AHORA POR SÓLO 4,50 €

[...]ERA REVISTA DE PROGRAMACIÓN EN CASTELLANO

SO[...] PROGRAMADORES

AÑO VIII. 2.ª ÉPOCA • Nº 93 • UNA PUBLICACIÓN DE: REVISTAS PROFESIONALES S.L • Precio: 4,50 € (España) (IVA incluido)

DISEÑO POR CONTRATO

MICROSOFT NET
ADO.NET (INICIACIÓN)
EMAIL CON ASP.NET
MONITOR DE ARCHIVOS

MICROSOFT OFFICE
TRABAJO CON EXCEL

CRIPTOGRAFÍA
CRYPTO API

3D/OPENGL
SISTEMAS DE PARTÍCULAS

JAVA
ESPECIFICACIÓN JAVA 2

MÓVIL
MENSAJERÍA CON EMS

CANAL PANDA
SOCIEDAD VIRUS/PC

Noticias, Libros, Dudas, Web´s de interés...

Criptografía en Windows (I)

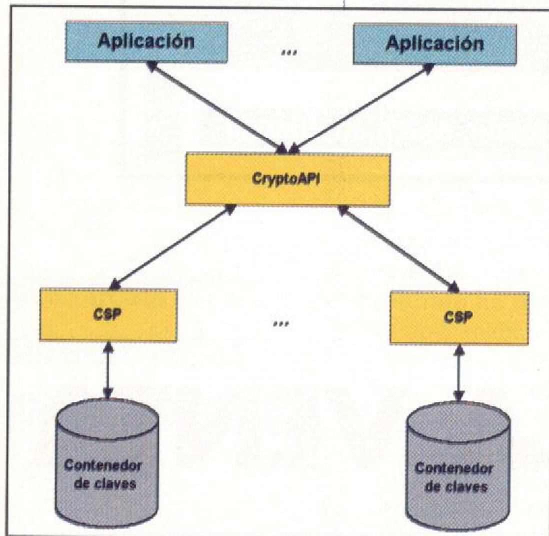
José Luis Blanco Claraco

A través de la CryptoAPI de Windows tenemos acceso a gran cantidad de potentes sistemas de cifrado, firmado digital, manejo de certificados, etc.. En este artículo empezaremos a estudiar sus posibilidades y veremos cómo usarlas.

Introducción

CryptoAPI es la interfaz que Windows nos ofrece para acceder a los servicios criptográficos y de seguridad del sistema. Sin embargo, estas operaciones no las realiza el sistema operativo en sí, sino unos módulos llamados CSP (Cryptographic Service Providers) de los cuales suele haber varios instalados en cada máquina. En la figura 1 se ve la relación entre las aplicaciones de usuario y los CSPs. El programador debe especificar qué CSP quiere usar, o en su defecto, qué tipo de servicios criptográficos usará su aplicación, de forma que el sistema operativo pueda asignarle un CSP apropiado. Existe un CSP por defecto, que suele ser el "Microsoft Base Cryptographic Provider", con capacidades criptográficas mas bien reducidas. Los CSP se caracterizan por soportar distintos algoritmos de cifrado, por la máxima longitud de las claves, etc. Una vez elegido un CSP, debemos elegir el contenedor de claves del CSP con el que queremos trabajar o crear uno nuevo. Con esto se aíslan las claves de una aplicación de las de otra.

CryptoAPI hace de interfaz para acceder a los servicios criptográficos



Hasta principios del año 2000, la ley de exportación de EEUU prohibía la salida del país de CSPs (y en general, cualquier programa criptográfico) que permitiera longitudes de claves de más de 56 bits para algoritmos simétricos o mas de 512 bits para RSA. A partir de entonces, la ley fue modificada y actualmente sí se pueden exportar al público en general, con excepción de Cuba, algunos países de Oriente Medio y zonas antes controladas por talibanes, los que necesitan una licencia especial.

Uso práctico

A lo largo de este artículo, veremos detalladamente algunas operaciones que podemos realizar con CryptoAPI, sin pararnos a detallar el funcionamiento interno de cada algoritmo de cifrado, sino la forma de utilizarlos. Al presentar CryptoAPI una misma interfaz independientemente del algoritmo de cifrado usado, podemos usar cualquiera de los algoritmos por complejos que sean de la misma manera, lo cual hace que cambiar de algoritmo o de longitudes de clave, en caso necesario, sea una tarea trivial.

