Cifrado con RSA y firma con RSA y SHA
Práctica 4

1. Objetivo

El objetivo de esta práctica es afianzar la comprensión de los conceptos teóricos sobre cifrado de información con algoritmos asimétricos y saber utilizarlos en un entorno de desarrollo. Además, también se afianza el concepto de firma digital basada en algoritmos asimétricos. Para realizar estas tareas, el alumno debe utilizar las clases disponibles en el entorno Visual Studio y .NET Framework en el espacio de nombres System.Security.Cryptography.

2. Generación de claves RSA

La tarea mínima a realizar en la práctica para cifrar/descifrar información con RSA consistirá en tres pequeños programas: uno para generar claves RSA y almacenarlas en ficheros. Otro programa utilizará la clave pública para cifrar un bloque de información. El tercer programa descifrará el bloque de información cifrada utilizando la clave privada.

Crear una solución de VS **para generar las claves** que deberá realizar las tareas siguientes:

1) Crear un objeto de ayuda que permita usar los métodos de ayuda desarrollados.

2) Crear un objeto de la clase **CspParameters** que permite especificar los parámetros deseados para un Proveedor de Servicios Criptográficos (*Cryptographic Service Provider, CSP*).

3) Crear un objeto proveedor de servicios criptográficos RSA a partir de la clase **RSACryptoServiceProvider**. Usar el constructor sin parámetros, que crea un proveedor con los parámetros por defecto. Después usar el constructor que recibe como parámetros la longitud de la clave y un objeto del tipo CspParameters, que permite especificar los parámetros del proveedor RSA. Finalmente, usando cualquiera de los constructores elegir una longitud de clave de 1024 bits que es el valor por defecto. La clave es generada aleatoriamente.

4) Mostrar en la consola las propiedades básicas del proveedor: los tamaños legales de las claves, el tamaño de la clave actual, si la clave debe conservarse en el proveedor actual, etc. Mirar la ayuda para ver las propiedades disponibles.

5) Exportar las claves utilizando los diversos métodos disponibles. Primeramente en formato XML utilizando el método **ToXmlString()** para convertir los parámetros básicos del objeto proveedor a una cadena XML.

6) El segundo método consiste en exportar los parámetros del proveedor RSA a una estructura del tipo **RSAParameters** usando el método **ExportParameters()**. Mostrar en la consola los elementos que constituyen las claves RSA almacenadas en la estructura.

7) El tercer método consiste en exportar las claves contenidas en el objeto proveedor a un objeto binario, comúnmente denominado Blob (*Binary Large Object*), utilizando el método

ExportCspBlob(). Desde el punto de vista de la programación, un Blob es simplemente un array de bytes. Primero exportar incluyendo las claves pública y privada, y guardar el Blob en un fichero binario en el disco denominado "zz_BlobRSA_Priva.bin". Después exportar nuevamente sin incluir la clave privada, y guardar el Blob en un fichero binario denominado "zz_BlobRSA_Publi.bin".

8) Liberar los recursos utilizados por el proveedor llamando al método **Dispose()**.

Si se desea comprobar que las claves se han generado y exportado correctamente, se puede cargar desde fichero el Blob de la clave privada (o la pública) en un array de bytes.

Crear un nuevo proveedor RSA e invocar su método **ImportCspBlob()** para comprobar que la clave se importa correctamente en el proveedor sin generar excepciones.

Los ficheros zz_BlobRSA_Priva.bin y zzBlobRSA_Publi.bin deben copiarse del directorio ..\bin\Debug\ a otro directorio en el que no sean sobrescritos con una nueva clave aleatoria cada vez que se ejecute el programa de generación de claves.

3. Cifrar y descifrar información con RSA

Crear una **nueva solución** de Visual Studio **para cifrar la información** que deberá realizar las tareas siguientes:

1) Crear un objeto de ayuda que permita usar los métodos de ayuda desarrollados.

2) Crear un objeto proveedor de servicios criptográficos RSA a partir de la clase **RSACryptoServiceProvider**. Elegir una longitud de clave igual a la generada anteriormente, que fue de 1024 bits (valor por defecto) y una nueva clave será generada aleatoriamente.

