The XML Security Suite: Increasing the security of e-business

Doug Tidwell (dtidwell@us.ibm.com)
Cyber Evangelist, developerWorks XML team
April 2000

As more and more companies use XML to transmit structured data across the Web, the security of documents becomes increasingly important. This article presents some basics of Web security, describes the components of the XML Security Suite, and gives examples that illustrate how the technologies in the XML Security Suite increase the security of Web commerce.

A brief overview of Web security

As more and more companies use XML to transmit structured data across the Web, the security of documents becomes increasingly important. The World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) are currently defining an XML vocabulary for digital signatures. The Tokyo Research Lab has created the XML Security Suite, a prototype implementation of the XML signature specification. The XML Security Suite, available from IBM's alphaWorks, contains utilities to automatically generate XML digital signatures.

When sending secure data across the Web, you need four things:

1. Confidentiality -- No one else can access or copy the data.
2. Integrity -- The data isn't altered as it goes from the sender to the receiver.
3. Authentication -- The document actually came from the purported sender.
4. Nonrepudiability -- The sender of the data cannot deny that they sent it, and they cannot deny the contents of the data.

SSL provides the first three functions; the XML Security Suite provides the fourth.

Creating a secure session

There are several steps involved in creating a secure session:

1. The secure server obtains a certificate from the appropriate certificate authority (CA).
2. The secure server sends its public key to the client.
3. The client uses the server's public key to encrypt a premaster secret (a random number generated by the client. The phrase *premaster secret* sounds much more sophisticated than *random number*). The server uses its private key to decrypt the premaster secret.
4. The server generates a new key based on the premaster secret. The key is known to the server, and can only be decrypted and used by the client that generated the premaster secret.

This process creates a session in which all traffic between the client and server is encrypted. Only the server and the client can decrypt each other's data. Confidentiality, integrity, and authentication are covered. SSL doesn't address nonrepudiability, however; if I have a document on my system, you can deny that you ever sent it, or you can deny that you created the content that's currently in the document. Nonrepudiability is one of the functions provided by the XML Security Suite.

The XML Security Suite

The XML Security Suite provides several important functions:

- XML signatures. This implementation is based on the XML-Signature Core Syntax and Processing specification currently being developed by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF). (See

Resources.)

- An implementation of the W3C's Canonical XML working draft. (See Resources.)
- Element-level encryption.

The following sections discuss each of these functions and how they contribute to Web security.

XML Signatures

The W3C and the IETF are currently working together on a proposal for XML-based digital signatures. The proposal defines a `<signature>` element that contains all the information needed to process a digital signature. Each digital signature refers to one of three things:

- An XML element contained inside the `<signature>` element
- An external XML document, referenced by a URI
- An external non-XML resource, referenced by a URI

Examples in this article show you how to create each of these resources. See the latest draft of the XML Signature proposal (see Resources) if you'd like to read all the gory details.

To illustrate how XML Signatures work, I generated a signature file called `signature.xml` for the Shakespearean sonnet used for so many of my XML examples. (View it in HTML; you can also download this file.)

In the signature file, the signed element is the sonnet, contained inside the `<dsig:Object>` element. The actual signature is contained in the `<SignatureValue>` element, with the signer of the document indicated in the `<X509Data>` element.

About the sample programs

Several sample programs illustrate the various functions of the XML Security Suite. If you want to follow along with the sample programs discussed here, there are several files you need. See Resources for information on where to get these files.

- The XML Security Suite itself Add both `xss4j.jar` and the path `/xss4j/samples` to your `classpath`.
- A Java 2 Development Kit, Version 1.2 or higher Make sure that this is the default JDK for your system.
- The `mail.jar` file from the JavaMail package Add this file to your `classpath`.
- An implementation of the Java Cryptography Extension A list of cryptographic service providers can be found at the Sun Web site. To avoid export complications, the examples here use OpenJCE, a free, open-source library. The file `jce.zip` is added to the classpath.
- Version 2 of IBM XML Parser for Java Version 3.0.1 The file `xml4j.jar` must be in your classpath.

