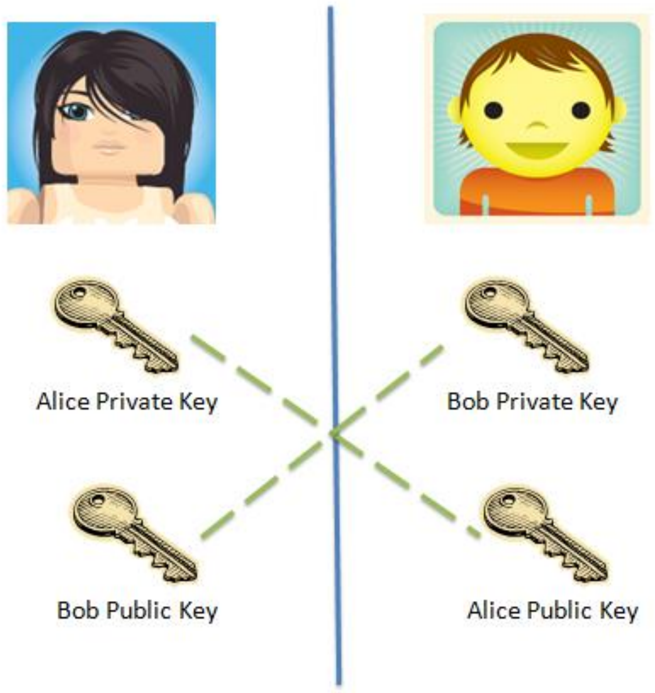


Figure 24: Alice and Bob's asymmetric keys

The process would look like the following for Alice sending data to Bob:

Encryption Using the Hybrid Approach

1. **Alice** generates a 256-bit (32-byte) AES Key. This key is called a session key in this process.
2. **Alice** generates a 128-bit (16-byte) IV.
3. **Alice** encrypts the data with AES using the session key and the IV.
4. **Alice** encrypts the session key with RSA and **Bob's** public key.

5. **Alice** stores the encrypted data, encrypted AES session key, and IV in a separate structure or file. This is the packet of data that is sent to **Bob**.

Decryption Using the Hybrid Approach

1. **Bob** decrypts the encrypted AES session key by using RSA and **Bob's** private key.
2. **Bob** decrypts the encrypted data by using the decrypted AES session key and the IV.
3. **Bob** reads the decrypted message.

Now, let's look at the same process again but this time, Bob is sending a reply to Alice where Bob uses Alice's public key and Alice uses her private key.

Encryption Using the Hybrid Approach

1. **Bob** generates a 256-bit (32-byte) AES Key. This key is called a session key in this process.
2. **Bob** generates a 128-bit (16-byte) IV.
3. **Bob** encrypts the data with AES by using the session key and the IV.
4. **Bob** encrypts the session key with RSA and **Alice's** public key.
5. **Bob** stores the encrypted data, encrypted AES session key, and IV in a separate structure or file. This is the packet of data that is sent to **Alice**.

Decryption Using the Hybrid Approach

1. **Alice** decrypts the encrypted AES session key by using RSA and **Alice's** private key.
2. **Alice** decrypts the encrypted data by using decrypted AES session key and the IV.
3. **Alice** reads the decrypted message.

Hybrid Encryption Implementation

To demonstrate this hybrid encryption approach, we will use the same **`AesEncryption`** class from the earlier chapter on symmetric encryption. In the **`AesEncryption`** class, there is a new method added called **`GenerateRandomNumber`** which will be used to generate our session key. For this example, we will use AES with a 256-bit (32-byte) key and we will use the algorithm in the default cipher block chaining mode with PKCS7 padding.

```
public class AesEncryption
{
    public byte[] GenerateRandomNumber(int length)
    {
        using (var randomNumberGenerator = new RNGCryptoServiceProvider())
        {
            var randomNumber = new byte[length];
            randomNumberGenerator.GetBytes(randomNumber);

            return randomNumber;
        }
    }

    public byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] iv)
    {
        using (var aes = new AesCryptoServiceProvider())
        {
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            aes.Key = key;
            aes.IV = iv;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
aes.CreateEncryptor(), CryptoStreamMode.Write);
                cryptoStream.Write(dataToEncrypt, 0, dataToEncrypt.Length);
                cryptoStream.FlushFinalBlock();

                return memoryStream.ToArray();
            }
        }
    }

    public byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] IV)
    {
        using (var aes = new AesCryptoServiceProvider())
        {
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;

            aes.Key = key;
            aes.IV = IV;

            using (var memoryStream = new MemoryStream())
            {
                var cryptoStream = new CryptoStream(memoryStream,
aes.CreateDecryptor(), CryptoStreamMode.Write);

                cryptoStream.Write(dataToDecrypt, 0, dataToDecrypt.Length);
                cryptoStream.FlushFinalBlock();

                var decryptBytes = memoryStream.ToArray();

                return decryptBytes;
            }
        }
    }
}
```

Code Listing 23

For the asymmetric encryption part of the hybrid encryption technique, we will use the same **`RSAWithRSAParameterKey`** class from our demonstration of RSA encryption. This class will store the generated RSA public and private keys as private members of the class for simplicity in this example. In a normal usage scenario, you would want to securely store your private key on a server or in a database.

```
public class RSAWithRSAParameterKey
{
    private RSAParameters publicKey;
    private RSAParameters privateKey;

    public void AssignNewKey()
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            publicKey = rsa.ExportParameters(false);
            privateKey = rsa.ExportParameters(true);
        }
    }

    public byte[] EncryptData(byte[] dataToEncrypt)
    {
        byte[] cipherbytes;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;
            rsa.ImportParameters(publicKey);


            cipherbytes = rsa.Encrypt(dataToEncrypt, false);
        }

        return cipherbytes;
    }

    public byte[] DecryptData(byte[] dataToEncrypt)
    {
        byte[] plain;

        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            rsa.PersistKeyInCsp = false;

            rsa.ImportParameters(privateKey);
            plain = rsa.Decrypt(dataToEncrypt, false);
        }

        return plain;
    }
}
```

Code Listing 24

First of all, we need a class to store our encrypted data packet. There are three parts to the packet that you will have once you have done the encryption. They are:

- **Encrypted Session Key**: This is the 256-bit AES session key that is generated and then encrypted with the RSA private key.
- **Encrypted Data**: This is our actual data that has been encrypted with the AES 256-bit session key.
- **Initialization Vector (IV)**: This is the 128-bit (16-byte) IV that is passed into the AES encryption algorithm. This does not need to be kept secret and can be stored inside our data packet.

```
public class EncryptedPacket
{
    public byte[] EncryptedSessionKey;
    public byte[] EncryptedData;
    public byte[] Iv;
}
```

Code Listing 25

The encryption process is very straightforward as illustrated in the following code example. The **`EncryptData`** method takes a string which, in this case, is our data to encrypt and the **`RSAWithRSAParameterKey`** object which contains our RSA keys.