3) Mostrar alguna información en la consola que permita comprobar que el proveedor RSA tiene las características que deseamos. Por ejemplo, la longitud de la clave que utiliza.

4) Obtener el tamaño en bytes del fichero "zz_BlobRSA_**Publi**.bin" y declarar un array de bytes del mismo tamaño para almacenar un Blob. Cargar el array de bytes con el contenido del fichero, así el Blob contendrá la clave RSA pública generada anteriormente.

5) Importar el Blob en el objeto proveedor RSA usando el método **ImportCspBlob()**.

6) Comprobar que la importación fue correcta. Para ello se sugiere exportar los parámetros del proveedor RSA a una estructura del tipo **RSAParameters** usando el método **ExportParameters(false)**. Observar que el parámetro false indica que no se exportan los parámetros privados de la clave. Mostrar en la consola el exponente público y el módulo. Comprobar que son correctos.

7) Crear un array de bytes en memoria para almacenar el texto plano a cifrar. El tamaño deberá estar definido por un número entero, por ejemplo 64. Inicializar los bytes con la secuencia 0, 1, 2, 3, etc., hasta llenar el array totalmente. Mostrar el contenido del array en la consola para comprobar la corrección de la información a cifrar.

8) Declarar un array de bytes para el texto cifrado y asignarle el resultado de llamar al método **Encrypt()** del proveedor RSA. Para cifrar, hay que elegir una técnica para rellenar el texto plano. Si el segundo parámetro de Encrypt() es true se usa el relleno OAEP y si es false se usa el relleno

PKCS#1 v1.5. Mostrar el texto cifrado en la consola y guardarlo en el fichero "zz_TextoCifrado.bin".

9) Liberar los recursos utilizados por el proveedor llamando al método **Dispose()**.

Antes de ejecutar el programa de cifrado, hay que copiar en el directorio ..\bin\Debug\ el fichero "zz_BlobRSA_Publi.bin".

Crear una **nueva solución** de Visual Studio **para descifrar la información** que deberá realizar las tareas siguientes:

1) Crear un objeto de ayuda que permita usar los métodos de ayuda desarrollados.

2) Crear un objeto proveedor de servicios criptográficos RSA a partir de la clase **RSACryptoServiceProvider**. Elegir una longitud de clave igual a la generada anteriormente, que fue de 1024 bits (valor por defecto) y una nueva clave será generada aleatoriamente.

3) Mostrar alguna información en la consola que permita comprobar que el proveedor RSA tiene las características que deseamos. Por ejemplo, la longitud de la clave que utiliza.

4) Obtener el tamaño en bytes del fichero "zz_BlobRSA_**Priva**.bin" y declarar un array de bytes del mismo tamaño para almacenar un Blob. Cargar el array de bytes con el contenido del fichero, así el Blob contendrá las claves RSA pública y privada generadas anteriormente.

5) Importar el Blob en el objeto proveedor RSA usando el método **ImportCspBlob()**.

6) Comprobar que la importación fue correcta. Para ello se sugiere exportar los parámetros del proveedor RSA a una estructura del tipo **RSAParameters** usando el método **ExportParameters(true)**. Observar que el parámetro true indica que se exportan los parámetros privados de la clave además de los públicos. Mostrar en la consola todos los parámetros de la clave, los exponentes privado y público, el módulo y los números primos 1(P) y 2(Q). Comprobar que son correctos.

7) Obtener el tamaño en bytes del fichero "zz_TextoCifrado.bin" y declarar un array de bytes del mismo tamaño para almacenar el texto cifrado. Cargar el array de bytes con el contenido del fichero. Mostrar el contenido del array en la consola para comprobar la corrección de la información cifrada.

8) Declarar un array de bytes para el texto descifrado y asignarle el resultado de llamar al método **Decrypt()** del proveedor RSA. Para descifrar, hay que usar la misma técnica empleada para rellenar el texto plano en el proceso de cifrado. Si el segundo parámetro de Encrypt() es true se asume un relleno OAEP y si es false se asume un relleno PKCS#1 v1.5. Mostrar el texto descifrado en la consola.

9) Liberar los recursos utilizados por el proveedor llamando al método **Dispose()**.

