Programming

1. Overview

There are two main modes of operation for the libraries. Signing and verifying. Verifying is the simplest operation, as it (generally) operates on a DOM <Signature> structure that has already been created.

Signing on the other hand can be more difficult, as there may be a requirement to create the DOM structure necessary for the signature prior to the actual signing operation.

The rest of this section provides a very high level overview on how to use the library for signing and verificataion of signatures.

> **Note:**
> Full API documentation for the current official release can be found here. The API documentation is also generated nightly from the CVS repository.

Two samples are provided :

- Simple HMAC Signing
- Simple DSA Validation

The code snippets are taken directly from some of the sample code provided in the src/samples directory in the distribution. More information on the API can be found in the API Documentation.

2. A simple HMAC Signing example

The first example is based on the simpleHMAC.cpp code in samples. It creates an XML letter, the appends a dummy signature to the end, using an enveloped-signature transform.

2.1. Setup

The following code snippet initialises Xerces, Xalan and XSEC. Note that the enveloped transform is implemented using an XPath expression, so it is imperitive the Xalan libraries are initialised.

```
int main (int argc, char **argv) {
```

```
    try {
        XMLPlatformUtils::Initialize();
#ifndef XSEC_NO_XALAN
        XalanTransformer::initialize();
#endif
        XSECPlatformUtils::Initialise();
    }
    catch (const XMLException &e) {

        cerr << "Error during initialisation of Xerces" << endl;
        cerr << "Error Message = : "
                    << e.getMessage() << endl;

    }

    // Create a blank Document

    DOMImplementation *impl =
        DOMImplementationRegistry::getDOMImplementation(MAKE_UNICODE_STRING("Core"));

    // Create a letter
    DOMDocument *doc = createLetter(impl);
    DOMElement *rootElem = doc->getDocumentElement();
```

In the sample application, the call to *createLetter(impl)* simply creates a letter DOM structure with a to and from address and some text. This is done using standard DOM calls via Xerces.

Once the system is initialised and the DOM document is created, a DSIGSignature object is created via the *XSECProvider* interface class. The signature object is then used to create a blank signature DOM node structure which is then inserted at the end of the document.

```
    XSECProvider prov;
    DSIGSignature *sig;
    DOMElement *sigNode;

    try {

        // Create a signature object

        sig = prov.newSignature();
        sig->setDSIGNSPrefix("ds");

        // Use it to create a blank signature DOM structure from the doc

        sigNode = sig->createBlankSignature(doc,
                                    CANON_C14N_COM,
                                    SIGNATURE_HMAC,
                                    HASH_SHA1);
```

The call to *newSignature* creates a signature object only. No DOM nodes are created at this

point. The call to *setDSIGNSPrefix* tells the XSEC library what namespace prefix to use for the signature object when it starts to create DOM nodes (in this case "ds" will be used). By default, the library will use "dsig" as the prefix for the name space for Digital Signatures.

Finally, the call to sig->createBlankSignature sets up both the DOM structure and the XSEC objects for a new signature with no <Reference> elements. In this case, the signature will be made using Commented C14n canonicalisation, and a HMAC-SHA1 signature.

Warning:

The XSECProvider class still "owns" the DSIGSignature object. To delete the object, the original provider.release(sig) call should be used. Never delete a DSIGSignature object directly.

2.2. Create a Reference and Sign

Now that the signature object is created, the signature is inserted into the document, and a reference is created and set for an enveloping transform.

```
        // Insert the signature DOM nodes into the doc

        rootElem->appendChild(doc->createTextNode(MAKE_UNICODE_STRING("\n")));
        rootElem->appendChild(sigNode);
        rootElem->appendChild(doc->createTextNode(MAKE_UNICODE_STRING("\n")));

        // Create an envelope reference for the text to be signed
        DSIGReference * ref = sig->createReference("");
        ref->appendEnvelopedSignatureTransform();
```

The "" parameter to *createReference* sets the URI attribute for the reference to be "" - indicating the root element of the document in which the signature resides. The call to *appendEnvelopedSignatureTransform* adds a standard eneveloped-signature transform to the Reference node.