This method first creates an instance of the **`AesEncryption`** object and generates the 256-bit (32-byte) session key. Then, the 128-bit (16-byte) IV is generated and stored in the encrypted packet object. Next, the data we want to be encrypted is encrypted with AES by using the generated session key and IV. The result is also stored in the encrypted packet object. Finally, the generated AES session key is encrypted with the RSA public key. This encrypted session key is then stored in the encrypted packet.

This means our data is now encrypted using AES, but the key is protected using the RSA key pair.

```
public byte[] Encrypt(byte[] dataToEncrypt, byte[] key, byte[] iv)
{
    using (var aes = new AesCryptoServiceProvider())
    {
        aes.Mode = CipherMode.CBC;
        aes.Padding = PaddingMode.PKCS7;

        aes.Key = key;
        aes.IV = iv;

        using (var memoryStream = new MemoryStream())
        {
            var cryptoStream = new CryptoStream(memoryStream,
aes.CreateEncryptor(), CryptoStreamMode.Write);
            cryptoStream.Write(dataToEncrypt, 0, dataToEncrypt.Length);
            cryptoStream.FlushFinalBlock();
            return memoryStream.ToArray();
        }
    }
}
```

Code Listing 26

The decryption process is just as straightforward. There is a method called **`DecryptData`** that takes the encrypted packet and the RSA encryption object that contains our public and private encryption keys. First, the encrypted AES session key is decrypted using the **`RSAWithRSAParameterKey`** object. This decrypts using the private key. Once the session key has been decrypted, the encrypted data is then decrypted with AES by using the session key and initialization vector. Then, the decrypted data is turned back into a string and returned to the caller.

```
public byte[] Decrypt(byte[] dataToDecrypt, byte[] key, byte[] IV)
{
    using (var aes = new AesCryptoServiceProvider())
    {
        aes.Mode = CipherMode.CBC;
        aes.Padding = PaddingMode.PKCS7;

        aes.Key = key;
        aes.IV = IV;

        using (var memoryStream = new MemoryStream())
        {
            var cryptoStream = new CryptoStream(memoryStream,
aes.CreateDecryptor(), CryptoStreamMode.Write);

            cryptoStream.Write(dataToDecrypt, 0, dataToDecrypt.Length);
            cryptoStream.FlushFinalBlock();

            var decryptBytes = memoryStream.ToArray();

            return decryptBytes;
        }
    }
}
```

Code Listing 27

This example shows how to get the best of both RSA and AES: You get the benefits of RSA's asymmetric keys (which makes key exchange much easier) and you then get the benefit of AES for securely encrypting your data with the RSA-protected session key.

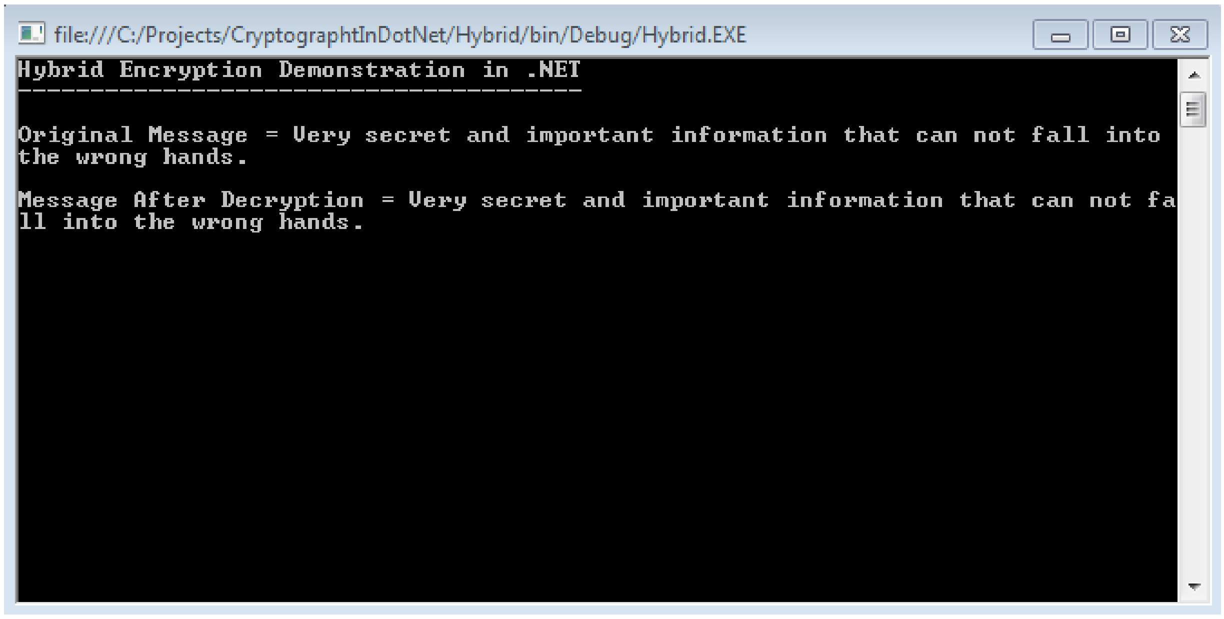
The following code listing shows the entire example for hybrid encryption.

```
public class EncryptedPacket
{
    public byte[] EncryptedSessionKey;
    public byte[] EncryptedData;
    public byte[] Iv;
}

class Program
{
    static void Main(string[] args)
    {
        const string original = "Very secret and important information that can
not fall into the wrong hands.";

        var rsaParams = new RSAWithRSAParameterKey();
        rsaParams.AssignNewKey();

        var encryptedBlock = EncryptData(original, rsaParams);
        var decrpyted = DecryptData(encryptedBlock, rsaParams);

        Console.WriteLine("Hybrid Encryption Demonstration in .NET");
        Console.WriteLine("---------------------------------------");
        Console.WriteLine();
        Console.WriteLine("Original Message = " + original);
        Console.WriteLine();
        Console.WriteLine("Message After Decryption = " + decrpyted);
        Console.ReadLine();
    }

    private static string DecryptData(EncryptedPacket encryptedPacket,
RSAWithRSAParameterKey rsaParams)
    {
        var aes = new AesEncryption();

        // Decrypt AES key with RSA and then decrypt data with AES.
        var decryptedSessionKey =
rsaParams.DecryptData(encryptedPacket.EncryptedSessionKey);
        var decryptedData = aes.Decrypt(encryptedPacket.EncryptedData,
decryptedSessionKey, encryptedPacket.Iv);
        return Encoding.UTF8.GetString(decryptedData);
    }

    private static EncryptedPacket EncryptData(string original,
RSAWithRSAParameterKey rsaParams)
    {
        var aes = new AesEncryption();

        var sessionKey = aes.GenerateRandomNumber(32);
        var encryptedPacket = new EncryptedPacket {Iv =
aes.GenerateRandomNumber(16)};

        // Encrypt data with AES and AES key with RSA.
        encryptedPacket.EncryptedData =
aes.Encrypt(Encoding.UTF8.GetBytes(original), sessionKey, encryptedPacket.Iv);
        encryptedPacket.EncryptedSessionKey = rsaParams.EncryptData(sessionKey);
```

```
            return encryptedPacket;
        }
    }
```

