CryptLib

Security Toolkit

API Handbook

Version 2.4.2

CryptLib Programmer's Handbook

Copyright

Trademarks

Table of contents

Chapter

1

Introduction

Cryptography is the science that deals with the coverage of messages.

The development of new approaches and methods to secure messages has been driven in the past primarily by military requirements. This can be traced back to the Roman Empire. The roman military used simple cipher methods in order to hide plaintext from their enemies. The so called 'Caesar-Cipher' may exemplify this fact.

In the years past, cryptographic applications have made their way into many areas of the real world. One of the driving reasons for this is the fact that the fast expansion of the World Wide Web has enabled the building of large computer networks which in turn has lead to increased communication possibilities.

These require a more in depth discussion regarding the security of the transmitted data. Herein various interests play a role. These may be of personal character such as in the area of online banking or of business character such as in the area of clearing business transactions over the Internet.

The increased demand for cryptographic procedures has pressed ahead the development of audited, secure, easy to use and publicly available algorithms.

Today it's possible for any computer user to protect his personal data using methods regarded as secure. In this context, secure means that even the application of computer power currently estimated as unrealistically large doesn't provide the opportunity to systematically ascribe ciphertext into plaintext investing a reasonable amount of time. This can only be carried out if a secure token of information, called the 'key' is well known.

In the course of time some procedures have established as standards. That's because of the fact that these procedures are well documented and thus are well tested regarding security and stability. It's important to note that the security of these procedures relies only on the used key. Public knowledge of the internals of the used algorithm doesn't weaken the security of the chosen procedures.

CryptLib from XPS Software GmbH offers to the programmer a library containing standardized cryptographic procedures for use in proprietary development. CryptLib is available for the following operating systems: Win32, Linux, OS/2, OS/400, IBM iSeries, VSE/ESA, MVS/ESA, OS/390 and IBM zSeries.

CryptLib offers among others methods to calculate hash values, methods for symmetrical and asymmetrical encryption, methods to process X.509 certificates, methods for creation and check of digital signatures and support of the Public Key Cryptography Standards PKCS#7 (S/MIME) and PKCS#12 (public key).

Chapter

2

Key generation

Common information

Random numbers play an integral part in cryptography. Generating a new symmetrical key as well as a new asymmetrical key begins with the generation of a random number. The integrity of the chosen random number is very important. If it is predictable the generated key can be recalculated given the chosen procedure is known. The security of the whole system depends on the privacy of the key. Therefore it's possible to provide a specific initialization value using the *seed* parameter which will be incorporated into the process of key generation.

A RSA public/private key pair can be generated using the *GenerateRSAKey* method. The method *GenerateKey* can be used to generate random keys, initialization vectors (iv) or any other random values.

Methods

GenerateKey

Generate a random key for symmetrical encryption and decryption.

Syntax	**void GenerateKey(BYTE *key, int keylength, BYTE *seed, int seedlength);**	
Return code	None.	
Parameter	**Description**	**Use**
key	Address of the storage area to be used to return the generated symmetrical key.	Output
keylength	Length of the key to generate.	Input
seed	Address of storage area holding variable data to be incorporated into the key generation.	Input
seedlength	Length of the variable data.	Input

Example:

```
#include "XPSCRYPT.H"

int main(void)
{
  BYTE block[32] = "XPS Software GmbH, Haar/Muenchen";
  BYTE seed1[16] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
  BYTE seed2[16] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
  BYTE key[32]   = {0};
  BYTE iv[16]    = {0};

  CIPHERCTX ctx     = {0};
  int       iOutlen = 0;
  int       rc      = 0;

  GenerateKey( key, sizeof(key), seed1, sizeof(seed1) );
```

```
  GenerateKey( iv,  sizeof(iv),  seed2, sizeof(seed2) );

  ctx.algorithm = AES;
  ctx.mode      = CBC;
  ctx.keylength = 256;
  ctx.key       = (BYTE *)key;
  ctx.iv        = (BYTE *)iv;

  /* encryption */
  InitCTX( &ctx );
  iOutlen = Encrypt( &ctx, block, sizeof(block), block, sizeof(block) );
  CleanupCTX( &ctx );
  return 0;
}
```

GenerateRSAKey

Generate a random RSA public/private key pair.

Syntax	**void GenerateRSAKey(RSA_PRIVATE_KEY *privkey, RSA_PUBLIC_KEY *pubkey, BYTE *seed, int seedlength, int keylength);**	
Return code	None.	
Parameter	**Description**	**Use**
privkey	Address of the storage area to be used to return the generated private RSA key.	Output
pubkey	Address of the storage area to be used to return the generated public RSA key.	Output
seed	Address of storage area holding variable data to be incorporated into the key generation.	Input
seedlength	Length of the variable data.	Input
keylength	Desired key length in bits (maximum 4096).	Input

Example:

```
#include "XPSCRYPT.H"

int main(void)
{
  BYTE  block[32]       = "XPS Software GmbH, Haar/Muenchen";
  BYTE  KeyRandom[16]   = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
  BYTE *output;

  RSA_PRIVATE_KEY rsa_priv;
  RSA_PUBLIC_KEY  rsa_pub;
  CIPHERCTX       ctx     = {0};
  int             iOutlen = 0;
  int             rc      = 0;

  GenerateRSAKey( &rsa_priv, &rsa_pub, KeyRandom, sizeof(KeyRandom), 1024 );

  ctx.algorithm    = RSA;
  ctx.mode         = PUBLIC;
  ctx.key          = (BYTE *)&rsa_pub;

  /* rsa encryption        */
  InitCTX( &ctx );
  iOutlen = GetResultLength( &ctx, sizeof(block) );
  output  = malloc( iOutlen );
  iOutlen = Encrypt( &ctx, block, sizeof(block), output, iOutlen );
  CleanupCTX( &ctx );
  free( output );
  return 0;
}
```

Chapter

3

Encryption

Common information

A distinction is drawn between two basic encryption modes:

Symmetrical

When using a symmetrical encryption method, message encryption and decryption is carried out using the identical key. This demands the sender (encryption) and the receiver (decryption) of the message to know and use the same key. If CBC is chosen as mode of operation an additional initialization vector is needed. The length of this initialization vector equals the block length of the chosen algorithm (DES, TripleDES, RC2, RC4, Blowfish = 8 Byte, AES = 16 Byte).

Asymmetrical (public key)

Methods using non identical keys for encryption and decryption are called asymmetrical. Both keys are created during the process of key generation and it is impossible to suggest the one key from the other. This enables one key to be made publicly available why it is called the public key. Using the public key the sender of a message can encrypt the message and send it to the owner of the public/private key pair who in turn is able to decrypt the public key encrypted message using the private key only known to him.

CryptLib supports a number of symmetrical encryption algorithms (AES, DES, TripleDES, RC2, RC4, and Blowfish) as well as the public/private key encryption algorithm RSA. Using the method *InitCTX* the programmer can chose the desired mode for encryption and decryption. After context initialization is complete the methods *Encrypt a*nd *Decrypt* can be used to encrypt and decrypt messages regardless of the chosen encryption algorithm.

Methods

InitCTX

Initialize the cryptographic context.

Syntax	**int InitCTX(PCIPHERCTX ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Address of storage area holding the context needed for encryption and decryption.	Input
Felder	**Description**	

ctx.algorithm	Algorithm used to encrypt/decrypt the data. CryptLib supports the following algorithms: AES — Advanced Encryption Standard (Rijndael) DES — Data Encryption Standard with ctx.keylength = 56 TripleDES EDE2 — Data Encryption Standard with ctx.keylength = 112 TripleDES EDE3 Data — Encryption Standard with ctx.keylength = 168 RC2 — Rivest Cipher No. 2 RC4 — Rivest Cipher No. 4 Blowfish — Schneier RSA — Public-/Private-Key method from Rivest, Shamir and Adleman
ctx.mode	Mode of operation. CryptLib supports the following modes for symmetrical encryption (AES, DES, RC2, RC4 and Blowfish): ECB — Electronic-Codebook-Mode CBC — Cipher-Block-Chaining CryptLib supports the following modes for asymmetrical encryption (RSA): PUBLIC — (public key encryption/decryption) *ctx.key* must specify the storage address of a RSA_PUBLIC_KEY structure PRIVATE — (private key encryption(decryption) *ctx.key* must specify the storage address of a RSA_PRIVATE_KEY structure
ctx.key	Storage address of the key.
ctx.keylength	Key length. CryptLib supports the following key lengths: AES — 128, 192, 256 DES — 56, 112, 168 RC2 — 40, 64, 128 RC4 — 40, 64, 128 Blowfish — 128 RSA — 512, 1024, 2048, 4096
ctx.iv	Address of storage area holding the initialization vector to be used for symmetrical encryption.

Encrypt

Encrypt data. The type of encryption is dependent on the *ctx.algorithm* parameter chosen for context initialization using the *InitCTX* method.

Syntax	**int Encrypt(PCIPHERCTX ctx, BYTE *input, int inputlength, BYTE *output, int outputlength);**	
Return code	Length of encrypted data or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context to be used for encryption.	Input
input	Storage address of the data to encrypt.	Input
inputlength	Length of the data to encrypt.	Input
output	Address of storage area to be used to hold the encrypted data.	Output
outputlength	Length of the storage area specified using the *output* parameter.	Input

Decrypt

Decrypt data. The type of decryption is dependent on the *ctx.algorithm* parameter chosen for context initialization using the *InitCTX* method.

Syntax	**int Decrypt(PCIPHERCTX ctx, BYTE *input, int inputlength, BYTE *output, int outputlength);**	
Return code	Length of decrypted data or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context to be used for decryption.	Input
input	Storage address of the data to decrypt.	Input
inputlength	Length of the data to decrypt.	Input
output	Address of storage area to be used to hold the decrypted data.	Output
outputlength	Length of the storage area specified using the *OUTPUT* parameter.	Input

GetResultLength

Determine the length of the storage needed to store the encryption/decryption result.
Hint: Using RSA this method can only be used to determine the resulting storage size for encryption. In the case of decryption the resulting storage size is always lower or equal to the input size.

Syntax	**int GetResultLength(PCIPHERCTX ctx, int inputlength);**	
Return code	Length of encrypted/decrypted data or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context to be used for encryption/decryption.	Input
inputlength	Length of the data to encrypt/decrypt.	Input

ResetCTX

Reset the initialization vector (iv) for symmetrical encryption and decryption.

Syntax	**int ResetCTX(PCIPHERCTX ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	The storage address of the context.	Input

CleanupCTX

Deallocation of the storage needed for encryption and decryption.

Syntax	**int CleanupCTX(PCIPHERCTX ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	The storage address of the context.	Input

Example (AES):

```
#include <stdlib.h>
#include "XPSCRYPT.H"

int main(void)
```

```
{
  BYTE block[32] = "XPS Software GmbH, Haar/Muenchen";
  BYTE seed1[16] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
  BYTE seed2[16] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
  BYTE key[32]   = {0};
  BYTE iv[16]    = {0};

  CIPHERCTX ctx     = {0};
  int       iOutlen = 0;
  int       rc      = 0;

  GenerateKey( key, sizeof(key), seed1, sizeof(seed1) );
  GenerateKey( iv,  sizeof(iv),  seed2, sizeof(seed2) );

  ctx.algorithm = AES;
  ctx.mode      = CBC;
  ctx.keylength = 256;
  ctx.key       = (BYTE *)key;
  ctx.iv        = (BYTE *)iv;

  /* encryption            */
  InitCTX( &ctx );
  iOutlen = Encrypt( &ctx, block, sizeof(block), block, sizeof(block) );
  if( iOutlen < 0 )
  {
    printf( "Encrypt error %d\n", iOutlen );
    exit( -1 );
  }
  /* now check decryption: */
  rc = ResetCTX( &ctx );
  iOutlen = Decrypt( &ctx, block, iOutlen, block, sizeof(block) );
  if( iOutlen < 0 )
  {
    printf( "Decrypt error %d\n", iOutlen );
    exit( -1 );
  }
  CleanupCTX( &ctx );
  return 0;
}
```

Example (TripleDES):

```
#include <stdlib.h>
#include "XPSCRYPT.H"

int main(void)
{
  BYTE block[32] = "XPS Software GmbH, Haar/Muenchen";
  BYTE key[24]   = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                    0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08};
  BYTE iv[8]     = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};

  CIPHERCTX ctx     = {0};
  int       iOutlen = 0;
  int       rc      = 0;

  ctx.algorithm = DES;
  ctx.mode      = CBC;
  ctx.keylength = 168;
  ctx.key       = (BYTE *)key;
  ctx.iv        = (BYTE *)iv;

  /* encryption            */
  InitCTX( &ctx );
  iOutlen = Encrypt( &ctx, block, sizeof(block), block, sizeof(block) );
  if( iOutlen < 0 )
  {
    printf( "Encrypt error %d\n", iOutlen );
    exit( -1 );
  }
  /* now check decryption: */
  ResetCTX( &ctx );
  iOutlen = Decrypt( &ctx, block, sizeof(block), block, sizeof(block) );
  if( iOutlen < 0 )
  {
    printf( "Decrypt error %d\n", iOutlen );
    exit( -1 );
  }
  CleanupCTX( &ctx );
  return 0;
}
```