The macro MAKE_UNICODE_STRING is defined within the library header files and is used to transcode local code page strings.

Note:

There is no need to insert the reference object into the DOM structure. This is done automatically by the *createReference* call.

Finally we create a signing key and sign the document.

```
        // Set the HMAC Key to be the string "secret"

        OpenSSLCryptoKeyHMAC * hmacKey = new OpenSSLCryptoKeyHMAC();
```

```
        hmacKey->setKey((unsigned char *) "secret", strlen("secret"));
        sig->setSigningKey(hmacKey);

        // Add a KeyInfo element
        sig->appendKeyName("The secret key is \"secret\"");

        // Sign

        sig->sign();
    }

    catch (XSECException &e)
    {
        cerr << "An error occured during a signature load\n   Message: "
             << e.getMsg() << endl;
        exit(1);

    }
```

The first two code lines create an OpenSSLCryptoKeyHMAC object, and set the key value to the string "secret". The OpenSSL... classes are the interface layer between XSEC and OpenSSL. More information can be found in the API documentation, but the main point of note is that the XSEC library never deals directly with OpenSSL - it works via the XSECCrypto abstract classes which are implemented in the OpenSSLCrypto code. This would allow another person to re-implement the XSECCrypto code to use any cryptographic provider required.

Note:

Once the key is passed to the signature it is owned by the signature. The signature object will delete the key when it is itself deleted, or a new key is passed in.

The call to *sig->appendKeyName()* is used to append a <KeyName> element into the <KeyInfo> block. The KeyInfo block was created as part of this call.

After the call to *sig->sign()* the DOM structure has the correct hash and signature values. The owner program can write, store or further manipulate the document as required. If a document manipulation might affect the signature (in this case almost anything would, as we are using an enveloping transform which effectively signs everything that is not part of the signature), then a further call to *sig->sign()* will re-sign the changes.

The last part of the code does some work to output the new DOM structure. The output should look something like the following:

```
<Letter>
<ToAddress>The address of the Recipient</ToAddress>
<FromAddress>The address of the Sender</FromAddress>
<Text>
To whom it may concern
```

```
...
</Text>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm=
"http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments"/>
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
<ds:Reference URI="">
<ds:Transforms>
<ds:Transform Algorithm=
"http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>askxS/A3BaLCjFjZ/ttU9c12kA4=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>oYEdQYG1IHzbkR1UcJ9Q5VriRPs=
</ds:SignatureValue>
<ds:KeyInfo>
<ds:KeyName>The secret key is "secret"</ds:KeyName>
</ds:KeyInfo>
</ds:Signature>
</Letter>
```

Note that the DigestValue and SignatureValue elements have been filled in.

3. A simple validation example

The second example takes a pre-signed document and an associated certificate and verifies the embedded signature. The document in question is a simple purchase order, and changes are made to the value of the order to demonstrate a signature failing verification.

3.1. Setup

As in the first example, Initialisation of the libraries is performed, and Xerces is used to read in the document (which in this case is stored in a string in the source code).

In order to be able to modify the contents of the document later on, we also quickly find the string containing the value of the purchase order.

For the sake of brevity, the code relating to parsing the in-memory document has been removed from the snippet below.

```
int main (int argc, char **argv) {

    try {
        XMLPlatformUtils::Initialize();
#ifndef XSEC_NO_XALAN
        XalanTransformer::initialize();
```

```
#endif
        XSECPlatformUtils::Initialise();
    }
    catch (const XMLException &e) {

        cerr << "Error during initialisation of Xerces" << endl;
        cerr << "Error Message = : "
             << DOMString(e.getMessage()) << endl;

    }

    ...

    Xerces is used to parse the document here



    DOM_Document doc = parser->getDocument();

    // Find the Amount node
    DOMNode *amt = doc->getDocumentElement();

    if (amt != NULL)
        amt = amt->getFirstChild();

    while (amt != NULL &&
           (amt->getNodeType() != DOMNode::ELEMENT_NODE ||
           !strEquals(amt->getNodeName(), "Amount")))
        amt = amt->getNextSibling();

    if (amt != NULL)
        amt = amt->getFirstChild();

    if (amt == NULL || amt->getNodeType() != DOMNode::TEXT_NODE) {
        cerr << "Error finding amount in purchase order" << endl;
        exit (1);
    }
```

