AHORA POR SÓLO 4,50 €

...IERA REVISTA DE PROGRAMACIÓN EN CASTELLANO

SOPROGRAMADORES

AÑO VIII. 2.ª ÉPOCA • Nº 94 • UNA PUBLICACIÓN DE: REVISTAS PROFESIONALES S.L. • Precio: 4,50 € (España) (IVA incluido)

SERVICIOS DE LOCALIZACIÓN: GPS/GIS

CRIPTOGRAFÍA
- CRYPTO API

PROTOCOLOS
- NNTP (News)

JAVA
- DISEÑO POR CONTRATO
- ESPECIFICACIÓN JAVA 2

ALMACENAMIENTO
- FILEMAPPING

CANAL PANDA
- FIREWALLS

GRÁFICOS 3D
- PARTÍCULAS

MICROSOFT OFFICE
- ACCEDIENDO A EXCEL

CRIPTOGRAFÍA BAJO WINDOWS (II)

José Luis Blanco Claraco

En esta segunda parte, estudiaremos algunas de las funciones más prácticas de CryptoAPI, sobre todo orientadas a comunicaciones seguras a través de Internet.

Exportación e importación de claves

Ya hemos visto hasta ahora como cifrar y descifrar datos con algoritmos simétricos, lo que nos obliga a usar en los dos extremos una misma clave, o en su defecto, una misma contraseña para derivar la clave.

Esto lleva inherente un problema y es la necesidad de comunicar al otro extremo la clave (Ks) que queremos usar en cada sesión (Figura 1-A). Si estamos cifrando los datos por considerar al canal inseguro, es evidente que no podemos enviar la clave por el mismo canal, a menos que hagamos algo más. Una posible solución sería cifrar la clave con otra clave, pero estamos en el mismo problema: obligamos a ambos extremos a conocer una clave común inicial. En el entorno de Internet, que es donde queremos llevar a la práctica estas técnicas, eso casi nunca es posible.

Los algoritmos asimétricos son la base de la criptografía moderna

La solución pasa por el uso de algoritmos asimétricos en lugar de simétricos. En estos algoritmos, cada clave se compone de un par de claves pública (E) y privada (D). Con la clave E se pueden cifrar datos, y con la D se pueden descifrar. Están diseñados para que sea extremadamente difícil obtener D a partir de E. Así, una solución podría ser la que se ve en la Figura 1-B: Un ordenador A quiere comunicarse de forma segura con otro ordenador B, así que A crea una clave asimétrica (Ea,Da) y envía a B solamente la pública (Ea). Entonces los datos que B quiera transmitir a A, son cifrados con esta clave, de forma que solamente A puede descifrarlos. Que alguien escuche la clave pública de A no es un problema de seguridad, ya que es muy difícil obtener Da a partir de Ea. Si se quieren enviar datos de A a B, este último debe generar también una clave asimétrica y el proceso es idéntico.

Aunque este método es muy seguro, casi nunca se usa tal y como se ha descrito, sino con una variante: El mecanismo de clave asimétrica es muy lento, por lo que sólo se usa para el intercambio inicial de una clave simétrica tradicional (Ya sea DES, 3-DES, RC4, etc...) como se ve en la figura 1-C. Es más, con las funciones de cifrado de CryptoAPI que vimos en el número anterior, ya vimos que sólo podíamos usar claves simétricas. CryptoAPI usa internamente el cifrado asimétrico en las operaciones que lo requieran, pero no nos permite cifrar datos directamente con estos algoritmos con *CryptEncrypt*. La excepción está en el CSP *Enhanced* de Windows 2000, que permite usar RSA directamente sobre datos.

EJEMPLO PRÁCTICO

Siguiendo este último modelo (Figura 1-C), vamos a crear una pequeña aplicación tipo *chat*, aunque se puede modificar fácilmente para que soporte todo tipo de intercambio de datos. Se distingue entre un cliente y un servidor, que harán el papel de los nodos B y A de la figura, respectiva-

mente. Todos los detalles de la comunicación nos quedan ocultos por los componentes de C++ Builder, *TClientSocket* y *TServerSocket*. No se va a entrar en los detalles de estos componentes para centrarnos en el tema del cifrado.