Example (RSA):

```
#include <stdlib.h>
```

```
#include "XPSCRYPT.H"

int main(void)
{
  BYTE  block[32]       = "XPS Software GmbH, Haar/Muenchen";
  BYTE  KeyRandom[16]   = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
  BYTE *output;
  BYTE  output2[32]    = {0};

  RSA_PRIVATE_KEY rsa_priv;
  RSA_PUBLIC_KEY  rsa_pub;
  CIPHERCTX       ctx     = {0};
  int             iOutlen = 0;
  int             rc      = 0;
  GenerateRSAKey( &rsa_priv, &rsa_pub, KeyRandom, sizeof(KeyRandom), 1024 );

  ctx.algorithm    = RSA;
  ctx.mode         = PUBLIC;
  ctx.key          = (BYTE *)&rsa_pub;

  /* encryption            */
  InitCTX( &ctx );
  iOutlen = GetResultLength( &ctx, sizeof(block) );
  output  = malloc( iOutlen );
  iOutlen = Encrypt( &ctx, block, sizeof(block), output, iOutlen );
  if( iOutlen < 0 )
  {
    printf( "Encrypt error %d\n", iOutlen );
    exit( -1 );
  }
  CleanupCTX( &ctx );

  /* now check decryption: */
  ctx.algorithm    = RSA;
  ctx.mode         = PRIVATE;
  ctx.key          = (BYTE *)&rsa_priv;

  InitCTX( &ctx );
  iOutlen = Decrypt( &ctx, output, iOutlen, output2, sizeof(output2) );
  if( iOutlen < 0 )
  {
    printf( "Decrypt error %d\n", iOutlen );
    exit( -1 );
  }
  CleanupCTX( &ctx );
  free( output );
  return 0;
}
```

Chapter

4

Digital signature

Common information

The implementation of digital signatures is one of the major applications for asymmetrical encryption. To create a digital signature at first a hash value for the data about to be digitally signed is created. Then, in a second step this hash value will be encrypted using the private key. Later on the result can be decrypted using the associated public key. Based on re-generation of the hash value and comparison of the result with the decrypted hash value, the integrity of the originally digitally signed data can be guaranteed.

CryptLib supports digital signatures created with the **RSA** algorithm. The supported methods for hash value generation are **MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512** and **RipeMD160**.

Use the following steps to create a digital signature:

§ Initialize the context using the *SignInit* method to specify the desired hash method.

§ Add data to be digitally signed using the *SignUpdate* method. This method can be called as often as needed.

§ Call the *SignFinal* method to finalize the procedure and to get hold of the digital signature.

Use the following steps to verify a digital signature:

§ Initialize the context using the *VerifyInit* method to specify the desired hash method.

§ Add data to be verified using the *VerifyUpdate* method. This method can be called as often as needed.

§ Call the *VerifyFinal* method to finalize the procedure and to verify the digital signature.

Methods

SignInit

Initialisierung des Signaturkontexts.

Syntax	**int SignInit(SIGNATURE_CTX *ctx, int digestAlgo);**	
Return code	0 oder Fehlercode (< 0).	
Parameter	**Description**	**Use**
ctx	Address pointer to receive the storage address of the created context. This	Output

	will be used for subsequent processing.	
digestAlgo	The type of hash to be used. CryptLib supports MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.	Input

SignUpdate

Add data to be signed.

Syntax	**int SignUpdate(SIGNATURE_CTX *ctx, unsigned char *inputdata,int datalength);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context initialized for creating the signature.	Input
inputdata	Storage address of the data to be signed.	Input
intputlength	Length of the data to be signed.	Input

SignFinal

Create the digital signature using the private key.

Syntax	**int SignFinal(SIGNATURE_CTX *ctx, unsigned char *signdata, int *signlength, RSA_PRIVATE_KEY privkey);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context to be used to sign the data.	Input
signdata	Address of storage area about to hold the created digital signature.	Output
signlength	Address of storage area about to hold the length of the created digital signature.	Output
privkey	The signer's private RSA key.	Input

VerifyInit

Initialize the context for verification.

Syntax	**int VerifyInit(SIGNATURE_CTX *ctx, int digestAlgo);**	
Return code	0 oder Fehlercode (< 0).	
Parameter	**Description**	**Use**
ctx	Address pointer to receive the storage address of the created context. This will be used for subsequent processing.	Output
digestAlgo	The type of hash to be used. CryptLib supports MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.	Input

VerifyUpdate

Add data to be verified.

Syntax	int VerifyUpdate(SIGNATURE_CTX *ctx, unsigned char *inputdata, int datalength);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context initialized for verification.	Input
inputdata	Storage address of data to be verified.	Input
inputlength	Length of the data to be verified.	Input

VerifyFinal

Verify the digital signature using the public key.

Syntax	int VerifyFinal(SIGNATURE_CTX *ctx, unsigned char *signdata, int signlength, RSA_PUBLIC_KEY pubkey);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context initialized for verification.	Input
signdata	Storage address of the digital signature.	Input
signlength	Length of the digital signature.	Input
pubkey	The signer's public RSA key.	Input

Example:

```
#include <stdlib.h>
#include "XPSCRYPT.H"
#define keylen 1024

int main(void)
{
  BYTE  text1[32]       = "XPS Software GmbH, Haar/Muenchen";
  BYTE  text2[]         = "Muenchener Strasse 17";
  BYTE  randomSeed[16]  = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
  BYTE  sign[keylen/8]  = {0};
  int   signlen;
  int   rc;

  SIGNATURE_CTX   ctx;
  RSA_PRIVATE_KEY rsa_priv;
  RSA_PUBLIC_KEY  rsa_pub;

  GenerateRSAKey( &rsa_priv, &rsa_pub, randomSeed, sizeof(randomSeed), keylen );

  SignInit  ( &ctx, SHA1 );
  SignUpdate( &ctx, text1, sizeof(text1) );
  SignUpdate( &ctx, text2, sizeof(text2) );
  SignFinal ( &ctx, sign, &signlen, &rsa_priv );

  VerifyInit  ( &ctx, SHA1 );
  VerifyUpdate( &ctx, text1, sizeof(text1) );
  VerifyUpdate( &ctx, text2, sizeof(text2) );
  rc = VerifyFinal( &ctx, sign, signlen, &rsa_pub );

  printf( "\nVerify completed: RC=%d\n", rc );
  return 0;
}
```

Chapter

5

Hash methods

Common information

Hash methods play an important role in the field of security. They are used every time a definite value needs to be calculated for input data of arbitrary size. The resulting check sum can later on be used to verify the integrity of the data.

CryptLib supports the hash methods **MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512** and **RipeMD160**. Besides these **HMAC** (Keyed-Hashing for Message Authentication) is supported.

Use the following steps to create a hash value:

- Initialize the context using the *DigestInit* function to specify the desired hash method.
- Add data to be hashed using the *DigestUpdate* function. This method can be called as often as needed.
- Call the *DigestFinal* method to finalize the procedure and to get hold of the hash value.

Methods

DigestInit

Initialize the hash context.

Syntax	**int DigestInit(DIGEST_CTX *ctx, int digestAlgo);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Address pointer to receive the storage address of the created context. This will be used for subsequent processing.	Output
digestAlgo	The type of hash to be used. CryptLib supports MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.	Input

DigestUpdate

Add data to be hashed.

Syntax	**int DigestUpdate(DIGEST_CTX *ctx, unsigned char *inputdata, int datalength);**
Return code	0 or error code (< 0).

Parameter	Description	Use
ctx	Storage address of the context initialized for hashing.	Input
inputdata	Storage address of data to be hashed.	Input
inputlength	Length of the data to be hashed.	Input

DigestFinal

Create the hash value.

Syntax	int DigestFinal(DIGEST_CTX *ctx, unsigned char *hashdata, int *hashlength);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context initialized for hashing.	Input
hashdata	Address of storage area about to hold the created hash value.	Output
hashlength	Address of storage area about to hold the length of the created hash value.	Output

Example:

```
#include <stdlib.h>
#include "XPSCRYPT.H"

int main(void)
{
  BYTE  text1[32]       = "XPS Software GmbH, Haar/Muenchen";
  BYTE  text2[]         = "Muenchener Strasse 17";
  char  md5Hash[16]     = {0};
  char  sha1Hash[20]    = {0};
  char  ripeHash[20]    = {0};
  int   digestlen;

  DIGEST_CTX   ctx;

  DigestInit  ( &ctx, MD5 );
  DigestUpdate( &ctx, text1, sizeof(text1) );
  DigestUpdate( &ctx, text2, sizeof(text2) );
  DigestFinal ( &ctx, md5Hash, &digestlen );

  DigestInit  ( &ctx, SHA1 );
  DigestUpdate( &ctx, text1, sizeof(text1) );
  DigestUpdate( &ctx, text2, sizeof(text2) );
  DigestFinal ( &ctx, sha1Hash, &digestlen );

  DigestInit  ( &ctx, RIPEMD160 );
  DigestUpdate( &ctx, text1, sizeof(text1) );
  DigestUpdate( &ctx, text2, sizeof(text2) );
  DigestFinal ( &ctx, ripeHash, &digestlen );
  return 0;
}
```

HMAC

Create a MAC. A Message-Authentication-Code or MAC is a key dependent one way hash method. Therefore a MAC can only be created or verified using a key. This prevents firstly a MAC secured message to be changed by an unauthorized user who does not possess the key and secondly the MAC from unauthorized re-calculation. Thus a MAC can be used to guarantee the integrity of data without the need for data encryption.

Syntax	int HMAC(BYTE *input, int inputlength, BYTE *key, int keylength, BYTE *digestMessage, int digestAlgo);	
Return code	Length of the HMAC or error code (< 0).	
Parameter	**Description**	**Use**

input	Storage address of the data.	Input
inputlength	Length of the data.	Input
key	Storage address of the key.	Input
keylength	Length of the key.	Input
digestMessage	Address of storage area about to hold the created hash value.	Output
digestAlgo	The type of hash to be used. CryptLib supports MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.	Input

Chapter

6

X.509 Certificates

Common information

Certificates exist since the invention of public key algorithms. Sometimes certificates are called digital passports. Simply spoken a certificate is no more than a signed piece of data. Two parties are involved in the process of creating a certificate: the certificate issuer and the certificate subject. Both of them need to have an asymmetrical pair of keys.

In order for the subject to be certified he has to transmit his public key to the signer. The issuer creates a data record including the name of the issuer, the name of the subject and the subjects' public key. Finally the resulting data record will be signed with the issuer's private key.

This data record in conjunction with the signature is called the certificate. Thus the issuer attests the owner of the certificate that he is also the owner of the associated public key. Assuming the subject's public key is available the unsophistication of the electronical passport can be verified.

CryptLib offers the program developer methods to verify the unsophistication of certificates as well as methods to extract data stored in the certificate.

Methods

ImportCertificate

Reading a certificate object and checking its formal correctness.

Syntax	**int ImportCertificate(void *certObject, void **certctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
certObject	Storage address of a X.509 certificate. CryptLib supports binary and Base64 encrypted certificates.	Input
certctx	Address pointer to receive the storage address of the created certificate context. This will be used for subsequent processing of the certificate.	Output

GetPublicKey

Extract the public key from the certificate.

Syntax	**int GetPublicKey(void *certctx, RSA_PUBLIC_KEY *pubkey, int outlen);**

Return code	Length of the public key or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
pubkey	Address of the storage area about to hold the extracted public key.	Output
outlen	Maximum size of the storage area reserved for the public key.	Input

GetCryptAlgo

Extract encryption algorithm of the public key.

Syntax	**int GetCryptAlgo(void *certctx, char *algo, int outlen);**	
Return code	Length of the name of the encryption algorithm or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
algo	Address of the storage area about to hold the extracted name of the encryption algorithm.	Output
outlen	Maximum size of the storage area reserved for the name.	Input

GetCryptKeylen

Extract the key length of the public key.

Syntax	**int GetCryptKeylen(void *certctx);**	
Return code	Length of the public key or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input

GetVersionInfo

Extract version number of the certificate.

Syntax	**int GetVersionInfo(void *certctx, char *version, int outlen);**	
Return code	Length of the version number or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
version	Address of the storage area about to hold the extracted version number.	Output
outlen	Maximum size of the storage area reserved for the version number.	Input

GetSerialNumber

Extract the serial number of the certificate.

Syntax	**int GetSerialNumber(void *certctx, BYTE *serial, int outlen);**
Return code	Length of the serial number or error code (< 0).

Parameter	Description	Use
certctx	Storage address of the certificate context.	Input
serial	Address of the storage area about to hold the extracted serial number.	Output
outlen	Maximum size of the storage area reserved for the serial number.	Input

GetIssuerDN

Extract information about the issuer.

Syntax	**int GetIssuerDN(void *certctx, char *dn, int outlen);**	
Return code	Length of issuer information or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
dn	Address of the storage area about to hold the extracted information about the issuer. Single elements (CN, O, OU etc.) will be separated by comma.	Output
outlen	Maximum size of the storage area reserved for the issuer information.	Input

GetSubjectDN

Extract information about the subject.

Syntax	**int GetSubjectDN(void *certctx, char *dn, int outlen);**	
Return code	Length of subject information or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
dn	Address of the storage area about to hold the extracted information about the subject. Single elements (CN, O, OU etc.) will be separated by comma.	Output
outlen	Maximum size of the storage area reserved for the subject information.	Input

GetSignatureAlgo

Extract the signature algorithm used by the certificate signer.