3.2. Create the Signature and Key objects

Now that the document is in memory, an XSECProvider is created and used to create a new DSIGSignature object. In addition, the OpenSSL interface routines are used to read in a certificate and obtain the associated public key.

```
    XSECProvider prov;

    DSIGSignature * sig = prov.newSignatureFromDOM(doc);


    try {
        // Use the OpenSSL interface objects to get a signing key
```

```
        OpenSSLCryptoX509 * x509 = new OpenSSLCryptoX509();
        x509->loadX509Base64Bin(cert, strlen(cert));

        sig->load();
```

In this case, the signature is create with the *newSignatureFromDOM* method. This tells the library that the signature structure (although not necessarily a signed structure) already exists in the DOM nodes. The library attempts to find the <Signature> node so that the load will work. (The library will throw an XSECException if it cannot find the Element.)

The later call to *sig->load()* tells the library to read the DOM structure and create the appropriate DSIG elements.

In this case an OpenSSLCryptoX509 object is also created. It is used to read in the *cert* string and convert to an X509 structure. This could also be done using standard calls directly to OpenSSL, but this is a quick shortcut.

3.3. Find a key

As we already know the key, the following code snippet loads the key directly from the related X509. However prior to doing this, the code demonstrates using the DSIGKeyInfo structures to find the key name that was embedded in the certificate. In an application, this could be used to reference the correct key to be passed in. (Maybe via an XKMS call.)

the *safeBuffer* type is used extensively within the XSEC library to safely handle variable length strings and raw buffers. The call to *rawCharBuffer()* simply returns a (char *) type pointer to the buffer within the *safeBuffer*

The call to *clonePublicKey()* returns a copy of the public key embedded in the certificate. It is owned by the caller, so in this case it can safely be passed to the DSIGSignature object where it will be destroyed when another key is loaded or the object is released by the XSECProvider.

```
        DSIGKeyInfoList * kinfList = sig->getKeyInfoList();

        // See if we can find a Key Name
        safeBuffer kname;
        DSIGKeyInfo * kinf = kinfList->getFirstKeyInfo();
        while (kinf != NULL) {
            kname = kinf->getKeyName();
            if (kname.sbStrcmp("")) {
                cout << "Key Name = "
                     << kname.rawCharBuffer() << endl;
            }
            kinf = kinfList->getNextKeyInfo();
        }
```

```
        sig->setSigningKey(x509->clonePublicKey());
```

3.4. Validate the signature

Finally the signature is validated. In this case, we validate it three times. First with the original DOM structure, then with the price changed and finally with the price set back to the original value.

```
        cout << "Amount = " << amt << " -> ";

        if (sig->verify()) {
            cout << "Signature Valid\n";
        }
        else {
            cout << "Incorrect Signature\n";
        }

        amt.setNodeValue("$0.50");

        cout << "Amount = " << amt << " -> ";

        if (sig->verify()) {
            cout << "Signature Valid\n";
        }
        else {
            cout << "Incorrect Signature\n";
        }

        amt.setNodeValue("$16.50");

        cout << "Amount = " << amt << " -> ";

        if (sig->verify()) {
            cout << "Signature Valid\n";
        }
        else {
            cout << "Incorrect Signature\n";
        }
```

When run, the program outputs the following:

```
Key Name = C=AU, ST=Vic, O=XML-Security-C Project,
CN=Samples Demo Certificate
Amount = $16.50 -> Signature Valid
Amount = $0.50 -> Incorrect Signature
Amount = $16.50 -> Signature Valid
```