Uso de Certificados y CMS/PKCS#7
Práctica 7

1. Objetivo

Los algoritmos criptográficos vistos hasta ahora permiten realizar funciones criptográficas básicas: cifrar/descifrar, resumir, y firmar digitalmente. Pero estos algoritmos generan las firmas en el formato PKCS#1 v1.5 que es bastante elemental.

Actualmente es más común almacenar las firmas digitales y otra información criptográfica (certificados, claves, etc.) en el formato PKCS#7 o en su evolución, el formato CMS (Cryptographic Message Syntax).

Para trabajar con el formato CMS, la biblioteca de clases de .NET Framework proporciona el espacio de nombres System.Security.Cryptography.Pkcs, que incluye múltiples clases para realizar operaciones criptográficas con contenedores CMS.

Pero para utilizar las clases de este espacio de nombres, hay que agregar manualmente una referencia al ensamblado de .NET Framework denominado **System.Security**.

El objetivo de esta práctica es realizar las operaciones criptográficas más elementales vistas en prácticas previas con CMS y utilizando certificados.

2. Obtener un certificado

Crea una nueva solución con Visual. Al comienzo del método Main() crea un objeto de **Ayuda**.

Seguidamente, crea un **mensaje**: el array de 64 bytes Msg y rellénalo con los bytes 0x00, 0x01, ... mediante un bucle.

Reutiliza el método estático **ExtraeCertificado()** para obtener un certificado del almacén de certificados del usuario. Para firmar extraeríamos el certificado del emisor del mensaje con una clave privada asociada al certificado. Para cifrar extraeríamos el certificado del receptor del mensaje sin clave privada asociada.

Por simplicidad, en esta práctica extraemos un único certificado con una clave privada asociada que usaremos para todo. Invocar al método desde el método Main() así:

```
X509Certificate2 Certificado = ExtraeCertificado(string NombreSujetoCer);
```

CERTIFICADO A UTILIZAR (MUY IMPORTANTE):

Se puede utilizar el certificado **zpusu.as** y su correspondiente certificado raíz **zpac.as**, generados con PowerShell. Pero este certificado puede presentar limitaciones al descifrar:

Si el certificado se ha creado con el script New-SelfSignedCertificate usando como proveedor de servicios criptográficos…

1) –Provider "Microsoft Base Cryptographic Provider v1.0" (o no se usa el parámetro –Provider)
ENTONCES SI habrá limitaciones (no se puede descifrar)

2) –Provider "Microsoft Enhanced Cryptographic Provider v1.0"
3) –Provider "Microsoft Enhanced RSA and AES Cryptographic Provider"
ENTONCES NO habrá limitaciones

Como alternativa, se puede utilizar el certificado (y su clave privada asociada) de **zmcli.as** y su correspondiente certificado raíz de **zmac.as**, generados con MakeCert, que tiene otras restricciones, pero ninguna para las operaciones realizadas en esta práctica.

Los certificados están disponibles en el CV y su generación esta detallada en la práctica 5A.

Además, en esta práctica se debe utilizar el certificado de persona física obtenido de la FNMT comprobando que se puede utilizar perfectamente para todas las operaciones criptográficas desarrolladas en la práctica.

3. Firmar con la clave privada

Crea el método estático **FirmaCMS()** que devuelve un mensaje CMS (array de bytes) usando tres parámetros de entrada: el mensaje a firmar, el certificado con clave privada asociada, y una variable booleana que indica si la firma es desasociada. El prototipo del método es:

```
static public byte[] FirmaCMS(Byte[] Msg, X509Certificate2 CertFirma, bool Desasociada)
```

El método debe comenzar declarando **CI,** un objeto de la clase **ContentInfo**, que representa los datos a firmar. Este objeto almacena lo que se denomina el contenido interno del mensaje CMS, o sea, los datos a firmar y un OID que identifica lo que son los datos.

Antes de crear CI declara el identificador del mensaje del modo siguiente:

```
Oid IdMsg = new Oid("1.2.840.113549.1.7.1");
```

Los valores posibles para el OID son:

```
// 1.2.840.113549.1.7.1 data
// 1.2.840.113549.1.7.2 signedData
// 1.2.840.113549.1.7.3 envelopedData
// 1.2.840.113549.1.7.4 signedAndEnvelopedData
// 1.2.840.113549.1.7.5 hashedData
// 1.2.840.113549.1.7.5 digestedData
// 1.2.840.113549.1.7.6 encryptedData
```

Crea el objeto **CI** usando un constructor que utilice **IdMsg** y **Msg**.

