# Microsoft Crypto API

CryptoAPI provides an abstraction layer that isolates you from the algorithm used to protect the data. An application will refer to contexts and keys and make calls to special functions that act as drivers for the encryption servers installed on the machine. These servers are called Cryptographic Service Providers (CSP) and are the modules that do the dirty work of encoding and decoding data.

The CryptoAPI interface to the application is composed of three classes of functionality: simplified cryptographic functions, certificate functions, and low-level services. Simplified cryptographic functions include high-level functions for creating and using keys and for encrypting and decrypting information.

At a lower programming level, there are other functions whose arguments require a specific CSP. Unless it is critical to your application, you should not access CSPs directly, and you should also avoid using features that are particular to a given provider. There may be applications that need their own CSP for one of a number of reasons, one of which might be special security concerns.

To ensure privacy, all the data a CSP manipulates (especially keys) is returned to the caller as opaque handles and remains inaccessible at the application level. Moreover, programs cannot affect the way the data is actually encoded. A program must limit itself to passing in the data and specifying the type of the encryption required. A provider can always return a type that explains what and how it is able to do. Different providers may use the same algorithms but adopt different logic for padding and different key sizes.

Each CSP is associated with a database of key containers that stores all the private and public keys for the users accessing that computer. Each container has a unique name that is the key to the CryptoAPI programming world. If this database doesn't exist, all CryptoAPI functions will fail. It's common to have a default key container with the logon name of each user. An application, however, may create a custom key container and key pairs during installation, assigning them its own name. Since the type of the provider affects the behavior of the cryptographic functions, two connected applications should use the same CSP, or at least CSPs with a common subset of functions.

However, before adding cryptography to real-world applications, you must become familiar with terms such as context, session keys, exchange keys, and signature keys. A context represents a session that's been established between CryptoAPI and the client application. To begin, you need to acquire a context. In doing so, you pass in the name of the key container you need and the name of the provider to which you want to connect. The handle you obtain must be used in all subsequent calls to the CryptoAPI routines.

A session key comes into play when it's time to encrypt or decrypt data. Session keys are volatile objects whose actual bytes never leave the CSP for reasons of privacy and security. The session key determines how a file is encrypted and must be inserted in a ciphered file to allow decryption. If you need to bring a session key out of the CSP for exchange or storage purposes, you need key blobs. A key blob is a binary chunk of data and may be considered an encrypted and exportable version of the key itself.

The safekeeping mechanism is completed with exchange keys. They are key pairs (one public and one private) that take care of encrypting the session keys inside the key blobs and handling the digital signatures. A session key is created dynamically from the information stored in the user's key container. Once you have a session key,

you are ready to make the calls that will scramble the bytes of the file.

Supporting cryptography in your applications doesn't require you to be familiar with details of RSA or DES algorithm, or subtle concepts such as public and private keys.

The basic routines cover four main areas: CSP, keys, hash objects, and signatures. Let's start with CSP. The first function you call in a typical CryptoAPI application is CryptAcquireContext, which provides a handle to the specified provider as a returned argument.

```
BOOL CryptAcquireContext(
HCRYPTPROV *phProv,
    LPCTSTR pszContainer,
    LPCTSTR pszProvider,
    DWORD dwProvType,
    DWORD dwFlags );
```

The returned HCRYPTPROV handle is proof that a working session was established between the application and the CSP. You may specify a particular CSP to which you want to connect, as well as the key container you want to access. You do this through the pszProvider and pszContainer arguments. You may restrict your query by indicating what type the provider should be. The function first attempts to locate a provider with the given name and characteristics. If it's successful, the function will then search for a pszContainer key container in that CSP. The dwFlags parameter allows you to create a new key container or delete an existing one. The handle obtained should be released via CryptReleaseContext.

None of the functions dealing with CSP (except for CryptAcquireContext) should be used at the application level; they must be called only from within system-level and administrative tools. CryptAcquireContext gives you a handle to an encryption provider. You can use this handle to create or derive keys.

There are two methods for generating session keys: using a random seed through CryptGenKey or using a hash method via CryptDeriveKey. CryptGenKey requires four parameters as shown below:

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

The first is a handle to the selected CSP, while the last is a buffer that will contain a valid key handle. The algid argument specifies a value identifying the algorithm to be used for creating the key. The algorithms available depend upon the provider's capabilities. The Microsoft RSA Base Provider offers two possible choices: CALG_RC4 and CALG_RC2. The key may or may not be exportable. Session key is a volatile object unless you assign it the CRYPT_EXPORTABLE attribute. This flag allows the key to originate a key blob—nothing more than a nonvolatile and encrypted version of the key. Often a key blob is attached to a message or file.

Exporting a key is a two-step process that involves another special object, the exchange key. The first step consists of asking the CSP for the user's public key viaCryptGetUserKey. Once you've got an exchange key handle, you are ready to get a key blob. The following does just that:

```
HCRYPTKEY hXKey=NULL;
LPBYTE pBuf=NULL;
CryptGetUserKey( hProv, AT_KEYEXCHANGE, &hXKey );
CryptExportKey( hKey, hXKey, SIMPLEBLOB, 0, NULL, &dwSize);
pBuf = (LPBYTE) malloc(dwSize)
CryptExportKey( hKey, hXKey, SIMPLEBLOB, 0, pBuf, &dwSize);
```

The function CryptExportKey returns the key blob, which is stored in the pBuf buffer. When the pBuf argument is NULL, the same function will instead return the amount of required memory. To decode a key blob and finally decipher the document, the destination user calls CreateImportKey, passing in the key blob and receiving an HCRYPTKEY handle.

A session key may also be generated through hashing. You just need to provide a stream of data to work on, and you get back a previously initialized hash object. The functions involved are CryptCreateHash (whose purpose is creating a hash object), CryptHashData (which hashes in the given data), and CryptDeriveKey (which generates an HCRYPTKEY handle from the hash object).

```
CryptCreateHash( hProv, CALG_MD5, 0, 0, &hHash );
CryptHashData( hHash, szData, lstrlen(szData) );
CryptDeriveKey( hProv, CALG_RC4, hHash, 0, &hKey );
```

The szData parameter represents the string whose content is hashed using an MD5 hash algorithm. Typically, szData will be a password. All the objects created have destructors that must be called when finished.

Up to this point, I've talked mostly about encryption and disregarded the signature topic. If you want to add a digital signature to your document, then take into account the CryptoAPI hash functions. To sign and verify documents, you should create a hash object, hash the content (exactly as described previously for passwords), and call CryptSignHash.

```
CryptCreateHash( hProv, CALG_MD5, 0, 0, &hHash );
CryptHashData( hHash, szDocContent, sizeOfDoc );
CryptSignHash( hHash, AT_SIGNATURE, szDesc, 0, NULL, &dwSize );
pBuf = (LPBYTE) malloc( dwSize )
CryptSignHash( hHash, AT_SIGNATURE, szDesc, 0, pBuf, &dwSize );
```

In this example, pBuf contains the signature information that will typically end up in a separate file, while szDesc is a description string. Both concur with the authentication process governed by CryptVerifySignature. In addition, this function requires a hashed version of the source file and the public signature key used to sign the file. This key is returned by CryptGetUserKey with the AT_SIGNATURE flag.

Sample programs for encrypting a file and signing data:
encrypt_file.cpp
signing.cpp

for correct run of program signing.cpp you need to create a certificate with accessible private key (in .NET by using command makecert)

c:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin>makecert -sk tes
t -ss labak_cert_store -$ individual -n CN=user_name