XML Security: Implement security layers, Part 2

Core technologies -- XML encryption and XML signature

Level: Introductory

Manish Verma (mverma@secf.com)
Principal Architect, Second Foundation
October 30, 2003

A host of emerging technologies, such as Web services, use XML extensively for data exchange. As a result, the security of XML, while in transit as well as when in storage, assumes very high importance. This series explores the technologies that help make XML secure. Part 1 covered the basic plumbing technologies required for XML security. This article builds on that base, covering the core technologies required for XML security -- XML encryption and XML signature. It also goes through the step-by-step process of using these technologies to secure an XML message.

A number of existing encryption techniques, such as Secure Sockets Layer (SSL), can encrypt an XML document just like any other document, so why is another encryption standard needed? Here, I'll start by exploring the objectives and motivations for coming up with yet another encryption standard -- XML encryption.

XML encryption

The primary objectives of XML encryption are:

- Support the encryption of any arbitrary digital content, including XML documents
- Ensure that the encrypted data, whether it's in transit or in storage, cannot be accessed by unauthorized persons
- Maintain the security of the data even beyond one message hop -- meaning, the security of the data is persisted not only when the data is being transferred (which is what SSL guarantees), but also when the data is at rest at a particular node
- Represent the encrypted data in XML form
- Make it possible for portions of the XML to be selectively encrypted

Compare this with what SSL over HTTP (also known as HTTPS) has to offer. Using SSL over HTTP, the entire message gets encrypted; the whole message is then decrypted at the first destination and is open for snooping before it is encrypted again as a whole for the second hop. The encryption offered by SSL over HTTP only exists for the duration of transit and is not persistent.

XML encryption overview

One of the defined objectives clearly states that encrypted XML data should be represented in XML form. In the resulting XML, two important elements are worth understanding: `<EncryptedData>` and `<EncryptedKey>`. `<EncryptedData>` contains all of the encrypted content other than the encryption key. When the key is encrypted, the resulting content is placed inside the `<EncryptedKey>` element.

In addition to the encrypted content, XML encryption allows you to specify the algorithm used for encryption or the encryption

key used as part of the two elements discussed above. This means you don't have to keep track of them separately for later reference, or send them to the receiving parties through some other transport mechanism.

Note: XML encryption does not define any new algorithms, but instead uses existing ones.

The XML encryption process

In this section, I start with a normal snippet of XML and go through various stages of encrypting it as per the XML Encryption standard (see Resources). I'll start with the XML shown in Listing 1.

Listing 1. Sample XML

```
<?xml version="1.0"?>
<PurchaseOrderRequest>
  <Order>
    <Item>
      <Code>Screw001</Code>
      <Description>Screw with half centimeter thread</Description>
    </Item>
    <Quantity>2</Quantity>
  </Order>
  <Payment>
    <CreditCard>
      <Type>MasterCard</Type>
      <Number>1234567891234567</Number>
      <ExpiryDate>20050501</ExpiryDate>
    </CreditCard>
    <PurchaseAmount>
      <Amount>30000</Amount>
      <Currency>INR</Currency>
      <Exponent>-3</Exponent>
    </PurchaseAmount>
  </Payment>
</PurchaseOrderRequest>
```

Listing 2 demonstrates how part of the XML in Listing 1 can be encrypted. Each step in the encryption process is explained after the listing.

Listing 2. XML encryption

```
//Get the DOM document object for the XML that you
// want to encrypt.
// getDocument method that takes XML file name as input
// and returns DOM document provided in Listing 3 (Step 1)
Document doc = XmlUtil.getDocument(xmlFileName);
String xpath = "/PurchaseOrderRequest/Payment";

// Step 2. Get the shared secret. This key is used to encrypt the
// XML content
Key dataEncryptionKey = getKey();
// Algorithm type used to generate shared secret
// i.e. content encryption key
AlgorithmType dataEncryptionAlgoType =  AlgorithmType.TRIPLEDES;
// Get the key pair. You are interested in the public key
// as that is the one you will use for encrypting the
// XML content
KeyPair keyPair = getKeyPair();
// Step 3. Get the public key of the key pair
Key keyEncryptionKey = keyPair.getPublic();
// Algorithm type used to generate the public
        // private key pair
AlgorithmType keyEncryptionAlgoType = AlgorithmType.RSA1_5;

KeyInfo keyInfo = new KeyInfo();

// Step 4
try {
        Encryptor enc =
                new Encryptor(
                        doc,
                        dataEncryptionKey,
                        dataEncryptionAlgoType,
                        keyEncryptionKey,
                        keyEncryptionAlgoType,
                        keyInfo);
        XPath xpath = new XPath(xPath);
        // Step 5
```

```
        try {
                enc.encryptInPlace(xpath);
        } catch (XPathException e1) {
                System.out.println("XPAth is not correct");
                e1.printStackTrace();
        }
        XmlUtil.XmlOnStdOut(doc);
} catch (Exception e) {
        System.out.println("Some exception");
        e.printStackTrace();
}
```

