CryptoAPI

Labs

Daniil Leksin

Structure

- CryptoAPI
- CSP (Cryptography Service Provider)
- CA
- Working with CryptoAPI, CSP, CA: algorithms, block-schemes and examples

CryptoAPI

CryptoAPI (Cryptographic Application Programming Interface, Microsoft Cryptography API, MS-CAPI or simply CAPI) is an application programming interface included with Microsoft Windows operating systems that provides services to enable developers to secure Windows-based applications using cryptography. It is a set of dynamically linked libraries that provides an abstraction layer which isolates programmers from the code used to encrypt the data. (CryptoAPI supports both public-key and symmetric key cryptography)

CAPI provides:

1. Secure data storing
2. Ability to transfer data
3. Validation from 3rd party users
4. Work with EDS
5. Work with cryptographic standards
6. Extension

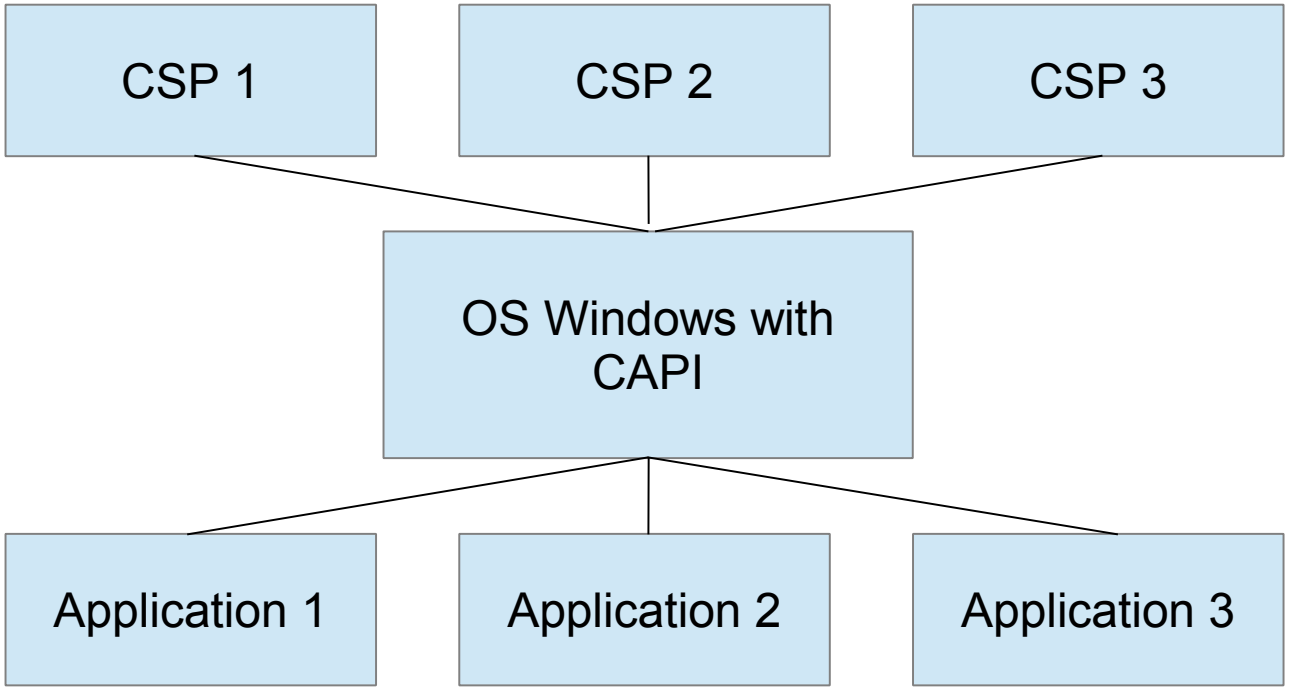
CAPI functionality groups:

1. Basic cryptographic functions:
    - 1.1 encoding / decoding
    - 1.2 hash function, EDS
    - 1.3 initializing CSP, working with context
    - 1.4 key generation
    - 1.5 key exchanging
2. Functions for working with certificates
3. High-level functions
4. Low-level functions