Code Listing 28

The following screenshot shows the hybrid encryption example working. Here, you see the original message that is to be encrypted and the message after it has been encrypted and then decrypted.


file:///C:/Projects/CryptographtInDotNet/Hybrid/bin/Debug/Hybrid.EXE

```
Hybrid Encryption Demonstration in .NET
---------------------------------------

Original Message = Very secret and important information that can not fall into
the wrong hands.

Message After Decryption = Very secret and important information that can not fa
ll into the wrong hands.
```

Figure 25: Hybrid encryption example

Hybrid Encryption with Added Integrity

In this section, we will build on the previous hybrid encryption example by adding message integrity. This will mean that, once the encrypted data block has been created, you can check whether or not it has been tampered with or corrupted while in transit to its recipient.

Before looking at this extra layer of integrity checking, it is important to go over byte array comparisons.

Comparing Byte Arrays

When dealing with byte arrays, it is common to want to check if one array is the same as another. Typically, you might have an implementation like the following.

```
private static bool CompareUnSecure(byte[] array1, byte[] array2)
{
    if (array1.Length != array2.Length)
    {
        return false;
    }

    for (int i = 0; i < array1.Length; ++i)
    {
        if (array1[i] != array2[i])
        {
            return false ;
        }
    }

    return true;
}
```

Code Listing 29

In this example, the **`Compare`** method takes two byte arrays and returns true if they are equal and false otherwise. This works by first checking if the array lengths are the same. If they are not, the method returns false. It then iterates through the first array and checks each element of the array against the second array. If it finds an element that is not equal, it will return false.

On the surface of it, this sounds like a reasonable implementation and efficient, too, as it will abort as soon as it detects that the arrays are different. However, when dealing with cryptographic processes, this isn't such a good idea. The reason is a potential attacker can use this type of comparison as a vulnerability and perform what is called a side channel timing attack.

Note: A side channel timing attack is any attack that is based on information gained from the physical implementation of a cryptosystem rather than from brute force or theoretical weaknesses in the algorithms. For example, timing information, power consumption, electromagnetic leaks, or even sound can provide an extra source of information which can be exploited to break the system.

This is where an attacker can glean information from the method because it will execute with different timings based on the arrays being fed into it. Ideally, what you need is a **`Compare`** method that will execute in the same time no matter how equal (or not) the arrays are. A better implementation would be to use something like the following.

```
private static bool Compare(byte[] array1, byte[] array2)
{
    var result = array1.Length == array2.Length;

    for(var i = 0; i < array1.Length && i < array2.Length; ++i)
    {
        result &= array1[i] == array2[i];
    }

    return result;
}
```

Code Listing 30

This version of **`Compare`** has the exact same method signature as the previous example. First, there is a length equality check done against the length of the two byte arrays. This will set the result field to either **`True`** or **`False`**. Even if the result is **`False`**, the method will still continue to run. Next, the method will iterate through the arrays and check each element. If any element is not equal, then the result gets set to **`False`** and the method carries on checking. This means that no matter whether the arrays are equal or not, the method will take the same length of time to execute and, therefore, will not leak information to an attacker.

Adding Integrity to the Hybrid Encryption Example

The first thing you need to do to add integrity to the hybrid encryption example is add a new field to the **`EncryptedPacket`** class. This new field will be used to store an HMAC of the encrypted data.

```
public class EncryptedPacket
{
    public byte[] EncryptedSessionKey;
    public byte[] EncryptedData;
    public byte[] Iv;
    public byte[] Hmac;
}
```

Code Listing 31

As described in Chapter 3, an HMAC may be used to simultaneously verify both the data integrity and the authentication of a message by combining a one-way hash function with a secret cryptographic key.

With this in mind, our new **`EncryptData`** method looks like the following.

```
private static EncryptedPacket EncryptData(string original, RSAWithRSAParameterKey
rsaParams)
{
    var aes = new AesEncryption();

    var sessionKey = aes.GenerateRandomNumber(32);
    var encryptedPacket = new EncryptedPacket {Iv = aes.GenerateRandomNumber(16)};

    // Encrypt data with AES and AES key with RSA.
    encryptedPacket.EncryptedData = aes.Encrypt(Encoding.UTF8.GetBytes(original),
sessionKey, encryptedPacket.Iv);
```

```
    encryptedPacket.EncryptedSessionKey = rsaParams.EncryptData(sessionKey);

    using (var hmac = new HMACSHA256(sessionKey))
    {
        encryptedPacket.Hmac = hmac.ComputeHash(encryptedPacket.EncryptedData);
    }

    return encryptedPacket;
}
```

Code Listing 32

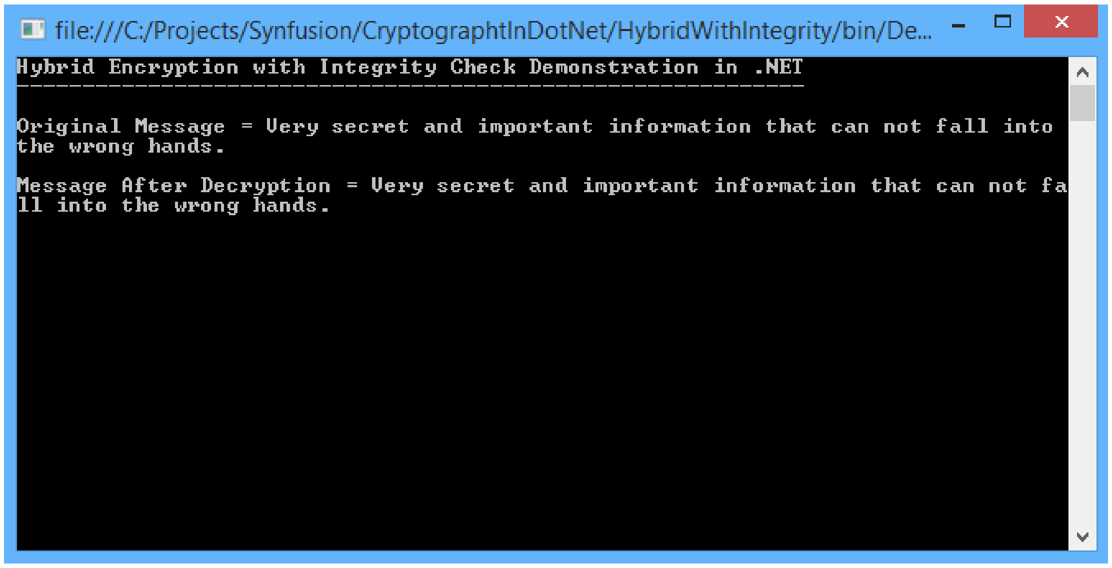
This version is the same as the previous version except for the HMAC calculated using the generated session key as input toward the bottom of the method. The HMAC is calculated against the data that was encrypted with the session key. It is important to use the unencrypted session key to calculate the HMAC because this means no one can regenerate the HMAC without having first successfully decrypted the session key with the RSA private key.