Step 1: Convert the XML into a DOM object as demonstrated in Listing 3:

Listing 3. Creating a DOM object from the XML

```
public static Document getDocument(String fileName) {
Document doc = null;

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

File f = new File(fileName);
DocumentBuilder builder = null;
try {
        builder = factory.newDocumentBuilder();
} catch (ParserConfigurationException e) {
        System.out.println("Parse configuration exception");
        e.printStackTrace();
}
try {
        doc = builder.parse(f);
} catch (Exception e1) {
        System.out.println("Some exception");
        e1.printStackTrace();
}
return doc;
}
```

Step 2: Get the shared secret. You will use this key to encrypt the XML content. The accompanying source code uses XML encryption that only recognizes the triple-DES algorithm, hence I create the key with that algorithm.

Step 3: Get the public key of the public-private key pair as described in Part 1 of this article series; you will need it to encrypt the shared secret. As you saw in Part 1, this public key would be based on the RSA algorithm.

Step 4: Create an `Encryptor` object with a data encryption key, a key encryption key, their associated algorithms, and key information to be included in the output. The algorithms specified while creating the `Encryptor` object must match the keys. `Encryptor` is the main object in the encryption process. Its class is available in `com.verisign.xmlenc` package. `Encryptor` encrypts according to the W3C XML Encryption specification. You can specify which type of encryption you want, Element or Content. In Listing 2, the encryption type is Element, which is the default. `Encryptor` understands XPath expressions in order to identify the XML element for encryption.

Step 5: In the final step, call either the `encrypt` or `encryptInPlace` method on the `Encrypt` object, passing XPath as the input parameter. XPath specifies the element within the XML that needs to be encrypted. All the child elements within the element and their attributes pointed to by XPath are also encrypted. In this example, you are encrypting the `/PurchaseOrderRequest/Payment` element of the XML. Both the `encrypt` and `encryptInPlace` methods encrypt the XML element as specified by the XPath using the secret key passed, and both methods encrypt the secret key using the public key and embed it with the XML-encrypted content. The only difference between the two methods is that `encrypt` returns a new DOM document containing the encrypted data, whereas `encryptInPlace` modifies the original document itself to contain the encrypted data. The encrypted XML that results is shown in Listing 4.

Listing 4. Encrypted XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <PurchaseOrderRequest>
    <Order>
      <Item>
        <Code>Screw001</Code>
          <Description>Screw with half centimeter thread</Description>
      </Item>
      <Quantity>2</Quantity>
    </Order>
    <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>
            <xenc:EncryptionMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
              <xenc:CipherData>
              <xenc:CipherValue>
       F1aIpdp3axm8nFofx/xX62VlsxilddHcxaevd7sbr+lv/fzZ7e8ovmKGQopAjclxPTybpkW
       YG8GVcOIbD4UGR24CNxeB7eZCws5/RKBTqKp+76FkVxf+G+EqgMmueRqoaF4oYOrTKquWLnR
       kiSOFmplRaJ8G7bR2j0eTFdiFRk=
              </xenc:CipherValue>
            </xenc:CipherData>
          </xenc:EncryptedKey>
        </ds:KeyInfo>
        <xenc:CipherData>
          <xenc:CipherValue>
       KMkufRUY7rs0i+4jX6VhviiUIbYWay1KbwhTQxH9SaqJ6HA+Qc2Ce7TVZUQuH0GGD4xTR8hB
       hOls+hgHA16EfmmxLd3E+YqO4sXQ+GkX9O9EcO4ULha/q1KmP2yNGNy/tavdj9a7JuZnnNGV
       /M4gxdt5fCJXT0A9bw9HwKR/Pc81rZYWa7fOrmvDvC7Q+//OCzkqcAaCmAHEySWbv2vK3T+a
       GlQOI2Wooxa9hm7Dx70BuLI8ihhSAV3moK+JAPdn1vdCpoFKdzzq2HSh/yOisYZvQOh+jIks
       MW8oUzWnVUe/DFztPtvvDKbPE/xoAasixlbDLa42gFFe9uzEeIG89XBMSkZtTio0zn9xppSf
       Dc0WFMy+UoLnCA==
          </xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedData>
  </PurchaseOrderRequest>
```

Listing 4 is an XML snippet with one part encrypted. This encrypted part can only be read by the receiver with the private key corresponding to the public key that was used to encrypt the data.

Closing the loop, the code in Listing 5 allows encrypted XML to be decrypted.

Listing 5. XML decryption

```
// Get the DOM document object for the XML that you
// want to decrypt (The one shown in Listing 4)
// The getDocument method that takes an XML file name as input
// and returns a DOM document is provided in Listing 3 (Step 1)
Document doc = XmlUtil.getDocument(encryptedXmlFileName);