CSP

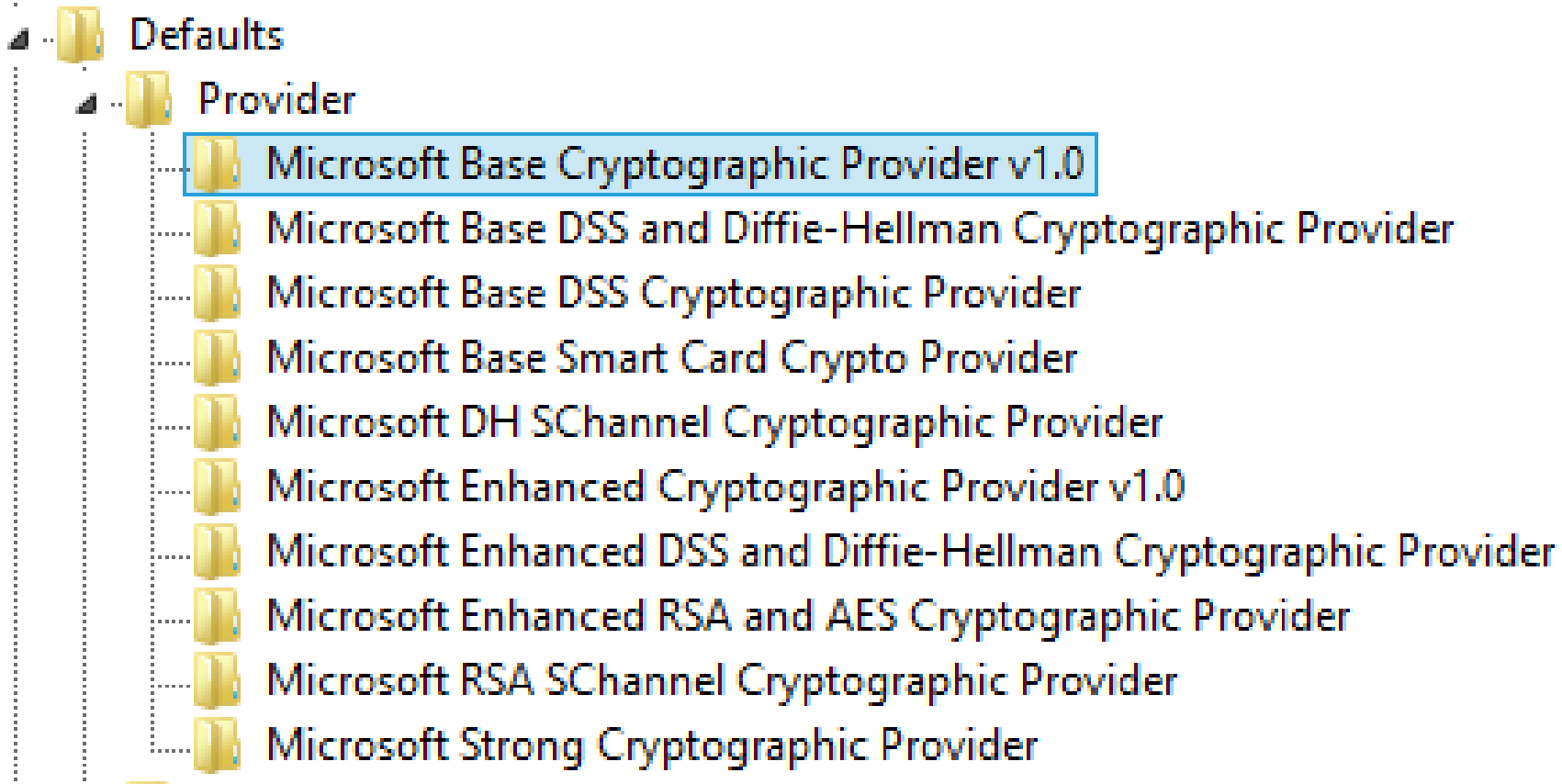
- CSP (Cryptography Service Provider) - is a software library that implements the Microsoft CryptoAPI (CAPI). CSPs implement encoding and decoding functions, which computer application programs may use.
- CSP provides:
    1. implementation of the standard interface
    2. work with encode / decode keys
    3. inability to interference from third parties
- 2 function groups for working with CSP:
    1. initialization of the context and getting CSP parameters
    2. Key generation and function for work with them
    3. encode / decode functions
    4. Hash functions and getting EDS