Aunque se muestran algunos ejemplos, en el CD se acompaña un pequeño programa en C++ Builder que desarrolla unos pocos ejemplos más, incluyendo cifrado y descifrado de ficheros con contraseña, usando el proveedor *Enhanced* de Microsoft y el algoritmo RC4.

INICIALIZACIÓN

Para usar las funciones de CryptoAPI en un programa en C, debemos incluir la cabecera *wincrypt.h*, donde están declaradas las funciones y constantes que vamos a usar.

Lo primero que un programa debe hacer antes de usar las funciones de CryptoAPI, es elegir qué CSP va a usar y que contenedor de claves dentro del CSP. Para eso se usa la función *CryptAcquireContext*, que nos devuelve un *handle* del proveedor:

```
BOOL WINAPI CryptAcquireContext(
  HCRYPTPROV *phProv,
  LPCTSTR pszContainer,
  LPCTSTR pszProvider,
```

```
    DWORD dwProvType,
    DWORD dwFlags
);
```

El primer parámetro es el puntero a donde queremos que nos devuelva el *handle* del proveedor. La cadena pszContainer debe ser el nombre del CSP que se quiere usar. En la tabla 1 se muestran algunos. Si solo se quiere que el CSP sea de un tipo determinado (es decir, que maneje codificación con claves publicas, privadas, firmado,...) y nos da igual que proveedor concreto se use, podemos dejar el nombre a NULL, colocando el tipo del proveedor en *dwProvType*, p.ej: PROV_RSA_FULL, nos asigna un CSP capaz de intercambiar claves y firmar usando RSA.

Para seleccionar un contenedor de claves existente, debe ponerse su nombre como cadena de texto en el parámetro *pszContainer*. Si se quiere usar el contenedor por defecto del CSP, se puede dejar a NULL. Este contenedor por defecto varía para cada usuario que inicie la sesión de Windows.

Esta función sirve además para el mantenimiento de contenedores de claves dentro de un CSP. Por ejemplo, se puede crear un nuevo contenedor de claves, colocando el nombre que se le quiere dar en *pszContainer* y usando el parámetro CRYPT_NEWKEYSET en dwFlags. También, por razones de seguridad, una aplicación podría querer borrar un contenedor de claves para que otras aplicaciones no puedan acceder a él. Esto se hace de la misma forma, pero usando el parámetro CRYPT_DELETEKEYSET en dwFlags. En el *listado 1* se ve una secuencia típica para abrir un contenedor de claves o crearlo si este no existe.

ERRORES

Aunque cada una de las funciones del CryptoAPI puede fallar por una multitud de razones, por lo menos todas comparten un sistema común de comunicarlo: Todas devuelven un tipo *BOOL*, que es *false* si hubo algún error, pudiendo obtener el código de error con *GetLastError()*. Gracias a esto es posible y recomendable sobre todo en las primeras fases de pruebas de una aplicación, crear alguna función de proceso de errores, al estilo de la que se ha usado en el ejemplo que se adjunta en el CD:

```
ProcesaErrores( [ Llamada a funcion
CryptoAPI], "Mensaje de error a mostrar");
...
void TForm1::ProcesaError(BOOL res,char *msg)
{
        AnsiString st;
        if (res) return;
    st.sprintf("%s (ERROR Nº=0x%X)" ,msg,
GetLastError());
        throw Exception(st);
}
```

El significado del valor numérico del error se puede ver en la documentación de Microsoft (http://msdn.microsoft. com/library) o en la cabecera estándar *winerror.h*