Now that **`EncryptData`** has been changed to support the generation of an HMAC for the encrypted packet, the **`DecryptData`** method needs to be changed to check the HMAC. This can be demonstrated with the following code.

```
private static string DecryptData(EncryptedPacket encryptedPacket,
RSAWithRSAParameterKey rsaParams)
{
    var aes = new AesEncryption();

    // Decrypt AES key with RSA and then decrypt data with AES.
    var decryptedSessionKey =
rsaParams.DecryptData(encryptedPacket.EncryptedSessionKey);

    using (var hmac = new HMACSHA256(decryptedSessionKey))
    {
        var hmacToCheck = hmac.ComputeHash(encryptedPacket.EncryptedData);

        if (!Compare(encryptedPacket.Hmac, hmacToCheck))
        {
            throw new CryptographicException("HMAC for decryption does not match
encrypted packet.");
        }
    }

    var decryptedData = aes.Decrypt(encryptedPacket.EncryptedData,
decryptedSessionKey, encryptedPacket.Iv);

    return Encoding.UTF8.GetString(decryptedData);
}
```

Code Listing 33

In this version of **`DecryptData`**, once the session key has been decrypted with the RSA private key, it is fed into the constructor of the **`HMACSHA256`** object. Next, an HMAC of the encrypted data in the packet is calculated, and then this newly calculated HMAC is compared to the HMAC in the original data packet using our more secure, time-independent **`Compare`** function. If the hashes match, then the encrypted data is decrypted with the session key. If the hashes do not match, then a **`CryptographicException`** is thrown.

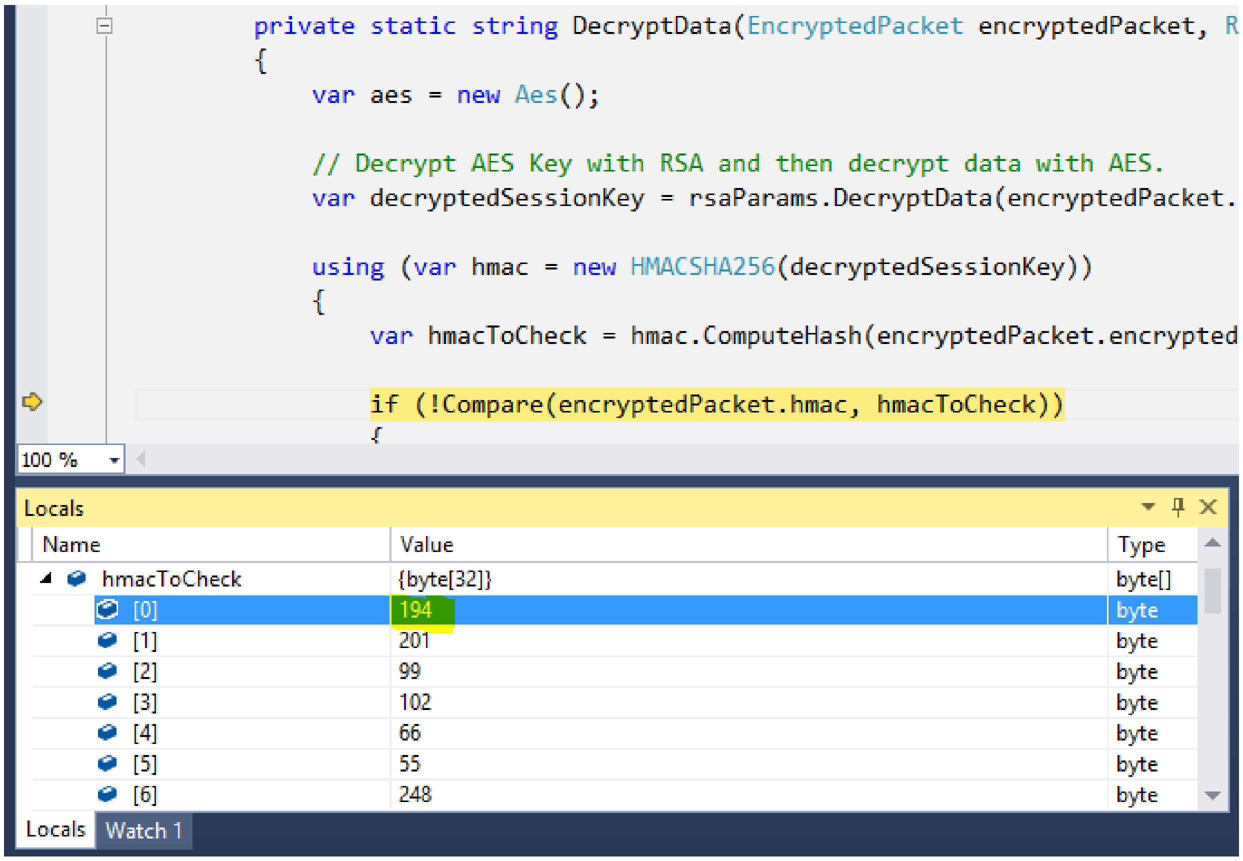


```
file:///C:/Projects/Synfusion/CryptographtInDotNet/HybridWithIntegrity/bin/De...
Hybrid Encryption with Integrity Check Demonstration in .NET
------------------------------------------------------------

Original Message = Very secret and important information that can not fall into
the wrong hands.

Message After Decryption = Very secret and important information that can not fa
ll into the wrong hands.
```

Figure 26: Hybrid encryption success example

In the preceding screenshot, you can see an example of the hybrid encryption app working where the message to be decrypted passes the integrity check. Let's now prove that the integrity check works.

In the following screenshot, there is a breakpoint placed on the comparison line. In the debugger, the **`hmacToCheck`** variable is modified to deliberately make it not match the HMAC in the **`encryptedPacket`**. This will make the **`Compare`** method return **`False`** and throw a **`CryptographicException`**.