CAPI & CSP & Apps



Find CSP on current machine

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider\



enum.cpp

```
CryptEnumProviderTypes(
        dwIndex,
        NULL,
        0,
        &dwType,
        NULL,
        &cbName)
```

The CryptEnumProviderTypes function retrieves the first or next types of cryptographic service provider (CSP) supported on the computer. Used in a loop, this function retrieves in sequence all of the CSP types available on a computer.

```
BOOL WINAPI CryptEnumProviderTypes(
  _In_      DWORD      dwIndex,
  _In_      DWORD      *pdwReserved,
  _In_      DWORD      dwFlags,
  _Out_     DWORD      *pdwProvType,
  _Out_     LPTSTR     pszTypeName,
  _Inout_ DWORD        *pcbTypeName
);
```

```
_tprintf (TEXT("     %4.0d          %s\n"),
                dwType,
                pszName);
```

```
Listing Available Provider Types.
Provider type    Provider Type Name
_____________    _____________________________________________
        1        RSA Full (Signature and Key Exchange)
        3        DSS Signature
       12        RSA SChannel
       13        DSS Signature with Diffie-Hellman Key Exchange
       18        Diffie-Hellman SChannel
       24        RSA Full and AES
```

enum.cpp

```
CryptEnumProviders(
        dwIndex++,
        NULL,
        0,
        &dwType,
        NULL,
        &cbName)
```

```
_tprintf (TEXT("     %4.0d          %s\n"),
                dwType,
                pszName);
```

The **CryptEnumProviders** function retrieves the first or next available cryptographic service providers (CSPs). Used in a loop, this function can retrieve in sequence all of the CSPs available on a computer.

```
BOOL WINAPI CryptEnumProviders(
  _In_     DWORD    dwIndex,
  _In_     DWORD    *pdwReserved,
  _In_     DWORD    dwFlags,
  _Out_    DWORD    *pdwProvType,
  _Out_    LPTSTR   pszProvName,
  _Inout_  DWORD    *pcbProvName
);
```

```
Listing Available Providers.
Provider type    Provider Name
_____________    ____________________________________________
        1        Microsoft Base Cryptographic Provider v1.0
       13        Microsoft Base DSS and Diffie-Hellman Cryptographic Provider
        3        Microsoft Base DSS Cryptographic Provider
        1        Microsoft Base Smart Card Crypto Provider
       18        Microsoft DH SChannel Cryptographic Provider
        1        Microsoft Enhanced Cryptographic Provider v1.0
       13        Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provide
r
       24        Microsoft Enhanced RSA and AES Cryptographic Provider
       12        Microsoft RSA SChannel Cryptographic Provider
        1        Microsoft Strong Cryptographic Provider
```

enum.cpp

```
CryptGetDefaultProvider(
        PROV_RSA_FULL,
        NULL,
        CRYPT_MACHINE_DEFAULT,
        pbProvName,
        &cbProvName))
```

The **CryptGetDefaultProvider** function finds the default cryptographic service provider (CSP) of a specified provider type for the local computer or current user. The name of the default CSP for the provider type specified in the dwProvType parameter is returned in the pszProvName buffer.

```
_tprintf(TEXT("\nThe default provider
name is \"%s\"\n"),
            pbProvName);
```

```
The default provider name is "Microsoft Strong Cryptographic Provider"
```

```
BOOL WINAPI CryptGetDefaultProvider(
  _In_      DWORD    dwProvType,
  _In_      DWORD    *pdwReserved,
  _In_      DWORD    dwFlags,
  _Out_     LPTSTR   pszProvName,
  _Inout_   DWORD    *pcbProvName
);
```

enum.cpp

```
CryptAcquireContext(
        &hProv,
        NULL,
        NULL,
        PROV_RSA_FULL,
        NULL)
```

```
CryptAcquireContext(
        &hProv,
        NULL,
        NULL,
        PROV_RSA_FULL,
        CRYPT_NEWKEYSET)
```

The **CryptAcquireContext** function is used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle is used in calls to CryptoAPI functions that use the selected CSP.