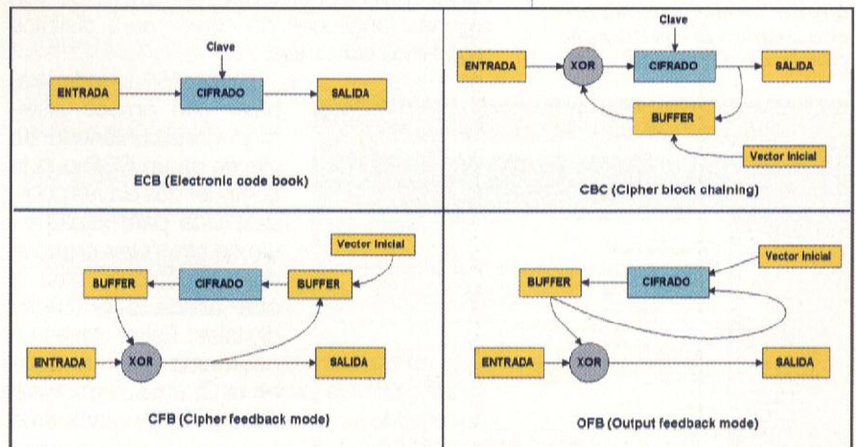
FUNCIONES DE HASH

CryptoAPI contiene una serie de funciones que nos permiten usar algoritmos de *hash* (o *digest, "resumen"*). Estos consisten en algoritmos que a partir de un bloque de datos, calculan un *hash* de los datos, normalmente de un tamaño fijo y relativamente pequeño. Un típico *checksum* o *CRC* son ejemplos sencillos de funciones de este tipo. A partir del *hash*, no se puede obtener el bloque de datos original, pero su utilidad es asegurar al receptor de un mensaje que éste no ha sido modificado desde que salió del emisor, ya sea para asegurar que no hubo errores o para evitar modificaciones malintencionadas. Para realizar esto, se añade al mensaje el *hash* del mensaje, de forma que el receptor puede volver a aplicar el mismo algoritmo y comprobar que son idénticos.

Otra utilidad en el campo de la seguridad, es el que se le da en algunas distribuciones de Linux para el fichero de contraseñas del sistema: En lugar de guardar las contraseñas tal cual en un fichero, se guardan sus *hash* (suele ser MD5), de forma que es muy difícil recuperarla en limpio, pero comprobar si un usuario ha escrito su contraseña correcta es tan sencillo como aplicarle el mismo algoritmo y compararla con la almacenada. Usando CryptoAPI, es fácil implementar este modelo de contraseñas en cualquier aplicación.

Algoritmos de este tipo que podemos usar

Modos de funcionamiento de algoritmos simétricos



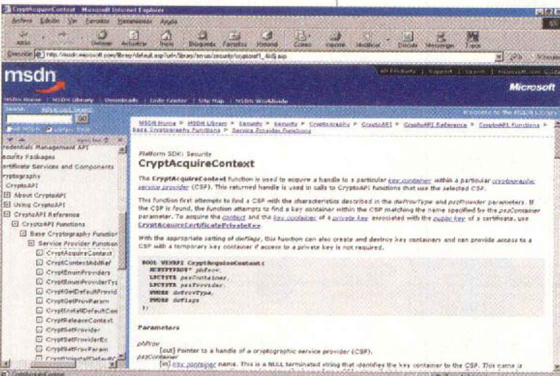
con CryptoAPI son:

- MD2,MD4,MD5: Sucesivas versiones del algoritmo *Message Digest*. Todos dan por resultado un *hash* de 128 bits.
- SHA: El *Secure Hashing Algorithm*, más reciente y considerado como más seguro que los anteriores. El resultado es de 160 bits.

Una vez tengamos abierto un *handle* a un CSP que soporte funciones *hash*, podemos crear un objeto *hash* con la función *CryptCreateHash*. Luego debemos pasarle los datos a los que queremos aplicar el algoritmo, con la función *CryptHashData*, y obtener el *hash* en sí llamando a *CryptGetHashParam*. Un ejemplo de todo esto se ve en el listado 2. En el ejemplo se conoce de antemano que el hash SHA ocupa 20 bytes, por lo que se reserva un buffer de ese tamaño. Si no se conoce el tamaño del hash a devolver, se puede dejar el puntero al buffer destino a NULL, con lo que se nos devolverá en *LonHash* el número de bytes necesarios, y podremos reservar la memoria dinámicamente.