Syntax	**int GetSignatureAlgo(void *certctx, char *algo, int outlen);**	
Return code	Length of the name of the algorithm or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
algo	Address of the storage area about to hold the name of the extracted encryption algorithm.	Output
outlen	Maximum size of the storage area reserved for the name of the encryption algorithm.	Input

GetSignature

Extract the signature of the certificate.

Syntax	int GetSignature(void *certctx, BYTE *signature, int outlen);	
Return code	Length of the signature or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
signature	Address of the storage area about to hold the extracted signature.	Output
outlen	Maximum size of the storage area reserved for the signature.	Input

GetStartDate

Extract begin of validity of the certificate.

Syntax	int GetStartDate(void *certctx, char *startdate, int outlen);	
Return code	Length of the date or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
startdate	Address of the storage area about to hold the extracted start of validity. The date will be returned as DD.MM.YYYY HH:MM:SS.	Output
outlen	Maximum size of the storage area reserved for the date.	Input

GetEndDate

Extract the end of validity of the certificate.

Syntax	int GetEndDate(void *certctx, char *enddate, int outlen);	
Return code	Length of the date or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
startdate	Address of the storage area about to hold the extracted end of validity. The date will be returned as DD.MM.YYYY HH:MM:SS.	Output
outlen	Maximum size of the storage area reserved for the date.	Input

GetIssuerDNBlob

Extract issuer's data in binary format including the ASN.1 control characters. The resulting BLOB may be used to follow the further path of certification.

Syntax	int GetIssuerDNBlob(void *certctx, BYTE **blob);	
Return code	Length of the certificate issuer BLOB or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
blob	Address pointer about to receive the address of the storage area holding the extracted issuer data BLOB.	Output

GetSubjectDNBlob

Extract subject's data in binary format including the ASN.1 control characters. The resulting BLOB may be used to follow the further path of certification.

Syntax	**int GetSubjectDNBlob (void *certctx, BYTE **blob);**	
Return code	Length of the certificate subject BLOB or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
blob	Address pointer about to receive the address of the storage area holding the extracted subject data BLOB.	Output

GetIssuerDNByType

Extract specific issuer element specified via parm *type*

Syntax	**int GetIssuerDNByType(void *certctx, int type, char *data, int outlen);**	
Return code	Length of issuer data or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
type	Issuers' DN type. The following types are available: DN_C Country DN_SP State/Province DN_L Locality DN_O OrganizationName DN_OU OrganizationUnit DN_CN CommonName DN_EMAIL E-Mail DN_STREET Street DN_PHONE Phone DN_POSTAL PostalCode DN_TITLE Title	Input
data	Address of the storage area about to hold the extracted issuer data.	Output
outlen	Maximum size of the storage area reserved for the issuer data.	Input

GetSubjectDNByType

Extract specific subject element specified via parm *TYPE*.

Syntax	**int GetSubjectDNByType(void *certctx, int type, char *data, int outlen);**	
Return code	Length of subject data or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
type	Subjects' DN type. The following types are available: DN_C Country DN_SP State/Province DN_L Locality DN_O OrganizationName DN_OU OrganizationUnit DN_CN CommonName	Input

	DN_EMAIL E-Mail DN_STREET Street DN_PHONE Phone DN_POSTAL PostalCode DN_TITLE Title	
data	Address of the storage area about to hold the extracted subject data.	Output
outlen	Maximum size of the storage area reserved for the subject data.	Input

GetFirstExtension

The standard fields supported by X.509 certificates may not be sufficient for some applications. Because of this beginning with version 3 the X.509 syntax has been modified introducing an extension component. The extension component makes it possible to include arbitrary data in the certificate.

Syntax	**int GetFirstExtension(void *certctx, CERTEXT *ext);**	
Return code	Length of the extension area (0 if not available) or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
ext	Address pointing to the following structure.	Output
Felder	**Description**	
ext.oid_binary	Binary value of the OID (object identifier).	
ext.oid_char	Character value of the OID (object identifier).	
ext.isCritical	Flag to mark the extension as critical or uncritical. Critical extensions have to be considered always. If a program detects a critical extension and doesn't know it's meaning further use of the certificate should be avoided. Uncritical extensions may be ignored by the processing program.	
ext.fieldType	Field type of the extension. The following types are possible: BER_BOOLEAN BER_INTEGER BER_BITSTRING BER_OCTETSTRING BER_OBJECT_IDENTIFIER BER_OBJECT_DESCRIPTOR BER_EXTERNAL BER_REAL BER_ENUMERATED BER_EMBEDDED_PDV BER_STRING_UTF8 BER_RELATIVE_OID BER_STRING_NUMERIC BER_STRING_PRINTABLE BER_STRING_T61 BER_STRING_GRAPHIC BER_STRING_ISO646 BER_STRING_VIDEOTEXT BER_STRING_GENERAL BER_STRING_UNIVERSAL BER_CHAR_STRING BER_STRING_BMP	
ext.value	Integer value for the types BER_BOOLEAN, BER_INTEGER, BER_ENUMERATED.	
ext.data	Storage address of the extensions binary data area.	
ext.datalen	Length of the extensions binary data area.	

GetNextExtension

Extract the next certificate extension.

Syntax	**int GetNextExtension(void *certctx, CERTEXT *ext);**	
Return code	Length of the extension area (0 if not available) or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
ext	Storage address of the extracted extension. For an explanation of the	Output

	format of the extension area see method GetFirstExtension.	

GetExtensionByOID

Extract a certificate extension for a specific OID (object identifier).

Syntax	**int GetExtensionByOID(void *certctx, BYTE *oid, CERTEXT *ext);**	
Return code	Lentth of the extension area (0 if not available) or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
oid	Storage address of the binary object identifier about to be extracted.	Input
ext	Storage address of the extracted extension. For an explanation of the format of the extension area see method GetFirstExtension.	Output

GetFingerPrint

Create a fingerprint (hash value) for the certificate. A fingerprint may be used to enable visual examination of a certificate.

Syntax	**int GetFingerPrint(void *certctx, int digestAlgo, BYTE *data, int outlen);**	
Return code	Length of the fingerprint or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
digestAlgo	The type of hash to be used. CryptLib supports MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.	Input
data	Address of the storage area about to hold the created fingerprint.	Output
outlen	Maximum size of the storage area reserved for the fingerprint.	Input

VerifyCertificate

Check the validity of a certificate.

Syntax	**int VerifyCertificate(void *certctx, RSA_PUBLIC_KEY pubkey);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input
pubkey	Issuer's public key.	Input

CleanupCertificate

Deallocate storage used by certificate routines.

Syntax	**int CleanupCertificate(void *certctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
certctx	Storage address of the certificate context.	Input

Example:

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "XPSCRYPT.H"

int  getRootPublicKey( char *file, RSA_PUBLIC_KEY * );
void printExtension( CERTEXT *extension );

typedef struct _dn_tab
{
  char name[21];
  int  type;
}DN_TAB;


/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
/*     parm1: certificate                                       *
/*     parm2: root-certificate [OPTIONAL]                       *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
// ------------------------------------------------------------ //
{
  RSA_PUBLIC_KEY  pubkey   = {0};
  BYTE           *cert;
  BYTE           *certctx;
  int             iCount, iRc, i1;
  unsigned char   temp[512]  = {0};
  unsigned char   temp2[128] = {0};
  unsigned char   oid[] = {0x06,0x09,0x60,0x86,0x48,0x01,0x86,0xF8,0x42,0x01,0x04};
  char           *ptr1, *ptr2;
  CERTEXT         extension;


  DN_TAB          DN_TAB[11] =
  {
    "CN=",     DN_CN,
    "O=",      DN_O,
    "OU=",     DN_OU,
    "L=",      DN_L,
    "SP=",     DN_SP,
    "STREET=", DN_STREET,
    "POSTAL=", DN_POSTAL,
    "C=",      DN_C,
    "EMAIL=",  DN_EMAIL,
    "PHONE=",  DN_PHONE,
    "TITEL=",  DN_TITEL
  };

  iCount = readFile( argv[1], &cert );
  if( iCount < 1 )
  {
    printf( "Certificate \"%s\" not found.\n", argv[1] );
    return( -1 );
  }

  iRc = ImportCertificate( cert, &certctx );
  if( iRc != 0 )
  {
    printf( "invalid certificate \"%s\": rc=%d\n", iRc );
    return( -1 );
  }

  if( argc > 2 )     // if parm2 -> verify certificate
  {
    iRc = getRootPublicKey( argv[2], &pubkey );
    if( iRc != 0 )
      return( -1 );

    iRc = VerifyCertificate( certctx, &pubkey );
    if( iRc != 0 )
      printf("Verify Error: rc=%d\n\n", iRc);
  }

  iCount = GetVersionInfo( certctx, temp, sizeof(temp) );
  printf("Version           = %s\n", temp);

  iCount = GetSerialNumber( certctx, temp, sizeof(temp) );
  printf("Serialnr          = %02X", temp[0]);
  for( i1 = 1; i1 < iCount; i1++ )
    printf("%02X", temp[i1]);
  printf("\n");
```

```
  iCount = GetSignatureAlgo( certctx, temp, sizeof(temp));
  printf("SignatureAlgo     = %s\n", temp);

  iCount = GetIssuerDN( certctx, temp, sizeof(temp) );
  strcpy( temp2, "\nIssuer            = " );
  ptr1 = temp;
  while( iCount )
  {
    unsigned char   temp3[128] = {0};
    ptr2 = strchr( ptr1, ',' );
    if( ptr2 )
    {
      memcpy( temp3, ptr1, ptr2-ptr1 );
      strcat( temp2, temp3 );
      iCount -= (ptr2-ptr1);
    }
    else
    {
      memcpy( temp3, ptr1, iCount );
      strcat( temp2, temp3 );
      iCount = 0;
    }
    printf( "%s\n", temp2 );
    strcpy( temp2, "                    " );
    ptr1 = ptr2+2;
  }
  printf("\n");

  iCount = GetStartDate( certctx, temp, sizeof(temp) );
  printf("StartDate         = %s\n", temp);

  iCount = GetEndDate( certctx, temp, sizeof(temp) );
  printf("EndDate           = %s\n", temp);

  strcpy( temp2, "\nSubject           = ");
  for( i1 = 0; i1 < 11; i1++ )
  {
    iCount = GetSubjectDNByType( certctx, DN_TAB[i1].type, temp, sizeof(temp) );
    if( iCount > 0 )
      printf("%s%s%s\n", temp2, DN_TAB[i1].name, temp);
    strcpy( temp2, "                    ");
  }
  printf("\n");

  iCount = GetCryptAlgo( certctx, temp, sizeof(temp) );
  printf("Algorithm         = %s", temp);

  iCount = GetCryptKeylen( certctx );
  printf("(%d)\n", iCount);

  iCount = GetPublicKey( certctx, (RSA_PUBLIC_KEY *)temp, sizeof(RSA_PUBLIC_KEY) );
  DumpData( "PublicKey", temp, iCount );

  iCount = GetSignatureAlgo( certctx, temp, sizeof(temp) );
  printf("\nSignature-Algo    = %s\n", temp);

  iCount = GetSignature( certctx, temp, sizeof(temp) );
  DumpData( "Signature", temp, iCount );

  iCount = GetFirstExtension( certctx, &extension );
  if( iCount == 0 )
  {
    printExtension( &extension );
    while( iCount == 0 )
    {
      iCount = GetNextExtension( certctx, &extension );
      if( iCount == 0 )
       printExtension( &extension );
    }
  }

  iCount = GetExtensionByOID( certctx, oid, &extension );
  if( iCount == 0 )
    printExtension( &extension );

  iCount = GetFingerprint( certctx, SHA1, temp, sizeof(temp) );
  DumpData( "SHA1 Fingerprint", temp, iCount );
  iCount = GetFingerprint( certctx, MD5, temp, sizeof(temp) );
  DumpData( "MD5 Fingerprint", temp, iCount );

  CleanupCertificate( certctx );
  CleanupFile( cert );
  return 0;
}

/* ------------------------------------------------------------ *
 *                                                              */
   void printExtension
/*                                                              *
```

```
 * ------------------------------------------------------------ */
( CERTEXT    *extension)
// ------------------------------------------------------------ //
{
  printf("\n\nExtension         = %s\n", extension->oid_char );
  printf("   isCritical     = %d\n", extension->isCritical);
  printf("   Type           = %d\n", extension->fieldType);
  if( extension->datalen == 0 )
    printf("   value          = %d\n", extension->value);
  else
    DumpData( "   data", extension->data, extension->datalen );
}

/* ------------------------------------------------------------ *
 *                                                              */
   int getRootPublicKey
/*                                                              *
 * ------------------------------------------------------------ */
( char            *rootFile,
  RSA_PUBLIC_KEY  *pubkey
)
// ------------------------------------------------------------ //
{
  BYTE  *buffer;
  BYTE  *rootctx;
  int    iRc, iCount;

  iCount = readFile( rootFile, &buffer );
  if( iCount == 0 )
  {
    printf( "file \"%s\" not found\n", rootFile );
    return( -1 );
  }

  iRc = ImportCertificate( buffer, &rootctx );
  if( iRc != 0 )
  {
    printf( "invalid root certificate \"%s\": rc=%d\n", iRc );
    return( -1 );
  }

  iRc = GetPublicKey( rootctx, pubkey, sizeof(RSA_PUBLIC_KEY) );
  if( iRc < 1 )
  {
    printf( "invalid root public-key: rc=%d\n", iRc );
    return( -1 );
  }