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

phProv [out] A pointer to a handle of a CSP.

CRYPT_NEWKEYSET
Creates a new key container with the name specified by pszContainer. If pszContainer is NULL, a key container with the default name is created.

enum.cpp

```
CryptGetProvParam(
            hProv,
            PP_ENUMALGS,
            pbData,
            &cbData,
            dwFlags)
```

The **CryptGetProvParam** function retrieves parameters that govern the operations of a cryptographic service provider

```
BOOL WINAPI CryptGetProvParam(
  _In_    HCRYPTPROV hProv,
  _In_    DWORD      dwParam,
  _Out_   BYTE       *pbData,
  _Inout_ DWORD      *pdwDataLen,
  _In_    DWORD      dwFlags
);
```

A PROV_ENUMALGS structure that contains information about one algorithm supported by the CSP being queried. The first time this value is read, the dwFlags parameter must contain the CRYPT_FIRST flag. Doing so causes this function to retrieve the first element in the enumeration. The subsequent elements can then be retrieved by setting the CRYPT_NEXT flag in the dwFlags parameter. When this function fails with the ERROR_NO_MORE_ITEMS error code, the end of the enumeration has been reached.

```
typedef struct _PROV_ENUMALGS{
  ALG_ID    aiAlgid;
  DWORD     dwBitLen;
  DWORD     dwNameLen;
  CHAR      szName[20];
} PROV_ENUMALGS;
```

enum.cpp

```
 // Extract algorithm information from the
pbData buffer.
           dwFlags = 0;
           ptr = pbData;
           aiAlgid = *(ALG_ID *)ptr;
           ptr += sizeof(ALG_ID);
           dwBits = *(DWORD *)ptr;
           ptr += dwIncrement;
           dwNameLen = *(DWORD *)ptr;
           ptr += dwIncrement;
           strncpy(szName,(char *) ptr,
dwNameLen);
```

```
// Determine the algorithm type.
switch(GET_ALG_CLASS(aiAlgid))
{
    case ALG_CLASS_DATA_ENCRYPT:
        pszAlgType = "Encrypt  ";
        break;
    case ALG_CLASS_HASH:
        pszAlgType = "Hash     ";
        break;
    case ALG_CLASS_KEY_EXCHANGE:
        pszAlgType = "Exchange ";
        break;
    case ALG_CLASS_SIGNATURE:
        pszAlgType = "Signature";
        break;
    default:
        pszAlgType = "Unknown  ";
        break;
}
```

```
 // Print information about the algorithm.
           printf("    %8.8xh    %-4d    %s     %-2d          %s\n",
               aiAlgid,
               dwBits,
               pszAlgType,
               dwNameLen,
               szName);
```

enum.cpp

```
Enumerating the supported algorithms

     Algid       Bits       Type          Name          Algorithm
                                          Length          Name

    ----------------------------------------------------------------
    00006602h    128       Encrypt        4            RC2
    00006801h    128       Encrypt        4            RC4
    00006601h    56        Encrypt        4            DES
    00006609h    112       Encrypt        13           3DES TWO KEY
    00006603h    168       Encrypt        5            3DES
    00008004h    160       Hash           6            SHA-1
    00008001h    128       Hash           4            MD2
    00008002h    128       Hash           4            MD4
    00008003h    128       Hash           4            MD5
    00008008h    288       Hash           12           SSL3 SHAMD5
    00008005h    0         Hash           4            MAC
    00002400h    1024      Signature      9            RSA_SIGN
    0000a400h    1024      Exchange       9            RSA_KEYX
    00008009h    0         Hash           5            HMAC
```

c1.cpp

```
// Define the name of the store where the needed certificate
// can be found.
```

← step1

```
// The message to be signed
// Size of message. Note that the length set is one more than the
// length returned by the strlen function in order to include
// the NULL string termination character.
// Pointer to a signer certificate
// Create the MessageArray and the MessageSizeArray.
```

← step2

```
 // Begin processing. Display the original message.
 // Open a certificate store.
 // Get a pointer to the signer's certificate.
 // Initialize the signature structure.
```

← step3

```
// With two calls to CryptSignMessage, sign the message.
// First, get the size of the output signed BLOB.
// Second, Get the SignedMessageBlob.
```