Creating a certificate

Before you can create digital signatures, you need a certificate. Although you can get a certificate from a certificate authority, for the examples here, you'll act as your own CA. To create the X.509 certificate used in `signature.xml`, use the Java 2 `keytool` command:

Listing 1. keytool command

```
keytool -genkey -dname "CN=Doug Tidwell, OU=developerWorks, O=IBM,
L=Research Triangle Park, S=North Carolina, C=US" -keypass openstds
-storepass security -alias xss4j
```

In the `keytool` command, the distinguished name (dname) is composed of the common name (CN), organizational unit (OU), organization (O), location (L), state (S), and country (C). The distinguished name is designed to be unique across the Internet. The password for the key store `(-storepass)` is `security`, `openstds` is the password for the private key for this

Color-coding our colorful coding

This article features colorized code listings, something we're experimenting with here at dW. To generate our color-coded listings, I'm using a couple of open-source tools. First, I load the document (Java, HTML, XML, whatever) into Emacs. Emacs defines colors for keywords, comments, function names, and other programming language constructs -- about a dozen in all. After Emacs has loaded and colored a file, I use the HTMLize package, an open-source utility written in the ever-popular Emacs Lisp language. HTMLize takes a listing exactly as it appears in Emacs, then converts it to HTML. The result is a fully color-coded file that highlights keywords, comments, function names, and so on.

Let us know what you think about these new and improved code listings.

If you'd like to do this kind of thing yourself, see Resources for the appropriate links.

certificate (`-keypass`), and `xss4j` is the alias for this certificate (`-alias`).

Signing an internal XML resource
To create digital signatures, use the SampleSign application. This application is shipped with the XML Security Suite, and is found in the `xss4j/samples` directory. Our first signature will be for an internal XML resource. This means that the digital signature and the XML resource are all in the same file. Here's how `sonnet.xml` is signed to generate `signature.xml`:

Listing 2. SampleSign application

```
java SampleSign xss4j security openstds -embxml
file:///d:/xss4j/samples/sonnet.xml > signature.xml
```

(BTW, this command was entered as a single line.)

Notice that the alias, private key password, and key store password are the same as from Listing 1, the `keytool` command. Also notice the use of the `file: URL` instead of a simple filename, and that the output is piped (using the > operator) into the file `signature.xml`. The results of this command are the document shown in `signature.xml`.

Signing an external XML resource
Signing an external XML resource means that the `<Signature>` document contains the URI of the XML resource, not the resource itself. To create this kind of digital signature, use the `-extxml` option:

Listing 3. SampleSign application with -extxml option

```
java SampleSign xss4j security openstds -extxml
file:///d:/xss4j/samples/sonnet.xml > external-signature.xml
```

This produces a document similar to `signature.xml`, except the actual XML document is not contained in the `<Signature>`.

Signing a non-XML resource
The final signing example creates a digital signature for a non-XML resource. In this example, use a GIF file that contains the developerWorks logo: 

To create a digital signature, use the `-ext` option:

Listing 4. SampleSign application with -ext option

```
java SampleSign xss4j security openstds -ext
file:///d:/xss4j/samples/dwlogo.gif > external-gif-signature.xml
```

Verifying a digital signature
The XML Security Suite provides a utility, SampleVerify, that verifies a digital signature. You can check a given signature to ensure that the signed resource has not changed, and you can check that the signature matches the information in the sender's certificate. If a signature is valid, Listing 5 shows the results you'll get:

Listing 5. SampleVerify application

```
java SampleVerify -dom < external-signature.xml
Signer: CN=Doug Tidwell, OU=developerWorks, O=IBM, L=Research Triangle
Park, ST=North Carolina, C=US
SignedInfo Bytes: 1069
---------------------------------------
--> Location: file:///d:/xss4j/samples/sonnet.xml
    Validity: Ok
--> SignedInfo: Ok
--> All: Ok
---------------------------------------
```

If you change the signed file, the signature will no longer be valid. To illustrate this, add a blank space to the end of one of the <line> elements. Change `<line>My mistress' eyes are nothing like the sun,</line>` to `<line>My mistress' eyes are nothing like the sun, </line>`.