Como comprobación, escribe en la consola las dos propiedades de CI, **ContentType** y **Content**.

Crea el objeto **CmsFirmado** de la clase **SignedCms** utilizando el constructor con tres parámetros: El primero es del tipo **SubjectIdentifierType**, el segundo el objeto **CI**, y el tercero la variable booleana **Desasociada**.

- Si Desasociada == true -> El contenido del mensaje NO se integra en el fichero de la firma.
- Si Desasociada == false -> El contenido del mensaje SI se integra en el fichero de la firma.

Desasociar el mensaje de la firma es conveniente si se quiere evitar el duplicado de la información, sobre todo con documentos grandes. Otra utilidad en S/MIME (correo seguro) es que si la información va separada de la firma, siempre se puede ver la información aunque no se soporte la verificación de firmas.

Crea el objeto **FirmanteCMS** de la clase **CmsSigner** pasándole en el constructor el certificado del usuario que desea firmar. Este objeto, en la propiedad SignedAttributes permite integrar atributos que se firman junto con el contenido especificado, y en la propiedad UnsignedAttributes permite asociar atributos que no se firman. Los atributos no firmados se pueden modificar sin invalidar la firma.

Como comprobación, escribe en la consola FirmanteCMS.Certificate.Subject.

Ahora se puede generar la firma, invocando el método ComputeSignature(FirmanteCMS) del objeto CmsFirmado. La firma generada queda integrada en el objeto CmsFirmado.

Finalmente hay que codificar el objeto en una cadena de bytes, usando el método Encode() del objeto CmsFirmado y retornarlo.

En el método Main() invocar el método creado así:

```
byte[] MsgCmsFirmadoCod = FirmaCMS(Msg, Certificado, Desasociada);
```

Guarda Msg en el fichero "Fichero.dat". Guarda MsgCmsFirmadoCod en el fichero "Fichero.p7b".

4. Verificar con la clave pública

Crea el método estático **VerificaCMS()** que devuelve un valor booleano (true si la verificación es satisfactoria y false en caso contrario) usando tres parámetros de entrada: el mensaje a verificar, la cadena de bytes con la codificación de un objeto de la clase SignedCms, y una variable booleana que indica si la firma es desasociada. El prototipo del método es:

```
static public bool VerificaCMS(byte[] Msg, byte[] CmsFirmadoCodificado, bool Desasociada)
```

Comienza declarando el objeto **CmsFirmado** de la clase **SignedCms**.

El proceso de verificación varía en función de que el mensaje CMS contenga los datos firmados o no los contenga.

SI la firma es Desasociada

> Crea **IdMsg** de la clase **OID** y pásale el valor "1.2.840.113549.1.7.1" en el constructor. Después crea el objeto **CI** de la clase **ContentInfo** pasándole en el constructor **IdMsg** y **Msg**.
>
> Crea el objeto **CmsFirmado** usando el constructor de tres parámetros y pásale el SubjectIdentifierType, el objeto **CI** y la variable **Desasociada**, que debe valer true.

SINO

> Crea el objeto **CmsFirmado** usando el constructor sin parámetros.

Ahora introduce la información de firma en el objeto **CmsFirmado**, invocando a su método **Decode**(**CmsFirmadoCodificado**).

Una vez que el objeto CmsFirmado ha sido inicializado correctamente, se puede verificar la firma que contiene usando el método **CheckSignature**() del propio objeto. Pero el método CheckSignature() NO devuelve un valor. Simplemente, cuando la firma no es correcta genera una excepción. Por tanto hay que gestionar las excepciones obligatoriamente.

Comienza declarando la variable booleana **Resultado**.

Seguidamente declara una estructura try { } catch { }.

En el **bloque try**, comienza mostrando en la consola el mensaje "Comprobando firma ... ". Seguidamente invoca al método CheckSignature(true) del objeto CmsFirmado.

Muestra en la consola el mensaje "La firma es válida". Este mensaje se mostrará solamente si no se genera una excepción.