TIPOS DE ALGORITMOS Y CLAVES

Hay dos tipos de algoritmos de cifrado y dos tipos de claves correspondientes: Asimétricos y simétricos. Los algoritmos simétricos usan una misma clave para cifrar y para descifrar, mientras que en los asimétricos, se usa una clave pública para cifrar, y otra clave privada, para descifrar. Cuando un sistema quiere enviar a otro información cifrada con este sistema, debe antes recibir del receptor su clave pública. Esta clave se transmite por el canal de comunicación (siempre debemos pensar que no es seguro) sin problemas, ya que lo importante es que los datos codificados por el emisor no pueden ser descifrados si no es con la clave privada, que la guarda el receptor. La seguridad del sistema asimétrico radica en la supuesta imposibilidad, o gran dificultad, de obtener la clave privada conociendo la pública. Además, en general a mayor tamaño de clave mayor seguridad. En la tabla 2 se ven algunas longitudes de claves para distintos algoritmos simétricos.



En http://msdn.microsoft.com/library encontrará la especificación completa de CryptoAPI

CryptoAPI puede trabajar con ambos sistemas. Cada contenedor de claves de un CSP puede contener dos claves públicas: Una para intercambio de otras claves (privadas, simétricas, etc.) y otra usada para firmas digitales. Estas claves se guardarán internamente en el CSP de forma cifrada y supuestamente segura, con lo que se evita que la aplicación tenga que preocuparse de almacenarlas. El algoritmo asimétrico de cifrado que se usará con estas dos claves depende del CSP usado. Para el CSP por defecto (*Microsoft Base Cryptographic Provider*) el algoritmo usado es el RSA Key Exchange con clave de 512 bits, que usa el conocido algoritmo RSA.

Cuando se crea un contenedor de claves, éste está vacío y debemos crear estas claves públicas manualmente. Hay que resaltar que el contenedor solo guardará estas claves asimétricas. Las claves simétricas deberán ser exportadas y guardadas manualmente. Para crear una clave, independientemente del algoritmo usado, se usa la función:

```
BOOL WINAPI CryptGenKey(
  HCRYPTPROV hProv,
  ALG_ID Algid,
  DWORD dwFlags,
  HCRYPTKEY *phKey
);
```

Que nos devuelve en *phKey* un *handle* a la nueva clave. En el parámetro *algid* es donde especificamos el tipo de algoritmo para el que queremos crear la clave. Algunos algoritmos que podemos usar son:

SIMÉTRICOS:

- DES (Data Encryption Stantard): Sistema adoptado por el gobierno de EEUU en los años 70 como estándar para documentos no secretos. Siempre ha estado rodeado de polémica desde el comienzo, por ser considerado fácilmente atacable por organizaciones con suficiente capacidad computacional. Usa tamaño de clave fija de 56 bits y cifra en bloques de 64 bits, aunque se puede usar como se verá abajo en otras configuraciones.
- RC2/RC4 (Ron's Code 2/4): Desarrollados por Ronald Rivest para la compañía RSA Security. Permiten variar la longitud de la clave, y se consideran más seguros y mucho mas rápidos que DES. RC2 esta orientado a cifrado por bloques, y RC4 de flujos de bytes.
- Triple-DES: Mejora de DES. Se usa una secuencia cifrar-descifrar-cifrar DES, pudiendo usar una clave distinta en cada etapa o usando una misma clave en los cifrados y otra distinta en el descifrado. Aunque se puede atacar, se considera totalmente impracticable con los ordenadores actuales. Las claves de cada etapa siguen siendo de 56 bits, lo que da un total de 112 o 168 bits de clave, según se usen 2 o 3 subclaves distintas en cada etapa.

ASIMÉTRICOS:

- DSA (Digital Signature Algorithm): Publicado por el NIST como estandar de firmado digital. Está basado en el logaritmo discreto.
- RSA: Obtiene el nombre de sus tres inventores (Ronald Rivest, Adi Shamir y Leonard Adleman) que lo desarrollaron en 1977. Se basa en teoría de números y en el problema de la factorización (multiplicar dos primos grandes es fácil, factorizarlos casi imposible). Usa una clave privada y otra publica. Se puede usar para intercambio seguro de claves ,auntenticación y cifrado.
- D-H: También obtiene el nombre de sus inventores, Diffie y Hellman en 1976. Se basa en el problema del logaritmo discreto. Se usa para intercambiar claves en canales de comunicación inseguros.

