XML Encryption Programming

1. Overview

Note:

The Encryption functionality within the library is currently beta. Whilst the API is considered fairly functional, it may change in version 1.2 as a result of feedback received from version 1.1.

As with signatures, there are two main modes of operation for the library when performing encryption functions - Encryption and Decryption. Decryption is generally fairly simple, as the library will handle most of the work around de-referencing key material and re-creating a DOM document (or returning a byte stream).

Encryption is fairly simple if you are trying to encrypt a DOM structure. The library will encrypt the nodes and then replace them with the encrypted version. However if you want to embed an arbitrary encrypted object in the document, you will need to encrypt it first and then pass the encrypted text into the library.

The rest of this page looks at some simple examples around encrypting and decrypting nodes within an XML document

2. A simple encryption example

The next example encrypts an element (and all its children) from a pre-generated document. It uses a randomly generated key to handle the bulk encryption, and then encrypts this using an RSA public key. The resultant encrypted key is embedded in an <EncryptedKey> element.

This example can be found in the src/samples directory as *simpleEncrypt.cpp*.

2.1. Setup

The first step is initialisation of Xerces, Xalan (if used) and XML-Security. Once this is done, we create a document. For brevity, the details of the call to *createLetter* are not included on this page. The function is very simple - it creates an XML DOM document that represents a letter, and sets a global variable (*g_toEncrypt*) that will be used later on to determine what node to encrypt.

```
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);
```

2.2. Setup for Encryption

Once the library is initialised, we create a *XENCCipher* object in a manner similar to the creation of a *DSIGSignature* object. The *XENCCipher* object is used to actually perform encryption/decryption functions and to manipulate the various encryption objects provided by the library.

As well as creating the *XENCCipher* object, the sample uses the *RAND_bytes* function within the **OpenSSL** library to create a random key that will be used during the encryption process.

```
    try {

        /* Create the cipher object that we need */

        XSECProvider prov;
        XENCCipher *cipher;

        cipher = prov.newCipher(doc);

        /* Now generate a random key that we can use to encrypt the element
         *
         * First check the status of the random generation in OpenSSL
         */
```

```
        if (RAND_status() != 1) {

            cerr << "OpenSSL random generation not properly initialised" << endl;
            exit(1);

        }

        unsigned char keyBuf[24];
        if (RAND_bytes(keyBuf, 24) == 0) {

            cerr << "Error obtaining 24 bytes of random from OpenSSL" << endl;
            exit(1);

        }
```

2.3. Encryption of Element

The actual code to perform encryption is very small. Most of the complexity for standard encryption is hidden within the library.

The first two lines of code wrap the generated key bytes in an OpenSSL 3DES key. This is then passed into the *cipher* object with a call to *setKey(key)*.

The last line in the following block performs the actual encryption. the first parameter to *cipher->encryptElement* is the node that will be encrypted. The second is the algorithm to be used. This is used to calcualte the Algorithm URI to be set in the <EncryptedData> element.

This call to *EncryptElement* will encrypt the provided element using the key set previously. The passed in element will be replaced with an <EncryptedData> element containing the encrypted version of the element and all its children.

If no further information is required to be embedded in the <EncryptedData> structure (such as <KeyInfo> nodes), the usage of the library could be terminated here.

```
        /* Wrap this in a Symmetric 3DES key */

        OpenSSLCryptoSymmetricKey * key =
            new OpenSSLCryptoSymmetricKey(XSECCryptoSymmetricKey::KEY_3DES_192);
        key->setKey(keyBuf, 24);
        cipher->setKey(key);

        /* Encrypt the element that needs to be hidden */
        cipher->encryptElement(g_toEncrypt, ENCRYPT_3DES_CBC);
```

2.4. Create an <EncryptedKey>

The following snippet of code uses the previously created *XENCCipher* object to encrypt the pseudo random key using an RSA key loaded from a X.509 certificate.

The first two lines load the certificate into an OpenSSLCryptoX509 structure, which is then used to extract the public key from the certificate and pass into the cipher.

A call to *setKEK* is used rather than *setKey*. This call is used to tell the cipher object that the key being used is a Key Encryption Key, and should be used for encrypting/decrypting <EncryptedKey> elements.

The final line actually performs the encryption and created the <EncryptedKey> structure. The first two parameters define the buffer and its length to be encrypted. The last defines the encryption algorithm to be used.