← step4

```
// Verify the message signature.
// With two calls to CryptVerifyMessageSignature, verify and
// decode the signed message.
// First, call CryptVerifyMessageSignature to get the length
// of the buffer needed to hold the decoded message.
// Allocate memory for the buffer.
```

← step5

step1

```
// Define the name of the store where the needed certificate
// can be found.
```

```
#define CERT_STORE_NAME  L"labak_cert_store"
```

step2

```
// The message to be signed
// Size of message. Note that the length set is one more than the
// length returned by the strlen function in order to include
// the NULL string termination character.
// Pointer to a signer certificate
// Create the MessageArray and the MessageSizeArray.
```

```
BYTE* pbMessage = (BYTE*)"CryptoAPI is a good way to handle security";
//
DWORD cbMessage = strlen((char*) pbMessage)+1;
//
PCCERT_CONTEXT pSignerCert;
//
CRYPT_SIGN_MESSAGE_PARA  SigParams;
DWORD cbSignedMessageBlob;
BYTE  *pbSignedMessageBlob;
DWORD cbDecodedMessageBlob;
BYTE  *pbDecodedMessageBlob;
CRYPT_VERIFY_MESSAGE_PARA VerifyParams;
//
const BYTE* MessageArray[] = {pbMessage};
DWORD MessageSizeArray[1];
MessageSizeArray[0] = cbMessage;
```

step3

```
// Begin processing. Display the original message.
// Open a certificate store.
// Get a pointer to the signer's certificate.
// Initialize the signature structure.
```

```
if ( !( hStoreHandle = CertOpenStore(
    CERT_STORE_PROV_SYSTEM,
    0,
    NULL,
    CERT_SYSTEM_STORE_CURRENT_USER,
    CERT_STORE_NAME)))
{
      MyHandleError("The MY store could
not be opened.");
}
```

```
if(pSignerCert =
CertFindCertificateInStore(
   hStoreHandle,
   MY_TYPE,
   0,
   CERT_FIND_SUBJECT_STR,
   SIGNER_NAME,
   NULL))
{
   printf("The signer's certificate
was found.\n");
} else {
    MyHandleError( "Signer certificate
not found.");
}
```

The CertOpenStore function opens a certificate store by using a specified store provider type. While this function can open a certificate store for most purposes.

```
HCERTSTORE WINAPI CertOpenStore(
  _In_  LPCSTR     lpszStoreProvider,
  _In_  DWORD      dwMsgAndCertEncodingType,
  _In_  HCRYPTPROV_LEGACY hCryptProv,
  _In_  DWORD      dwFlags,
  _In_  const void   *pvPara
);
```

This function finds the first or next certificate context in a certificate store that matches search criteria established by the dwFindType parameter and its associated pvFindPara parameter.

```
PCCERT_CONTEXT WINAPI CertFindCertificateInStore(
  HCERTSTORE hCertStore,
  DWORD dwCertEncodingType,
  DWORD dwFindFlags,
  DWORD dwFindType,
  const void* pvFindPara,
  PCCERT_CONTEXT pPrevCertContext
);
```

```
SigParams.cbSize = sizeof(CRYPT_SIGN_MESSAGE_PARA);
SigParams.dwMsgEncodingType = MY_TYPE;
SigParams.pSigningCert = pSignerCert;
SigParams.HashAlgorithm.pszObjId = szOID_RSA_MD5;
SigParams.HashAlgorithm.Parameters.cbData = NULL;
SigParams.cMsgCert = 1;
SigParams.rgpMsgCert = &pSignerCert;
```

```
SigParams.cAuthAttr = 0;
SigParams.dwInnerContentType = 0;
SigParams.cMsgCrl = 0;
SigParams.cUnauthAttr = 0;
SigParams.dwFlags = 0;
SigParams.pvHashAuxInfo = NULL;
SigParams.rgAuthAttr = NULL;
```

```
typedef struct _CRYPT_SIGN_MESSAGE_PARA {
  DWORD                        cbSize;
  DWORD                        dwMsgEncodingType;
  PCCERT_CONTEXT               pSigningCert;
  CRYPT_ALGORITHM_IDENTIFIER   HashAlgorithm;
  void                         *pvHashAuxInfo;
  DWORD                        cMsgCert;
  PCCERT_CONTEXT               *rgpMsgCert;
  DWORD                        cMsgCrl;
  PCCRL_CONTEXT                *rgpMsgCrl;
  DWORD                        cAuthAttr;
  PCRYPT_ATTRIBUTE             rgAuthAttr;
  DWORD                        cUnauthAttr;
  PCRYPT_ATTRIBUTE             rgUnauthAttr;
  DWORD                        dwFlags;
  DWORD                        dwInnerContentType;
  CRYPT_ALGORITHM_IDENTIFIER   HashEncryptionAlgorithm;
  void                         pvHashEncryptionAuxInfo;
} CRYPT_SIGN_MESSAGE_PARA, *PCRYPT_SIGN_MESSAGE_PARA;
```