When we check the signature again, Listing 6 shows the results:

Listing 6. SampleVerify application (changed file)

```
java SampleVerify -dom < external-signature.xml
Signer: CN=Doug Tidwell, OU=developerWorks, O=IBM, L=Research Triangle
Park, ST=North Carolina, C=US
SignedInfo Bytes: 1069
---------------------------------------
--> Location: file:///d:/xss4j/samples/sonnet.xml
    Validity: NG
      Reason: Digests were mismatched.
--> SignedInfo: Ok
--> All: NG
---------------------------------------
```

Because the changed XML document doesn't match the digital signature, you know not to trust the document. (If someone other than the original document signer had tried to pass themselves off as the creator of the digital signature, the SignedInfo message would have indicated that.)

The joys of nonrepudiability

The main benefit of digital signatures is that they provide nonrepudiability. If you send me a signed document, I know that you're the one who sent it, because the signature contains your public key. In addition, because the signature is based on the contents of the document, the signature won't match if any changes have been made to the document.

Canonical XML

To calculate the digital signature, you need a common way to represent all XML documents. You can use the W3C's Canonical XML standard for this.

canonical form
The simplest form of something (Merriam-Webster's Collegiate Dictionary, Online Edition)

Even though two XML documents may not be identical, they may be equivalent for the purposes of an XML application. Consider the following two elements:

Listing 7. Equivalent but not identical XML

```
    <img alt="" src="dwlogo.gif" width="225" height="30"/>
    <img alt="" src="dwlogo.gif" height="30" width="225"/>
```

If you do a simple string comparison on these two elements, they're obviously not the same. From an XML processing perspective, though, they're equivalent. According to the XML 1.0 Recommendation, the order of attributes is not significant. Other insignificant differences between XML source documents are the amount of white space between attributes, and whether attributes with default values were actually included in the source document. To solve this problem, the W3C is currently

defining a canonical form for XML documents.

The XML Security Suite provides an XML Canonicalizer utility, a prototype implementation of the W3C's emerging standard for Canonical XML. To convert an XML document to its canonical form, use one of the following two commands:

Listing 8. XML Canonicalizer

```
java C14nDOM
 canonical-sonnet-DOM.xml
java C14nSAX
 canonical-sonnet-SAX.xml
```

The C14nDOM application uses a DOM parser, while the C14nSAX application uses a SAX parser. Although you can use these applications to generate the canonical form of an XML document, they are primarily used by the XML signature code. (BTW, the *c14n* abbreviation means that *canonicalization* is spelled beginning with a *c*, has 14 letters, and ends with an *n*. You'll often see *internationalization* similarly written as *i18n*.)

Because the XML Signature code uses canonicalization to generate digital signatures, you can make certain changes to the original document without affecting the validity of the digital signature. As an example, add some white space inside a tag. Change the line:

```
<sonnet type="Shakespearean">
```

to

```
<sonnet
type="Shakespearean">
```

After you make this change, run the SampleVerify application again to make sure the digital signature is still valid:

Listing 9. SampleVerify application (canonical form of file)

```

java SampleVerify -dom < external-signature.xml
Signer: CN=Doug Tidwell, OU=developerWorks, O=IBM, L=Research Triangle
Park, ST=North Carolina, C=US
SignedInfo Bytes: 1069
----------------------------------------
--> Location: file:///d:/xss4j/samples/sonnet.xml
    Validity: Ok
--> SignedInfo: Ok
--> All: Ok
----------------------------------------
```

Even though the document is different, the differences aren't semantically significant. Because the XML Security Suite uses the canonical form of the XML documents, the semantically insignificant differences are ignored.

Element-level encryption
One of the strong points of XML is that you can choose element names so that marked-up documents are much more readable. For example, look at a customer order in XML in Listing 10. (You can also download this file.)

Listing 10. custorder.xml

```
<?xml version="1.0"?>
<!DOCTYPE customer_order SYSTEM "custord.dtd">
<customer_order>
  <items>
    <item>
      <name>Turnip Twaddler</name>
      <qty>3</qty>
      <price>9.95</price>
    </item>
    <item>
      <name>Snipe Curdler</name>
      <qty>1</qty>
      <price>19.95</price>
    </item>
  </items>
  <customer>
    <name>Doug Tidwell</name>
    <street>1234 Main Street</street>
    <city state="NC">Raleigh</city>
    <zip>11111</zip>
  </customer>
  <credit_payment>
    <card_issuer>American Express</card_issuer>
    <card_number>1234 567890 12345</card_number>
    <expiration_date month="10" year="2004"/>
  </credit_payment>
</customer_order>
```

There are three sections to the document, an `<items>` element that lists all the items ordered by the customer, a `<customer>` element that contains information about the customer, and a `<credit_payment>` element that describes the credit card used to pay for this order. (Note to apprentice cyberthieves: The information above is not, in fact, my credit card number. If you are able to purchase goods and services using this information, please let me know.)