  CleanupCertificate( rootctx );
  CleanupFile( buffer );
  return( 0 );
}
```

Chapter

7

S/MIME Objects (PKCS#7)

Common information

PKCS#7 also known as *Cryptographic-Message-Syntax-Standard* describes methods to secure data using cryptographic procedures such as digital signatures or encryption. CryptLib supports the following content types:

Data	Simply used to model data. This type offers no cryptographic functions.
Signed-data	Describes a format to ensure data integrity and sender authenticity by means of digital signatures and certificates.
Enveloped-data	Used to encrypt data in a receiver specific way to disable trespassers from reading the message (confidentiality).
Encrypted-data	Used to encrypt data.

Methods

ImportPKCS7Data

Read a PKCS#7 data object and guarantee its formal correctness. The content type *Data* describes an arbitrary sequence of data bytes. Data may be retrieved using the methods *GetFirstData* and *GetNextData*.

Syntax	**int ImportPKCS7Data(BYTE *pkcs7object, int lengthobject, void **pkcs7ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7Object	Storage address of the PKCS#7 data object. Binary, Base64 encrypted and S/MIME formats are supported.	Input
lengthobject	Length of the PKCS#7 data object.	Input
pkcs7ctx	Storage address of the imported PKCS#7 context. This object will be needed for subsequent processing.	Output

ImportSignedData

Read a PKCS#7 signed data object and guarantee its formal correctness. The content type *Signed-data* defines syntax for calculation and transport of digital signatures. The message may be signed by an arbitrary number of signers.

Data can be extracted using the methods *GetFirstData* and *GetNextData* respectively. Signers can be leached using the methods *GetFirstSigner* and *GetNextSigner* respectively. The methods *VerifySigner* and *VerifyAllSigner* can be used to examine data integrity. Finally the *AddSignerCert* method can be used to add certificate issuers.

Syntax	**int ImportSignedData(BYTE *pkcs7object, int lengthobject, void **pkcs7ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7Object	Storage address of the PKCS#7 data object. Binary, Base64 encrypted and S/MIME formats are supported.	Input
lengthobject	Length of the PKCS#7 data object.	Input
pkcs7ctx	Storage address of the imported PKCS#7 context. This object will be needed for subsequent processing.	Output

ImportEnvelopedData

Read a PKCS#7 enveloped data object and guarantee its formal correctness. The content type *Enveloped-data* defines syntax for receiver specific message encryption. This means that information about the intended receiver may be part of the message. This is carried out using a technique called 'digital enveloping'.

Equivalent to the *Signed-data* type accepting an arbitrary number of signers the *Enveloped-data* type permits the inclusion of multiple message receivers. Information about specific receivers is included using the type 'RecipientInfo'.

Data can be extracted using the methods *GetFirstData* and *GetNextData* respectively.

Syntax	**int ImportEnvelopedData(BYTE *pkcs7object, int lengthobject, BYTE *pkcs12object, int lengthpkcs12, BYTE *pwd, int lengthpwd, void **pkcs7ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7Object	Storage address of the PKCS#7 data object. Binary, Base64 encrypted and S/MIME formats are supported.	Input
lengthobject	Length of the PKCS#7 encrypted data object.	Input
pkcs12Object	Storage address of a PKCS#12 object. PKCS#12 objects define syntax for the exchange of keys and certificates. Included in PKCS#12 objects are key-bags and certificate-bags. Using the certificate-bags it will be possible to find out if the PKCS#7 object contains a RecipientInfo for the user. Assuming a RecipientInfo is contained the users private key can be extracted from the key-bag. This private key is used to encrypt the *Content-Encryption* key which in turn can be used to decrypt the data.	Input
lengthpkcs12	Length of the PKCS#12 object.	Input
pwd	PKCS#12 objects are sealed with a password. Using this parameter the storage address of the password has to be made known.	Input
lengthpwd	Length of the password.	Input
pkcs7ctx	Storage address of the imported PKCS#7 context. This object will be needed for subsequent processing.	Output

ImportEncryptedData

Read a PKCS#7 encrypted data object and guarantee its formal correctness. The content type *Encrypted-data* defines syntax for message encryption. Unlike *Enveloped-data* processing it's assumed the receiver does already posses the *Content-Encryption* key thus making it superfluous to explicitly specify it.

This type mainly qualifies for storing encrypted data. As a prominent application the *Personal-Information-Syntax-Standard* PKCS#12 can be named.

Data can be extracted using the methods *GetFirstData* and the *GetNextData* respectively.

Syntax	**int ImportEncryptedData(BYTE *pkcs7object, int lengthobject, BYTE *pwd, int lengthpwd, void **pkcs7ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7Object	Storage address of the PKCS#7 data object. Binary, Base64 encrypted and S/MIME formats are supported.	Input
lengthobject	Length of the PKCS#7 encrypted data object.	Input
pwd	Address of storage area holding the password used to encrypt the *Content-Encryption* key.	Input
lengthpwd	Length of the password.	Input
pkcs7ctx	Storage address of the imported PKCS#7 context. This object will be needed for subsequent processing.	Output

CreatePKCS7Data

Create a PKCS#7 data object. PKCS#7 data objects are used to model data and don't offer any cryptographic methods. Data can be added using the method *AddPKCS7Data*.

Syntax	**int CreatePKCS7Data(int option, void **pkcs7ctx);**		
Return code	0 or error code (< 0).		
Parameter	**Description**		**Use**
option	HEADER_INCLUDED	If this option is given the ContentType will be added.	Input
pkcs7ctx	Storage address of the created PKCS#7 context.		Output

CreateSignedData

Create a PKCS#7 signed data object. The content type *Signed-data* defines syntax for calculation and transport of digital signatures. The number of parties signing a message can be arbitrary. Signers can be added using the method *AddSigner*, using the method *AddPKCS7Data* data can be added. Additional certificates can be added using the method *AddSignerCert*.

Syntax	**int CreateSignedData(int option, void **pkcs7ctx);**		
Return code	0 or error code (< 0).		
Parameter	**Description**		**Use**
option	HEADER_INCLUDED	If this option is given the ContentType will be added.	Input
	DATA_IMPLICIT	If this option is specified the message text will be added to the signed-data object. This means that the content field will be available to include the message text. If	

		this option is not specified the content field will be absent and the message text has to be transferred using a different way.	
	CERT_IMPLICIT	If this option is specified all signer certificates contained in the PKCS#12 file will be included in the signed-data object.	
pkcs7ctx	Storage address of the created PKCS#7 context.		Output

CreateEnvelopedData

Create a PKCS#7 enveloped data object. The content type *Enveloped-data* defines syntax for receiver specific message encryption. This means that information about the intended receiver may be part of the message. This is carried out using a technique called 'digital enveloping'.

Equivalent to the *Signed-data* type accepting an arbitrary number of signers the *Enveloped-data* type permits the inclusion of multiple message receivers.

Receivers can be added using the method *AddRecipient* while data can be added using the method *AddPKCS7Data*.

Syntax	**int CreateEnvelopedData(int option, void **pkcs7ctx, int encryptionAlgo);**		
Return code	0 or error code (< 0).		
Parameter	**Description**		**Use**
option	HEADER_INCLUDED	If this option is given the ContentType will be added.	Input
pkcs7ctx	Storage address of the created PKCS#7 context.		Output
encryptionAlgo	Algorithm to use for data encryption. CryptLib supports the following algorithms: DESEDE3CBC — Triple DES 168Bit DESCBC — DES 56Bit RC2CBC_128 — RC2 128Bit RC2CBC_64 — RC2 64Bit RC2CBC_40 — RC2 40Bit RC4_128 — RC4 128Bit RC4_64 — RC4 64Bit RC4_40 — RC4 40Bit		Input

CreateEncryptedData

Create a PKCS#7 encrypted data object. The content type *Encrypted-data* defines syntax for message encryption. Unlike *Enveloped-data* processing it's assumed the receiver does already posses the *Content-Encryption* key thus making it superfluous to explicitly specify it.

This type mainly qualifies for storing encrypted data.

Data can be added using the method *AddPKCS7Data*.

Syntax	**int CreateEncryptedData(int option, void **pkcs7ctx, int pbeAlgo, BYTE *pwd, int lengthpwd);**		
Return code	0 or error code (< 0).		
Parameter	**Description**		**Use**
option	HEADER_INCLUDED	If this option is given the ContentType will be added.	Input

pkcs7ctx	Storage address of the created PKCS#7 context.	Output
pbeAlgo	Algorithm to use for data encryption. CryptLib supports the following algorithms PBE3DES_3Key Triple DES 168Bit PBE3DES_2Key Triple DES 112Bit PBERC2_128 RC2 128Bit PBERC2_40 RC2 40Bit PBERC4_128 RC4 128Bit PBERC4_40 RC4 40Bit	Input
pwd	Storage address of the password used for key generation.	Input
lengthpwd	Length of the password.	Input

AddPKCS7Data

Import methods
Signed data objects are able to process data IMPLICIT or EXPLICT. Choosing IMPLICIT mode has the effect that data will be included in the signed data object. This means more precisely that the *content* field including the message content will be present. If EXPLICIT mode is chosen the *content* field will be absent meaning the message content has to be transferred any other way. This method offers the possibility to transmit data to the signed data object.

Create methods
Hereby the methods *CreatePKCS7Data, CreateSignedData, CreateEnvelopedData* and *CreateEncryptedData* are used to add data to the PKCS#7 object.

Syntax	**int AddPKCS7Data(void *pkcs7ctx, BYTE *data, int lengthdata);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
data	Storage address of the data about to add.	Input
lengthdata	Length of data.	Input

AddMessageDigest

Signed data objects being processd in EXPLICIT mode can have a Message Digest (hash value calculated for the data) added. If an externally calculated Message Digest is added the method *AddPKCS7Data* must not be called.

Syntax	**int AddMessageDigest(void *pkcs7ctx, BYTE *data, int lengthdata);**	
Returncode	0 or error code (< 0).	
Parameter	**Beschreibung**	**Verwendung**
pkcs7ctx	The storage address of the PKCS#7 context.	Eingabe
data	The storage address of the Message Digest.	Eingabe
lengthdata	The length of the Message Digest.	Eingabe

AddSigner

A Signed-data object needs to be signed by one or more signers. Using this method based on a PKCS#12 file which includes the secret key as well as the signer's X.509 certificate a *SignerInfo* structure ready to sign the object will be created.

Syntax	int AddSigner(void *pkcs7ctx, BYTE *pkcs12obj, int iLenpkcs12, BYTE *pwd, int lengthpwd);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
pkcs12Object	Storage address of the signers PKCS#12 object.	Input
lengthobject	Length of the PKCS#12 object.	Input
pwd	Storage address of the password used to encrypt the PKCS#12 object.	Input
lengthpwd	Length of the password.	Input

AddSignerExtern

This method works identically as the *AddSigner* method. The only difference is the fact that the signature to use won't be generated by XPS CryptLib. This task has to be carried out by a transmitted callback method. Besides this the X.509 certificate has to be made known to XPS CryptLib using another callback method.

For example this method can be used to let XPS CryptLib create a PKCS#7 object while the signature is created from another source such as a smartcard.

Syntax	int AddSignerExtern(void *pkcs7ctx, void *userdata, int (*SignData)(), int (*ReadX509Cert)());	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
userdata	Storage address of a user area which will be made available to the callback methods.	Input
*int (*SignData)()*	Callback method to be used to sign the PKCS#7 object.	Input
*int (*ReadX509Cert)()*	Callback method to be used to read the X.509 certificate.	Input

Syntax	int (*SignData)(void *userdata, BYTE *hash, int iLenHash, BYTE *signature, int *iLenSign);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
userdata	Storage address of a user area made available to the *AddSignerExtern* method.	Input
hash	Storage address of the hash value about to sign.	Input
iLenHash	Length of the hash value about to sign.	Input
signature	The storage address of the signature. The callback method has to store the signature into this storage area. The maximum length of the signature is 512 byte. A signature length of 512 byte corresponds to a 4096 bit RSA key.	Output
iLenSign	The length of the signature. This may not exceed 512 byte.	Output

Syntax	int (*ReadX509Cert)(void *userdata, BYTE **cert, int *iLenCert);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**

userdata	Storage address of a user area made available to the *AddSignerExtern* method.	Input
cert	Storage address of the X.509 certificate.	Output
iLenCert	Length of the X.509 certificate.	Output

AddRecipient

While creating an *EnvelopedData* object, information about the intended receiver has to be added. This means the message has to be receiver specificly encrypted. Using this method the receiver's X.509 certificate is made known. The certificate includes the public key which will be used to encrypt the symmetrical *Content-Encryption* key.

Syntax	**int AddRecipient(void *pkcs7ctx, BYTE *cert, int lengthcert);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
cert	Storage address of the receivers X.509 certificate.	Input
lengthcert	The length of the certificate.	Input

AddSignerCert

Import methods
In order to verify a *Signed-data* object the hierarchy of signer certificates needs to be validated. If the PKCS#7 object doesn't contain signer certificates this methods provides the possibility to deliver signer certificates to the *Signed-data* object.

Create methods
When the *CreateSignedData* method is executed all signer certificates stored in the PKCS#12 file will be added. Additional certificates can be added using this method.

Syntax	**int AddSignerCert(void *pkcs7ctx, BYTE *cert, int lengthcert);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
cert	Storage address of the signers X.509 certificate.	Input
lengthcert	The length of the certificate.	Input

AddTrustedSigner

During the process of verification of a Signed-data object the signer's certificate can be examined for a trusted signer. Using this method, certificates of trusted signers can be added.

Syntax	**int AddTrustedSigner(void *pkcs7ctx, BYTE *cert, int lengthcert);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
cert	Storage address of the trusted signers X.509 certificate.	Input
lengthcert	The length of the certificate.	Input

ForceTrustedSigner

Using this function prior to the verification of a *Signed-data* object it's possible to overwrite a certificate with an equal identity (issuer- and subject-blob) that might be already contained in the the PKCS#7 object.

Syntax	**int ForceTrustedSigner(void *pkcs7ctx, BYTE *cert, int lengthcert);**	
Returncode	0 oder Fehlercode (< 0).	
Parameter	**Beschreibung**	**Verwendung**
pkcs7ctx	Speicheradresse des PKCS#7-Kontexts.	Eingabe
cert	Speicheradresse des X.509 Zertifikates des vertrauenswürdigen Ausstellers.	Eingabe
lengthcert	Länge des Zertifikates.	Eingabe

GetFirstSigner

Signed-data objects may be examined for information about the signers (*SignerInfos*). Using this method, the first SignerInfo structure can be extracted.

Syntax	**int GetFirstSigner(void *pkcs7ctx, SIGNERINFO **signer);**	
Return code	0 if a signer is available else < 0.	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
signer	Storage address of the extracted SignerInfo. This structure includes information about the certificate, the serial number, the issuer, the attributes, the Message Digest, the digest algorithm and the signature.	Output

GetNextSigner

Signed-data objects may be examined for information about the signers (*SignerInfos*). Using this method, subsequent SignerInfo structures can be extracted.

Syntax	**int GetNextSigner(void *pkcs7ctx, SIGNERINFO **signer);**	
Return code	0 if a signer is available else < 0.	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
signer	Storage address of the extracted SignerInfo. This structure includes information about the certificate, the serial number, the issuer, the attributes, the Message Digest, the digest algorithm and the signature.	Output

GetSigningAlgo

Signed-data objects may be examined for information about the used encryption- and hash-algorithm. This method needs a SingerInfo structure for input which will be returned from calling *GetFirstSigner* and *GetNextSigner*.

Syntax	**int GetSigningAlgo(void *pkcs7ctx, SIGNERINFO *signer, char *AlgoInfo, int AlgoLen);**

Return code	Length of the extracted *AlgoInfo*.	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
signer	Storage address of the SIGNERINFO.	Input
AlgoInfo	Address of storage area to use to store the required information. This information will be returned as a string containing the encryption algorithm and the hash algorithm separated by a slash character e.g "rsaEncryption/sha-1".	Input/Output
AlgoLen	Length of the storage area available for the *AlgoInfo*.	Input

GetSigningTime

Signed-data objects may be examined for information about the signing time. This method needs a SingerInfo structure for input which will be returned from calling *GetFirstSigner* and *GetNextSigner*.

Syntax	**int GetSigningTime(void *pkcs7ctx, SIGNERINFO *signer, char *SigningTime, int SigningTimeLen);**	
Return code	Length of the extracted *SigningTime*.	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
signer	Storage address of the SIGNERINFO.	Input
SigningTime	Address of storage area to use to store the required information. The signing time will be returned as a string with the following format: "YYMMDDHHMMZ". The single characters have the following meaning: YY year (00 – 99) MM month (01 - 12) DD day (01 - 31) HH hour (00 - 23) MM minute (00 - 59) Z The "Z" character indicates Greenwich Mean Time (GMT).	Input/Output
SigningTimeLen	Length of the storage area available for the *SigningTime*.	Input

GetNextSignerCert

Using this method the current signer's next signer certificate will be extracted.

Syntax	**int GetNextSignerCert(void *pkcs7ctx, BYTE **cert);**	
Return code	0 if no more certificates are available, else the length of the certificate.	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
cert	Storage address of the extracted signer certificate.	Output

VerifySigner

Using this method the *Signed-data* object will be checked for validity regarding a specific signer. The required SignerInfo strucuture has to be previously extracted calling one of the *GetFirstSigner* and *GetNextSigner* methods.

Syntax	**int VerifySigner(void *pkcs7ctx, SIGNERINFO **signer, int checkcertchain);**	
Return code	0 if signer can be verified, else error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
signer	Storage address of the SignerInfo structure about to verify.	Input
checkcertchain	If set to 0 the certificate paths of the signer and the trusted signers won't be checked. If set to 1 the path of certificates will be checked up to the root. In this case the issuer's certificate has to be loaded in advance using the AddTrustedSigner method.	Input

VerifyAllSigner

All signers of the Signed-data object will be checked for validity.

Syntax	**int VerifyAllSigner(void *pkcs7ctx, int checkcertchain);**	
Return code	0 if all signers can be verified, else error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
checkcertchain	If set to 0 the certificate paths of the signer and the trusted signers won't be checked. If set to 1 the path of certificates will be checked up to the root. In this case the issuer's certificate has to be loaded in advance using the AddTrustedSigner method.	Input

GetFirstPKCS7Data

The PKCS#7 objects data will be extracted. This method is available for all supported PKCS#7 types.

Syntax	**int GetFirstPKCS7Data(void *pkcs7ctx, BYTE **data);**	
Return code	Length of data. If 0, no data is available, else error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
data	Storage address of the extracted data.	Output

GetNextPKCS7Data

The next data from the PKCS#7 object will be extracted. This method is available for all supported PKCS#7 types.

Syntax	**int GetNextPKCS7Data(void *pkcs7ctx, BYTE **data);**	
Return code	Length of data. If 0, no data is available, else error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input

data	Storage address of the extracted data.	Output

CreateObject

This method finalizes the creation of a PKCS#7 object.

Syntax	**int CreateObject (void *pkcs7ctx, BYTE **object);**	
Return code	Length of the PKCS#7 object or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input
data	Storage address of the created PKCS#7 object.	Output

CleanupPKCS7

Deallocate storage areas previously reserved by diverse PKCS#7 methods.

Syntax	**int CleanupPKCS7(void *pkcs7ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs7ctx	Storage address of the PKCS#7 context.	Input

Example (Create PKCS7Data):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
/*                                                              *
 * write PKCS#7 Data object                                     *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  BYTE       *object;
  void       *ctx          = {0};
  int         iRc;
  int         option       = HEADER_INCLUDED;
  char        data1[]      = "XPS Software GmbH";
  char        data2[]      = "Muenchner Str. 17";
  char        filename[]   = "pk7data.p7m";

  iRc = CreatePKCS7Data( option, &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  AddPKCS7Data( ctx, data1, strlen(data1) );
  AddPKCS7Data( ctx, data2, strlen(data2) );

  iRc = CreateObject( ctx, &object );
  while( iRc < 0 )
  {
    printf( "Object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  writeFile( filename, object, iRc, 0 );

  CleanupPKCS7( ctx );
```

```
  return( 0 );
}
```

