Amar explians the implementation of encryption using the Win32 Crypto API.
January 2002
Amar Galla

With computers handling more and more of our information, the need for privacy and secrecy has only been increasing, and everyone expects their applications and devices to be able to provide the strongest security, irrespective of requirements or practicality. This has placed an additional burden on us developers to come out with better, user friendly products that are also secure and reliable. Until recently, cryptography was an arcane skill limited to a very few brave souls and, practically, sometimes it is just not possible to implement some complex encryption algorithm in order to encrypt some sensitive data.
Microsoft's CryptoAPI makes this task much easier. Just by adding some function calls, a developer can embed powerful crypto algorithms, into their applications. The CryptoAPI first made its appearance in the Windows 95 OEM Service Release (OSR2), and then was soon seen again in the release of Windows NT 4.0. It has been a standard feature of both Operating Systems from then on, and has slowly grown and evolved into a fully mature Cryptographic API with the introduction of Windows 2000.
The CryptoAPI is implemented as a system level interface that Win32 applications can call to avail cryptographic services. It is built in parts with the uppermost layer, the interface layer, exposed to the clients. The rest is abstracted below. The interface layer interacts with the service provider later, called the Cryptographic Service Provider (CSP). The CSP contains the actual implementations of cryptographic standards and algorithms. It is generally deployed as a Dynamic Link Library (DLL). Microsoft provides us with some CSP's, while also gives us the flexibility to include our own service provider if we wish. The application can choose which CSP it needs to access and the CryptoAPI will transparently render the services of that particular CSP to the application via a common set of function calls. The overall process is shown in Figure 1.
Version 1.0 of the CryptoAPI contained just basic functionality to handle cryptography. These are now included into the Microsoft Base Cryptography Functions and contained routines to handle privacy and data integrity. With the introduction of version 2.0 authentication functions and digital certificate management functions were included into the API. In addition, version 2.0 also contains functions to encode and decode PKCS #7 (ASN.1) messages.
Windows 2000 many different technologies for better security management, which are tightly integrated with the base operating system. Among these is the support for Certificate Services, PKI and Kerberos Authentication. These further enhance the power of the API to call upon these system services.
Let's dive straight into the topic with an example. In this sample application, we would like to encrypt a file with the help of a password and decrypt it when the same password is supplied. A similar 'C' version of the application is included with the MSDN samples, for your study, but there are some bits of information missing from it, which may get you stuck before you even begin.

Setting up the compiler

I am assuming that you are using VC++ 6.0 for your development. The procedures here may differ a bit for other compilers. In order to successfully use the CryptoAPI you will need to set up the compiler to link to the correct library. In most cases you will have to link to crypt32.lib. So add this in the object/library module for the link tab of the Project Settings. There are some functions, which may require you to link to advapi32.lib. Check the MSDN documentation for exact details.

Setting up the environment

You will also need to include the header file wincrypt.h in your program. This contains all the declarations for the CryptoAPI functions and types. One small detail, which is very important, but is not at all included in the MSDN documents and all the samples, is to include the following compiler derivative before you include the header file.

```
#define _WIN32_WINNT 0x0400
```

This can be any value greater or equal to 0x0400. Without this the header file will fail to include the declarations required for the CryptoAPI functions.
You will also need to specify an encoding type like:
#define MY_ENCODING_TYPE (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)
PKCS_7_ASN_ENCODING signifies message encoding and X509_ASN_ENCODING signifies certificate encoding.
At this point you will be ready to start writing the actual program. But for clarity's sake, let me again show you the full header, as it should be in your program.

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <wincrypt.h>
#define MY_ENCODING_TYPE (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)
```

Building the function
We can now proceed to create the application. The procedure will differ a bit depending on what kind of an application you are trying to create in VC++. So, for having a general option, I will create a simple function, which will work in any Win32, based application, to encrypt or decrypt a file. The prototype of our function is:

```
BOOL Crypt(
// Source file for either encryption or decryption
PCHAR szSource,
// Destination file for either encryption or decryption
PCHAR szDestination,
// a password used to generate a key
PCHAR szPassword,
// ID for the operation. Encryption or Decryption
INT Operation);
```

Encryption Process
Lets start with building the encryption function first. You will have to start with creating the appropriate data types. The ones, which we are interested in, are:

```
HCRYPTPROV hCryptProv; // a handle to a CSP
HCRYPTKEY hKey; // a handle to the Key
HCRYPTHASH hHash; // a handle to a Hash Object
```

One thing I would like to mention here. The API does not permit us to access the keys or key data directly. Instead, all key related manipulations are done by using the handle to the key object. This protects the actual key from any modification or low-level access.
The first and foremost task now is to request the CryptoAPI for a service from a particular CSP. This can be done with the use of the CryptAcquireContext function. You can always check up the exact prototype and all the gory details from MSDN.
This would acquire a handle to a particular key container within a particular cryptographic service provider (CSP).

