CHAPTER 4

Introduction to XML Digital Signatures

In June 2000, the U.S. Congress approved the Electronic Signatures in Global and National Commerce Act (E-SIGN Act). This broad legislation gave electronically generated signatures a new legitimacy by preventing contest of a contract or signature based solely on the fact that it is in electronic form. In other words, an electronic transaction cannot be denied authenticity because of its electronic nature alone. The E-SIGN Act is expected to facilitate business-to-business commerce by mitigating the need for logistically expensive paper signatures. The portability of XML as a data format makes it ideal for business-to-business transactions that require a robust mechanism for both data integrity and authentication. In response to these business requirements, as well as requirements from the legal industry, the "XML-DSig" Charter was established. The goals of this joint IETF/W3C Charter include the creation of a highly extensible signature syntax that is not only tightly integrated with existing XML technologies, but also provides a consistent means to handle composite documents from diverse application domains.

XML Signature Basics

An XML Signature is a rich, complex, cryptographic object. XML Signatures rely on a large number of disparate XML and cryptographic technologies. The culmination of these technologies results in a signature syntax that can be quite abstract and daunting, even to those well versed in both security technology and XML syntax and tools. The XML Signature syntax is designed with a high degree of extensibility and flexibility; these notions add to the abstract nature of the syntax itself, but provide a signature syntax that is conducive to almost any signature operation.

The XML-Signature Syntax and Processing W3C Recommendation defines the XML Signature syntax and its associated processing rules. This recommendation, like most of the additional XML-related recommendations, can be found at the World Wide Web Consortium Web site, **http://www.w3.org**. The XML Signature Recommendation will likely change in subtle ways as XML Signatures become more pervasive and gain implementation experience. However, we are not concerned with the nooks and crannies of the specification, but instead with the basic reason for its existence, examples, and the fundamental properties that define an XML Signature. One might question why we need such a rich signature syntax that differs markedly from our existing signature infrastructure. If we compared an existing messaging syntax, such as PKCS#7[1], to XML Signatures, we would see drastic differences in the intent and implementation of the syntax.

We will first attempt to describe XML Signatures from an abstract point of view. This will establish broad notions and definitions that can be built upon in a systematic way towards practical examples. Readers with little experience with digital signatures can refer to the primer in Chapter 2 or to a similar section in one of the references listed in Chapter 2. Our first definition is shown below.

Definition 4.1

XML Signature The specific XML syntax used to represent a digital signature over any arbitrary digital content.

[1]For more information on PKCS#7 see the primer in Chapter 2.

At first glance this definition seems remedial to anyone who has created a digital signature even once. The only marked difference is that the signature is defined to *be* XML. This point is especially important and provides insight into the purpose of the XML Signature. Currently, a digital signature (either RSA or DSA) over arbitrary digital content results in raw binary output of a relatively fixed size. The output of an RSA signature is related to the key-size used; the output of a DSA signature is related to the representation of the encoding used. Moreover, to verify a raw digital signature, the signer must provide additional information to the verifier, including the type of algorithm used as well as information about the recipient and verification key. Once these parameters are configured, it is often difficult to change them or have a mechanism in place that is robust in different scenarios. Before the advent of XML and its related technologies, several solutions emerged to aid in this type of extensible processing—ASN.1 and BER encoding, coupled with a hierarchical set of Algorithm Object Identifiers are currently used to facilitate this type of flexible processing. Readers unfamiliar with ASN.1 and BER/DER encoding should refer to the primer in Chapter 2. The ASN.1 definition of an Algorithm Identifier that is used to encode algorithm specific information is shown in Listing 4-1.

The actual value of `OBJECT IDENTIFIER` is defined by various standards bodies and is intended to be a unique bit-string that is encoded in a raw binary format that conforms to BER/DER. For example, an `AlgorithmIdentifier` that designates an RSA Signature with the SHA-1 hash function might be encoded as in Listing 4-2.

Listing 4-1

ASN.1 definition of AlgorithmIdentifier

```
AlgorithmIdentifier :: SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL }
```

Listing 4-2

AlgorithmIdentifier for RSA with SHA-1

```
30 0D 06 09 2A 86 48 86 F7 0D 01 01 05 05 00
```

This bit-string is intended to merely identify a type of signature algorithm. This type of compact binary representation is a rather tedious and complex way of accomplishing the simple task of informing someone what type of signature algorithm is to be used during application processing. In contrast to the algorithm identifier used above, an XML Signature would use the following identifier to denote the same RSA Signature with the SHA-1 hash function:

```
http://www.w3.org/2000/09/xmldsig#rsa-sha1
```

Because XML is a text-based format, this type of algorithm identifier lends itself to the type of text-based processing common for XML documents. Whereas the previous algorithm identifier is a more compact (and therefore smaller) representation, the pervasiveness of XML parsers makes such a text-identifier more viable and much simpler. XML Signatures have tried to remove themselves from this type of compact binary representation when possible, although the binary-encoded identifiers are still used in the creation of the signature for backwards compatibility.