El cliente es el que inicia la comunicación conectando con el servidor. En cuanto la conexión queda establecida, lo primero que hace es coger su clave pública de intercambio (o generarla si no existiera). Hay que fijarse que en este punto no podemos elegir el algoritmo a usar para intercambio de claves, sino que es fijo dependiendo del CSP usado. En nuestro ejemplo, para el CSP *Enhanced* es RSA, el cual usamos con clave de 1024 bits, que nos da una seguridad más que suficiente para prácticamente cualquier tarea. Para transportar esta clave al otro extremo, no nos vale con el *handle* de esta clave porque sólo tiene significado local, por lo que debemos exportar la clave en un buffer (*BLOB*, en la terminología de Microsoft), usando la función:

```
BOOL WINAPI CryptExportKey(
  HCRYPTKEY hKey,
  HCRYPTKEY hExpKey,
  DWORD dwBlobType,
  DWORD dwFlags,
  BYTE *pbData,
  DWORD *pdwDataLen
);
```

Por seguridad, no todas las claves pueden ser exportadas en limpio (sin cifrar), sino que se deben cifrar como veremos más adelante, con otra clave. Pero en este caso, como estamos exportando la clave pública de intercambio del cliente, no hay ningún problema en exportarla sin cifrar y enviarla al servidor. Todo este proceso se ve resumido en el listado 1, sin comprobaciones de errores.

Una vez que le llega al servidor el bloque de bytes que sabemos que representa la clave pública del cliente (Listado 2), se importa con la función *CryptImportKey*. Una vez importada, genera una clave de sesión simétrica. Para nuestro caso, se ha obligado a ambos extremos a usar el CSP *Enhanced*, que nos permite usar el algoritmo RC4 con clave de 128 bits, en lugar de los sólo 40 bits del CSP *Base*.

El servidor a continuación vuelve a exportar esta clave, pero esta vez usando para cifrar la clave de intercambio del cliente. Así nos aseguramos que nadie mas pueda descifrarla y por lo tanto, podemos cifrar y descifrar el resto de la conversación sin problemas con la clave de sesión conocida ya por ambos extremos.

LISTADO 1 Exportación de la clave pública de intercambio del cliente

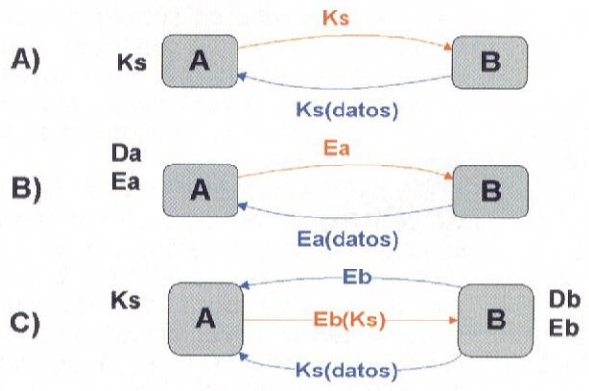
```
// Coger la clave de intercambio:
if (! CryptGetUserKey(
        hProv,                  // handle al CSP
        AT_KEYEXCHANGE,         // Queremos la clave de intercambio
        &hXchgKey) )
   // Ha fallado la funcion, ver por que:
        if (GetLastError()==NTE_NO_KEY)
        {       // La clave no existia. Crearla:
        ProcesaError(
          CryptGenKey(
          hProv,          // handle al CSP
          AT_KEYEXCHANGE, // Algorimo
          (LONGITUD_CLAVE)<<16 | CRYPT_EXPORTABLE, // Longitud de clave
          &hXchgKey       // Donde guardar la nueva clave
          ) ,"ERROR al crear nueva clave de intercambio");
        } else  return;    // Ha sido otro error:

// Exportar la clave a un bufer
bufer=new BYTE[MAX_STR];
len=MAX_STR;
CryptExportKey(
    hKey,           // Key a exportar
    0,              // Queremos sacarla en limpio, sin cifrar
    PUBLICKEYBLOB, // No cifrar los bytes de la clave
    0,              // Flags
    bufer,
    len );
// Mandarla al servidor:
Socket->SendBuf(&len,sizeof(DWORD));
Socket->SendBuf(bufer, len );
```