```
// this will get a handle to the correct CSP, if found
CryptAcquireContext(&hCryptProv,
// Use the default Key Container
NULL,
```

```
// Use the default Provider
NULL,
// Type of the Provider to acquire
PROV_RSA_FULL,
// Extra Flags. Usually set to 0.
0)
```

One word of caution here. You will see that many times a call to CryptAcquireContext will fail with an error value NTE_BAD_KEYSET. You will have to check this error value by using the GetLastError() function. This is a very common error value, which arises, because the key container did not exist, and hence the function failed. If this happens, again give a call to CryptAcquireContext with the same parameters, and pass CRYPT_NEWKEYSET as the flag. This will create a default key set for your program, and the function call will not succeed in acquiring a context to a particular CSP.
After this, the next thing is to create a key to be used for encryption. This key will be derived from the password, which is used for encryption, and can be recreated only if the correct password is supplied at the time of decryption.
The following functions will create a key from the password supplied by the user.

```
CryptCreateHash(
hCryptProv, // handle to a CSP
CALG_MD5, // ID to hash Algorithm
0, // handle to key for keyed algorithms. // Set to 0 for non key algorithms.
0, // Flags. Must be 0
&hHash) // address of the handle to hash object
```

First is to get a handle to the CSP hash object. The CryptCreateHash function does exactly the thing and returns the handle in the hHash (HCRYPTHASH), which we had created at the beginning.
Once the hash object is ready, we need to give the object the data we need to hash. In our case, we need to hash the Password in order to generate a unique key . The following function will add password data to the hash object.

```
CryptHashData(
// handle to the Hash Object, obtained previously
hHash,
// pointer to data to be added to the hash object
(BYTE *)szPassword, strlen(szPassword), // length of the data
0) // Flags
```

The next step would be to generate a unique key from this password. The CryptDeriveKey function generates cryptographic session keys from some given base, data, in our case, the password. This function will always generate the same session key, for the particular base data, and hence can be used to regain the session key from the password, while decrypting.

```
CryptDeriveKey(
hCryptProv, // handle to CSP
CALG_RC2, // Algorithm ID used to generate the key
hHash, // handle to hash object containing data
0, // flags to indicate type of key
&hKey) // handle to the Key
```

Lastly, we should destroy the hash object once we are finished with it, with a call to CryptDestroyHash.
CryptDestroyHash(hHash);
Now, our key is ready. There is just one more function call, which is left in order to reach our goal of encrypting the data. But before I explain it, I need to explain a bit about block ciphers. There are basically two kinds of ciphers, stream and block. Stream ciphers are ciphers where data in encrypted 1 bit at a time, while block ciphers rely on encrypting data in a unit of blocks. We will be using block cipher here (as RC2 algorithm is a block cipher algorithm), and thus the questions arise on what the block size will be. Well, block sizes differ from algorithm to algorithm and hence, be sure to set the correct size for the algorithm you will be using. For RC2, it is 64 bits or 8 bytes.
Further, you will also have to take care that the data is supplied in a multiple of the block size. If the data is less than that, you generally need to pad the data so as to complete the block length. But thankfully, you will not need to do this, as the API will do it by itself. All you have to do is, inform the API if, the current block is the last block or not.

```
CryptEncrypt(
hKey, //handle to key used for encryption
0, //for simultaneous hashing and
// encryption. 0 in our case.
bFinished, // flag indicating last block
0, // flags
pbBuffer, // pointer to the plaintext buffer
&dwByteCount, // number of bytes to be encrypted.
// Must be multiples of block size,
//unless final block
dwBufferLen) // length of the data buffer
```

This completes the final encryption process. I have not covered the file read and write, as you can very well see it in the sample code. One thing I would like to mention here is about buffer sizes. Do check out the documentation for the exact in and out buffer sizes each crypto algorithm requires, and make sure that you have allocated the right amount of memory. I general, if stream ciphers are used, the ciphertext is of the same length as the plaintext, but in case of block ciphers the ciphertext is a block length larger than the plaintext supplied.

Decryption Process
The decryption process is infact, exactly the reverse of the encryption process. You will first have to acquire a handle to the CSP by using CryptAcquireContext, and then create a key in the same way done in the encryption process. The only difference is the last part, i.e. instead of using the CryptEncrypt, you will now use another function called, CryptDecrypt.

```
CryptDecrypt(
hKey, // handle to key used for decryption
0, // for simultaneous hashing and
//decryption. 0 in our case.
bFinished, // flag indicating last block
0, // flags
pbBuffer, // pointer to the ciphertext buffer
&dwByteCount) // number of bytes to be decrypted.
// Must be multiples of block size,
//unless final block
```

That's it. Your data would be decrypted and you will get back the original plaintext, provided the password used was the same. Again, same as in encryption do check the buffer lengths and make sure you have enough memory allocated to hold the data.

I have provided the full working examples of both the encryption and decryption functions along with this article. The only major addition in it is the loops, that handle multiple blocks of data at a time.

Windows 2000 also provides us with 2 utility functions called EncryptFile and DecryptFile. These do not use a password, and use a session key generated by the users certificate and security setting. So, you might consider using them, if you do not want backward compatibility and you want a user level encryption/decryption functionality.

The CryptoAPI provides a lot of treasures to the developer. Apart from simple encryption and decryption, it also has a host of functions for public/private key generation, certificate encoding and decoding, and a support for a lot of different crypto algorithms. As a developer, now you have the power to call the services of this extremely powerful API in order to build world-class security into your products. A word of caution here-Do not overdo its usage, just because it's so simple to use. Cryptographic operations come at a high degree of CPU cost, and if overused, may slow down your applications. Instead, try to identify the points in your application where your priority data is unsecured and most vulnerable. This would be the point where you will be interested in encrypting your data.