An XML Signature is itself an XML document; it carries with it all of the properties of a well-formed XML document. All of the information needed to process the signature can be embedded within the signature representation itself, including the verification information. Furthermore, all XML Signatures can undergo minimal processing even when applications do not have XML signing or verification capabilities. The elements, attributes, and text (if present) can all be processed as "normal" XML (except the actual signature and digest values). The added complexity of an XML Signature lies not in the signature process or cryptographic operations used, but in the additional processing features demanded by XML documents. XML Signatures are more closely related to a messaging syntax such as PKCS#7, rather than raw binary digital signatures. An XML Signature specifies the structure of the signature in relation to the source documents; it also has the capability to encompass a cryptographic key or X.509 certificate for signature verification. We can define the high-level procedure for generating an XML Signature as follows:

Definition 4.2

An **XML Signature** is generated from a hash over the canonical form of a signature manifest.

This "meta-algorithm" gives us a flavor for what is involved in signature generation. It does not encapsulate the specifics of signature generation, which will be covered later. Perhaps the most curious part of the definition is the use of the term "manifest." This term is often used in conjunction with XML to refer to a master list of sorts, but it has its origins in the description of cargo on a sailing vessel. It may be useful to think of "manifest" as the collection of resources that are signed[2]; these may be local to the signature itself or Web resources that are accessible via a Uniform Resource Identifier (URI). One can think of the XML Signature as a sailing vessel that carries with it a cargo list (manifest) that must brave unknown networks to arrive at its destination unscathed.

The signature manifest, or list of resources to be signed, is expressed using XML. XML allows for syntactic variations over logically equivalent documents. This means that it is possible for two XML documents to differ by one byte but still be semantically equivalent. A simple example is the addition of white space inside XML start and end tags. This liberal format causes problems for hash functions, which are sensitive to single byte differences. Readers unfamiliar with cryptographic hash functions may refer to the primer in Chapter 2. To alleviate this problem, a canonicalization algorithm is applied to the signature manifest to remove syntactic differences from semantically equivalent XML documents. This algorithm ensures that the same bytes are hashed and subsequently signed. For example, consider the following arbitrary empty XML element.

```
<Manifest Id="ReferenceList"/>
```

If one were to apply a SHA-1 hash to the above element, the hash output would be the following octet-string:

```
61 16 EC F9 32 60 A1 20 65 8B DD 6C DB 96 23 3B E5 1D 33 C2
```

Consider what would happen if we were to modify the element by adding some spaces:

```
<Manifest    Id="ReferenceList"/>
```

The SHA-1 hash would now produce the following completely different octet-string:

```
78 54 7D E6 2C 3C 4E 39 25 00 63 F7 61 08 A2 33 DC 0D 29 92
```

[2]The manifest actually contains a list of digests of the resources.

The hash values do not match, but the semantics of the empty XML element in each case are exactly the same. This subtle complication with how XML is processed is clear evidence for the use of a robust normalization algorithm within XML Signature processing.

The last item of interest is the use of the hash function itself. Hash functions are used so pervasively in conjunction with digital signatures that it often seems they are a necessary, defining component. It is possible to generate a digital signature using only a signing key and acceptable public-key algorithm. Hash functions are convenient and when used with digital signatures, reduce the size of what is being signed and effectively speed up the signing operation. A hash function is used in two different scenarios when XML Signatures are generated. Each resource included in the manifest is hashed, and this list or collection of resources is then hashed a second time during the signing operation. One might ask why a manifest or list is required at all. Consider what would happen if the number of resources to be included in the signature grows. Applying a signature algorithm to each resource would be time-consuming and would hinder the creation and verification of an XML Signature. The manifest or list is a means to side-step this problem. Instead of signing each resource, we hash each resource, which is much faster; then, we include the hash value and resource location in the manifest.

At this point, we have briefly discussed the definition of an XML Signature along with an extremely high-level signature generation procedure. The definitions given thus far are terse but precise and should give the reader a strong foundation for understanding XML Signatures. The next topic concerns the semantics of an XML Signature. Presenting a clear idea of what it *means* for something to be signed by an XML Signature will help the reader understand the limits of XML Signatures from a conceptual standpoint. (See Definition 4.3.)

Definition 4.3

An **XML Signature** associates the contents of a signature manifest with a key via a strong one-way transformation.

These semantics are very precise—an XML Signature defines a one-way signature operation based on a signing key. It is important to note that the term "one-way" is used informally in this context. Most people believe that there is no feasible way to reverse the signature transformation. The most common signature transformations that are used in conjunction with XML Signatures are RSA Signatures, DSA Signatures, and symmetric key message authentication codes. The term "one-way" refers to the cryptographic properties of these or any other signature algorithms that may be used. XML Signatures have the ability to utilize symmetric key message authentication codes (HMACs), which can also be used as a strong signature transformation. For more information on HMAC, refer to the primer in Chapter 2.

