Asymmetric crypto

Using Symbian Asymmetric Crypto

The Symbian crypto libraries have long been unavailable to end users due to the UK's ridiculous export restrictions. This problem was overcome last year, and at long last developers can have access to a trimmed down version of the crypto library for all versions of Symbian greater than v9. Of course, it's not much good having the crypto library if you don't know how to use it, and while it's fairly easy to figure out how to use the symmetric ciphers, asymmetric is somewhat harder. This wiki article goes in to a few common tasks you might want to carry out with RSA, DSA and DH keys.

This article assumes you are in general familiar with the theory and practice of asymmetric crypto, PKI and trust.

Some notes on the crypto implementation

Again, due to the outdated and just plain stupid laws in the UK, the crypto library can be shipped in two "strengths", Strong and Weak. The Weak crypto library is frankly useless, it limits you to Asymmetric keys of less than 512 bits, and symmetric keys of around 40 bits. From a security standpoint, 1024 and 128 bit keys respectively are an absolute minimum. These days, even those are becoming questionable. For a reasonable guarantee of security, one should consider using 2048 and 256 bit keys by default.

The crypto library discovers its own strength at runtime. All calls are proxied to strong_cryptography.dll if it exists, or weak_cryptography.dll if it doesn't. Recent S60 SDKs ship with a version of strong_cryptography.dll which is useful. Some recent UIQ SDKs did not. You can solve this problem by simply copying the strong crypto dll from the S60 SDK to the UIQ SDK to get strong crypto everywhere. All actual devices will have strong crypto on board.

Then we have the problem that the Symbian released crypto plugin doesn't actually have the same headers as the Symbian internal crypto implementation does. This means that some headers that are publishedAll do not actually parse without modification. A prime example of this is "x509keys.h" This header is actually excluded from many S60 SDKs, and must be modified from the UIQ SDK. This is extremely tedious. Renaming headers in this way is a compatibility nightmare. For your convenience, the include directives at the top of x509keys.h should look like this in order to parse sucessfully:

Header files.

```
#include <e32base.h>
#include <e32std.h>
#include <cryptobasic.h>
#include <cryptoasymmetric.h>
#include <hash.h>
#include <signed.h>
```

Libraries.

```
LIBRARY x509.lib
```

That done, you now have most of what you need to start using asymmetric crypto.

X.509 Certificates

In the most part, you'll find that the main way public keys are conveyed to people is embedded in a certificate. The certificate not only includes a public key, but also some information on its owner and who issued it that enable you to establish trust.

Dealing with X.509 certificates in Symbian is trivial. You must however ensure that a certificate you are attempting to load is raw binary DER encoding, rather than PEM. Symbian has no library for dealing with Base64 decoding PEM certificates, although this is a fairly trivial thing to do yourself should you need to.

Some rather contrived code for loading an X.509 certificate from a file follows:

```
#include <f32file.h>
#include <x509cert.h>

RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

RFile certFile;
User::LeaveIfError(certFile.Open(fs, _L("c:\\certfile.cer"), EFileRead));
CleanupClosePushL(certFile);

TInt size(0);
User::LeaveIfError(certFile.Size(size));

HBufC8* data = HBufC8::NewLC(size);
TPtr8 buf = data->Des();
User::LeaveIfError(certFile.Read(buf));

CX509Certificate* cert = CX509Certificate::NewLC(*data);

CleanupStack::PopAndDestroy(4, &fs); //certFile, data, cert
```


(C) Copyright Nokia 2013. All rights reserved.

Code using the CX509Certificate class must link against x509.lib.

Validating certificates

TODO: talk about CPKIXCertChain, and the certstore.

Extracting a certificate's public key

You can query the X.509 certificate to see the type of public key it contains. This is achieved as follows:

```
CX509Certificate* cert = /* blah blah blah */
TAlgorithmId algorithm = cert->PublicKey().AlgorithmId();
```

Once you know the type of key, to perform any cryptographic operation you'll need to get at its public key. This is done as follows:

For RSA keys:

```
#include <x509keys.h>

CX509Certificate* cert = /* blah blah blah */
CX509RSAPublicKey* key = CX509RSAPublicKey::NewLC(cert->PublicKey().KeyData());
```

TODO: DSA keys, possibly fixed DH params too if I get enthusiastic.

Once you have a public key, you can use it to encrypt data.

Performing asymmetric encryption

For RSA encryption, a standard PKCS#15 encryptor class is available, CRSAPKCS1v15Encryptor. This standard method of performing RSA encryption with padding is used by any number of protocols and standards.

Its use it pretty trivial:

```
#include <cryptoasymmetric.h>
#include <x509keys.h>

CX509RSAPublicKey* key = /* blah blah blah */
CRSAPKCS1v15Encryptor* encryptor = CRSAPKCS1v15Encryptor::NewLC(*key);

TBuf8<64> plaintext;
TBuf8<128> ciphertext;

/* fill the plaintext buffer with some data here */

encryptor.EncryptL(plaintext, ciphertext);
```

Of course, asymmetric crypto is designed to be used to encrypt very small amounts of data, typically key for a symmetric cipher which will be less than 512 bytes long. You can get the maximum input and output sizes from the CRSAPKCS1v15Encryptor class via the MaxInputLength and MaxOutputLength methods.

Decryption

TODO: talk about the ASN.1 library and how to load standard key DER encodings, and the use of those keys.

Signing

The third operation type you can do with asymmetric cryptography is to sign data, again this is fairly easy.

The missing keystore

In the full Symbian security subsystem, there exists a component called "keystore" which is responsible for the secure storage of keys. That keystore isn't available to developers is a great shame, because it makes the task of finding keys and signing data trivial. It also means that you aren't responsible for the security of your keys, they are securely stored and managed by the system, which is very convenient.

However, simply because you don't have access to the keystore doesn't mean you can't sign data, it just means you have to load your private keys from file and manage them yourself. We have already seen how to do this in the decryption section above how to decode RSA keys, but not DSA keys.

DSA is purely a signature algorithm, it cannot perform encryption to facilitate a key exchange. For this purpose, it is often supplemented by either RSA or DH.

TODO: ASN.1 dec example for decoding DSA keys that openssl spits out.

What to sign

If you wish to vouch for a data set, for purposes of ensuring its integrity, etc. It is usual to take the hash of the data, and then to sign that relying on the

(C) Copyright Nokia 2013. All rights reserved.

property of the cryptographic hash that meaningful collisions are monumentally unlikely to provide a guarantee that the signed data has not been tampered with.

The signing classes provided by Symbian will not hash the data you provide, that must be done separately. Symbian provides some common hash functions such as MD5 and SHA1 to accomplish this task.

TODO: link to an article on hash functions

How to sign data

TODO: contrived code sample for RSA and DSA signatures.

Performing a DH key exchange

The DH algorithm is not able to sign data, but is a fairly fast method of performing a secure key exchange.

Diffie Hellman key exchange

// TODO: DH key exchange example

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.