Example (Read PKCS7Data):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

#define true  1
#define false 0

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * extract PKCS#7 Data object                                   *
 *                                                              *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  char       *buffer;
  char       *data;
  void       *ctx         = {0};
  int         iRc, iCount;

  iCount = readFile( argv[1], &buffer );
  if( iCount == 0 )
  {
    printf( "file \"%s\" not found\n", argv[1] );
    return( -1 );
  }

  iRc = ImportPKCS7Data( buffer, iCount, &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = GetFirstPKCS7Data( ctx, &data );
  while( iRc )
  {
    DumpData( "PKCS#7-Data:", data, iRc );
    iRc = GetNextPKCS7Data( ctx, &data );
  }

  CleanupPKCS7( ctx );
  return( 0 );
}
```

Example (Create SignedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * write PKCS#7 signedData object                               *
 *                                                              *
 * Parameter-1: pkcs#12-file  privateKey Signer   (xpsuser1.p12) *
 * Parameter-2: password pkcs#12-privateKey file  (xpsuser1)    *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  BYTE       *buffer, *object;
  void       *ctx, *pemctx;
  BYTE       *PEM;
  int         iRc, iCount1;
  int         option       = DATA_IMPLICIT | CERT_IMPLICIT | HEADER_INCLUDED;
  char        filename[]   = "pk7sd.p7m";
  char        data1[]      = "XPS Software GmbH";
  char        data2[]      = "Muenchner Str. 17";
```

```
  char        data3[]      = "85540 Haar";

  iCount1 = readFile( argv[1], &buffer );
  if( iCount1 == 0 )
  {
    printf( "pkcs#12-object \"%s\" not found\n", argv[1] );
    return( -1 );
  }

  iRc = CreateSignedData( option, &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  AddPKCS7Data( ctx, data1, sizeof(data1)-1 );
  AddPKCS7Data( ctx, data2, sizeof(data2)-1 );
  AddPKCS7Data( ctx, data3, sizeof(data3)-1 );

  iRc = AddSigner( ctx, buffer, iCount1, argv[2], sizeof(argv[2]) );
  if( iRc != 0 )
  {
    printf( "Signer invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = CreateObject( ctx, &object );
  while( iRc < 0 )
  {
    printf( "Object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = ASN2PEM( object, iRc, filename, &PEM, &pemctx );
  writeFile( filename, PEM, iRc, createFile );

  CleanupFile( buffer );
  CleanupPEM( pemctx );
  CleanupPKCS7( ctx );

  return( 0 );
}
```

Example (Read SignedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

#define true  1
#define false 0


/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * read PKCS#7 signedData object                                *
 *                                                              *
 * Parameter-1: pkcs#7-signedData file            (pk7sd.p7m)   *
 * Parameter-2: X.509 Certificate trustedSigner   (xpstest.cer) *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  char       *buffer;
  char       *cbuffer;
  char       *data;
  void       *ctx         = {0};
  int         iRc, iCount1, iCount2;
  PSIGNERINFO signer;

  if( argc < 3 )
  {
    printf( "Parameter missing: xpsread2 pk7sd.p7m xpstest.cer\n" );
    return( -1 );
  }

  /* read pkcs#7-object */
  iCount1 = readFile( argv[1], &buffer );
  if( iCount1 == 0 )
  {
    printf( "pkcs#7-file \"%s\" not found\n", argv[1] );
```

```
    return( -1 );
  }

  /* read trusted signer */
  iCount2 = readFile( argv[2], &cbuffer );
  if( iCount2 == 0 )
  {
    printf( "trusted-signer \"%s\" not found\n", argv[2] );
    return( -1 );
  }

  iRc = ImportSignedData( buffer, iCount1, &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  AddTrustedSigner( ctx, cbuffer, iCount2 );

  iRc = GetFirstSigner( ctx, &signer );
  if( iRc < 0 )
    return( iRc );

  while( iRc == 0 )
  {
    DumpData( "Signer Certificate:", signer->pCert, signer->iLenCert );
    iRc = GetNextSigner( ctx, &signer );
  }

  iRc = VerifyAllSigners( ctx, true );
  printf( "\nVerify All Signers: rc=%d\n", iRc );

  iRc = VerifySigner( ctx, signer, true );
  printf( "\nVerify Last Signer: rc=%d\n", iRc );

  iRc = GetFirstPKCS7Data( ctx, &data );
  while( iRc )
  {
    DumpData( "PKCS#7-Data:", data, iRc );
    iRc = GetNextPKCS7Data( ctx, &data );
  }

  CleanupPKCS7( ctx );
  return( 0 );
}
```

Example (Create EnvelopedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * write PKCS#7 envelopedData object                            *
 *                                                              *
 * Parameter-1: X.509-file  publicKey Recipient   (xpsuser1.cer) *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  BYTE       *object;
  void       *ctx, *pemctx;
  BYTE       *cert;
  BYTE       *PEM;
  int         iRc, iCount1;
  int         option        = HEADER_INCLUDED;
  char        data1[]       = "XPS Software GmbH";
  char        data2[]       = "Muenchner Str. 17";
  char        data3[]       = "85540 Haar";
  char        filename[]    = "pk7env.p7m";

  iCount1 = readFile( argv[1], &cert );
  if( iCount1 == 0 )
  {
    printf( "x509-cert \"%s\" not found\n", argv[1] );
    return( -1 );
  }

  /* iRc = CreateEnvelopedData( option, &ctx, rc2CBC_128 ); */
  iRc = CreateEnvelopedData( option, &ctx, desEDE3CBC );
```

```
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  AddPKCS7Data( ctx, data1, strlen(data1) );
  AddPKCS7Data( ctx, data2, strlen(data2) );
  AddPKCS7Data( ctx, data3, strlen(data3) );

  iRc = AddRecipient( ctx, cert, iCount1 );
  if( iRc != 0 )
  {
    printf( "Recipient invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = CreateObject( ctx, &object );
  while( iRc < 0 )
  {
    printf( "Object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = ASN2PEM( object, iRc, filename, &PEM, &pemctx );
  writeFile( filename, PEM, iRc, createFile );

  CleanupFile( cert );
  CleanupPEM( pemctx );
  CleanupPKCS7( ctx );

  return( 0 );
}
```