Finalmente asigna true a **Resultado**.

En el **bloque catch**, muestra en la consola la excepción generada y asigna false a **Resultado**.

El método termina retornando la variable booleana **Resultado**.

En el método Main() comienza la sección de código dedicada a la verificación de la firma con las instrucciones siguientes:

```
Console.WriteLine("Altera el fichero de datos o firma si quieres ...");
Console.ReadKey();
```

Esto permite parar la ejecución del programa para alterar los ficheros con HxD, por ejemplo.

Carga la firma en la nueva variable MsgCmsFirmadoCod2 de tipo byte[] desde el archivo "Fichero.p7b".

Carga el mensaje desde archivo si es necesario. Para ello, comienza declarando Msg2 de tipo byte[] e inicialízalo a null. Después, SI la firma es Desasociada, asigna espacio de almacenamiento para Msg2 con el operador new y cárgale el mensaje desde el archivo "Fichero.dat".

Declara la variable booleana Verificacion y asígnale el valor que devuelve el método VerificaCMS() creado previamente.

Escribe en la consola el valor de la variable Verificacion.

PRUEBAS: Las firmas creadas y almacenadas con formato CMS en ficheros .p7b con este programa pueden ser verificadas con la aplicación XolidoSign, y viceversa, las firmas creadas con XolidoSign pueden ser verificadas con este programa. ¡Compruébalo!.

5. Cifrar (usando la clave pública)

CONCEPTO ESENCIAL: Se utiliza la clave pública del certificado del receptor para cifrar la clave del algoritmo simétrico (3DES, AES, ...) usado para cifrar los datos. Los datos se envían al receptor cifrados con una clave simétrica, junto con la clave simétrica cifrada con la clave pública asimétrica del receptor.

Crea el método estático **CifraCMS()** que devuelve un array de bytes y usa dos parámetros de entrada: el mensaje a cifrar, y el certificado del receptor (destinatario) del mensaje. El prototipo del método es:

```
static public byte[] CifraCMS(Byte[] Msg, X509Certificate2 CertReceptor)
```

El método debe comenzar declarando el objeto **CI** de la clase **ContentInfo**, que representa los datos a cifrar. Este objeto almacena lo que se denomina el contenido interno del mensaje CMS, o sea, los datos a cifrar y un OID que identifica lo que son los datos. En este caso pasa directamente Msg en el constructor de ContentInfo().

Crea el objeto **CmsCifrado** de la clase **EnvelopedCms** utilizando el constructor con un solo parámetro: un objeto de la clase **ContentInfo**. Pasa como parámetro el objeto **CI**.

Muestra el algoritmo usado para cifrar el contenido. Para ello, usa la propiedad ContentEncryptionAlgorithm del objeto CmsCifrado que devuelve un objeto de la clase AlgorithmIdentifier. Escribe en la consola la propiedad Oid de este objeto, usando las dos propiedades que proporciona un Oid, su Nombre y su valor. Puedes mostrar también la longitud de la clave utilizada por el objeto de la clase AlgorithmIdentifier.

Crea el objeto **ReceptorCMS** de la clase **CmsRecipient** utilizando un constructor con dos parámetros. Para el primer parámetro usa uno de los valores de la enumeración **SubjectIdentifierType**, como por ejemplo, **SubjectKeyIdentifier**. El segundo parámetro es el objeto **CertReceptor**, pasado como parámetro al método.

Cifra el contenido llamando al método **Encrypt()** del objeto **CmsCifrado** y pasando al método el objeto **ReceptorCMS**. El contenido cifrado queda integrado en el objeto CmsCifrado.

Finalmente hay que codificar el objeto CmsCifrado en una cadena de bytes, usando el método Encode() del objeto CmsCifrado y retornarlo.

En el método Main() invocar el método creado así:

```
byte[] MsgCmsCifradoCod = CifraCMS(Msg, Certificado);
```

Guarda MsgCmsCifradoCod en el fichero "Fcifrado.dat".

6. Descifrar (usando la clave privada)

CONCEPTO ESENCIAL: Se utiliza la clave privada asociada al certificado del receptor para descifrar la clave del algoritmo simétrico (3DES, AES, ...) usado para cifrar los datos. Los datos llegan al receptor cifrados con una clave simétrica, junto con la clave simétrica cifrada con la clave pública asimétrica del receptor.