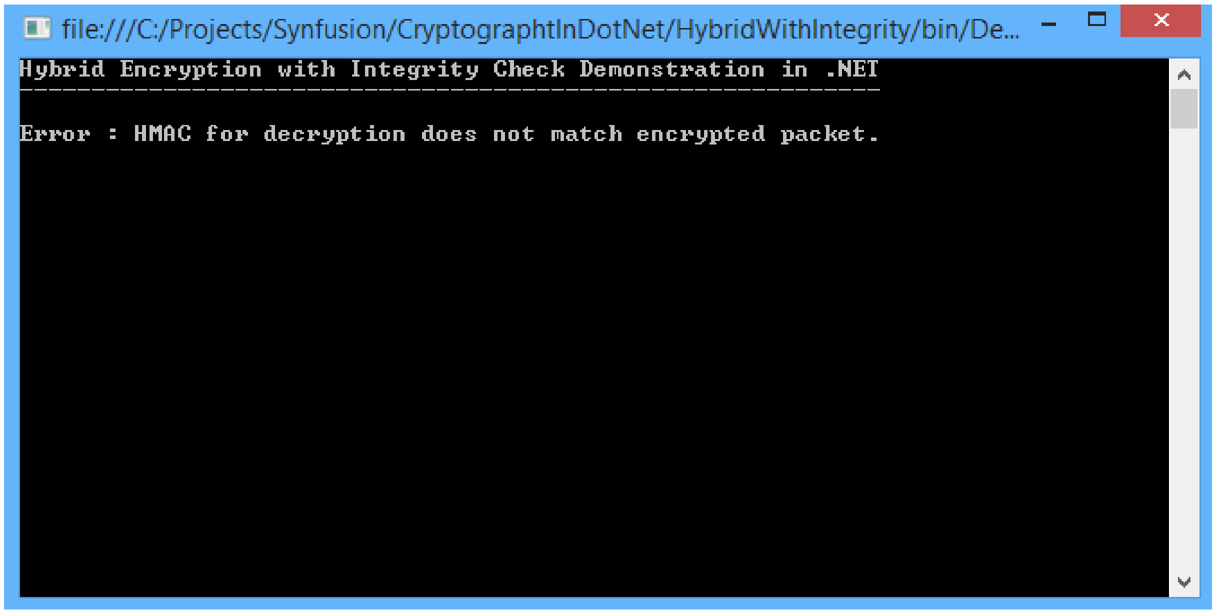
```
private static string DecryptData(EncryptedPacket encryptedPacket, R
{
    var aes = new Aes();

    // Decrypt AES Key with RSA and then decrypt data with AES.
    var decryptedSessionKey = rsaParams.DecryptData(encryptedPacket.

    using (var hmac = new HMACSHA256(decryptedSessionKey))
    {
        var hmacToCheck = hmac.ComputeHash(encryptedPacket.encrypted

        if (!Compare(encryptedPacket.hmac, hmacToCheck))
        {
```

Locals

Name	Value	Type
hmacToCheck	{byte[32]}	byte[]
[0]	194	byte
[1]	201	byte
[2]	99	byte
[3]	102	byte
[4]	66	byte
[5]	55	byte
[6]	248	byte

Figure 27: Use the debugger to break message integrity

By failing the integrity check, the application won't proceed to decrypt the data and will display the following error message in the console window.



file:///C:/Projects/Synfusion/CryptographtInDotNet/HybridWithIntegrity/bin/De...

```
Hybrid Encryption with Integrity Check Demonstration in .NET
------------------------------------------------------------

Error : HMAC for decryption does not match encrypted packet.
```

Figure 28: Hybrid encryption example that fails the integrity check

The following code listing provides the entire example for hybrid encryption with an integrity check.

```
public class EncryptedPacket
{
    public byte[] EncryptedSessionKey;
    public byte[] EncryptedData;
    public byte[] Iv;
    public byte[] Hmac;
}

class Program
{
    static void Main(string[] args)
    {
        const string original = "Very secret and important information that cannot 
fall into the wrong hands.";

        var rsaParams = new RSAWithRSAParameterKey();
        rsaParams.AssignNewKey();

        Console.WriteLine("Hybrid Encryption with Integrity Check Demonstration in 
.NET");
        Console.WriteLine("-------------------------------------------------------
-----");
        Console.WriteLine();

        try
        {
            var encryptedBlock = EncryptData(original, rsaParams);
            var decrpyted = DecryptData(encryptedBlock, rsaParams);

            Console.WriteLine("Original Message = " + original);
            Console.WriteLine();
            Console.WriteLine("Message After Decryption = " + decrpyted);
        }
        catch (CryptographicException ex)
        {
            Console.WriteLine("Error : " + ex.Message);
        }

        Console.ReadLine();
    }

    private static bool Compare(byte[] array1, byte[] array2)
    {
        var result = array1.Length == array2.Length;

        for(var i = 0; i < array1.Length && i < array2.Length; ++i)
        {
            result &= array1[i] == array2[i];
        }
        return result;
    }

    private static string DecryptData(EncryptedPacket encryptedPacket,
RSAWithRSAParameterKey rsaParams)
    {
        var aes = new AesEncryption();

        // Decrypt AES key with RSA and then decrypt data with AES.
        var decryptedSessionKey =
rsaParams.DecryptData(encryptedPacket.EncryptedSessionKey);

        using (var hmac = new HMACSHA256(decryptedSessionKey))
        {
            var hmacToCheck = hmac.ComputeHash(encryptedPacket.EncryptedData);

            if (!Compare(encryptedPacket.Hmac, hmacToCheck))
            {
                throw new CryptographicException("HMAC for decryption does not
match encrypted packet.");
            }
        }

        var decryptedData = aes.Decrypt(encryptedPacket.EncryptedData,
decryptedSessionKey, encryptedPacket.Iv);

        return Encoding.UTF8.GetString(decryptedData);
    }

    private static EncryptedPacket EncryptData(string original,
RSAWithRSAParameterKey rsaParams)
    {
        var aes = new AesEncryption();

        var sessionKey = aes.GenerateRandomNumber(32);
        var encryptedPacket = new EncryptedPacket {Iv =
aes.GenerateRandomNumber(16)};

        // Encrypt data with AES and AES key with RSA.
        encryptedPacket.EncryptedData =
aes.Encrypt(Encoding.UTF8.GetBytes(original), sessionKey, encryptedPacket.Iv);
        encryptedPacket.EncryptedSessionKey = rsaParams.EncryptData(sessionKey);

        using (var hmac = new HMACSHA256(sessionKey))
        {
            encryptedPacket.Hmac =
hmac.ComputeHash(encryptedPacket.EncryptedData);
        }

        return encryptedPacket;
    }
}
```

Chapter 8 Digital Signatures

An important function of cryptography is to ensure nonrepudiation of a sent message. This is where the receiver of the message cannot deny that the message is authentic. A digital signature is a technique used to help demonstrate this authenticity and the integrity of the message. Digital signatures are useful for demonstrating authenticity of documents and contracts, software downloads, and financial transactions.

Digital signatures are based on asymmetric cryptography. For the receiver of the message, a digital signature allows the receiver to believe the message was sent by the correct sender. This can be thought of as a digital equivalent to a signature on a letter, except a digital signature is much harder to forge.