When early Internet users were reluctant to use credit cards to shop online, many e-business proponents pointed out that all credit card purchases have a level of risk. I commonly give my credit card to a waiter in a restaurant; I trust that my card won't be used for anything other than the meal I've just eaten. Likewise, when I buy something online, I'm trusting the merchant not to use my credit card for unauthorized purchases.

With the element-level encryption feature of the XML Security Suite, you could encrypt the sensitive information so that even the merchant couldn't see it. The merchant would pass the encrypted information on to the credit card processing agency, which would have the proper keys to decrypt the sensitive information. This would enable a greater level of security than the typical transactions of today.

To demonstrate element-level encryption, I've slightly modified the `CipherTest.java` file that ships with the XML Security Suite. (Here is the source for the modified file.) I'll review the changes I made, then illustrate how element-level encryption works.

The first change I made to `CipherTest.java` was to import the OpenJCE libraries, then define the ABAProvider class as a cryptography provider:

Listing 11. CipherTest.java change (import OpenJCE libraries)

```
import au.net.aba.crypto.*;
import au.net.aba.crypto.provider.*;
import au.net.aba.crypto.spec.*;
...
public class CipherTest {
  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println("Usage: CipherTest -e|-d passphrase infile outfile");
      return;
    }

    java.security.Security.
         addProvider(new au.net.aba.crypto.provider.ABAProvider());
```

Instead of calling the java.security.Security.addProvider method, you could modify the `java.security` file (in `JavaHome/lib/security`), replacing this line:

```
"security.provider.1=sun.security.provider.Sun"
```

with this one:

```
"security.provider.1=au.net.aba.crypto.provider.ABAProvider"
```

The only other change is to modify the code so that it will encrypt any `<credit_payment>` elements:

Listing 12. CipherTest.java change (encrypt elements)

```
if (n.getNodeType() == Node.ELEMENT_NODE) {
    //System.out.println(((Element)n).getTagName());
    if ((ne != null &&
        ((Element)n).getTagName().equals("credit_payment"))
        ||
        (nd != null &&
        ((Element)n).getTagName().equals("EncryptedElement"))) {
        child = n;
        break;
```

To illustrate this function, run CipherTest against the original XML file. The CipherTest application uses a password on the command line:

Listing 13. CipherTest application

```
java CipherTest -e security custorder.xml encrypted-custorder.xml
```

This creates the file

```
encrypted-custorder.xml
```

shown in Listing 14. (You can also download this file.)

Listing 14. encrypted-custorder.xml

```
<?xml version="1.0"?><customer_order>
  <items>
    <item>
      <name>Turnip Twaddler</name>
      <qty>3</qty>
      <price>9.95</price>
    </item>
    <item>
      <name>Snipe Curdler</name>
      <qty>1</qty>
      <price>19.95</price>
    </item>
  </items>
  <customer>
    <name>Doug Tidwell</name>
    <street>1234 Main Street</street>
    <city state="NC">Raleigh</city>
    <zip>11111</zip>
  </customer>
  <EncryptedElement algorithm="DES/CBC/PKCS5Padding" contentType="text/xml"
  encoding="base64" iv="S5Rirg//pNQ=">vJqNpDrQT1vmCVbyGJfIwdIDBYoGXGmutgz6TVGoPuKVG7I
  xNEN50iKw8pmtxFixz5hOChOXgTtPqktQhEHO5+vLOLAFgIioDIRQGHHmHng3CLd+8tvrT8wxPBCRSMUpx4
  d2TGXW2tqSepam0ZxdmwUXwNSAgaR8hmiromD+bh+tDomPv7eFZ4no5ft3JG3t0trLlwVupF/5vaIJimUSm
  uUkkgyG8x9AcS/kXJxHpmM=peqGzIMf+8A=</EncryptedElement>
</customer_order>
```