Antes de ejecutar el programa de descifrado, hay que copiar en el directorio ..\bin\Debug\ los ficheros "zz_BlobRSA_Priva.bin" y "zz_TextoCifrado.bin".

Haz pruebas con los programas para comprobar que la longitud del mensaje a cifrar junto con el relleno utilizado (PKCSv1.5 o OAEP) debe ser inferior a la longitud de la clave RSA.

4. Crear una firma digital con SHA y RSA y verificarla

Crear una **nueva solución de Visual Studio** que realice el cálculo del resumen de un mensaje y su cifrado (firma) con el algoritmo RSA. Posteriormente se verifica la firma.

Tras crear la solución copia el fichero "zz_BlobRSA_Priva.bin" en el directorio ..\bin\Debug\ de la solución.

El programa deberá realizar las tareas siguientes:

Crear un objeto de ayuda que permita usar los métodos de ayuda desarrollados.

Declarar un array de 64 bytes como la variable Mensaje. Usar un bucle para generar el Mensaje con los valores 0x00, 0x01, ..., 0x3F.

FASE-1: CÁLCULO DEL HASH

1) Crear un objeto proveedor de servicios criptográficos para el algoritmo SHA256 a partir de la clase **SHA256CryptoServiceProvider**. También es posible usar las clases **SHA256Managed** y **SHA256Cng**.

2) Mostrar en la consola algunas propiedades del algoritmo de Hash, como por ejemplo el tamaño del hash generado, usando la propiedad **HashSize**.

3) Declarar un array de bytes denominado Resumen y cargarlo con el valor del hash obtenido llamado al método **ComputeHash()** del objeto proveedor de SHA256. Mostrar en la consola el hash calculado.

4) Liberar todos los recursos utilizados por el proveedor SHA llamando al método **Dispose()**.

FASE-2: CREAR UN PROVEDOR RSA PARA CIFRAR EL VALOR DEL HASH

1) Crear un objeto proveedor de servicios criptográficos RSA a partir de la clase **RSACryptoServiceProvider**. Elegir una longitud para la clave que coincida con la longitud de las claves almacenadas en el fichero "zz_BlobRSA_Priva.bin".

2) Mostrar en la consola la longitud de la clave que utiliza el proveedor RSA.

3) Cargar el Blob contenido en el fichero "zz_BlobRSA_Priva.bin" en el array de bytes BlobRSA_Priva y luego impórtalo en el proveedor RSA utilizando el método **ImportCspBlob()**.

FASE-3: GENERAR LA FIRMA

1) Declarar un array de bytes denominado Firma e inicializarlo con la firma devuelta por el método **SignHash()** del objeto proveedor RSA. A este método hay que pasarle el Resumen calculado previamente y el identificador del algoritmo con el que se creó el resumen. Mostrar la Firma en la consola.

2) Declarar otro array de bytes Firma2 e inicializarlo con la firma devuelta por el método **SignHash()** del objeto proveedor RSA a partir del Resumen usado previamente. Mostrar la Firma en la consola.

La firma debe ser igual a la anterior, pues este esquema de firma es determinista: para el mismo resumen y la misma clave, la firma siempre es la misma.

3) Comprobar que se puede obtener la firma en un solo paso, llamando al método **SignData()** del objeto proveedor RSA. Declarar una variable Firma3 e inicializarla con el resultado de llamar al método. Mostrar en la consola Firma3 y comprobar que coincide con Firma.

FASE-4: COMBINAR EL MENSAJE CON LA FIRMA

En esta práctica no se combinará el mensaje (bloque de información) con la firma. Simplemente volcar el mensaje en el fichero "zz_Mensaje.bin" y volcar la firma en el fichero "zz_Firma.bin".

FASE-5: VERIFICAR LA FIRMA ANTES DE ENVIARLA CON EL MENSAJE

1) Incluir tres instrucciones que modifiquen un byte de los arrays de bytes Mensaje, Resumen y Firma. Inicialmente comentar las tres instrucciones. Si se desea que la firma sea NO válida descomentar una o más de estas instrucciones.

