



Criptografía práctica

El encriptador que lo encripte... debe guardar las claves

El descryptador que lo descrypte... debe conocer las claves

La criptografía nos permite proteger los datos de forma tal, que la visualización o modificación de los mismos solo sea posible para aquellos que conozcan la forma en que han sido encriptados. Esto es aplicable no solo a la comunicación entre dos puntos, sino también al contenido de nuestros ficheros. Además de proteger la información, también podemos utilizar la criptografía para garantizar que los datos no han sido alterados y que provienen de una fuente fiable.

Para realizar las tareas criptográficas, .NET Framework nos ofrece una serie de clases en las que se implementan algoritmos criptográficos estandarizados; en este artículo veremos algunos de esos algoritmos y cómo usarlos desde C# (en el ZIP que acompaña al artículo se incluye también el código para Visual Basic).

Cifrar y descifrar datos

Podemos realizar el cifrado de datos de dos formas distintas, según sea grande o pequeño el tamaño de los mismos. En el primer caso, podemos usar el cifrado simétrico, en el cual intervienen secuencias (*streams*),

en particular una de tipo `CryptoStream`, que será la que pasemos a las clases que utilizan este tipo de cifrado. La segunda forma de cifrado normalmente lo aplicaremos a cantidades pequeñas de datos, los cuales estarán almacenados habitualmente en un array de bytes.

Pero para poder realizar el cifrado (y posteriormente el descifrado) necesitamos crear las claves para cifrar y descifrar esos datos. En el caso de los algoritmos simétricos, además de una clave también hay que generar un vector de inicialización (IV - *Initialization Vector*);

Primitivas criptográficas

Existen diferentes formas de encriptación, conocidas como **primitivas criptográficas**. A continuación enumeramos esas primitivas y el uso que podemos darle.

- **Cifrado de clave secreta (criptografía simétrica)**
Realiza la transformación de los datos, impidiendo que terceros los lean. Este tipo de cifrado utiliza una clave secreta compartida para cifrar y descifrar los datos.
- **Cifrado de clave pública (criptografía asimétrica)**
Realiza la transformación de los datos, impidiendo que terceros los lean. Este tipo de cifrado utiliza un par de claves pública y privada para cifrar y descifrar los datos.
- **Firmas criptográficas**
Ayudan a comprobar que los datos se originan en una parte específica mediante la creación de una firma digital única para esa parte. En este proceso también se usan funciones *hash*.
- **Valores *hash* criptográficos**
Asigna datos de cualquier longitud a una secuencia de bytes de longitud fija. Los valores *hash* son únicos estadísticamente; el valor *hash* de una secuencia de bytes distinta no será el mismo.

Guillermo «Guille» Som

Es Microsoft MVP de Visual Basic desde 1997. Es redactor de **dotNetManía**, mentor de **Solid Quality Iberoamericana**, tutor de **campusMVP**, miembro de Ineta Speakers Bureau Latin America, y autor de los libros "Manual Imprescindible de Visual Basic .NET" y "Visual Basic 2005".
<http://www.elguille.info>

ambos hay que mantenerlos en secreto y solo deben conocerlos las dos partes interesadas. Las clases utilizadas en este tipo de algoritmos incluyen métodos para generar esas claves, en particular los métodos `GenerateKey()` y `GenerateIV()`, aunque también podemos generarlas de forma manual, por ejemplo basándonos en una cadena; en cualquier caso, al instanciar la clase, se generan tanto la clave como el IV de forma automática.

En el código del fuente 1 podemos ver las dos formas de generar esas claves para usar con la clase `DESCryptoServiceProvider`. Si usamos una cadena a partir de la que generar tanto la clave como el vector de inicialización, debemos asegurarnos de que tiene los bytes necesarios; ese valor lo obtenemos por medio de la propiedad `KeySize`, que devuelve el número de bits necesarios (1 byte = 8 bits).