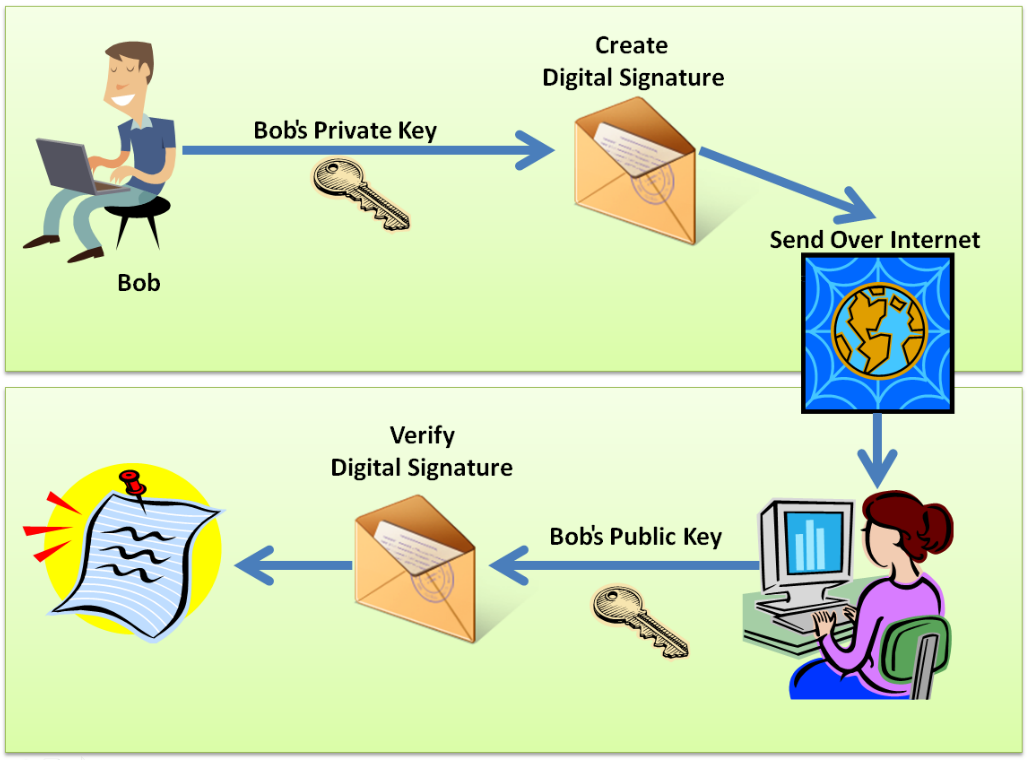


Figure 29: Example of digital signature flow

A digital signature consists of the following three algorithms:

- Public and private key generation using RSA.
- A signing algorithm that uses the private key to create the signature.
- A signature verification algorithm that uses the public key to test if the message is authentic.

A digital signature requires two properties. First, the signature is generated for a message using a private key and this signature can be verified using the associated public key. Second, you should not be able to generate a valid signature for a message without knowing the private key.

```
public void AssignNewKey()
{
    using (var rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.PersistKeyInCsp = false;
        publicKey = rsa.ExportParameters(false);
        privateKey = rsa.ExportParameters(true);
    }
}
```

Code Listing 34

For the digital signature example project, there is a class called **`DigitalSignature`**. The first method in this class is **`AssignNewKey`**. This method will generate a public and private key pair to be used for creating and verifying the digital signature. Next, there is the **`SignData`** method. This is the method that will create a digital signature. Let's say you want to sign some data that represents a document although what you are signing doesn't matter as it is represented as a byte array. You don't sign the actual data itself; you sign a hash of the data.

```
public byte[] SignData(byte[]hashOfDataToSign)
{
    using (var rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.PersistKeyInCsp = false;
        rsa.ImportParameters(privateKey);

        var rsaFormatter = new RSAPKCS1SignatureFormatter(rsa);
        rsaFormatter.SetHashAlgorithm("SHA256");

        return rsaFormatter.CreateSignature(hashOfDataToSign);
    }
}
```

Code Listing 35

The **`SignData`** method takes a byte array which is the hash of the data you wish to sign. The first thing that happens in this method is an instance of the **`RSACryptoServiceProvider`** class is created and then the already created private key (from the **`AssignNewKey`** method) is loaded into that instance. Then, an instance of **`RSAPKCS1SignatureFormatter`** is created and the instance of **`RSACryptoServiceProvider`** is passed in.

Next, the hash algorithm has to be set on the **`RSAPKCS1SignatureFormatter`** instance. The algorithm set here has to match the hash algorithm you used to hash your data before creating the digital signature. For example, if you hash your data with SHA-1, then the string passed into **`SetHashAlgorithm`** needs to be **`"SHA1"`**. If you used SHA-256 (which this example program does), then you need to pass **`"SHA256"`** into **`SetHashAlgorithm`**.

Note: If you have a mismatch between the hash algorithm used to hash your data and the algorithm set with the SetHashAlgorithm method, a CryptographicException will be thrown with the message Bad Hash.

Once this has been done, the last thing to do is call **`CreateSignature`** on the **`RSAPKCS1SignatureFormatter`** instance. This will return a byte array containing your digital signature.

The next method in the **`DigitalSignature`** class is the **`VerifySignature`** method. This method is used to verify that a digital signature is valid for a particular hash of the data you want to verify.

```
public bool VerifySignature(byte[]hashOfDataToSign, byte[] signature)
{
    using (var rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.ImportParameters(publicKey);

        var rsaDeformatter = new RSAPKCS1SignatureDeformatter(rsa);
        rsaDeformatter.SetHashAlgorithm("SHA256");

        return rsaDeformatter.VerifySignature(hashOfDataToSign, signature);
    }
}
```

Code Listing 36

The **`VerifySignature`** method takes two parameters, a byte array containing a hash of the data that the signature was originally created for, and a byte array of the digital signature itself. First, an instance of the **`RSACryptoServiceProvider`** class is created. Then, the public key that was generated with the **`AssignNewKey`** method is imported into the RSA instance.