The *encryptedKey* method returns an *XENCEncryptedKey* object. This contains the DOM structure for the object, but it is not yet rooted in a particular document. (Although it is created using the *DOMDocument* that was passed in during the call to *newCipher*.)

```
        /* Now lets create an EncryptedKey element to hold the generated key */

        /* First lets load the public key in the certificate */
        OpenSSLCryptoX509 * x509 = new OpenSSLCryptoX509();
        x509->loadX509Base64Bin(cert, strlen(cert));

        /* Now set the Key Encrypting Key (NOTE: Not the normal key) */
        cipher->setKEK(x509->clonePublicKey());


        /* Now do the encrypt, using RSA with PKCS 1.5 padding */

        XENCEncryptedKey * encryptedKey =
            cipher->encryptKey(keyBuf, 24, ENCRYPT_RSA_15);
```

2.5. Append <EncryptedKey> to <EncryptedData>

The final part (other than outputting the result) is to retrieve the <EncryptedData> element that was previously created and append the newly created <EncryptedKey> as a <KeyInfo> element.

```
        /*
         * Add the encrypted Key to the previously created EncryptedData, which
         * we first retrieve from the cipher object.  This will automatically create
         * the appropriate <KeyInfo> element within the EncryptedData
         */

        XENCEncryptedData * encryptedData = cipher->getEncryptedData();
        encryptedData->appendEncryptedKey(encryptedKey);
```

The above code results in a document that contains the newly created <EncryptedData> as follows:

```
<Letter>
<ToAddress>The address of the Recipient</ToAddress>
<FromAddress>The address of the Sender</FromAddress>
<xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
<xenc:CipherData>
<xenc:CipherValue>Wh8pAkDsQceHiktGxnlhXGfEMPDOLB6FwWp8PLedFEB3L3F6xHUoCOerIvA7Pgvv
VYzVqLv4a5x5YdnCqikkFBLE/fruAUe2Z8ZTEn/CaPYmpzU6qYHALCl7Q61LcbqH
R87TzroBYsYwfHmXmrKHL9K9sB6zmuec1TjVzm2c/Xs=
</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedKey>
</ds:KeyInfo>
<xenc:CipherData>
<xenc:CipherValue>YhqQciiFkLG1z0I1TJC6Pewnzw/gmVuGqcTvHtWpgak/b3NQDRAlv07lJOmBLoHX
23LQ1CdPSxvnyerlJGwkY6xJ0M5tjpDregTVcECXo/bd+x8eIsF2kaawoZGCqD1K
96T36Fx9rHek9bY/Hp1OiQ==
</xenc:CipherValue>
</xenc:CipherData>
</xenc:EncryptedData></Letter>
```

3. A simple decryption example

The final example shows how to use the library to decrypt an EncryptedData structure. A private key is loaded as a Key Encryption Key (KEK), and a call is made to the library which decrypts the encrypted data and inserts the resulting DOM nodes back into the original document.

This example can be found in the src/samples directory as *simpleDecrypt.cpp*.

3.1. Setup

The setup process is much the same as for simpleVerify. The document (which is the document created in simpleEncrypt) is parsed using Xerces and a *DOMDocument* is returned.

3.2. Load Private Key

The *simpleDecrypt* uses a preloaded RSA private key for the decryption. A key resolver (*XSECKeyInfoResolver*) can also be used to provide a callback mechanism such that applications can determine the correct key at run time.

The following code uses a *XSECProvider* to obtain a *XENCCipher*uses OpenSSL to load the private key from the *s_privateKey* char array.

The key is loaded using a call to *setKEK*. This method loads the key as a Key Encryption Key - which means it will be used to decrypt an <EncryptedKey> structure.

```
        XSECProvider prov;
        XENCCipher *cipher;

        cipher = prov.newCipher(doc);

        /* Load the private key via OpenSSL and then wrap in an OpenSSLCrypto construct
        BIO * bioMem = BIO_new(BIO_s_mem());
        BIO_puts(bioMem, s_privateKey);
        EVP_PKEY * pk = PEM_read_bio_PrivateKey(bioMem, NULL, NULL, NULL);

        /* NOTE : For simplicity - no error checking here */

        OpenSSLCryptoKeyRSA * k = new OpenSSLCryptoKeyRSA(pk);
        cipher->setKEK(k);
```

3.3. Perform Decryption

Now that the key is loaded, the actual decryption is performed using two lines of code. The first finds the node to be decrypted. In this case, the *findXENCNode* library function is used.

The second line, *decryptElement* actually performs the decryption. It performs the following steps :

- Load the <EncryptedData> structure into an *XENCEncryptedData* structure.
- if no decryption key is loaded (in this case, none is), search the <KeyInfo> list for an <EncryptedKey> element (one will be found in this case).
- Use the previously loaded KEK to decrypt the key found in the previous step.
- Use the decrypted key to decrypt the <EncryptedData> data
- Parse the decrypted data into DOM nodes
- Replace the <EncryptedData> with the DOM fragment returned in the previous step

```
        /* Find the EncryptedData node */
        DOMNode * encryptedNode = findXENCNode(doc, "EncryptedData");

        /* Do the decrypt */
        cipher->decryptElement((DOMElement *) encryptedNode);
```

The result of these steps is the decrypted letter.

```
<Letter>
<ToAddress>The address of the Recipient</ToAddress>
<FromAddress>The address of the Sender</FromAddress>
```

```
<Text>
To whom it may concern, my secret credit card number is :
  0123 4567 89ab cdef

...
</Text></Letter>
```