GENERAR CLAVES A PARTIR DE CONTRASEÑAS

Con la función *CryptGenKey* se generan claves de forma aleatoria. Por lo tanto si se cifran datos con una clave simétrica la aplicación deberá guardar también esta clave para poder descifrarlos luego. En algunas ocasiones puede ser más cómodo o práctico generar la clave a partir de una contraseña, número PIN, etc. Para realizar esto, hay que seguir estos pasos:

- Crear un objeto hash temporal, como se describía más arriba.
- Añadir a este objeto hash un bloque de datos con la contraseña.
- Llamar a la función:

```
BOOL WINAPI CryptDeriveKey(
  HCRYPTPROV hProv,
  ALG_ID Algid,
  HCRYPTHASH hBaseData,
  DWORD dwFlags,
  HCRYPTKEY *phKey
);
```

Que funciona igual que *CryptGenKey*, solo que en lugar de generar la clave aleatoriamente, lo hace a partir del objeto hash pasado en el parámetro *hBaseData*. Una vez obtenido el *handle* a la clave simétrica, se puede eliminar el objeto *hash* y usar la clave de forma normal. Es importante usar los mismos CSPs, algoritmos (de cifrado y de *hash*) y longitudes de clave para asegurar que la clave sea idéntica cada vez que se genere. Recordar también que se distinguen mayúsculas y minúsculas.

CIFRADO SIMÉTRICO

CryptoAPI solo nos permite usar algoritmos simétricos para cifrar datos directamente. Como ya se ha dicho arriba, estos algoritmos necesitan usar la misma clave en la decodificación y en la codificación. Para el cifrado de los datos se usa la función:

```
BOOL WINAPI CryptEncrypt(
  HCRYPTKEY hKey,
  HCRYPTHASH hHash,
  BOOL Final,
  DWORD dwFlags,
  BYTE *pbData,
  DWORD *pdwDataLen,
  DWORD dwBufLen
);
```

En donde:

- El parámetro *hKey* determina la clave (y por lo tanto el algoritmo) a usar para el cifrado.
- *Final* normalmente será *true*, a menos que queramos introducir los datos en fragmentos, lo que se hará llamando a esta función con *Final* a *false* hasta el último fragmento, en el que se pondrá a *true*.
- Los datos a cifrar y su longitud se pasan en *pbData* y *pdwDataLen*, respectivamente. Los datos cifrados se devuelven en el mismo buffer, y su longitud en la misma variable. Los datos cifrados serán de longitud igual o un poco mayor a la de los datos en claro.

Además, se permite pasar un objeto de *hash*, de forma que se realice el *hash* y el cifrado al mismo tiempo (Si no se quiere usar *hash*, se deja este parámetro a 0). Si no conocemos cuál va a ser la longitud de los datos cifrados, podemos dejar *pbData* a *NULL*, con lo que no nos devuelve el bloque cifrado, pero sí cual será su longitud en *pdwDataLen*. En cuanto al descifrado, se realiza de forma similar, con la función:

```
BOOL WINAPI CryptDecrypt(
  HCRYPTKEY hKey,
  HCRYPTHASH hHash,
  BOOL Final,
  DWORD dwFlags,
  BYTE *pbData,
  DWORD *pdwDataLen
);
```

Si todo el descifrado se va a realizar de una sola vez, se pone *Final* a *true*, el texto cifrado en *pbData*, y en el mismo buffer se obtiene su

descifrado. En este caso, la longitud de los datos descifrados será igual o menor que la de los datos cifrados, luego no será necesario comprobar si se necesita más espacio que el que ocupa el buffer. Si se le pasa un objeto de *hash*, automáticamente se le añaden los datos descifrados, con lo que se puede llamar inmediatamente a *CryptGetHashParam* para obtener el *hash* en sí. Esto es útil para saber con certeza que los datos que se han descifrado se han hecho con la clave correcta, aunque la mayoría de las ocasiones, el mismo *CryptDecrypt* da un error de *NTE_BAD_DATA* si se usa una clave errónea. Como ejemplo, en el programa adjunto en el CD, se cifra un fichero y se guarda junto con los datos cifrados el valor del *hash*. Al descifrar el fichero, se comprueba que el *hash* devuelto por *CryptDecrypt* es el mismo que el que esta almacenado en el fichero. Si no es así, la contraseña usada no es la correcta o el fichero cifrado ha sido modificado.