Por otra parte, los algoritmos asimétricos deben crear una clave pública y otra privada; la clave pública es la que se usará para cifrar el contenido y la privada la usaremos para descifrarlo. Si no indicamos lo contrario en el constructor de las clases que utilizan este tipo de algoritmos, se generarán automáticamente claves con una longitud de 1024 bits, (128 bytes); el rango puede ser de 512 a 1024 en incrementos de 64 bits. Por supuesto, solo debemos exponer la clave pública, que será la usada para cifrar el mensaje, mientras que la clave privada la usaremos para descifrarlo.

Estas clases implementan métodos para exportar e importar las claves públicas y/o privadas. Por ejemplo, si queremos generar una cadena en formato XML con la clave pública podemos usar el método `ToXmlString()` pasándole un valor falso

```

static void generarAuto()
{
    // Crear claves simétricas automáticamente
    DESCryptoServiceProvider des = new DESCryptoServiceProvider();
    // Estas llamadas no son necesarias
    // ya que al crear la instancia se generan las claves
    des.GenerateIV();
    des.GenerateKey();
    // Guardar las claves generadas
    byte[] bIV = des.IV;
    byte[] bKey = des.Key;
    Console.WriteLine("La longitud de la clave (KeySize) es de {0} bits", des.KeySize);
    Console.WriteLine("La longitud de la clave es de {0} bytes", bKey.Length);
}

static void generarManual(string clave)
{
    // Crear la clave manualmente a partir de una cadena de texto
    DESCryptoServiceProvider des = new DESCryptoServiceProvider();
    // Averiguar la longitud de las claves
    int bits = des.KeySize;
    // Establecer la clave secreta
    // La longitud de la cadena debe ser de al menos (bits/8) bytes
    int cant = bits / 8;
    if (clave.Length < cant)
    {
        clave += new string('F', cant - clave.Length);
    }
    // Convertir la cadena en un array de bytes
    des.Key = Encoding.Default.GetBytes(clave.Substring(0, cant));
    des.IV = Encoding.Default.GetBytes(clave.Substring(0, cant));
    // Guardar las claves generadas
    byte[] bIV = des.IV;
    byte[] bKey = des.Key;
    Console.WriteLine("La longitud de la clave (KeySize) es de {0} bits", des.KeySize);
    Console.WriteLine("La longitud de la clave es de {0} bytes", bKey.Length);
}

```

Fuente 1. Generación de claves

```

// La longitud de la clave puede ser de 512 a 1024 bits
// en incrementos de 64 bits
DSACryptoServiceProvider dsa = new DSACryptoServiceProvider(640);
Console.WriteLine("La longitud de la clave es de {0} bits", dsa.KeySize);

// Exportar las claves generadas como cadena XML
string clavePublica = dsa.ToXmlString(false);
Console.WriteLine("Clave pública:\n{0}", clavePublica);
Console.WriteLine();
// Las claves pública y privada
string clavePrivada = dsa.ToXmlString(true);
Console.WriteLine("Clave pública y privada:\n{0}", clavePrivada);

```

Fuente 2. Exportar las claves en formato XML

como parámetro; si queremos exportar las dos claves, tendremos que usar un valor verdadero como parámetro de esa llamada.

En el código del fuente 2 vemos cómo usar el método `ToXmlString()` para guardar las claves usadas al crear la instancia de la clase `DSACryptoServiceProvider`.

Cifrado y descifrado asimétrico

Podemos usar la clase `RSACryptoServiceProvider` para cifrar y descifrar textos de forma fácil ya que, a diferencia de la clase `DSACryptoServiceProvider`, expone métodos para realizar esas tareas. Para cifrar un texto (en realidad un *array* de bytes), usaremos el método `Encrypt()`, que devolverá un *array* con el texto cifrado. Para descifrar un texto previamente cifrado, usaremos el método `Decrypt()`. En ambos casos debemos usar las claves públicas y privadas. Si lo que queremos es cifrar los datos, solo necesitamos la clave pública; pero para descifrar esos datos, necesitaremos también la clave privada.