2) Verificar la validez de la firma contra el resumen de los datos calculado previamente. Declarar la variable booleana VFR (**V**erificación de la **F**irma **R**ecibida) y asignarle el resultado devuelto por el método **VerifyHash()** del objeto proveedor RSA. Si VFR es true la firma es válida.

3) Verificar la validez de la firma contra el mensaje a partir del cual se ha obtenido el resumen. Declarar la variable booleana VFM y asignarle el resultado devuelto por el método **VerifyData()** del objeto proveedor RSA. Si VFM es true la firma es válida.

4) Finalmente, liberar todos los recursos utilizados por el proveedor RSA llamando al método **Dispose()**.

TRABAJO OPCIONAL: Realizar un nuevo programa (nueva solución de VS) que verifique la firma generada por el programa anterior leyendo de ficheros el mensaje, la firma del mensaje y la clave pública RSA del usuario que generó el mensaje y su firma.

REFLEXIONAR: Al terminar esta fase de la práctica es importante **DIFERENCIAR** claramente las dos formas de trabajo de RSA y como esas formas dependen de qué métodos del proveedor de RSA se utilizan.

FORMA 1: PARA CIFRAR + DESCIFRAR INFORMACIÓN (ej. clave de sesión)

1) **B**enito (receptor de la información)

Genera su pareja de claves (pública + privada)

2) **A**licia (emisor de la información)

Cifra INFO con la clave pública de B

Hay que inicializar el objeto RSA de A con la clave pública de B

Cifrar INFO con Encrypt() que usa la clave pública disponible

3) **B**enito (receptor de la información)

Descifra INFO con la clave privada de B

Hay que inicializar el objeto RSA de B con ambas claves de B

Descifrar INFO con Decrypt() que usa la clave privada disponible

FORMA 2: PARA AUTENTICAR INFORMACIÓN (Firma Digital)

1) **A**licia (emisor de información)

Genera su pareja de claves (pública + privada)

2) **A**licia

Cifra HASH con la clave privada de A

Hay que inicializar el objeto RSA de A con ambas claves de A

Cifrar HASH con SignHash() que usa la clave privada disponible

NO se puede usar Encrypt() pues cifraría con la clave pública

3) **B**enito (receptor de información)

Descifra HASH con la clave pública de A

Hay que inicializar el objeto RSA de B con la clave pública de A

Descifra HASH con VerifyHash() que usa la clave pública disponible

NO se puede usar Decrypt() pues descifraría con la clave privada

5. Anexo: Contenedores de claves RSA

Las claves RSA, incluyendo la parte pública y la privada, pueden almacenarse de modo permanente en un computador en ubicaciones predefinidas y en condiciones seguras.

En esta sección de la práctica se creará un programa que almacene una clave RSA en un contenedor. Un contenedor de una clave es un fichero que contiene los elementos de una clave RSA de un modo protegido. Los contenedores se integran en un "almacén" de contenedores, que es simplemente un directorio en el sistema de ficheros del computador.

Los pasos para crear el programa son los siguientes:

FASE.-1: Crear un contenedor de una clave RSA y ver sus propiedades

1) Crea el objeto ParamCSP de la clase **CspParameters** usando un constructor sin parámetros.

2) Asigna valores a los 4 campos del objeto:

A **ProviderType** asígnale 1 (valor por defecto) para RSA.

A **ProviderName** asígnale el nombre de un Proveedor de Servicios Criptográficos, tal como "Microsoft Strong Cryptographic Provider".

A **KeyContainerName** asígnale un nombre, por ejemplo, "PRACTICAS".

A **KeyNumber** asígnale (int) KeyNumber.Exchange ó 1 si la clave se utiliza para intercambiar información cifrada, o bien, (int) KeyNumber.Signature ó 2 si la clave se utiliza para firma digital.

3) Se pueden asignar valores a Flags. Consulta la ayuda para ver los valores utilizables.

Mediante los Flags es posible definir el almacén a usar para un contendor. Por defecto el contenedor de claves se almacena en el almacén de claves del usuario. Si se desea utilizar el almacén de claves de la máquina, usar:

```
ParamCSP.Flags = CspProviderFlags.UseMachineKeyStore;
```

4) Crea el objeto ProvRSA, proveedor de servicios criptográficos RSA, usando en el constructor una longitud de clave, 1024 por ejemplo, y el objeto ParamCSP.