In the encrypted XML document, the `<credit_payment>` element is replaced by an `<EncryptedElement>` element. Nothing in the document indicates how many elements are encrypted, the names of the encrypted elements, or the structure or sequence of those elements. To preserve the security of the encrypted document, we don't use a `<!DOCTYPE` declaration in our source document. The encrypted document wouldn't follow the DTD, and including any reference to it would indicate the structure of the encrypted elements. If you do encrypt an XML document with a `<!DOCTYPE` declaration, the decryption process won't work. (The error message you'll get won't help you much; it'll say, `java.lang.NullPointerException at CipherTest.main(CipherTest.java:83)`, or something equally useless.)

To restore the original document, use the `-d` (decrypt) option instead of the `-e` (encrypt) option. Be sure the password you use is the same.

Listing 15. CipherTest application (-d option)

```
java CipherTest -d security encrypted-custorder.xml restored-custorder.xml
```

This restores the encrypted file to its original state. Be aware that element-level encryption uses Canonical XML, so the restored file may not have the exact same syntax as the original. Any differences that do appear will not be semantically significant, however.

Other utilities

The XML Security Suite provides two other utilities, an ASN.1-to-XML translator and a variety of DOMHASH tools. The ASN.1-to-XML translator automatically translates between ASN.1 data (such as X.509 certificates and LDAP data) and XML. (See Resources for information about ASN.1.) DOMHASH is an algorithm that generates a unique hash value for a given node in an XML document tree. The DOMHASH tools included with the XML Security Suite calculate hash values for given nodes, and provide a suite of DOMHASH test tools. alphaWorks contains a DOMHASH application called XMLTreeDiff (see Resources); it uses DOMHASH to determine the differences between two DOM trees.

Summary

The XML Security Suite provides a variety of functions that enhance the security of XML documents. All of these technologies can be used with existing Web security. As the exchange of XML documents becomes increasingly important, the technologies in the XML Security Suite will provide vital security functions. Best of all, these technologies are based on open, emerging standards, and they work on any Java-enabled platform. If you're curious to see the kinds of security technologies the world will be using in the very near future, the XML Security Suite is definitely worth a look.

Resources

- Find the specification of the ASN.1 basic notation and the complete set of ASN.1 standards at the International Telecommunication Union Web site.
- For details about canonical XML, see the Canonical XML working draft from W3C.
- For a great book on cryptography, check out Applied Cryptography by Bruce Schneier. Described by *Wired* magazine as "The book the National Security Agency wanted never to be published," it has everything you ever wanted to know about cryptography, and then some.
- Read about and download the Java Cryptography Extension and find a list of cryptographic service providers.
- Download the Java 2 Development Kit, Version 1.2 or higher, from Sun, IBM, and other places on the Web.
- Read about and download the JavaMail API implementation, a set of abstract classes that model a mail system.
- If you'd like to know the ropes of Java security, check out the Security Trail of the Java Tutorial.
- Read about and download Open JCE, a free, open-source implementation of the JCE API.
- Download the IBM XML Parser for Java Version 3.0.1 from alphaWorks.
- Download the XML Security Suite from alphaWorks. The download package includes the code, documentation, and all the sample programs discussed in this article.
- Reference the home page of the XML-signature working group, a joint activity of the W3C and the IETF. The latest draft of the XML signature proposal is available.
- Use XMLTreeDiff from alphaWorks to spot differences between DOM trees, using the DOMHASH algorithm discussed briefly in this paper.
- Find the standard for X.509 certificates as defined by the ITU. You can get your very own X.509 certificate from any of several certificate authorities, including VeriSign.
- Learn about color-coding your listings with Emacs at www.gnu.org.

About the author

Doug Tidwell is a Senior Programmer at IBM. He has well over a sixth of a century of programming experience, and has been working with markup-based applications for more than a decade. His job as a Cyber Evangelist is, in the words of IBM's Simon Phipps, to "compellingly trivialize the complex." Due to a special arrangement with his employer, he is now paid entirely in chocolate truffles. He holds a Masters Degree in Computer Science from Vanderbilt University and a Bachelors Degree in English from the University of Georgia, and can be reached at dtidwell@us.ibm.com.