Crea el método estático **DescifraCMS()** que devuelve el mensaje descifrado como un array de bytes y usa un parámetro de entrada: el mensaje CMS a descifrar. El prototipo del método es:

```
static public byte[] DescifraCMS(byte[] CmsCifradoCodificado)
```

Comienza el método creando el objeto **CmsCifrado** de la clase **EnvelopedCms** utilizando el constructor sin parámetros.

Inicializa el objeto **CmsCifrado** llamando a su método **Decode()** pasándole a este método el argumento recibido en CmsCifradoCodificado.

Como comprobación de una correcta decodificación del mensaje CMS, comprueba que el número de receptores del mensaje es uno solo. Declara un objeto de la clase **RecipientInfoCollection** e inicialízalo con la propiedad **RecipientInfos** de **CmsCifrado**. Si el número de elementos de la colección es uno extrae el único elemento y cárgalo en el objeto **Receptor** de la clase **RecipientInfo**. Escribe en la consola el tipo y el valor del identificador del receptor.

Descifra el mensaje llamando al método **Decrypt()** del objeto **CmsCifrado**. Este método admite varias sobrecargas para el parámetro que recibe. Pásale el objeto **Receptor** obtenido de **CmsCifrado**. Tras llamar al método, los datos descifrados están disponibles en un objeto de la clase **ContentInfo**, al que se puede acceder mediante la propiedad **ContentInfo** de **CmsCifrado**.

Retorna los datos contenidos en el objeto de la clase **ContentInfo**, a los que se accede mediante la propiedad **Content** del objeto.

En el método Main(), carga el array de bytes MsgCmsCifradoCod2 con el mensaje cifrado desde el fichero que lo contiene, y después, invoca el método creado así:

```
byte[] MsgDescifrado = DescifraCMS(MsgCmsCifradoCod2);
```

Finalmente, muestra en la consola el mensaje descifrado.

7. Anidación de operaciones

El formato CMS proporciona la capacidad de aplicar los dos tipos de protección (Firma y Cifrado) a un mismo mensaje. Primero se aplica uno y luego el otro.

Es común firmar un mensaje y luego cifrarlo (envolverlo). Para ello hay que firmar el mensaje usando un objeto de la clase **SignedCms** y luego codificarlo. El array de bytes que se obtiene de la codificación se utiliza para construir un objeto de la clase **ContentInfo**, que se usa a su vez, para construir un objeto de la clase **EnvelopedCms**.

Ahora el objeto **EnvelopedCms** tiene un objeto **SignedCms** anidado dentro de él, como su contenido interno. Finalmente cifra (envuelve) el contenido del objeto **EnvelopedCms**.

CONCEPTO IMPORTANTE: Para que el mensaje quede envuelto o cifrado en el objeto EnvelopedCms, la firma **NO** debe ser **desasociada**: la firma y el mensaje firmado deben estar ambos incluidos en el objeto SignedCms que es envuelto en el objeto EnvelopedCms.

Como ejercicio, al final del método **Main()**, se puede firmar y cifrar un mensaje de forma anidada.

El EMISOR Firma y Cifra el mensaje así:

Crea el array de bytes **MsgCmsFirmadoCod3** y asígnale lo que devuelve el método **FirmaCMS()** al firmar el mensaje **Msg** utilizando una firma NO desasociada.

Crea al array de bytes **MsgCmsCifradoCod3** y asígnale lo que devuelve el método **CifraCMS()** al cifrar el array de bytes **MsgCmsFirmadoCod3**.

El RECEPTOR Descifra y Verifica el mensaje así:

Crea el array de bytes **MsgCmsDescifrado3** y asígnale lo que devuelve el método **DescifraCMS()** al que se le pasa el array **MsgCmsCifradoCod3**.

Crea el variable booleana **Verifica2** y asígnale lo que devuelve el método **VerificaCMS()** al pasarle el array **MsgCmsDescifrado3**. Muestra el resultado de la verificación en la consola.

Ahora, modifica el método CifraCMS() para que **utilice como algoritmo de cifrado** AES256 en vez del algoritmo por defecto 3DES.