5) Controla la persistencia de la clave. Asigna la propiedad **PersistKeyInCsp** a true para indicar que la clave debe conservarse y a false para indicar que no debe conservarse.

La propiedad PersistKeyInCsp se establece automáticamente a true al especificar un nombre de contenedor de claves en un objeto CspParameters y al usarlo para inicializar un objeto RSACryptoServiceProvider.

Concretamente, cuando se especifica el nombre de un contenedor de claves en un objeto CspParameters y se le pasa a un objeto AsymmetricAlgorithm con la propiedad PersistKeyInCsp asignada a true ocurre lo siguiente:

Si el contenedor con el nombre especificado ...

- NO existe, se crea el contenedor y la clave asimétrica se almacena en el contenedor
- YA existe, la clave almacenada se carga automáticamente en el objeto AsymmetricAlgorithm

6) Muestra en la consola las propiedades del objeto RSACryptoServiceProvider relativas a la clave, por ejemplo, KeySize, PersistKeyInCsp, PublicOnly, KeyExchangeAlgorithm, SignatureAgorithm. También se puede mostrar la propiedad estática UseMachineKeyStore.

7) Muestra en la consola los elementos de la clave RSA en formato XML.

8) Muestra información sobre el contenedor de la clave. Para ello crea el objeto InfoConten de la clase **CspKeyContainerInfo** a partir de la propiedad CspKeyContainerInfo del objeto proveedor RSA.

Para mostrar las propiedades del objeto InfoConten incluye el método estático VerContenedor() al final del método Main(), el cual recibe como parámetro un objeto de la clase CspKeyContainerInfo y muestra sus propiedades en la consola.

9) Invoca al método Dispose() del objeto proveedor RSA para liberar todos sus recursos.

FASE.-2: Cargar la clave RSA de un contenedor en un nuevo proveedor RSA

1) Crea el objeto ProvRSA2, usando la misma longitud de clave y el mismo objeto ParamCSP que en la fase anterior.

2) Muestra la clave RSA en formato XML. Debe ser idéntica a la mostrada en la fase 1.

3) LLama al método Dispose() de ProvRSA2 para liberar todos sus recursos.

FASE.-3: Observa el almacén en el que se creó el contenedor

1) Introduce las dos líneas siguientes en el programa:

```
Console.WriteLine("\npulsa una tecla para continuar ...");
Console.ReadKey();
```

Al parar la ejecución se puede ver en el directorio (almacén) el fichero (contenedor) con la clave.

Un CSP de la "Legacy CryptoAPI" de Microsoft guarda las claves en:

SI SE USA EL ALMACÉN DEL USUARIO:

En %APPDATA%\Microsoft\Crypto\RSA\User SID\

```
echo %APPDATA% = C:\Users\Daniel\AppData\Roaming\
   \User SID\ = S-1-5-21-79863388-467637616-1809882203-1000
```

NOTA: \AppData\ es un directorio oculto

Lo mejor es borrar el contenido de esta la carpeta si no contiene nada importante

Después de ejecutar el programa hasta aquí tiene que aparecer un archivo contenedor

SI SE USA EL ALMACÉN DE LA MÁQUINA:

En %ALLUSERSPROFILE%\Microsoft\Crypto\RSA\MachineKeys

```
echo %ALLUSERSPROFILE% = C:\ProgramData\
```

NOTA: \ProgramData\ es un directorio oculto

No borrar archivos de esta carpeta; ordenarla por fecha para ver el último archivo creado

FASE.-4: Eliminación de un contenedor

En esta fase se elimina el contenedor del almacén.

1) Crea el objeto ProvRSA3, usando la misma longitud de clave y el mismo objeto ParamCSP que en el caso anterior.

2) Asigna la propiedad PersistKeyInCsp de ProvRSA3 a false.

3) Muestra la clave RSA en formato XML. Debe ser idéntica a la mostrada en las fases anteriores.