En el código del fuente 3 vemos cómo podemos usar esta clase para cifrar y descifrar un texto.

Comprobar firma digital

Una de las características de las clases que utilizan cifrado asimétrico

es permitirnos generar un valor *hash*, por ejemplo del tipo SHA1, con el contenido del texto a cifrar. Mediante ese valor *hash* podemos comprobar si el mensaje (o texto) ha sido modificado. De esta forma, el mensaje realmente no se cifra, sino solo los valores de comprobación, ya que una de las funcionalidades de este tipo de algoritmos es obtener firmas digitales de mensajes, de forma que podamos comprobar la autenticidad del mismo. Si quisiéramos cifrar el mensaje, podremos usar cualquiera de las clases de cifrado simétrico o bien usar los

```

static void pruebaRSA()
{
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    string claves = rsa.ToXmlString(true);
    string clavePublica = rsa.ToXmlString(false);
    string texto = "Hola, Mundo";

    // Para cifrar solo es necesario la clave pública,
    // aunque también podemos usar las dos claves
    byte[] datosEnc = cifrarRSA(texto, clavePublica);

    // Para descifrar necesitamos la clave pública y privada
    string res = descifrarRSA(claves, datosEnc);
    Console.WriteLine(res);
}

static byte[] cifrarRSA(string texto, string clavePublica)
{
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    // La clave pública usada para cifrar el texto
    rsa.FromXmlString(clavePublica);
    // Convertimos la cadena a un array de bytes
    byte[] datos = Encoding.Default.GetBytes(texto);
    // Generamos los datos encriptados y los devolvemos
    return rsa.Encrypt(datos, false);
}

static string descifrarRSA(string claves, byte[] datosCifrados)
{
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    // Las claves usadas para cifrar el texto
    rsa.FromXmlString(claves);
    // Generamos los datos descifrados
    byte[] datos = rsa.Decrypt(datosCifrados, false);
    // Devolvemos la cadena original
    return Encoding.Default.GetString(datos);
}

```

Fuente 3. Cifrar y descifrar usando RSA

métodos que nos permiten tanto el cifrado como el descifrado.

A la hora de usar la firma digital, el que la genera debe disponer de las dos claves, (pública y privada), pero el que comprueba si es correcta solo necesita la clave pública.

En el código del fuente 4 podemos ver cómo generar el valor *hash* cifrado a partir de una cadena de texto; también usamos ese valor *hash* para comprobar que el texto coincide con el valor *hash* usado para verificarlo. Para generar el valor *hash* SHA1 utilizamos un objeto del tipo `SHA1CryptoSer-`

```
static void probarDSA()
{
    DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();
    string claves = dsa.ToXmlString(true);
    string texto = "Hola, Mundo";
    // Creamos la firma hash usando el texto
    // y las claves pública y privada
    // El texto convertido en un valor hash SHA1
    byte[] datosHash = claveSHA1(texto);
    // La firma para el valor hash indicado
    byte[] firmaHash = crearFirmaDSA(datosHash, claves);

    // Comprobamos si es correcto, usando la clave pública
    string clavePublica = dsa.ToXmlString(false);
    // Comprobamos si los datos son correctos
    bool res = comprobarFirmaDSA(clavePublica, firmaHash, datosHash);
    if (res)
        Console.WriteLine("La comprobación es correcta.");
    else
        Console.WriteLine("La comprobación no es correcta.");
}

static byte[] crearFirmaDSA(byte[] datosHash, string clavesXML)
{
    DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();
    // Asignamos las claves indicadas
    dsa.FromXmlString(clavesXML);
    //
    DSASignatureFormatter dsaFormatter = new DSASignatureFormatter(dsa);
    // Indicamos el algoritmo hash a usar
    dsaFormatter.SetHashAlgorithm("SHA1");
    // Creamos la firma y la devolvemos
    return dsaFormatter.CreateSignature(datosHash);
}

static bool comprobarFirmaDSA(string clavePublica, byte[] firma, byte[] datosHash)
{
    DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();
    // Utilizamos la clave pública
    dsa.FromXmlString(clavePublica);
    DSASignatureDeformatter dsaDeformatter = new DSASignatureDeformatter(dsa);
    // El tipo de algoritmo hash a usar
    dsaDeformatter.SetHashAlgorithm("SHA1");
    // Devolver un valor boolean si la firma es correcta
    return dsaDeformatter.VerifySignature(datosHash, firma);
}

static byte[] claveSHA1(string texto)
{
    // Crear una clave SHA1 a partir del texto indicado.
    // Devuelve un array de bytes con la clave SHA1 generada
    UTF8Encoding enc = new UTF8Encoding();
    byte[] datos = enc.GetBytes(texto);

    SHA1CryptoServiceProvider sha = new SHA1CryptoServiceProvider();
    return sha.ComputeHash(datos);
}
```

