XML security : Implement security layers, Part 1

Basic plumbing technologies

Level: Introductory

Manish Verma (mverma@secf.com)
Principal Architect, Second Foundation
October 21, 2003

As a format for exchanging information over the Internet, XML's popularity is continuing to grow -- and one of the key issues associated with information exchange is security. No information exchange format is complete without a mechanism for ensuring the security and reliability of the information. This is the first in a series of articles by Manish Verma that will discuss the technologies that play a crucial role in securing XML. This article focuses on the basic plumbing technologies, defining security in an XML context, XML canonicalization, and PKI infrastructure, and providing a step-by-step guide to generating keys. Part 2 will discuss XML encryption and XML signature. This series will give you a practical grasp of the basic technology used for securing XML messages.

Related content:

Enabling XML security
Exploring XML Encryption, Part 1
Exploring XML Encryption, Part 2
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

Also in the XML zone:

Tutorials
Tools and products
Code and components
Articles

For the purposes of this article, the term **security** covers the round-trip protection of XML between a client and a final destination through a variable number of intermediaries. Please note that different parts of a single XML message may have different final destinations. Different parts of the payload are secured such that only the intended set of recipients are able to read them while they remain encrypted to all other intermediaries.

The basic unit of granularity for securing XML is an **element**. Encryption granularity can be further refined by specifying whether the encryption is of type **element** or **content**. Element encryption encrypts the entire element, including attributes, and replaces it with an `EncryptedData` element. Content encryption essentially means that only the child nodes of the element are encrypted and replaced with an `EncryptedData` element.

What does security mean?

The term security encompasses all of the following:

- **Confidentiality** ensures that only the intended recipient reads the intended part of the XML. To ensure confidentiality, it's important to encrypt the XML -- XML Encryption is the standard for doing this.
- **Integrity** ensures that the XML is not altered during transit from the source to its final destination. The XML Signature standard allows the sender to attach a digital signature to the content whose integrity is to be ensured.
- **Authenticity** is the ability to ensure that the XML was actually sent by the person who claims to have sent it. XML signatures sent along with content help ensure the identity of the sender. The recipient of the payload can validate the digital signature using the public key of the sender. If the digital signatures are valid, then the

identity is confirmed; otherwise it isn't.

- **Non-repudiation** is used to ensure that a sender cannot deny sending the XML. An XML signature generated from the private key of the sender and affixed to the XML ensures the non-refutability of the sender.

What is XML canonicalization?
You can create XML documents that appear to be different, but have identical data or identical semantical value. Differences may lie in entity structure, attribute ordering, character encoding, or insignificant whitespace. Because of such physical differences, equivalence testing cannot be done at byte level for arbitrary XML documents. Herein lies the problem: Digital signatures rely on byte-level equivalence, whereas it is possible to have two XML documents that are logically the same, but contain different byte sequences.

Hence, if you digitally sign some XML markup and then try to verify the digital signature after modifying the order of some attributes -- or adding or removing some insignificant whitespace without logically changing the XML -- the verification will fail. To ensure that you get success every time you try to verify the digital signatures of logically equivalent XML -- irrespective of its physical representation -- you must make sure that the XML is in an agreed-upon standard format. That standard is called **canonicalization** and it is a standard mechanism for serializing XML. Canonicalization comes in two forms:

- **Normal canonicalization:** When a sub-part of the XML is serialized, the ancestor element's context and all namespace declarations and attributes in the `xmlns` namespace are included.
- **Exclusive canonicalization:** When a sub-part of the XML is serialized, the ancestor element's context is not included.

So which one should you use, normal or exclusive canonicalization? Consider this: With digital signatures, the digitally signed payload may have to be inserted into a different context after it is removed from its original message. If normal canonicalization is used, the payload will include the context of its original message's ancestor elements, and all namespace declarations and attributes in the `xmlns` namespace. The payload, thus extracted from the original message, may not be inserted faithfully into a different context.

Exclusive canonicalization is required for digital signatures in which the ancestor element's context, attribute, and declaration of the `xmlns` namespace are excluded, thus making the digitally signed payload portable to different contexts.

APIs that do digital signatures manage canonicalization in the background; you do not have to do anything extra for canonicalization while digitally signing the XML.

I have covered enough of canonicalization to help you understand its importance and how it fits into the basic framework of the XML security infrastructure. The next topic focuses on another important component of the XML security infrastructure -- PKI.

PKI basics
Public Key Infrastructure, or PKI, helps make technologies like digital signature and XML encryption generally available to the public at large. At the heart of digital signature and XML encryption are **keys**. Keys are used for digitally signing documents and verifying signatures, and they help with the encryption and decryption processes. PKI is entrusted with managing everything related to the creation, manipulation, and management of these keys.

PKI ensures the following:

- Trusted and efficient management of public and private keys
- Any time you use a public key, you can be sure that the associated private key is indeed owned by the subject whose private key you are using

Keys are a critical element of PKI. This section describes different types of keys, and how to select an algorithm for your key.

Public-private key combination is at the heart of Public Key Infrastructure (PKI), and is based on asymmetric cypher. In this section, I'll explain how to create asymmetric keys and how to use them to encrypt and digitally sign XML data.