Certificados

Usando métodos como el expuesto, podemos tener total seguridad de que nuestra comunicación no puede ser modificada ni espiada por nadie, pero nos queda un problema por resolver: ¿Cómo puede un extremo de la comunicación asegurarse de que el otro es quien dice ser?

podéis encontrar los listados adicionales en nuestra Web: www.revistasprofesionales.com



Distintas formas de intercambiar claves.

LISTADO 2 El servidor cifra la clave de sesión con la pública del cliente

```
// Acabamos de recibir la clave publica de intercambio del cliente:
CryptImportKey(
   hProv,                 // Handle del CSP
   buf,                   // El BLOB donde esta la clave del cliente
   nBytes,                // Numero de bytes en BLOB
   0,                     // El blob no esta cifrado
   CRYPT_EXPORTABLE,   // Flags
   &hClienteXchgKey); // Donde guardar la clave

// Crear la clave de sesion para esta conexion:
CryptGenKey(
   hProv,              // Handle del CSP
   CALG_RC4,           // algoritmo simetrico: RC4
   CRYPT_EXPORTABLE, // Tamaño clave: por defecto
   &hSesionKey);

// Exportar la clave de sesion usando la del cliente:
CryptExportKey(
   hSesionKey,        // Clave a exportar
   hClienteXchgKey, // Clave con la que exportar
   SIMPLEBLOB,        // Clave de sesion
   0,                  // Flags
   buf,                // Buffer de salida
   &nBytes);          // Puntero a contador de bytes
// Enviar clave de sesion cifrada al cliente:
Socket->SendBuf(&nBytes,sizeof(DWORD) );
Socket->SendBuf(buf,nBytes);
```

Esto se soluciona con el uso de certificados digitales y con el común acuerdo de confianza en una serie de compañías (Verisign, Thawte, etc...) que se denominan *Certification Authorities* (CA). El sistema operativo mantiene una lista de certificados en los que confía, en los que se incluyen por defecto todas estas empresas bien conocidas. Una CA puede dar autoridad a otra nueva empresa para distribuir certificados, lo que irá indicado en el certificado de esta última. De esta forma, se crean árboles de confianza, cuya raíz debe de ser una CA conocida o el certificado no será considerado fiable.

Aunque un certificado contiene información como el rango de fechas de validez, nombre de la compañía, etc... el dato más importante sin duda es la clave pública del creador del certificado. De esta forma, una compañía puede firmar todo tipo de documentos, ficheros, etc... usando su clave privada que mantiene en secreto, y si sus clientes (que tienen su certificado, y por tanto su clave pública) verifican que la firma es correcta, eso significa sin duda que el fichero no ha sido modificado desde que lo firmó la compañía.

CryptoAPI guarda los certificados en lo que llama almacenes de certificados, que son divisiones lógicas para que cada usuario o aplicación del sistema puedan guardar sus propios certificados. Los mas comunes son *root* (almacén raíz), *My* (almacén personal del usuario), *Trust* (lista de certificados en los que se confía),...

Los certificados basan su seguridad en algoritmos asimétricos

Para la codificación de los certificados se pueden usar los estándares X.509 y el PKCS #7 de *RSA Laboratories*, que usan el conocido ASN.1. No hay que confundir codificación con cifrado. Cifrar los datos es realizar una transformación para evitar que sean entendidos sin las claves correspondientes. Codificar una estructura de datos es especificar esa estructura en forma de flujo de bytes. Esto es necesario porque cada máquina puede tener una representación interna de los datos distinta. Un ejemplo típico de esto son las máquinas con almacenamiento en extremista mayor y en extremista menor. Por ello, es necesario adoptar una forma estándar de codificar las estructuras complejas como éstas de forma que todas las máquinas las interpreten correctamente independientemente de su representación interna.