LONGITUDES DE CLAVE

Aunque los algoritmos como DES o triple DES son de tamaño de clave fija, hay otros como RC2/4, RSA, etc. en los que podemos elegir el tamaño de la clave, dentro de lo que nos lo permita el CSP (Ver *tabla 2*). El tamaño hay que especificarlo al crear la clave. Volvamos a la función de creación de claves:

```
BOOL WINAPI CryptGenKey(
  HCRYPTPROV hProv,
  ALG_ID Algid,
  DWORD dwFlags,
  HCRYPTKEY *phKey
);
```

El parámetro *dwFlags*, es de 32 bits, dos palabras. Pues la longitud de la clave en bits se debe poner en la palabra alta de este parámetro, a menos que queramos usar la longitud por defecto del CSP, en cuyo caso sólo tenemos que dejarlo a cero. Si se usa la función *CryptDeriveKey* el mecanismo es idéntico. Un ejemplo: para crear una clave de 128 bits y además exportable (en el siguiente artículo veremos para qué sirve esto), debemos poner en el parámetro *dwFlags* un *OR* lógico con los dos valores:

(128 << 16) | CRYPT_EXPORTABLE

MODOS DE FUNCIONAMIENTO

Un algoritmo de cifrado simétrico funciona tratando un bloque de bytes de entrada para transformarlo en otro bloque de salida cifrado. Pero esta forma de funcionar presenta ciertos problemas. Por ejemplo, si se está cifrando un fichero grande, se tiene que dividir este en fragmentos que el algoritmo pueda procesar. Si el fichero contiene varios de estos fragmentos iguales, esto se notará en la salida cifrada, facilitando un posible ataque. Esto es así porque si dos bloques a la entrada son iguales, los bloques cifrados a la salida también son iguales, lo que puede facilitar obtener la clave por fuerza bruta. Además se podrían introducir bloques extra en el flujo de datos cifrados, sin que el receptor lo notase. Para evitar esto se definen unos modos de funcionamiento (ver figura 2), que son aplicables a cualquier algoritmo simétrico:

- ECB (*Electronic code book*). Es el modo de funcionamiento más sencillo: Un bloque de entrada se procesa para dar un bloque de salida cifrado, siendo cada bloque independiente de los demás, si es que existen. Su nombre viene de que es equivalente a usar una tabla de valores a sustituir para cada posible bloque de entrada.
- CBC (*Chain block cipher*): Este método se basa en hacer un XOR lógico de los datos a cifrar con el ultimo bloque cifrado. Con esto se elimina la posibilidad de que el mismo bloque de entrada produzca el mismo bloque de salida, ya que cada bloque cifrado depende de todos los anteriores. Al descifrar, se realiza la operación complementaria, realizando un XOR del bloque actual descifrado con el anterior bloque cifrado. Además, se define un vector inicial de bytes, que se usará para el XOR en el primer paso, cuando no hay datos anteriores.
- CFB (*Cipher feedback mode*). La cualidad de este modo de funcionamiento es que no se depende del tamaño de bloque del algoritmo simétrico, además de eliminar también el problema del ECM. Por ejemplo, si lo configuramos para que funcione de 8 en 8 bits, se va

TABLA 1 Los CSPs de Microsoft

CSP	W9X	W2000	
Microsoft Base Cryptographic Provider	X	X	Versión inicial (W95 e IE 3.0)
Microsoft Strong Cryptographic Provider		X	Mejora del CSP Base (Más algoritmos y longitudes de claves)
Microsoft Enhanced Cryptographic Provider	X	X	Mejora del CSP Strong
Microsoft DSS Cryptographic Provider	X	X	Hash, firmado, certificados.
Microsoft Base DSS and Diffie-Hellman Cryptographic Provider	X	X	Mejora del anterior. Incluye intercambio de claves.
Microsoft DSS and Diffie-Hellman/Schannel Cryptographic Provider	X	X	Cifrado clave publica, intercambio claves.
Microsoft RSA/Schannel Cryptographic Provider		X	Hash, firmado, certificados.