step4

```
// With two calls to CryptSignMessage, sign the message.
// First, get the size of the output signed BLOB.
// Second, Get the SignedMessageBlob.
```

```
if(CryptSignMessage(
    &SigParams,
    FALSE,
    1,
    MessageArray,
    MessageSizeArray,
    NULL,
    &cbSignedMessageBlob))
{
    printf("The size of the BLOB is
%d.\n",cbSignedMessageBlob);
}
else
{
    MyHandleError("Getting signed BLOB
size failed");
}
```

```
if(!(pbSignedMessageBlob = (BYTE*)malloc(cbSignedMessageBlob)))
{
    MyHandleError("Memory allocation error while signing.");
}
```

```
if(CryptSignMessage(
      &SigParams,
      FALSE,
      1,
      MessageArray,
      MessageSizeArray,
      pbSignedMessageBlob,
      &cbSignedMessageBlob))
{
    printf("The message was signed
successfully. \n");
}
else
{
    MyHandleError("Error getting signed
BLOB");
}
```

The CryptSignMessage function creates a hash of the specified content, signs the hash, and then encodes both the original message content and the signed hash.

step5

```
// Verify the message signature.
// With two calls to CryptVerifyMessageSignature, verify and
// decode the signed message.
// First, call CryptVerifyMessageSignature to get the length
// of the buffer needed to hold the decoded message.
// Allocate memory for the buffer.
```

```
VerifyParams.cbSize = sizeof(CRYPT_VERIFY_MESSAGE_PARA);
VerifyParams.dwMsgAndCertEncodingType = MY_TYPE;
VerifyParams.hCryptProv = 0;
VerifyParams.pfnGetSignerCertificate = NULL;
VerifyParams.pvGetArg = NULL;
```

```
typedef struct _CRYPT_VERIFY_MESSAGE_PARA {
  DWORD                                 cbSize;
  DWORD                                 dwMsgAndCertEncodingType;
  HCRYPTPROV_LEGACY                     hCryptProv;
  PFN_CRYPT_GET_SIGNER_CERTIFICATE      pfnGetSignerCertificate;
  void                                  *pvGetArg;
  PCCERT_STRONG_SIGN_PARA               pStrongSignPara;
} CRYPT_VERIFY_MESSAGE_PARA, *PCRYPT_VERIFY_MESSAGE_PARA;
```

```
if(CryptVerifyMessageSignature(
    &VerifyParams,
    0,
    pbSignedMessageBlob,
    cbSignedMessageBlob,
    NULL,
    &cbDecodedMessageBlob,
    NULL))
{
    printf("%d bytes need for the buffer.\n",
cbDecodedMessageBlob);
}
else
{
    printf("Verification message failed. \n");
}
```

```
if(CryptVerifyMessageSignature(
    &VerifyParams,
    0,
    pbSignedMessageBlob,
    cbSignedMessageBlob,
    pbDecodedMessageBlob
    &cbDecodedMessageBlob,
    NULL))
{
    printf("The verified message is \n->
%s \n", pbDecodedMessageBlob);
}
else
{
    printf("Verification message failed. \n");
}
```

```
if(!(pbDecodedMessageBlob =
   (BYTE*)malloc(cbDecodedMessageBlob)))
{
    MyHandleError("Memory allocation error allocating decode BLOB.");
}
```

The CryptVerifyMessageSignature function verifies a signed message's signature.