CryptoAPI incluye unas funciones útiles para la codificación y decodificación de campos ASN.1 de los certificados: *CryptDecodeObject* y *CryptEncodeObject*. En el listado 4 se ve un fragmento del ejemplo de firmado, donde muestra el método para decodificar el campo del certificado que indica el nombre del dueño del mismo.

CERTIFICADO DE PRUEBAS

Para poder probar el siguiente ejemplo, hay que crear un certificado de pruebas con el que poder firmar. Necesitaremos la herramienta de Microsoft *makecert.exe* disponible gratuitamente junto a otras a través de su web con el *Microsoft JDK* o con el *Platform SDK*. Ejecutando:

```
makecert -ss My -n "CN=Jose Luis"
```

donde lo que sigue a "CN=" es el nombre del destinatario del certificado, se crea y se instala en el contenedor de certificados personal ("My") un nuevo certificado. Señalar que los nombres no son sensibles a mayúsculas o minúsculas.

Estos certificados de pruebas, al no distribuirlos ninguna de las CA de confianza, se crean a partir del "Root Agency", una CA estándar en el cual no se confía por defecto ya que cualquiera puede crear un certificado a partir de esta sin ningún problema, como vemos.

También podemos crear el certificado en un fichero .cer e importarlo haciendo doble clic sobre él en el explorador:

```
makecert -n "CN=SoloProg" SoloProg.cer
```

Cuando nos pregunte en qué almacén de certificados queremos guardarlo, elegimos por ejemplo, el personal ("My").

Para comprobar que efectivamente el certificado ha sido creado y está guardado, podemos verlos en Panel de control -> Opciones de Internet -> Contenido -> Certificados.

Firmado digital

CryptoAPI dispone de unas funciones que llama "simplificadas" para realizar operaciones de firmado y verificación de mensajes, aunque éstas requieren una gran cantidad de parámetros, por lo que vamos a verlas detenidamente con otro ejemplo. En éste, vamos a firmar un texto usando un certificado de pruebas, guardando el mensaje firmado en un fichero, para luego poder verificar la firma. Como se ve en la figura 5, junto con un mensaje firmado se guarda suficiente información como para asegurarnos que no ha sido modificado sin necesidad de información adicional. Por lo tanto, no tenemos porqué fiarnos de un certificado para poder verificar un mensaje firmado con él.

Lo primero que hay que hacer es abrir un almacén de certificados y buscar uno que contenga clave privada para poder firmar. En el listado 3 se ve como realizar esto. Hay que resaltar que los nombres tanto del firmante como del almacén de certificados no son cadenas tipo lenguaje C estándar, sino UNICODE, por lo que debemos tenerlo en cuenta para transformarlos a sus correspondientes cadenas de *wchar_t*. Aprovechando la clase *AnsiString* de *Builder*, podemos hacer esto fácilmente así:

```
edStore->Text.WideChar(Almacen,1000);
```

Donde *edStore->Text* es la *AnsiString* con el texto normal, y *Almacen* es el "*wchar_t*".

La firma en sí, se realiza llamando a :

```
CryptSignMessage(
  PCRYPT_SIGN_MESSAGE_PARA pSignPara,
  BOOL fDetachedSignature,
  DWORD cToBeSigned,
  const BYTE *rgpbToBeSigned[ ],
  DWORD rgcbToBeSigned[ ],
  BYTE *pbSignedBlob,
  DWORD *pcbSignedBlob)
```

Como se ve, usa bastantes parámetros, siendo además el primero un puntero a una estructura con bastantes más campos, como se ve en la *figura 4*.