4) Libera recursos llamando al método Dispose() de ProvRSA3. También se podría llamar al método Clear(). Al liberar los recursos y estar la propiedad PersistKeyInCsp a false, el contenedor es borrado del directorio. Compruébalo.

Si no deseas que se elimine el contenedor no ejecutes la fase 4. Cuando el programa este parado en el método ReadKey() de la fase 3, aborta la ejecución pulsando **Ctrl+C**.

Como **ejercicio final**, puedes desarrollar un nuevo programa que utilice la clave RSA almacenada por este programa, en vez de generar una nueva aleatoriamente.

Tendrás que combinar los flags deseados con el flag UseExistingKey utilizando el operador OR bit a bit | para evitar la creación automática de una nueva clave RSA.

6. Extensión: Comparación de firmas SHA-RSA deterministas y probabilistas

Las firmas basadas en SHA y RSA que se han realizado previamente en esta práctica son del tipo RSASSA-PKCS1 de tipo determinista. Pero también es posible realizar firmas del tipo RSASSA-PSS de tipo probabilístico.

Crea una nueva solución de Visual Studio para realizar esta extensión de la práctica. Copia en esta solución el programa principal desarrollado en la sección 4 para firma y verificar.

Para despreocuparse de la clave a utilizar elimina la sección del programa que carga las claves pública y privada desde un fichero y utiliza el par de claves que se crean aleatoriamente en cada ejecución del programa. Podemos hacer esto, porque usamos la clave privada para firmar y la pública para verificar en el mismo programa, usando un único proveedor RSA.

PARA FIRMAR:

Usa la sobrecarga del método SignHash() que usa tres parámetros: el resumen, el nombre del algoritmo de resumen y el relleno utilizado para la firma RSA.

Es muy recomendable parametrizar un poco el programa declarando el string **AlgResumen** que contenga la cadena que define el algoritmo usado para calcular el resumen, esto es "SHA1" "SHA256" "SHA384" "SHA512". Después crea el objeto **NomAlgRes** de la clase **HashAlgorithmName** usando en el constructor la cadena definida en AlgResumen. Usa AlgResumen como segundo parámetro en la llamada al método SignHash().

Ahora crea el objeto **RellenoFirma** de la clase **RSASignaturePadding** asignándole para su creación la propiedad **Pkcs1** de la clase RSASignaturePadding. Usa RellenoFirma como tercer parámetro en la llamada al método SignHash().

Utiliza los mismos parámetros en la segunda llamada al método SighHash().

Modifica la llamada al método SignData para que utilice los dos parámetros definidos previamente, NomAlgRes y RellenoFirma.

PARA VERIFICAR:

Usa la sobrecarga del método VerifyHash() que usa cuatro parámetros: el resumen, la firma, el nombre del algoritmo de resumen y el relleno utilizado para la firma RSA. Para el tercer y el cuarto parámetros utiliza los dos parámetros definidos previamente, NomAlgRes y RellenoFirma.

Usa la sobrecarga del método VerifyData() que usa cuatro parámetros: los datos, la firma, el nombre del algoritmo de resumen y el relleno utilizado para la firma RSA. Para el tercer y el cuarto parámetros utiliza los dos parámetros definidos previamente, NomAlgRes y RellenoFirma.

PRUEBAS:

Comprueba que el programa funciona como se espera al usar un RellenoFirma del tipo **Pkcs1** (determinista) generando el mismo valor de firma las tres veces que se realiza la firma. Comprueba también que la verificación de las firmas es correcta.

Ahora usa un RellenoFirma del tipo **Pss** (probabilista). El programa generará una excepción. Eso se debe a que un proveedor RSA del tipo **RSACryptoServiceProvider** no soporta este tipo de relleno en las firmas. Comenta la línea de proveedor RSA y usa otra con un proveedor RSA del tipo **RSACng**.

Comprueba que las tres firmas generadas son diferentes.

Comprueba también que las dos verificaciones funcionan correctamente con cualquiera de las firmas, siempre que no se estropee deliberadamente la firma y/o el mensaje.

Ahora utiliza un algoritmo de resumen SHA512 y una longitud de clave de 1024 bits. ¿Funciona correctamente el programa? ¿Cuál es el problema? ¿Cómo lo puedes solucionar?