Example (Read EnvelopedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

#define true  1
#define false 0

/* ------------------------------------------------------------- *
 *                                                               */
   int main
/*                                                               *
 * decrypt envelopedData object:                                 *
 *                                                               *
 * Parameter-1: pkcs#7 envelopedData file          (pk7env.p7m)  *
 * Parameter-2: pkcs#12-privateKey file (Recipient) (xpsuser1.p12) *
 * Parameter-3: password privateKey file            (xpsuser1)   *
/*                                                               *
 * ------------------------------------------------------------- */
( int    argc,
  char **argv )
/* ------------------------------------------------------------- */
{
  char       *pkcs7obj;
  char       *pkcs12obj;
  char       *data;
  void       *ctx         = {0};
  int         iRc, iCount1, iCount2;
  char        pw[32]      = {0};


  if( argc < 4 )
  {
    printf( "Parameter missing: xpsread3 pk7env.p7m p12-file password\n" );
    return( -1 );
  }

  strcpy( pw, argv[3] );
  iCount1 = readFile( argv[1], &pkcs7obj );           // pkcs7-file
  if( iCount1 == 0 )
  {
    printf( "file \"%s\" not found\n", argv[1] );
    return( -1 );
  }

  iCount2 = readFile( argv[2], &pkcs12obj );          // pkcs12-file
  if( iCount2 == 0 )
  {
    printf( "file \"%s\" not found\n", argv[2] );
    return( -1 );
  }
```

```
  iRc = ImportEnvelopedData( pkcs7obj, iCount1, pkcs12obj, iCount2,
                             pw, sizeof(pw), &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = GetFirstPKCS7Data( ctx, &data );
  while( iRc )
  {
    DumpData( "PKCS#7-Data:", data, iRc );
    iRc = GetNextPKCS7Data( ctx, &data );
  }

  CleanupPKCS7( ctx );
  return( 0 );
}
```

Example (Create EncryptedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * write PKCS#7 encryptedData object                            *
 *                                                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  BYTE       *object;
  void       *ctx, *pemctx;
  BYTE       *PEM;
  int         iRc;
  int         option       = HEADER_INCLUDED;
  char        data1[]      = "XPS Software GmbH";
  char        data2[]      = "Muenchner Str. 17";
  char        data3[]      = "85540 Haar";
  char        filename[]   = "pk7enc.p7m";
  char        password[]   = "testpassword";

  iRc = CreateEncryptedData( option, &ctx, pbe3DES_3Key,
                             password, strlen(password) );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  AddPKCS7Data( ctx, data1, strlen(data1) );
/*  AddPKCS7Data( ctx, data2, strlen(data2) ); */
/*  AddPKCS7Data( ctx, data3, strlen(data3) ); */

  iRc = CreateObject( ctx, &object );
  while( iRc < 0 )
  {
    printf( "Object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = ASN2PEM( object, iRc, filename, &PEM, &pemctx );
  writeFile( filename, PEM, iRc, createFile );

  CleanupPKCS7( ctx );
  CleanupPEM( pemctx );

  return( 0 );
}
```

Example (Read EncryptedData):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

#define true  1
```

```
#define false 0

/* ------------------------------------------------------------- *
 *                                                               */
   int main
/*                                                               *
 * decrypt encryptedData object:                                 *
 *                                                               *
 * Parameter-1: pkcs#7 encryptedData file           (pk7enc.p7m)   *
/*                                                               *
 * ------------------------------------------------------------- */
( int    argc,
  char **argv )
/* ------------------------------------------------------------- */
{
  char       *buffer;
  char       *data;
  void       *ctx          = {0};
  int         iRc, iCount;
  char        password[]   = "testpassword";

  iCount = readFile( argv[1], &buffer );
  if( iCount == 0 )
  {
    printf( "file \"%s\" not found\n", argv[1] );
    return( -1 );
  }

  iRc = ImportEncryptedData( buffer, iCount, password, strlen(password), &ctx );
  if( iRc != 0 )
  {
    printf( "pkcs#7-object invalid: rc=%d\n", iRc );
    return( iRc );
  }

  iRc = GetFirstPKCS7Data( ctx, &data );
  while( iRc )
  {
    DumpData( "PKCS#7-Data:", data, iRc );
    iRc = GetNextPKCS7Data( ctx, &data );
  }

  CleanupPKCS7( ctx );
  return( 0 );
}
```

Chapter

8

PKCS#12 private key

Common information

PKCS#12 objects (*Personal-Information-Exchange-Syntax-Standard*) define syntax for keys and certificates exchange. PKCS#12 objects contain key-bags and certificate-bags. PKCS#12 is accounted standard for securely storing private keys and certificates. Internet browser programs such as Netscape (.p12) and Microsoft Internet Explorer (.pfx) support PKCS#12.

Methods

ImportPKCS12

Reading a PKCS#12 object and checking its formal correctness.

Syntax	**int ImportPKCS12(BYTE *pkcs12object, int lengthobject, BYTE *pwd, int lengthpwd, void **pkcs12ctx);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs12Object	The address of the PKCS#12 object.	Input
lengthobject	The length of the PKCS#12 object.	Input
pwd	Storage address of the password that has been used to encrypt the PKCS#12 object.	Input
lengthpwd	The length of the password.	Input
pkcs12ctx	Storage address of the created PKCS#12 context. This object will be needed for subsequent processing.	Output

GetPrivateKey

Extract the private key from the PKCS#12 object.

Syntax	**int GetPrivateKey(void *pkcs12ctx, RSA_PRIVATE_KEY *privkey);**	
Return code	Length of the private key structure or error code (< 0).	
Parameter	**Description**	**Use**
pkcs12ctx	Storage address of the PKCS#12 context.	Input
privkey	Strorage address to be used to store the extracted private key.	Output

GetFirstCert

Extract the first user certificate from the PKCS#12 object.

Syntax	int GetFirstCert(void *pkcs12ctx, BYTE **x509cert);	
Return code	Length of the X.509 certificate or 0 if no certificate is available.	
Parameter	**Description**	**Use**
pkcs12ctx	Storage address of the PKCS#12 context.	Input
x509cert	Storage address of the extracted X.509 certificate.	Output

GetNextCert

Extract the next certificate from the PKCS#12 object. Besides the user's certificate a PKCS#12 object can contain all signer certificates up to the root certificate.

Syntax	int GetNextCert (void *pkcs12ctx, BYTE **x509cert);	
Return code	Length of the X.509 certificate or 0 if no certificate is available.	
Parameter	**Description**	**Use**
pkcs12ctx	Storage address of the PKCS#12 context.	Input
x509cert	Storage address of the extracted X.509 certificate.	Output

CleanupPKCS12

Deallocate storage areas previously allocated by diverse PKCS#12 methods.

Syntax	int CleanupPKCS12(void *pkcs12ctx);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
pkcs12ctx	Storage address of the PKCS#12 context.	Input

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "XPSCRYPT.H"

/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * get private key from PKCS#12-File (.p12)                     *
 *                                                              *
 * Parameter 1: Name of PKCS#12-File        (xpsuser1.p12)      *
 * Parameter 2: Password                    (xpsuser1)          *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
// ------------------------------------------------------------ //
{
  RSA_PRIVATE_KEY privkey   = {0};
  BYTE           *p12obj    = 0;
  BYTE           *x509Cert  = 0;
  void           *ctx;
  int             iCount, iRc;

  if( argc < 3 )
  {
    printf( "Parameter missing: pk12test p12-file password\n" );
    return( -1 );
```

```
  }

  iCount = readFile( argv[1], &p12obj );
  if( iCount < 1 )
  {
    printf( "PKCS12 Object \"%s\" not found.\n", argv[1] );
    return( -1 );
  }

  iRc = ImportPKCS12( p12obj, iCount, argv[2], strlen(argv[2]), &ctx );
  if( iRc != 0 )
  {
    printf( "PKCS#12 Error: rc=%d\n", iRc );
    return( -1 );
  }

  iRc = GetPrivateKey( ctx, &privkey );
  DumpData( "Private Key:", (BYTE *)&privkey, sizeof(privkey) );

  iCount = GetFirstCert( ctx, &x509Cert );

  while( iCount != 0 )
  {
    DumpData( "X.509 Certificate:", x509Cert, iCount );
    iCount = GetNextCert( ctx, &x509Cert );
  }

  CleanupPKCS12( ctx );
  CleanupFile( p12obj );
  return 0;
}
```

Kapitel

9

SSL/TLS

Common information

Secure Sockets Layer (SSL) or Transport Layer Security (TLS) is a hybrid protocol that's used to provide secure data transmission over the internet. TLS is the standardized evolution of SSL 3.0 (TLS 1.0 has been introduced to replace SSL 3.1). SSL will now be developed under the name TLS. In this documentation the term SSL will be used for both evolutions of the protocol.

SSL has been developed by Netscape. SSL prevents data from being read by unauthorized sources and ensures that data won't be manipulated during transmission.

Methods

SSL_Init

Initialise the SSL context. Using this method the decision is to make if a client or a server session shall be established. In case of a client session the desired protocol (SSL or TLS) can be specified. In case of a server session the decision can be made if client requests must provide a certificate.

Syntax	**int SSL_Init(BYTE **ctx, int iProtocolSide, int iOption);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address to receive the address of the created SSL-object. This will be needed for the further processing of the object.	Output
iProtocolSide	Decision if a client or a server sessions hall be created. The following values are possible:: **SSL_ClientSide**: Create a SSL client connetion. **SSL_ServerSide**: Create a SSL server connection.	Input
iOption	**when iProtocolSide = SSL_ClientSide:** Desired protocol SSL_Version_3_0 or TLS_Version_1_0. **when iProtocolSide = SSL_ServerSide:** TRUE(1) – Client certificate required. FALSE(0) – Client certificate not required.	Input

SSL_Set_PrivateKey

Allocate a private key to a SSL connection.

This method must be called for client as well as for server sessions, because during the SSL handshake a RSA public/private key exchange occurs. XPS-CryptLib is capable of processing .p12 and .pfx formatted files.

Syntax	**int SSL_Set_PrivateKey(BYTE **ctx, BYTE *pkcs12obj, int iLenpkcs12, BYTE *pwd, int lengthpwd);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
pkcs12obj	Storage address of the PKCS#12 object.	Input
iLenpkcs12	Length of the PKCS#12 object.	Input
pwd	Storage address of the password used to enctrypt the PKCS#12 object.	Input
lengthpwd	Length of the password.	Input

SSL_Add_x509Cert

Add another X.509 certificate.

During the SSL handshake certificates will be exchanged. XPS-CryptLib transmits all certificates stored in the pkcs12 file added previously to the session using the method SSL_Set_PrivateKey. If required certificates such as the root certificate are not contained in the pkcs12 file, these can be added using this method.

This method can be called for client as well as for server connections. Server connections always send their certificates. Client connections only if it is explicitly required by the server.

Syntax	**int SSL_Add_x509Cert(BYTE **ctx, BYTE *pX509, int iLenX509);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
pX509	Storage address of the X.509 certificate.	Input
iLenX509	Length of the X.509 certificate.	Input

SSL_Set_Cipher

A SSL client declares its preferred symmetric encryption algorithms.

After the handshake is complete the SSL protocol requires symmetric encryption of transmitted data. The SSL client can declare its preferred symmetric encryption algorithms calling this method as often as necessary. The SSL server will then select one of the symmetric encryption algorithms offered by the client.

Syntax	**int SSL_Set_Cipher(BYTE **ctx, CipherSuite Cipher);**	
Return ode	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
Cipher	XPS-CryptLic supports the following symmetric ciphers: SSL_RSA_WITH_RC4_MD5_EXP SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_RC4_128_SHA SSL_RSA_WITH_RC2_CBC_MD5_EXP	Input

	SSL_RSA_WITH_RC2_CBC_128_MD5 SSL_RSA_WITH_DES_CBC_MD5 SSL_RSA_WITH_DES_CBC_SHA SSL_RSA_WITH_3DES_EDE_CBC_SHA SSL_RSA_WITH_3DES_EDE_CBC_MD5 TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA	

SSL_Add_DN

Specify permitted session partners.

Using this optional method either side of the SSL connection can specify a pool of partners permitted to establish a connection.

Client connections can specify a pool of permitted servers through adding their certificates using this method.

Server connections can limit the pool of permitted clients to those whose certificates have been signed with the signer certificates specified using this method.

If this method won't be called certificates won't be checked.

Syntax	**int SSL_Add_DN(BYTE **ctx, BYTE *pX509, int iLenX509);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
pX509	Storage address of the X.509 certificate.	Input
iLenX509	Length of the X.509 certificate.	Input

SSL_Handshake

Negotiate the session key with the partner.

This method has to be called by the SSL client as well as by the SSL server in order to negotiate the symmetric algorithm and the key. The SSL client has to provide the TCP/IP socket received from the ***connect*** call. The SSL server has to provide the TCP/IP socket received from the ***accept*** call.

Syntax	**int SSL_Handshake(BYTE **ctx, SOCKET socket);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
socket	TCP/IP socket received during connection establishment.	Input

SSL_Get_Peer_Cert

Extract peer's user certificate.