Before jumping into public and private keys and the nitty-gritty of generating them, I'd like to show you some other

options for encrypting and digitally signing XML data.

In encryption and digital signatures, everything ultimately boils down to the cryptography algorithms used to generate keys. Here are the criteria on which the key generation algorithms are based:

Algorithm type
Algorithm Type defines whether the cryptography algorithm is symmetric, asymmetric, or a message digest. This is the primary differentiating factor amongst the cryptography algorithms.

- **Symmetric algorithms** are used to generate single keys, also called shared secret keys. The same key is used for both encryption and decryption. The initiating party uses the shared secret to encrypt or digitally sign the payload; the receiving party, in turn, also uses the same shared secret to decrypt or verify the digital signature. Passing the shared secret around can become a potential liability: The shared secret can be exchanged out of band, or the shared secret itself may be encrypted using another key. Both of these shared secret exchange mechanisms are unwieldy. Asymmetric keys solve this problem.

  Though symmetric keys have their disadvantages (primary among them being how to pass them among the trusted parties), the major advantage they offer is speed of encryption and decryption. Encrypting messages using a shared secret is much faster than with asymmetric keys.

- **Asymmetric algorithms** are used to generate keys that exist in pairs. This is the bedrock of PKI. One key of each pair is a public key and the other is a private key. The private key is only known (or *should* only be known) to the entity that owns the key pair. The sender of the message uses the recipient's public key to encrypt the message. The intended recipient then decrypts the message with the corresponding private key and reads it.

  Unlike symmetric keys, public keys can be made available to the public in general without the need to keep them confidential. With the flexibility of allowing public keys to be open comes the issue of mechanisms for making them available to the general public. The solution is XML Key Management System (XKMS) -- a topic I will cover in a future article.

  One disadvantage of asymmetric keys is that they take much longer to encrypt a message than do symmetric keys, and the encryption time is directly proportional to the size of the message.

  Using public and private keys to encrypt only the shared secret instead of the entire message solves the problem of long encryption time. The encrypted shared secret is embedded with the message itself. The shared secret, in turn, is used to encrypt the rest of the message.

- **Message Digests** are secure, one-way hash functions that convert arbitrary length data into fixed-length check sum / hash code. These algorithms are used for digital signature applications. For digitally signing large files, the files are first compressed (the fixed-length hash is calculated) in a secure manner using these algorithms, and then signed with the private key.

Key size
Different algorithms belonging to the algorithm types defined above use varying key sizes. These sizes are defined as sufficient for different types of algorithms -- these key sizes are listed below:

- For symmetric cyphers, 128 bits is sufficient.
- Different asymmetric algorithms provide different levels of security with the same bit size. The most talked about algorithm in asymmetric cyphers is RSA, which uses 2048-bit keys -- equivalent to 128-bit symmetric keys.
- For a message digest, the strength of the algorithm is determined by the size of the hash or check sum that the algorithm generates. An output size of 128 bits is sufficient to ensure a high degree of confidentiality. The encryption using crypto algorithms is ensured by the fact that it is computationally infeasible to generate plaintext that matches the hash value.

Quality of algorithm
Quality of an algorithm refers to the quality of an algorithm's *implementation*. This is gauged by the faithfulness of the implementation to the algorithm's specifications, the compactness of the algorithm, and the speed with which it

encrypts. Various tests can be run to determine which implementation is better based on various criteria. This is the least prominent of the criteria used for choosing an algorithm. The number of algorithms in each category is limited and new ones emerge at very slow rate.

How is the asymmetric algorithm type used in PKI?
Key pairs work in the following manner: Assume you have two entities -- Entity A and Entity B. Entity A has to send an encrypted message to Entity B. Entity A obtains Entity B's public key and uses it to encrypt the message. The message thus encrypted can only be decrypted using the corresponding private key, which is held by Entity B. This ensures that anyone can encrypt the message using the receiver's public key but only the receiver (the one with the corresponding private key) can decrypt it.

Similarly, with a digital signature the sender of the message signs the message using his or her own private key. The receiver can verify the authenticity of the message by using the public key of the sender to verify the digital signature; if the known public key of the sender is able to verify the digital signature, the message is considered to have come from its claimed sender.

So, to summarize the use of PKI in XML encryption and XML digital signature...

In case of XML encryption:

1. The sender of the message uses the public key of the receiver to encrypt the message to be sent.
2. The receiver of the message uses the corresponding private key, which it owns, to decrypt the message. No one other than the one that owns the corresponding private key can decrypt the message, thus ensuring its confidentiality.

In case of XML digital signatures:

1. The sender uses its own private key to digitally sign the message that is to be sent.
2. The receiver uses the sender's corresponding public key to verify the digital signature.

Generating a public-private key pair
This section describes the process of generating the public-private key pair, which is the basis for encryption and digital signature.

Step 1. Basic setup
You need to install and register a cryptographic provider for the Java Cryptographic Extension (JCE). The JCE provider must support the RSA algorithm type for generating public and private keys. This requirement stems from the fact that the XML encryption implementation used in the code samples accompanying this article (see Resources) supports only the RSA1_5 algorithm type for public-private key combination.