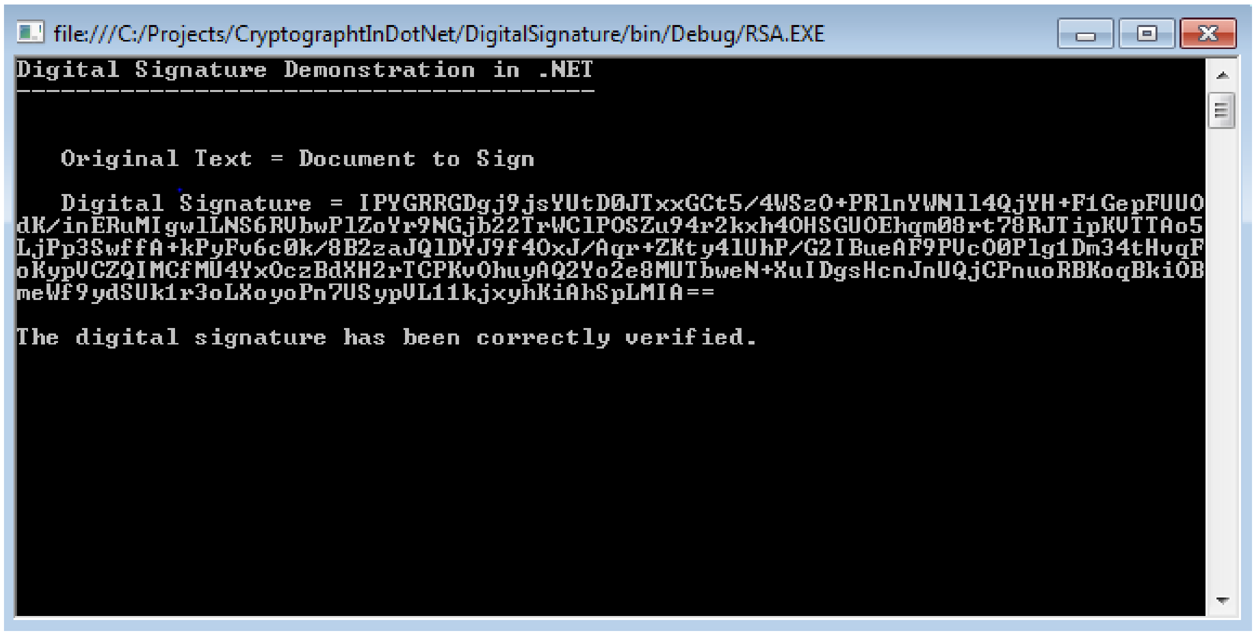
After that, an instance of the **`RSAPKCS1SignatureDeformatter`** is created and the hash algorithm is set to **`“SHA256”`**. Next, to verify the signature, you call **`VeryifySignature`** on the **`RSAPKCS1SignatureDeformatter`** instance by providing the hash of the data that was signed and the actual signature itself. If the signature is valid, **`True`** is returned. **`False`** is returned if the signature is not valid.

Use of this class is demonstrated in the following code sample.

```
class Program
{
    static void Main(string[] args)
    {
        var document = Encoding.UTF8.GetBytes("Document to Sign");
        byte[] hashedDocument;
        using (var sha256 = SHA256.Create())
        {
            hashedDocument = sha256.ComputeHash(document);
        }

        var digitalSignature = new DigitalSignature();
        digitalSignature.AssignNewKey();

        var signature = digitalSignature.SignData(hashedDocument);
        var verified = digitalSignature.VerifySignature(hashedDocument,
signature);

        Console.WriteLine("Digital Signature Demonstration in .NET");
        Console.WriteLine("---------------------------------------");
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("   Original Text = " +
System.Text.Encoding.Default.GetString(document));
        Console.WriteLine();
        Console.WriteLine("   Digital Signature = " +
Convert.ToBase64String(signature));
        Console.WriteLine();

        if (verified)
        {
            Console.WriteLine("The digital signature has been correctly
verified.");
        }
        else
        {
            Console.WriteLine("The digital signature has NOT been correctly
verified.");
        }

        Console.ReadLine();
    }
}
```

Code Listing 37

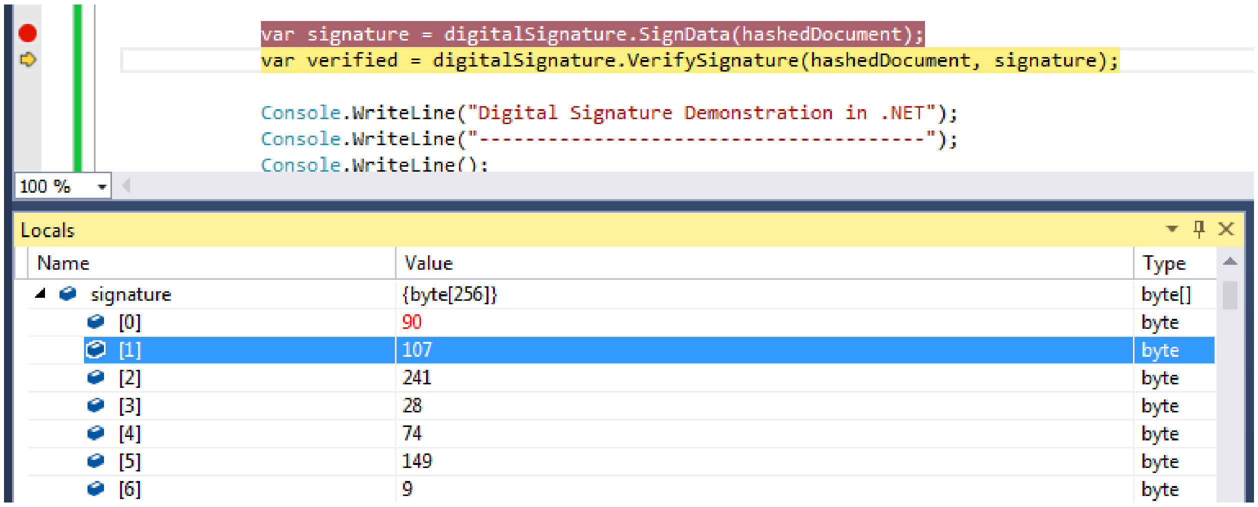
When this program is executed and a valid digital signature is verified, you will see the following output.


file:///C:/Projects/CryptographtInDotNet/DigitalSignature/bin/Debug/RSA.EXE

```
Digital Signature Demonstration in .NET
---------------------------------------


   Original Text = Document to Sign

   Digital Signature = IPYGRRGDgj9jsYUtD0JTxxGCt5/4WSzO+PRlnYWNll4QjYH+F1GepFUUO
dK/inERuMIgwlLNS6RVbwPlZoYr9NGjb22TrWClPOSZu94r2kxh4OHSGUOEhqm08rt78RJTipKUTTAo5
LjPp3SwffA+kPyFv6c0k/8B2zaJQlDYJ9f4OxJ/Aqr+ZKty41UhP/G2IBueAF9PVcO0Plg1Dm34tHvqF
oKypVCZQIMCfMU4YxOczBdXH2rTCPKvOhuyAQ2Yo2e8MUTbweN+XuIDgsHcnJnUQjCPnuoRBKoqBkiOB
meWf9ydSUk1r3oLXoyoPn7USypVL11kjxyhKiAhSpLMIA==

The digital signature has been correctly verified.
```

Figure 30: Digital signature verification

To prove that the **`VerifySignature`** method will return false for an invalid signature, you can deliberately tamper with the signature in the debugger by changing one of the values in the debugger as shown in the following screenshot.