Syntax	**int SSL_Get_Peer_Cert(BYTE **ctx, BYTE **pX509);**	
Return code	Length of the certificate or error code (<0).	
Parameter	**Description**	**Use**

ctx	Storage address of the SSL context.	Input
pX509	Storage address to receive the address of the extracted certificate.	Output

SSL_GetNext_Peer_Cert

Extract next peer's certificate.

Syntax	**int SSL_GetNext_Peer_Cert(BYTE **ctx, BYTE **pX509);**	
Return code	Length of the certificate or 0 (no more certificates available) or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
pX509	Storage address to receive the address of the extracted certificate.	Output

SSL_Read

Read data, decrypt it and check for validity.

Syntax	**int SSL_Read(BYTE **ctx, BYTE *data, int iLength);**	
Return code	Length of the data or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
data	Address of storage to receive the read and decrypted data.	Input
iLength	Maximum length of data to return (size of provided data area).	Input

SSL_Write

Encrypt data, sign it and write.

Syntax	**int SSL_Write(BYTE **ctx, BYTE *data, int iLength);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
data	Storage address of data to send.	Input
iLength	Length of data to send.	Input

SSL_Close_Session

Close the SSL session.

All session related data, especially the session key, will be deallocated while the TCP/IP-Socket remains open. Calling the SSL-Resume_Session method, client connections can establish a new SSL connection using this socket.

Syntax	**int SSL_Close_Session(BYTE **ctx);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**

ctx	Storage address of the SSL context.	Input

SSL_Resume_Session

Resume a SSL session.

Calling the method a client can resume a session which has afore been closed using the SSL_Close_Session method. The SSL handshake will be processed based on the data previously specified for the session.

Syntax	**int SSL_Resume_Session(BYTE **ctx);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
socket	TCP/IP socket received during connection establishment.	Input

SSL_Cleanup

Deallocation of the SSL session.

All storatge areas reserved for the SSL session will be released and the underlying TCP/IP socket will be closed.

Syntax	**int SSL_Cleanup(BYTE **ctx);**	
Return code	0 oder error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input

SSL_Get_Last_Error

Investigate the message belonging to the last error.

Mit dieser Funktion kann nach dem Auftreten eines Fehlers die zugehörige Nachricht extrahiert werden.

Syntax	**int SSL_Get_Last_Error(BYTE **ctx, BYTE *msg, int msglen);**	
Return code	0 or error code (<0).	
Parameter	**Description**	**Use**
ctx	Storage address of the SSL context.	Input
msg	Storage address of the message area.	Input
msglen	Length of the message area.	Input

SSL client example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>
#include <fcntl.h>
#include <io.h>
#include <sys/stat.h>
#include <time.h>

#include "xpscrypt.h"

int main( int argc, char* argv[] )
{
  int     iRc, i1;
  SOCKET  sockfd;
  WSADATA wsadata      = {0};
  short   sServerPort  = 5555;
  int    *piTemp;
  BYTE   *ctx;
  BYTE    buffer[4096]  = {0};

  struct  sockaddr_in   addr;
  struct  hostent *he;

  BYTE   *p12obj      = 0;
  BYTE   *x509Cert    = 0;
  BYTE   *pcert       = 0;
  char    pwd[8]      = "xpsuser1";
  char    szData01[8] = "test";
  int     protocol    = TLS_Version_1_0;

  /*************************************************/
  /* Initialize the socket address structure       */
  /*************************************************/
  iRc =  (short)WSAStartup( 0x0101, &wsadata );

  he = gethostbyname("localhost");

  addr.sin_family      = AF_INET;
  piTemp               = (int *)*he->h_addr_list;
  addr.sin_addr.s_addr = *piTemp;
  addr.sin_port        = htons(sServerPort);

  /*************************************************/
  /* Create an AF_INET stream socket               */
  /*************************************************/
  sockfd = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP );
  if (sockfd < 0)
  {
   printf("socket() failed");
      return (-1);
  }

  /*************************************************/
  /* Connect to the server                         */
  /*************************************************/
  iRc = connect(sockfd,
              (struct sockaddr *)&addr,
              sizeof(struct sockaddr));
  if( iRc < 0 )
  {
   printf("connect() failed");
      return(-1);
  }

  i1 = readFile( "xpsuser1.p12", &p12obj );
  if( i1 < 1 )
  {
    printf( "PKCS12 Object \"%s\" not found.\n", "xpsuser1.p12" );
    return( -1 );
  }

  SSL_Init( &ctx, SSL_ClientSide, protocol );

  SSL_Set_Cipher( &ctx, TLS_RSA_WITH_AES_256_CBC_SHA );
  SSL_Set_Cipher( &ctx, TLS_RSA_WITH_AES_128_CBC_SHA );
  SSL_Set_Cipher( &ctx, SSL_RSA_WITH_3DES_EDE_CBC_SHA );

  iRc = SSL_Set_PrivateKey( &ctx, p12obj, i1, pwd, strlen(pwd) );
  if( iRc != 0 )
    return( iRc );
  CleanupFile( p12obj );

  iRc = SSL_Handshake( &ctx, sockfd );
```

```
  if( iRc != 0 )
  {
    char szTemp[80];

    SSL_Get_Last_Error( &ctx, szTemp, sizeof(szTemp) );
    printf("%s\n", szTemp);
    return( iRc );
  }

  iRc = SSL_Get_Peer_Cert( &ctx, &pcert );
  while( iRc > 0 )
  {
    DumpData( "X.509 Cert", pcert, iRc );
    iRc = SSL_GetNext_Peer_Cert( &ctx, &pcert );
  }

  iRc = SSL_Write( &ctx, (char *)&szData01, 5 );
  if( iRc < 0 )
  {
    printf("send() failed");
       return(-1);
  }
  printf("write: %s\n", szData01 );

  iRc = SSL_Read( &ctx, buffer, sizeof(buffer) );
  if( iRc <= 0 )
    return( iRc );
  printf( "read: %s - Len=%d\n", buffer, iRc );

  SSL_Close_Session( &ctx );
  closesocket(sockfd);
  SSL_Cleanup( &ctx );

  WSACleanup();
  printf("Socket  e n d e d\n");
  return( 0 );
}
```

SSL server example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>
#include <time.h>
#include "xpscrypt.h"

int main( int argc, char* argv[] )
{
  int     iRc, i1;
  SOCKET  sd, asd;
  WSADATA wsadata               = {0};
  short   sServerPort           = 5555;

  struct  sockaddr_in   addr     = {0};
  struct  sockaddr      addr_acc = {0};
  char    szInput[100]           = {0};
  char    szOutput[100]          = {0};
  BYTE   *p12obj                 = 0;
  BYTE   *x509Cert               = 0;
  char    pwd[16]                = "xpssrv";
  char    file[32]               = "xpssrv.p12";
  int     iReadLen;
  int     iLen                   = sizeof(addr_acc);
  BYTE   *ctx;
  char    szTemp[80];
  int     iOption                = 1;

  /***********************************************/
  /* Initialize the socket address structure       */
  /***********************************************/
  iRc =  (short)WSAStartup( 0x0101, &wsadata );
  memset(&addr, 0, sizeof(addr));
  addr.sin_family      = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port        = htons(sServerPort);

  sd = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP );
  if (sd < 0)
  {
   printf("socket() failed");
    return(-1);
  }
  iRc = bind( sd, (struct sockaddr *)&addr, sizeof(addr));
  if (iRc < 0)
  {
   printf("bind() failed");
```

```
    return(-1);
  }

  iRc = listen( sd,1000 );
  if (iRc != 0)
  {
    printf("listen() failed");
    return(-1);
  }

//  iOption |= 0x80000000;
  SSL_Init( &ctx, SSL_ServerSide, iOption );

  i1 = readFile( file, &p12obj );
  if( i1 < 1 )
  {
    printf( "PKCS12 Object \"%s\" not found.\n", file );
    return( -1 );
  }
  iRc = SSL_Set_PrivateKey( &ctx, p12obj, i1, pwd, strlen(pwd) );
  if( iRc != 0 )
    return( iRc );
  CleanupFile( p12obj );

  printf("SSL Server started at port %d\n", sServerPort);

  while (1)
  {
    asd = accept( sd, &addr_acc, &iLen );
    if (asd == 0)
    {
      printf("accept() failed\n");
      return(-1);
    }
    printf("accept() ok sd=%d\n",asd);


    iRc = SSL_Handshake( &ctx, asd );
    if( iRc != 0 )
    {
      char szTemp[80];

      SSL_Get_Last_Error( &ctx, szTemp, sizeof(szTemp) );
      printf("%s\n", szTemp);
    }
    else
    {
      while( TRUE )
      {
        memset( szInput, 0, sizeof(szInput) );
        iReadLen = SSL_Read( &ctx, (char *)szInput, sizeof(szInput) );
        if( iReadLen < 0 )
        {
          SSL_Get_Last_Error( &ctx, szTemp, sizeof(szTemp) );
          printf("%s\n", szTemp);
             break;
        }

        if( iReadLen == 0 )
          continue;

        sprintf(szOutput, "ECHO: %s", szInput );
        printf("%s\n", szOutput);

        iRc = SSL_Write( &ctx, (char *)szOutput, strlen(szOutput) );
        if( iRc < 0 )
        {
          SSL_Get_Last_Error( &ctx, szTemp, sizeof(szTemp) );
          printf("%s\n", szTemp);
             break;
        }
      }

      closesocket(asd);
      printf("close sd=%d\n",asd);
    }

  }

  /*********************************************/
  /* Close down the socket                     */
  /*********************************************/
  closesocket(sd);
  SSL_Cleanup( &ctx );
  WSACleanup();
  return( 0 );
}
```

Chapter

10

GZIP

Common information

CryptLib offers the possibility to compress and decompress data using *gzip* and *gunzip*.

Methods

gzip

Data will be compressed according to the gzip format.

Syntax	**int gzip(char *input, int inputlength, char **output, int *outputlength, char *fileName);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
Input	Storage address of the data to be compressed.	Input
inputlength	Length of the data about to compress.	Input
output	Address pointer to receive the storage address of the compressed data.	Output
outputlength	Storage address of a field about to receive the length of the compressed data.	Output
fileName	File name to be stored in the ZIP header. If given the name must be low value terminated (x'00').	Input

gunzip

Decompress data previously compressed according to the gzip format.

Syntax	**int gunzip(char *input, int inputlength, char **output, int *outputlength, char *fileName);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
input	Storage address of the data to be decompressed.	Input
inputlength	Length of the data about to decompress.	Input
output	Address pointer to receive the storage address of the decompressed data.	Output
outputlength	Storage address of a field about to receive the length of the decompressed data.	Output

fileName	Storage address to receive the file name stored in the ZIP header. If no file name is stored in the ZIP header this address is set to NULL otherwise the file name will be low value terminated (x'00').	Output

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"

/* ------------------------------------------------------------ *
   int main
/*                                                              *
 * gzip/gunzip Routines                                         *
* ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  char  *pZipData;
  char  *pUnzipData;
  char   fileName[_MAX_PATH]     = {0};
  int    iZipLen = 0, iUnzipLen = 0;

  char HAMLET[] =
  {
   "To be, or not to be: that is the question:\n"
   "Whether 'tis nobler in the mind to suffer\n"
   "The slings and arrows of outrageous fortune,\n"
   "Or to take arms against a sea of troubles,\n"
   "And by opposing end them. To die, to sleep-\n"
   "No more- and by a sleep to say we end\n"
   "The heartache, and the thousand natural shocks\n"
   "That flesh is heir to| 'Tis a consummation\n"
   "Devoutly to be wished. To die, to sleep-\n"
   "To sleep-perchance to dream; ay, there's the rub,\n"
   "For in that sleep of death what dreams may come'\n"
   "When we have shuffled off this mortal coil,\n"
   "Must give us pause. There's the respect\n"
   "That makes calamity of so long life:\n"
   "For who would bear the whips and scorns of time,\n"
   "Th' oppressor's wrong, the proud man's contumey,\n"
   "The pangs of despised love, the law's delay,\n"
   "The insolence of office, and the spurns\n"
   "That patient merit of th' unworthy takes,\n"
   "When he himself might his quietus make\n"
   "With a bare bodkin? Who would fardels bear,\n"
   "To grunt and sweat under a weary life,\n"
   "But that the dread of something after death,\n"
   "The undiscovered country, from whose bourn\n"
   "No traveler returns, puzzles the will,\n"
   "And makes us rather bear those ills we have,\n"
   "Than fly to others that we know not of?\n"
   "Thus conscience does make cowards of us all,\n"
   "And thus the native hue of resolution\n"
   "Is sicklied o'er with the pale cast of thought,\n"
   "And enterprises of great pitch and moment,\n"
   "With this regard their currents turn awry,\n"
   "And lose the name of action.-Soft you now,\n"
   "The fair Ophelia|-Nymph, in thy orisons\n"
   "Be all my sins remembered.\n"
  };

  gzip( HAMLET, sizeof(HAMLET), &pZipData, &iZipLen, "hamlet.txt" );
  DumpData( "Zip Data:", pZipData, iZipLen );

  gunzip( pZipData, iZipLen, &pUnzipData, &iUnzipLen, fileName );
  /* writeFile("hamlet.txt", pUnzipData, iUnzipLen, createFile ); */
  DumpData( fileName, pUnzipData, iUnzipLen );