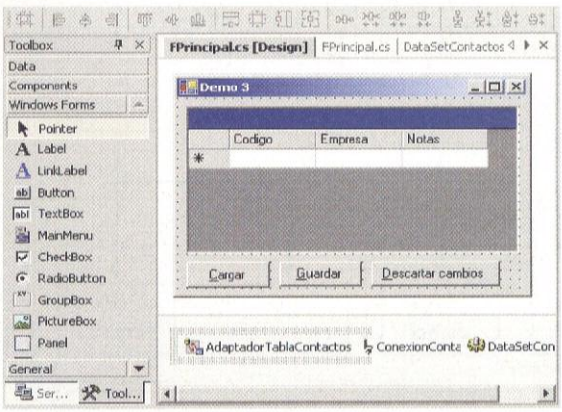
Algunos de los parámetros más importantes son:

- *pSignPara*: Puntero a una estructura con los parámetros más importantes, como son: tipo de codificación a usar (PKCS #7, X.509), puntero al certificado del firmante y el tipo de algoritmo a usar en el hash (MD5, SHA, etc..).
- *rgpbToBeSigned*: Es un array de punteros a bloques de datos a cifrar. Así se evita tener que llamar varias veces a esta función para firmar varios bloques con el mismo certificado.
- *pbSignedBlob*: El buffer de salida donde se guardará el mensaje firmado y codificado.

Este último bloque de datos es el que está listo para grabarlo en un fichero, enviarlo a través de la red o lo que sea necesario. Cuando en el mismo o en otro programa queramos verificar que la firma es correcta, es decir que el mensaje no ha sido manipulado desde que fue firmado, solo tenemos que llamar a:

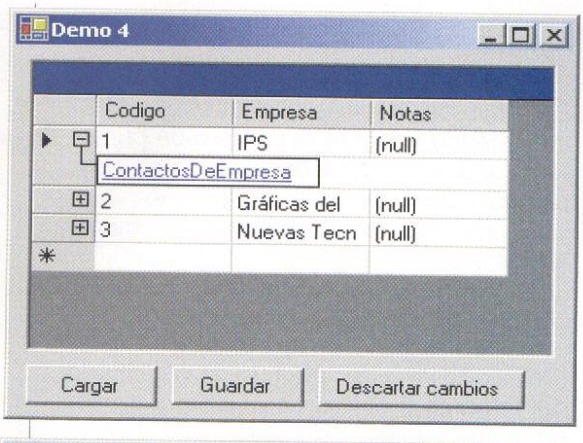
```
CryptVerifyMessageSignature(
  PCRYPT_VERIFY_MESSAGE_PARA pVerifyPara,
  DWORD dwSignerIndex,
  const BYTE *pbSignedBlob,
  DWORD cbSignedBlob,
  BYTE *pbDecoded,
  DWORD *pcbDecoded,
  PCCERT_CONTEXT *ppSignerCert
)
```

podéis encontrar los listados adicionales en nuestra Web: www.revistasprofesionales.com

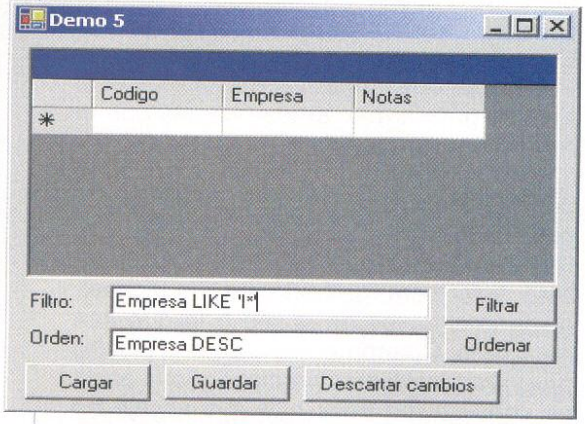


La aplicación de ejemplo es un "chat seguro".

El certificado de pruebas.



Parámetros a rellenar para firmar un mensaje.