// Step 2. Get the private key of the pair whose public key was
// used to encrypt the XML
Key privateKey = keyPair.getPrivate();
// Specifying the XPath at which encrypted data is lying
// in the XML

// Step 3. Specify XPath expression
String xpath = "//xenc:EncryptedData";
// Specify the namespace that relates to the XPath
// expression
String[] ns =
        { "xenc", "http://www.w3.org/2001/04/xmlenc#" };
// Create the XPath helper with the XPath expression and a map
// of namespaces that relate to the XPath expression

XPath xpath = new XPath(xPath, ns);

// Step 4. Create the Decryptor object with decryption key and
// location of the encrypted data to be decrypted
Decryptor decrypt = null;
try {
        decrypt = new Decryptor(doc, privateKey, xpath);
} catch (Exception e) {
        System.out.println("Some exception");
        e.printStackTrace();
}
// Step 5. Method decryptInPlace is called to decrypt the
// encrypted contents of the XML
```

```
try {
        decrypt.decryptInPlace();
} catch (Exception e1) {
        System.out.println("Some exception");
        e1.printStackTrace();

}
```

Listing 5 demonstrates how you can decrypt the encrypted data if you have the right private key. The following steps explain the process of decryption.

Step 1: Convert the encrypted XML into a DOM object as was done before encryption.

Step 2: Get the private key of the key pair whose public key was used to encrypt the XML. Note that decryption uses the corresponding private key of the public key used for encrypting the XML.

Step 3: Create the XPath along with the relevant namespaces where the encrypted data is located in the encrypted XML. The XPath in this case is `//xenc:EncryptedData`. Irrespective of what elements are encrypted, the encrypted data will always be under the element `xenc:EncryptedData` in the encrypted XML. The XPath `//xenc:EncryptedData` says to look for the `EncryptedData` element anywhere it might be exiting in the XML.

Step 4: Create the `Decryptor` object with decryption key and the location of the encrypted data that is to be decrypted. `Decryptor` is the main object in the decryption process. Its class is available in the `com.verisign.xmlenc` package. `Decryptor` decrypts as per the W3C XML Encryption specification (see Resources). Decryption of both the Element and Content types is supported. `Decryptor` understands XPath expressions in order to identify the XML element for decryption.

Step 5: The final step in the decryption process is to call either the `decryptInPlace` or `decrypt` method on the `Decryptor` object. Both method calls use the supplied private key to decrypt the shared secret key (which is part of the encrypted message), and then use the shared secret to decrypt the rest of the message. The only difference in the calls is that `decrypt` decrypts the XML and creates a new DOM object, whereas `decryptInPlace` decrypts the message in the same DOM object that it receives as input.

XML signatures

Digital signatures have been in use for quite some time, where any digital content could be digitally signed using Public Key Cryptography Standards (the most common being the PKCS#7 signature). Secure Multipurpose Internet Mail Extensions (S/MIME) allows digital signatures to be attached to e-mail messages, such that the recipient can verify the identity of the signer.

XML signature is an extension of already existing digital signature infrastructure. Some objectives and motivations for creating an XML signature are:

- Structure around the digital signature allows digital signatures to be represented as XML documents
- Possibility to digitally sign part of the XML, leaving the rest of the document unsigned
- Possibility to have more than one digital signature sign different parts of the same XML document
- Need to persist the signatures and not simply use them for document transportation and communication

XML signature overview

XML signatures can be used to sign any digital content including XML documents. Signature of digital content is a two-stage process. In the first stage, the digital content is digested and the resulting value is placed in an XML element. In stage two, the digested value is picked and signed. The reason for doing this is simple: Digesting the original content generates a small but unique encrypted value (digest) that takes less time to sign as compared to the original content.

After the XML (or some part thereof) is digitally signed, the resulting XML signature is represented as an XML element that is identified as `<Signature>`. The original content is related to the digital signature based on these XML signature type definitions:

- **Enveloping signature:** The <Signature> element includes the element that is digitally signed. The digitally signed element becomes the child of the <Signature> element.
- **Enveloped signature:** The <Signature> element becomes a child element of the data being signed. The <Signature> element refers to the signed element by using information in its <Reference> element.
- **Detached signature:** The <Signature> element and the signed element are kept separate. The element being signed and the <Signature> element could be siblings in the same document, or the <Signature> element could be in an altogether different document.

In addition to a reference to the digital content being signed, the <Signature> element includes information about the following:

- The method used to canonicalize the digital content
- The algorithm used to generate the signature for the canonicalized element to be signed
- Additional information that specifies how to process the element to be signed before it is digested

The XML signature process

Continuing with the example in Listing 1, I will now take you through the process of digitally signing an element of the XML. Listing 6 demonstrates how part of the XML is digitally signed; each step in the signing process is explained after the listing.

Listing 6. The digital signature process