TABLA 2 Algunas longitudes de clave

PROVEEDOR	ALGORITMO	LONGITUD MIN.	LONGITUD MAX.	LONGITUD DEFECTO
MS Base	RC2/RC4	40	56	40
MS Enhanced	RC2/RC4	40	128	128
MS Strong	RC2/RC4	40	128	40
DSS Base	RC2/RC4	40	56	40
DSS Enhanced	RC2/RC4	40	128	40
MS Base	DES	56	56	56
MS Enhanced	DES	56	56	56
MS Strong	DES	56	56	56
DSS Base	DES	56	56	56
DSS Enhanced	DES	56	56	56
MS Enhanced	3DES	112	112	112
MS Strong	3DES	168	168	168
DSS Enhanced	3DES	168	168	168

generando un byte codificado, por cada byte de entrada. Se usa en algunos dispositivos hardware que necesitan cifrar byte a byte.

- OFB (*Output feedback mode*). Parecido al anterior, como se ve en el esquema, pero los bloques para el XOR se van generando a partir del vector inicial, independientemente de los datos a cifrar.

Por lo tanto, el modo de ECB es poco recomendable, y de hecho, al crear una clave simétrica, por defecto está en modo CBC. Podemos cambiar manualmente el modo de funcionamiento sobre el *handle* de una clave, así:

```
DWORD modo= CRYPT_MODE_XXX;
CryptSetKeyParam(
hKey,              // Handle a la clave
KP_MODE, // Param. a cambiar
(BYTE*)&modo,      // Nuevo modo
      0                  // Flags
);
```

Donde los distintos modos se representan por las constantes:

CRYPT_MODE_ECB (Electronic code book)

CRYPT_MODE_CBC (Chain block cipher)

CRYPT_MODE_CFB (Cipher feedback mode)

CRYPT_MODE_OFB (Output feedback mode)

Si usamos los modos CFB o OFB debemos especificar el numero de bits por cada bloque de entrada / salida. Por ejemplo, para ponerlo a un byte (8 bits) se realizaría así:

```
DWORD bits= 8;
CryptSetKeyParam(
hKey,              // Handle a la clave
KP_MODE_BITS,      // Param. a cambiar
(BYTE*)&bits,      // Nuevo valor
      0                  // Flags
);
```

Conclusión

Hasta ahora hemos introducido la estructura de CryptoAPI y las funciones básicas de cifrado y *hash*, así como la creación de claves. Con esto, se puede crear ya un sistema de cifrado/descifrado de ficheros basado en petición de contraseña bastante seguro, como el que se ha implementado en el ejemplo. Sin embargo, surgen nuevos problemas, como es la forma de enviar la contraseña a través de la red con seguridad, sin tener ninguna clave común inicialmente. En el siguiente artículo, resolveremos éste y otros problemas, como la autenticación y uso de certificados, usando más herramientas que nos proporciona esta API. SP

TABLA 3 Algunos valores comunes de algid

VALOR	ALGORITMO A USAR CON LA CLAVE
CALG_DES	DATA ENCRIPTION STANDARD (CIFRADO)
CALG_3DES	TRIPLE DES (CIFRADO)
CALG_RC2	RC2 (CIFRADO)
CALG_RC4	RC4 (CIFRADO)
CALG_RSA_SIGN	RSA (FIRMA DIGITAL)
CALG_DSS_SIGN	DSA (FIRMA DIGITAL)
CALG_RSA_KEYX	RSA (INTERCAMBIO DE CLAVES)
CALG_DH_SF	DIFFIE-HELLMAN (INTERCAMBIO DE CLAVES)
CALG_DH_EF	DIFFIE-HELLMAN (INTERCAMBIO DE CLAVES)
CALG_KEA_KEYX	KEY EXCHANGE ALGORITHM FORTEZZA (INTERCAMBIO DE CLAVES)
CALG_MD2	MESSAGE DIGEST 2 (HASH)
CALG_MD4	MESSAGE DIGEST 4 (HASH)
CALG_MD5	MESSAGE DIGEST 5 (HASH)
CALG_SHA	Secure Hash Algorithm (HASH)
CALG_HMAC	HMAC (HASH CON CLAVE)
CALG_MAC	MESSAGE AUTHENTICATION CODE (HASH)