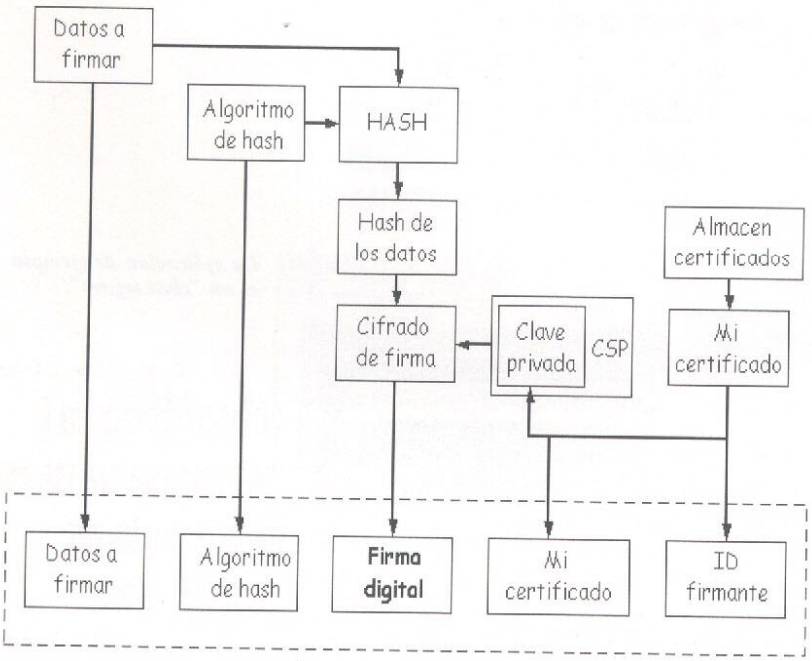


Donde el *pbSignedBlob* es un buffer que contiene el mensaje firmado. Esta función además decodifica el texto del mensaje y lo guarda en *pbDecoded*. Si queremos saber de quién es el certificado usado para firmar, solo tenemos que pasarle el puntero *ppSignerCert*, con lo que nos dará el certificado codificado que contiene el mensaje. En el listado 4 se ve cómo decodificar el campo del propietario del certificado, para obtener la cadena de texto *char** con el nombre del firmante.

El programa completo se incluye en el CD, junto con un mensaje firmado por el autor como ejemplo. Para verificar el mensaje, no hace falta el certificado de forma independiente, ya que como se ve en la figura 5, un mensaje por sí solo contiene la clave pública necesaria del certificado.

Para poder compilar programas que usen estas funciones con C++ Builder, hay que usar la cabecera *<wincrypt.h>* e incluir en el proyecto la librería *crypt32.lib* que se encuentra en el directorio "lib\psdk".

Contenido de un mensaje firmado digitalmente.



Alternativas a CryptoAPI

A pesar de las ventajas de usar CryptoAPI, también existen en la Red algunas librerías gratuitas, principalmente para lenguaje C/C++, que realizan las mismas tareas que el CryptoAPI, pero de forma independiente a éste. Esto puede ser interesante si se quieren usar algoritmos y longitudes de claves mayores de los que permiten los CSPs de una determinada versión de Windows, o incluso más algoritmos de los que estos implementan (PGP, por ejemplo), con la ventaja adicional de independencia del sistema operativo. Aquí se pueden encontrar algunas de estas librerías gratuitas:
http://www.thefreecountry.com/developercity/encryption.shtml

Conclusión

Aunque solo hayamos usado una pequeña parte de las posibilidades de esta completa y compleja API de Windows, hemos repasado los mecanismos mas básicos de criptografía que pueden ser muy útiles a cualquier aplicación, sobre todo comparado con la mínima dificultad y tiempo necesario para sus implementaciones. Además, con la tendencia actual a una arquitectura cada vez más distribuida en las aplicaciones, y teniendo en cuenta los riesgos de la Red, se va haciendo cada vez más necesario y fundamental un uso de estas herramientas para conseguir un mínimo de seguridad.