```
// Get the DOM document object for the XML that you
// want to digitally sign.
// getDocument method that takes XML file name as input
// and returns DOM document is provided in Listing 3 (Step 1)
Document doc = XmlUtil.getDocument(xmlFileName);
// XPath expression that is to be digitally signed
String xpath = "/PurchaseOrderRequest/Payment";

// Step 2. Get both the public and private keys. The private key is
// used to digitally sign the content, whereas the public key
// is sent along with the digitally signed content for the
// receiver to verify the digital signature.

contentKeyPair keyPair = getKeyPair();
PublicKey verificationKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();

// Step 3. Create a signer object passing the document whose part
// is to be signed, the private key to be used for
// signing, and the public key that is to be used
// for verification.

Signer signer = null;

// Step 4
try {
        signer = new Signer(doc, privateKey,
                                            verificationKey);
        XPath location1 =
        new XPath("/PurchaseOrderRequest/Payment/CreditCard");
        signer.addReference(location1);

        XPath location2 =
        new XPath("/PurchaseOrderRequest/Payment/PurchaseAmount");
        signer.addReference(location2);

} catch (Exception e) {
        e.printStackTrace();
}
// Step 5. Use the signer object to sign the document passing the
// XPath of the element that is to be signed.
try {
        d = signer.sign(new XPath(xPath));
} catch (Exception e1) {
        e1.printStackTrace();
}
```

Step 1: Convert the XML to be digitally signed into a DOM object as was done prior to encryption.

Step 2: Get the public as well as the private key of the key pair that you generated in the "Generating a public-private key pair"

of Part 1. You will use the private key to digitally sign the content, and send the public key along with the digitally signed message for the recipient to verify the digital signatures. As described earlier, the public-private key pair is based on the RSA algorithm.

Step 3: Create a `signer` object that passes the DOM document (part of which is to be signed), the private key to be used for signing, and the public key to be used for verification. Specifying the public key for verification is not mandatory. `Signer` is the main object in the digital signature process. Its class is available in the `com.verisign.xmlsig` package. `Signer` signs XML documents according to the W3C XML Signature specification (see Resources). All three modes of signatures -- enveloping, enveloped, and detached -- are supported.

Step 4: Specify the portions of the XML that need to be signed. XML locations are specified for digital signatures by adding references to the `Signer` object. References are added by calling the `addReference` method on the `Signer` object. The XPath of the element to be signed is provided as an argument to `addReference`. You can call `addReference` multiple times to specify different elements that you want signed.

Step 5: The final step is to sign the XML, which you do by calling the `sign` method on the `signer` object. The enveloping signature is created when the `sign` method is called without any arguments. Calling the `sign` method with an XPath argument where the digital signature is to be placed in the document helps create an enveloped signature.

Conclusion

Thus far in this series on XML security, I have defined what security means, and discussed XML canonicalization, PKI infrastructure, XML encryption, and XML signature. I have also attempted to demystify the XML security space by providing some of the theory behind it while keeping the focus on demonstrating how easy it is to encrypt and digitally sign XML.

In future articles, I will discuss Security Assertion Markup Language (SAML) and XML Key Management System (XKMS). SAML allows you to transfer credential and other related information about a subject, human or otherwise, between various cooperating entities without losing ownership of the information. XKMS allows for easy management of public key infrastructure (PKI) by abstracting the complexity of managing the PKI from client applications to a trusted third party.

Resources

- Download the source code used in this article.
- Read the Exclusive XML Canonicalization specification.
- Find out more about Canonical XML.
- Get a closer look at the XML Encryption Syntax and Processing specification.
- Read the XML-Signature Syntax and Processing specification.
- Read Murdoch Mactaggart's introduction to XML encryption and XML signature, "Enabling XML security" here on *developerWorks* (September 2001).
- Read Bilal Siddiqui's two-part article on XML Encryption. Part 1 explains how XML and security are proposed to be integrated into the W3C's Working Draft for XML Encryption (*developerworks*, March 2002). Part 2 examines the usage model of XML Encryption with the help of a use case scenario (*developerworks*, August 2002).
- Find more XML resources on the *developerWorks* XML zone.
- Learn how you can become an IBM Certified Developer in XML and related technologies.

About the author

Manish Verma is a Principal Architect at Second Foundation, a global IT services company. Manish has 10 years experience in all aspects of the software development life cycle and has designed integration strategies for client organizations running disparate systems. Manish's integration expertise is founded on his understanding of a host of technologies, including various legacy systems, .NET, Java technology, and the latest middleware. Prior to Second Foundation, Manish worked as a Software Architect and Technical Lead at Quark Inc., Hewlett Packard, Endura Software, and The Williams Company. You can contact Manish at mverma@secf.com.