Fuente 4. Comprobación de firmas digitales

NOTA

Hay que tener presente que los algoritmos *hash* como SHA1 solo permiten crear valores *hash* a partir de un *array* de bytes, pero no sirven para recuperar el *array* (o texto normal) a partir de un valor *hash*.

`viceProvider`, para asegurarnos de que el valor *hash* generado tiene la longitud adecuada (20 bytes).

Algoritmos simétricos

Las clases que utilizan el algoritmo simétrico para la encriptación de datos utilizan la misma clave tanto para cifrar como para descifrar los datos, además de que se suelen aplicar cuando tenemos gran cantidad de datos, principalmente los obtenidos a partir de secuencias. En estos casos, utilizaremos una secuencia de tipo `CryptoStream` para realizar las tareas de cifrado y descifrado.

En el código del fuente 5 utilizamos la clase `TripleDESCryptoServiceProvider` para cifrar y posteriormente descifrar el contenido de un fichero. La clase `CryptoStream` acepta como parámetros del constructor la secuencia usada para leer o escribir, además de un objeto de tipo `ICryptoTransform` (el cual obtenemos mediante el método `CreateEncryptor` de la clase de encriptación usada) y el modo en el que la usaremos, para leer o para escribir.

En los dos métodos usados en el ejemplo mostrado en el fuente 5, para cifrar/guardar y descifrar/leer, pasamos las claves correspondientes, las cuales deben coincidir. Esas claves las podemos generar nosotros, teniendo en cuenta la longitud esperada, que en el caso de la clase `TripleDESCryptoServiceProvider` debe ser de 128 a 192 bits. Esas claves también las podemos generar a partir de la creación de una ins-

```
static void cifrarFichero(string fichero, string ficSalida,
                        byte[] bKey, byte[] bIV)
{
    byte[] datos;

    using (StreamReader sr =
            new StreamReader(fichero, Encoding.Default, true))
    {
        datos = Encoding.Default.GetBytes(sr.ReadToEnd());
        sr.Close();
    }
    using (FileStream fs =
            new FileStream(ficSalida, FileMode.Create,
                          FileAccess.Write))
    {
        TripleDESCryptoServiceProvider td =
            new TripleDESCryptoServiceProvider();
        td.Key = bKey;
        td.IV = bIV;

        CryptoStream cs = null;
        try
        {
            // Crear una secuencia de cifrado
            cs = new CryptoStream(fs, td.CreateEncryptor(), CryptoStreamMode.Write);
            // Escribir los datos cifrados en el fichero
            cs.Write(datos, 0, datos.Length);
        }
        catch
        { }
        finally
        {
            if (cs != null)
                cs.Close();
        }
    }
}

static void descifrarFichero(string fichero, string ficSalida,
                            byte[] bKey, byte[] bIV)
{
    // El proveedor del cifrado y las claves usadas para cifrar
    TripleDESCryptoServiceProvider td = new TripleDESCryptoServiceProvider();
    td.Key = bKey;
    td.IV = bIV;
    //
    // Crear la secuencia para leer el fichero cifrado
    using (FileStream fs =
            new FileStream(fichero, FileMode.Open,
                          FileAccess.Read))
    {
        using (CryptoStream cs =
                new CryptoStream(fs, td.CreateDecryptor(),
                                CryptoStreamMode.Read))
        {
            // Guardar el contenido de fichero descifrado
            StreamWriter sw = new StreamWriter(ficSalida);
            StreamReader sr = new StreamReader(cs);
            sw.Write(sr.ReadToEnd());
            sw.Flush();
            sw.Close();
        }
    }
}
}
```