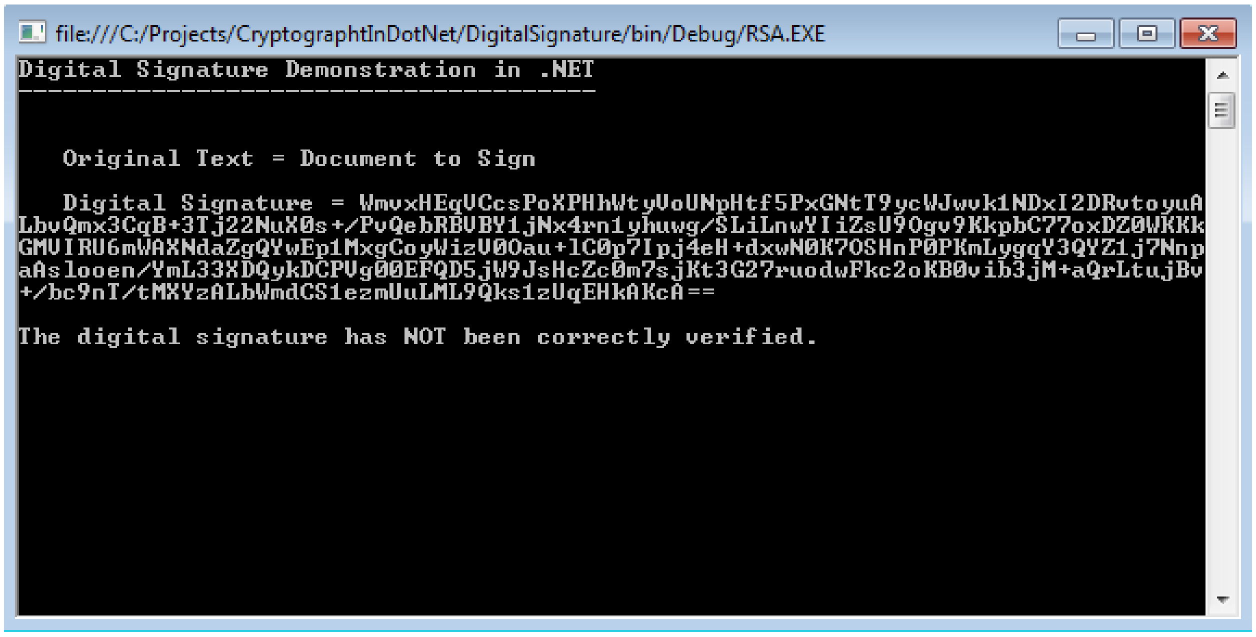


```
var signature = digitalSignature.SignData(hashedDocument);
var verified = digitalSignature.VerifySignature(hashedDocument, signature);

Console.WriteLine("Digital Signature Demonstration in .NET");
Console.WriteLine("--------------------------------------");
Console.WriteLine();
```

Name	Value	Type
signature	{byte[256]}	byte[]
[0]	90	byte
[1]	107	byte
[2]	241	byte
[3]	28	byte
[4]	74	byte
[5]	149	byte
[6]	9	byte

Figure 31: Digital signature tampered in the debugger

When the program is allowed to carry on executing with the tampered digital signature, you will see the following output.



file:///C:/Projects/CryptographtInDotNet/DigitalSignature/bin/Debug/RSA.EXE

```
Digital Signature Demonstration in .NET
---------------------------------------


   Original Text = Document to Sign

   Digital Signature = WmvxHEqVCcsPoXPHhWtyVoUNpHtf5PxGNtT9ycWJwvk1NDxI2DRvtoyuA
LbvQmx3CqB+3Tj22NuX0s+/PvQebRBVBY1jNx4rn1yhuwg/SLiLnwYIiZsU9Ogv9KkpbC77oxDZ0WKKk
GMVIRU6mWAXNdaZgQYwEp1MxgCoyWizV0Oau+lC0p7Ipj4eH+dxwN0K7OSHnP0PKmLygqY3QYZ1j7Nnp
aAslooen/YmL33XDQykDCPVg00EFQD5jW9JsHcZc0m7sjKt3G27ruodwFkc2oKB0vib3jM+aQrLtujBv
+/bc9nT/tMXYzALbWmdCS1ezmUuLML9Qks1zUqEHkAKcA==

The digital signature has NOT been correctly verified.
```

Figure 32: Invalid digital signature

The following is the complete code for the **`DigitalSignature`** example class used in this chapter.

```
using System.Security.Cryptography;

namespace CryptographyInDotNet
{
    public sealed class DigitalSignature
    {
        private RSAParameters publicKey;
        private RSAParameters privateKey;

        public void AssignNewKey()
        {
            using (var rsa = new RSACryptoServiceProvider(2048))
            {
                rsa.PersistKeyInCsp = false;
                publicKey = rsa.ExportParameters(false);
                privateKey = rsa.ExportParameters(true);
            }
        }

        public byte[] SignData(byte[] hashOfDataToSign)
        {
            using (var rsa = new RSACryptoServiceProvider(2048))
            {
                rsa.PersistKeyInCsp = false;
                rsa.ImportParameters(privateKey);

                var rsaFormatter = new RSAPKCS1SignatureFormatter(rsa);
                rsaFormatter.SetHashAlgorithm("SHA256");

                return rsaFormatter.CreateSignature(hashOfDataToSign);
            }
        }

        public bool VerifySignature(byte[] hashOfDataToSign, byte[] signature)
        {
            using (var rsa = new RSACryptoServiceProvider(2048))
            {
                rsa.ImportParameters(publicKey);

                var rsaDeformatter = new RSAPKCS1SignatureDeformatter(rsa);
                rsaDeformatter.SetHashAlgorithm("SHA256");

                return rsaDeformatter.VerifySignature(hashOfDataToSign,
signature);
            }
        }
    }
}
```

Code Listing 38

Closing Notes

Cryptography is a fascinating subject but it can get very complicated if you start to look into the math behind it. For most developers, this level of detail is not required; you just need to take a more pragmatic approach to securing your applications. This book's aim was to help you integrate cryptography into your applications by using proven and well-tested algorithms that are built into the .NET framework. The sample code has been designed to be very practical so that you can incorporate it into your system with minimal complications.

I hope that you have found this book useful and that it helps you to take a more practical approach to adding security into your systems. If you would like to contact me about anything in this book, you can contact me via my blog at www.stephenhaunts.com. Just go to the contact form at the top of the page.

Further Reading

If you find the subject of cryptography interesting and want to read more about it, then I highly recommend the following books:

The Code Book: The Secret History of Codes and Code-breaking by Simon Singh

Cryptography: A Very Short Introduction by Fred Piper and Sean Murphy

Security Driven .NET by Stan Drapkin (Available from http://www.securitydriven.net)

Everyday Cryptography: Fundamental Principles and Applications by Keith Martin

Applied Cryptography: Protocols, Algorithms, and Source Code in C by Bruce Schneier

Practical Cryptography by Niels Ferguson and Bruce Schneier