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.