  CleanupGzip( pZipData );
  CleanupGzip( pUnzipData );
  return( 0 );
}
```

Chapter

11

Additional methods

Common information

CryptLib offers the programmer a number of usefull methods for the development of cryptographic applications. Among these are methods for the conversion of ASN.1 objects from binary to US-ASCII and vice versa, methods to read and write files and methods for data conversion from EBCDIC to ASCII and vice versa.

Methods

ASN2PEM

BER/DER encoded ASN.1 objects are available in binary format. Because of the fact that some transmission protocols don't support transmission of binary data the need for translation into US-ASCII representation rises. This is carried out using the Base64 method as layed out in RFC 1521. Using this method a binary object can be translated into a Base64 object.

Syntax	**int ASN2PEM(BYTE *input, int inputlength, char *smime, BYTE **output, void **ctx);**	
Return code	Length of the Base64 object or error code (< 0).	
Parameter	**Description**	**Use**
input	The storage address of the binary ASN.1 object.	Input
inputlength	The length of the ASN.1 object.	Input

smime	If a S/MIME name is provided the following S/MIME header will be included in front of the PEM object:
	
	`Content-Disposition: attachment; filename="smime.p7m"`
	`Content-Type: application/x-pkcs7-mime; name="smime.p7m"`
	`Content-Transfer-Encoding: base64`
		Input

output	Storage address of the created Base64 object.	Output
ctx	Storage address of the created context. This will be required in order to deallocate any reserved working storage.	Output

PEM2ASN

Using this method a Base64 object can be re-translated into a binary object.

Syntax	**int PEM2ASN(BYTE *input, int inputlength, BYTE **output, void **ctx);**
Return code	Length of the binary object or error code (< 0).

Parameter	Description	Use
input	Storage address of the Base64 object.	Input
inputlength	Length of the Base64 object.	Input
output	Storage address of the created binary object.	Output
ctx	Storage address of the created context. This will be required in order to deallocate any reserved working storage.	Output

CleanupPEM

Deallocation of the storage used by the Base64 routines.

Syntax	int CleanupPEM(void *ctx);	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
ctx	Storage address of the context.	Input

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xpscrypt.h"


/* ------------------------------------------------------------ *
 *                                                              */
   int main
/*                                                              *
 * Translate  ASN.1 DER coded file to base64 coded PEM file, or *
 *            base64 coded PEM file to DER coded ASN.1 file     *
 *                                                              *
 * Parameter:                                                   *
 *            ASN.1/PEM file                                    *
 *            '1' = ASN.1 to PEM                                *
 *            '2' = PEM   to ASN.1                              *
 * ------------------------------------------------------------ */
( int    argc,
  char **argv )
/* ------------------------------------------------------------ */
{
  BYTE     *buffer;
  char     *PEM         = 0;
  int       count       = 0;
  int       func        = 0;
  void     *ctx;

  count = readFile( argv[1], &buffer );
  if( count == 0 )
  {
    printf( "File \"%s\" not found", argv[1] );
    return( -1 );
  }

  if( argv[2][0] == '1' )
    func = 1;
  else
    func = 2;

  if( func == 1 )
    count = ASN2PEM( buffer, count, &PEM, &ctx );
  else
    count = PEM2ASN( buffer, count, &PEM, &ctx );

  writeFile( "PEMout", PEM, count, createFile );

  CleanupPEM( ctx );

  return( count );
}
```

readFile

Using this function a file can be read.

Syntax	**int readFile(char *fileName, BYTE **buffer);**	
Return code	Length of the file or error code (< 0).	
Parameter	**Description**	**Use**
filename	Name of the file to read.	Input
buffer	Storage address of the read file.	Output

writeFile

Using this function a file can be written.

Syntax	**int writeFile(char *fileName, BYTE *buffer, int bufferlength, int fOption);**	
Return code	0 or error code (< 0).	
Parameter	**Description**	**Use**
filename	Name of the file about to write.	Input
buffer	Storage address of the file data to write.	Input
bufferlength	Length of the file data to write.	Input
fOption	Write option. The following options are supported: createFile The file will be newly created. appendFile The data will be appended to an already existing file.	Input

cleanupFile

Deallocation of the storage reserved for read file data.

Syntax	**void cleanupFile(BYTE *buffer);**	
Return code	None.	
Parameter	**Description**	**Use**
buffer	Address of the storage to deallocate.	Input

EBCDIC_to_ASCII

Using this method data can be converted from EBCDIC to ASCII.

Syntax	**void EBCDIC_to_ASCII(BYTE *data, int datalength);**	
Return code	None.	
Parameter	**Description**	**Use**
data	Storage address of the EBCDIC data about to convert.	Input/Output
datalength	Length of the EBCDIC data.	Input

ASCII_to_EBCDIC

Using this method data can be converted from ASCII to EBCDIC.

Syntax	void ASCII_to_EBCDIC(BYTE *data, int datalength);	
Return code	None.	
Parameter	**Description**	**Use**
data	Storage address of the ASCII data about to convert.	Input/Output
datalength	Length of the ASCII data.	Input

Chapter

12

Error codes

err_algorithm **-100**

Description: The algorithm transmitted to the InitCTX method is not supported. Supported algorithms are AES, DES, RC2, RC4, Blowfish and RSA.

err_keylength **-101**

Description: The key length transmitted to the InitCTX method is not supported.

err_mode **-102**

Description: The mode transmitted to the InitCTX method is not supported. Supported modes are ECB and CBC for symmetrical encryption, PUBLIC and PRIVATE for asymmetrical encryption.

err_buildkey **-103**

Description: An error occurred when the InitCTX method tried to initialize the encryption algorithm.

err_buildiv **-104**

Description: An error occurred when the InitCTX method tried to build the initialization vector.

err_CTX **-105**

Description: The transmitted context is invalid or not properly initialized.

err_outlen **-106**

Description: The storage area provided for the method is too small.

err_contentenc **-200**

Description: An error occurred when converting Base64 data to PEM.

err_data **-201**

Description: Data transmitted to a RSA method is invalid.

err_digalgo **-202**

Description: The hash type selected for the RSA method is not supported.

err_encoding **-203**

Description: An error occurred when converting PEM data to Base64.

err_rsakey **-204**

Description: The RSA key transmitted to the method is invalid.

err_rsalength **-205**

Description: The length of the data transmitted to the RSA method is invalid.

err_modulus **-206**

Description: The modulus of the transmitted RSA key is incorrect.

err_random **-207**

Description: A random structure couldn't be initialized.

err_privkey **-208**

Description: The private RSA key transmitted to the method is invalid.

err_pubkey **-209**

Description: The public RSA key transmitted to the method is invalid.

err_signature **-210**

Description: The signature transmitted for verification doesn't match the original.

err_encralgo **-211**

Description: The encryption algorithm specified for the RSA method is unknown.

err_certparm **-300**

Description: An invalid parameter has been transmitted to the ImportCertificate method.

err_certimport **-301**

Description: An unsupported X.509 certificate has been transmitted to the ImportCertificate method.

err_certlength **-302**

Description: The storage area provided for the extraction of a certificate is too small.

err_certalgo **-303**

Description: The only supported encryption algorithm for X.509 certificates is RSA.

err_certhash **-304**

Description: The only supported hash methods for X.509 certificates are MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.

err_certstart **-305**

Description: Within the validation process of a X.509 certificate a future certificate start date has been detected. The signature of the certificate is correct.

err_certend **-306**

Description: Within the validation process of a X.509 certificate an elapsed certificate end data has been detected. The signature of the certificate is correct.

err_certoid **-307**

Description: The certificate extension searched with the GetExtensionByOID method is inexistent.

err_asn1 **-400**

Description: The object currently being imported has a faulty or unsupported ASN.1 structure.

err_asn1table **-401**

Description: The imported ASN.1 object is not compatible with the called method.

err_hashOID **-402**

Description: The imported PKCS object uses an unsupported hash type.

err_contentinf **-403**

Description: The imported PKCS object contains unsupported content information.

err_hmac **-404**

Description: The HMAC calculated for the imported PKCS#12 object is invalid. Possibly the transmitted password is incorrect.

err_authsafe **-405**

Description: The imported PKCS#12 object contains an unsupported authenticated safe.

err_pbeType **-406**

Description: The imported PKCS#12 object contains an unsupported PBE type (PBE = password based encryption). CryptLib supports the following algorithms: pbeWithSHAAnd128BitRC4, pbeWithSHAAnd40BitRc4, pbeWithSHAAnd3KeyTripleDES-CBC, pbeWithSHAAnd2KeyTripleDES-CBC, pbeWithSHAAnd128BitRC2-CBC and pbeWithSHAAnd40BitRC2-CBC.

err_certbag **-407**

Description: The imported PKCS#12 object contains an unsupported certificate-bag.

err_certbagType **-408**

Description: The certificate-bag found in the imported PKCS#12 object contains an unspported certificate format. CryptLib supports the following formats: x509Certificate and sdsiCertificate.

err_noauthsafe **-409**

Description: The imported PKCS#12 object doesn't contain an authenticated safe.

err_safeBag **-410**

Description: The imported PKCS#12 object doesn't contain a safe bag type 'pkcs8-shroudedKeybag'.

err_privKey **-411**

Description: The imported PKCS#12 object doesn't contain a private key.

err_RSAEnc **-412**

Description: CryptLib only supports the RSA encryption algorithm for PKCS#12 objects.

err_nokeyBag **-413**

Description: The imported PKCS#12 object doesn't contain a key-bag.

err_hmacAlgo **-414**

Description: The imported PKCS#12 object contains an unsupported HMAC algorithm. CryptLib supports the following algorithms: MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.

err_Algo **-415**

Description: The imported PKCS#7 object contains an unsupported hash type. CryptLib supports the following hash types: MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RipeMD160.

err_EAlgo **-416**

Description: The imported PKCS#7 object contains an unsupported encryption algorithm. CryptLib only supports the RSA algorithm.

err_noSigner **-417**

Description: No trusted signer could be found for the signer specified in the imported PKCS#7 object.

err_noSignerCert **-418**

Description: No certificate could be found for the signer specified in the imported PKCS#7 object.

err_messagedig **-419**

Description: The Message Digest for the signer specified in the imported PKCS#7 signed-data object is invalid.

err_verify **-420**

Description: Verification of the signature of the signer of the imported PKCS#7 signed-data object failed.

err_unknownSigner **-421**

Description: The signer transmitted to the method *VerifySigner* isn't contained in the imported PKCS#7 signed-data object.

err_noData **-422**

Description: The imported PKCS#7 object doesn't contain data.

err_noCert **-423**

Description: The imported PKCS#7 object doesn't contain a certificate.

err_noRecipient **-424**

Description: The imported PKCS#7 enveloped-data object doesn't contain a recipient.

err_p12notvalid **-425**

Description: The PKCS#12 object tansmitted to the *ImportEnvelopedData* method is invalid.

err_invopt **-426**

Description: The option transmitted to the *Create…Data* method is invalid.

err_noTrustedSign **-427**

Description: No trusted signer could be located for the signer transmitted to the *VerifySigner* method.

err_maxData **-428**

Description: Executing the AddPKCS7Data method the maximum acceptable data size has been exceeded.

err_ssl_inv_version **-500**

Description: A client tries to establish a connection based on an unsupported SSL version.

err_ssl_inv_side **-501**

Description: Calling the SSL_Init method an invalid protocol side has been specified. The only accepted values are SSL_ClientSide and SSL_ServerSide.

err_ssl_cipher_invalid **-502**

Description: Calling the SSL_Set_Cipher method an unvalid or not support cipher has been specified.

err_ssl_cert_not_trusted **503**

Description: Verification of an X.509 certificate has failed.

err_ssl_send **-504**

Description: A TCP/IP error occurred while sending data.

err_ssl_select **-505**

Description: Failed to execute the TCP/IP method *select*.

err_ssl_read **-506**

Description: A TCP/IP error occurred while reading data.

err_ssl_unsupportedType **-507**

Description: SSL data has been sent using an unknown type.

err_ssl_unsupportedVersion **-508**

Description: While reading data an unknown or unsupported version has been detected.

err_ssl_unsupportedCipher **-509**

Description: In case of a client session the server has selected an unsupported cipher. In case of a server session the client didn't provide a server supported cipher.

err_ssl_toomanydata **-510**

Description: Data exceeding the permitted SSL maximum length of 32767 has been read.

err_ssl_protocolviolation **-511**

Description: A protocol violation occurred during the SSL handshake.

err_ssl_messageHash **-512**

Description: An ivalid message has has been detected during the SSL handshake.

err_ssl_alertRead **-513**

Description: The partner connection has sent an alert message.

err_ssl_reqCertInv **-514**

Description: An unsupported X.509 certificate has been detected.

err_ssl_noCertsAvailable **-515**

Description: No X.509 certificate has been found during the SSL handshake. Probably the SSL_Set_PrivateKey method has not been called previously.

---

err_ssl_unsupported_RSAkey -516

Description: The transmitted private key is not supported.

---

err_ssl_hash_RSAkey_inv -517

Description: The server key exchange failed.

---

err_ssl_premasterkey_inv -518

Description: An invalid pre master key has been detected.

---

err_ssl_certverify_inv -519

Description: The client certificate is invalid.

---

err_ssl_peer_not_trusted -520

Description: A server certificate has been received for a client connection that does not conform to the pool of certificates loaded using the SSL-Add_DN method.
A client certificate has been received for a server connection that does not conform to the pool of certificates loaded using the SSL-Add_DN method.