In JDK 1.4, the SunJCE provider comes pre-configured. The SunJCE provider is not suitable for generating asymmetric keys, as it does not contain the RSA algorithm type implementation. The accompanying sample code uses an open source JCE provider, ISNetworks, which includes support for RSA.

First, make the JCE provider available. You can do this by putting the provider .jar files in the CLASSPATH.

Second, configure the JCE provider. Edit the `java.security` file under the `JAVA_HOME\lib\security` directory, and add the following line:

```
security.provider.n=com.isnetworks.provider.jce.ISNetworksProvider
```

The Java Runtime Environment (JRE) selects the JCE provider to load into the JVM. Selection of the JCE provider is made based on preference order. Substitute `"n"` with the preference order in which you want it to be picked by the JRE -- the lower the number, the higher the order.

Registration of a provider can also be done dynamically through code. See Dynamically

Dynamically registering a JCE provider
To register the JCE Provider dynamically, call either `addProvider` or `insertProviderAt` in the `java.security.Security` class. The `addProvider` method adds the new provider to the end of the list of installed providers, whereas `insertProviderAt` adds the provider at the specified position. If security manager is installed, its `checkSecurityAccess` method is called to see if it is allowed to add a new provider dynamically.

registering a JCE provider.

Permissions must be granted for situations where applets or applications that use JCE are run with the security manager installed. Each provider specifies which permissions must be granted. Permissions can be granted in the policy file.

You can steer clear of this permission-granting business by installing the provider .jar file in the `"ext"` directory as a bundled extension.

Step 2. Key pair generation
The code snippet in Listing 1 demonstrates how a key pair is generated.

Listing 1. Key pair generation

```
KeyPairGenerator pairgen = null;
try {
   pairgen = KeyPairGenerator.getInstance("RSA", "ISNetworks");
} catch (NoSuchAlgorithmException e) {
   System.out.println("There is no such algorithm for generating the keys");
   e.printStackTrace();
} catch (NoSuchProviderException e) {
   System.out.println("Not a valid Provider");
   e.printStackTrace();
}
SecureRandom random = new SecureRandom();
int KEYSIZE = 1024;
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
```

First, get a `KeyPairGenerator` object that implements the RSA algorithm.

Provide the `KeyPairGenerator` object with a key size and a random number, and then call the `generateKeyPair` method to get the public-private key pair. Hold on tight to these keys -- you will use them extensively for encryption and digital signatures.

Generating a shared secret
You will also need a shared secret. As discussed in the "Algorithm type" section, a shared secret is used to encrypt the content of the XML, and the public-private key combination is used to encrypt the secret key, which is embedded in the content itself.

Such juggling is required in order to have good encryption response time without the need to send the shared secret out of band. Good encryption response time is ensured by using the shared secret to encrypt the XML content, whereas the key pair is used for encrypting the shared secret itself.

Encryption using a shared secret is fast. A shared secret is used to encrypt the bulk of the data -- meaning the actual content of the XML. Encryption using a public-private key is slow and is directly proportional to the size of the data being encrypted. Since the shared secret key is usually smaller than the content, a public key is used to encrypt only the shared secret key, which itself is embedded in the content.

The implementation of XML encryption used in the sample code accompanying this article only recognizes the Triple-DES algorithm for content encryption. Listing 2 demonstrates how a secret key is generated using Triple-DES.

Listing 2. Secret key generation

```
Key ky = null;
KeyGenerator kg = null;
try {
   kg = KeyGenerator.getInstance("DESede", "ISNetworks");
} catch (NoSuchAlgorithmException e) {
   System.out.println(
      "There is no such algorithm for generating the shared secret");
   e.printStackTrace();
} catch (NoSuchProviderException e) {
   System.out.println("Not a valid Provider");
   e.printStackTrace();
}
ky = kg.generateKey();
```

First, get a `KeyGenerator` object that implements the Triple-DES algorithm.

On the `KeyGenerator` object, call the `generateKey` method to get the shared key. This shared key is used to encrypt the content of the XML. The shared secret key itself is encrypted using the public key you created in the previous step, and is embedded in the XML content.

Conclusion

XML plays a prominent role in data interchange over public networks. Consequently, great concern is generated over XML security standards as well as how these standards are adopted, interpreted, and implemented by different technology platforms. In this article, I have objectively defined security and its basic technologies, namely XML canonicalization and the PKI infrastructure, which play an important role in making XML interchange secure. Building further on this foundation, the next article in this series will discuss XML encryption and XML signature, and will provide a step-by-step guide to using them.

Resources

- Download the source code used in this article.
- Read the Exclusive XML Canonicalization specification.
- Find out more about Canonical XML.
- Learn more about PKI at the National Institute of Standards and Technology's Computer Security Resource Center. Resources include the PKI technical specifications.
- Take a look at the XML Key Management Specification (XKMS).
- See the Sun Java Cryptography Extension (JCE) 1.2.2 Installation Instructions.
- 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. Part 2 examines the usage model of XML Encryption with the help of a use case scenario.

- 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 lifecycle 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.