Fuente 5. Cifrar y descifrar ficheros

tancia de esa clase y usarlas posteriormente. Si la clave usada para leer/descifrar no es la misma que la que usamos al cifrar el fichero, recibiremos una excepción. En este caso, esas claves las generamos antes de llamar a esos dos métodos, tal como vemos en el código del fuente 6.

medio del método `GetNonZeroBytes()` obtenemos números aleatorios, pero sin incluir el valor cero.

En el código del fuente 7 se generan números aleatorios usando los dos métodos comentados; el número de bytes a generar lo dará el tamaño del *array* pasado como parámetro a los métodos.

```
TripleDESCryptoServiceProvider td = new TripleDESCryptoServiceProvider();

Console.WriteLine("Generando el fichero encriptado");
cifrarFichero(fic1, fic2, td.Key, td.IV);
Console.WriteLine("Descifrando el fichero");
descifrarFichero(fic2, fic3, td.Key, td.IV);
```

Fuente 6. Al cifrar/descifrar los ficheros debemos usar las mismas claves

Generar números aleatorios criptográficos

Las clases de .NET Framework utilizan generadores de números aleatorios para generar las claves criptográficas. Nosotros podemos generar también esos números aleatorios por medio de la clase `RNGCryptoServiceProvider`, que está basada en la clase abstracta `RandomNumberGenerator`.

La forma de generar esos números aleatorios es por medio de un *array* de tipo `byte` en el que indicamos cuántas cifras debe generar, ya que en cada elemento del *array* se incluirá una cifra. El método usado para generar los números aleatorios es `GetBytes()`, el cual incluirá todos los valores, incluso el cero. Por

NOTA

Cuando usemos un objeto de tipo `CryptoStream`, debemos asegurarnos de llamar al método `Close()` para cerrar la secuencia y eliminar los datos de la memoria. Si mientras estamos usando un objeto de tipo `CryptoStream` se produce un error, la secuencia no se cerrará, por tanto es importante que incluyamos la manipulación de esa secuencia dentro de un bloque `using` o dentro de un bloque `try/catch`, para asegurarnos de que la cerramos en el bloque `finally`.

```
// El número de bytes a generar
byte[] numeros = new byte[10];
RNGCryptoServiceProvider rnd = new RNGCryptoServiceProvider();

// En los números generados puede haber ceros
rnd.GetBytes(numeros);
for (int i = 0; i < numeros.Length; i++)
    Console.Write("{0} ", numeros[i].ToString());

Console.WriteLine();

// Genera números sin incluir el cero
rnd.GetNonZeroBytes(numeros);
for (int i = 0; i < numeros.Length; i++)
    Console.Write("{0} ", numeros[i].ToString());
```

Fuente 7. Generar números aleatorios criptográficos

Conclusiones

En este artículo hemos visto de una forma práctica cómo podemos usar algunas de las clases que .NET Framework pone a nuestra disposición para cifrar y descifrar la información que queremos transmitir y que de este modo solo sea “entendible” para aquellos a los que queremos permitirselo.

Si usted prefiere usar Visual Basic, en el código que acompaña a este artículo, el cual puede bajar desde el sitio Web de **dotNetManía**, se incluyen todos los ejemplos aquí usados tanto en Visual Basic como en C#. ○