Digital signatures have wide applications for associating a document or data with an actual human, just as a normal paper signature does. Based on this notion, an XML Signature is widely believed to provide this sort of trust semantic. While this is extremely useful and practical, an XML Signature by itself does not associate a signing key with an individual. An XML Signature instead provides the *means* to establish this sort of association. This is accomplished by the convenient method of packaging the verification key within the XML Signature via an optional element. In a sense, the XML Signature may present the verification material (either raw public key or certificate that contains the public key) to the application, leaving the issue of trust to the application. Well-defined mechanisms for validating the identity of a signer based on public key information already exist, such as certificate path validation. This decoupling of entity verification from the actual signature gives the application more flexibility in deciding its own custom trust mechanisms. For example, an application might wish to check if a particular entity has the authority to sign a document or portion of a document. Not all private keys are authorized as signing keys. A trusted authority might have restrictions on private key usage for a particular individual, or an individual's key pair might have been revoked altogether. These additional trust semantics lie outside of the scope of an XML Signature. A few legal and technical organizations have pushed for stronger integration of additional trust semantics, but at present they are left out of the scope of XML Signatures. The XML Signature leaves the problem of establishing *trust* to another core XML Security technology called XKMS (XML Key Management Specification).

XML Signatures and Raw Digital Signatures

We can now supplement the XML Signature basics with some examples. We will first briefly examine the structure of an XML Signature in terms of its defining tags. At the outset we will hide much of the complexity of an XML Signature and attempt to relate it to raw digital signatures over binary data. As we proceed through the examples, we will see how the asymmetry of raw digital signatures makes them cumbersome, and how the XML Signature is a superior design for many cases. Before we begin, a definition of "raw digital signature" is in order. The referent here is the simple case of an RSA private key operation applied to a hash of the original document. DSA could be used for this example as well; the choice is rather arbitrary. This type of "raw" signature assumes a basic padding scheme (either PKCS#1 or some appropriate padding scheme) that does little more than transform the hashed data into a valid input for the RSA algorithm. The term "raw" does not necessitate the absence of padding (as in raw RSA encryption), but simply implies that the signature has no packaging mechanism applied to it that affords it additional semantics. Readers unfamiliar with PKCS#1 or padding schemes in general should refer to the primer in Chapter 2.

Listing 4-3 gives the outer structure or skeleton of an XML Signature. The elements are XML tags, and their structure defines the parent-child relationships of each element. The reader may also notice the use of cardinality operators. These operators denote the number of occurrences of each element within the parent `<Signature>` element. The definition of each cardinality operator is given in Table 4-1. The *absence* of a cardinality operator on an element or attribute denotes that exactly one occurrence of that element must appear.

Table 4-1

Cardinality Operators

Operator	**Description**
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrence

Listing 4-3

The XML Signature structure

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod>
      <SignatureMethod>
       (<Reference (URI=)?>
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
       </Reference>)+
     </SignedInfo>
  <SignatureValue>
 (<KeyInfo>)?
 (<Object>)*
</Signature>
```

At first glance the structure shown in Listing 4-3 may seem overly complex or even a bit daunting. Many readers are probably questioning whether the surface complexity is really necessary. We can apply an intellectual knife to simplify the structure to a vacuous case shown in Listing 4-4. This simplification hides the added features of the XML Signature and allows us to think of things in terms of a "raw" digital signature.

We are intentionally leaving out the cardinality operators in this instance. Here we assume that one and only one element of each type is allowed. Even this vacuous example may fail to make much sense without further context. Definition 4.1 refers to the idea of a signature *manifest*. Recall that the manifest is a list or collection of resources that are to be included in the signature. These resources can be remote Web resources, local resources, or even same document references. This list or manifest is the contents of the `<SignedInfo>` element as shown in Listing 4-4. For now, we will ignore the complexity of this element and assume that it somehow points to everything that we wish to sign and includes all the information necessary to produce the actual signature. This being the case, Listing 4-4 begins to make more sense. The parent `<Signature>`

Listing 4-4

The vacuous XML Signature

```
<Signature>
  <SignedInfo>
  </SignedInfo>
  (SignatureValue)
</Signature>
```

element contains two entities: an original document, or collection of original documents (`<SignedInfo>`), and an actual signature value (`SignatureValue`). At this point, the `<Signature>` element serves to group the two items for easy transmission to a third party. We are also intentionally omitting references to terms like *enveloped*, *enveloping,* or *detached* at this point. These terms have precise definitions when used in conjunction with XML Signatures and should not be confused with their use with other standards (such as PKCS#7 or S/MIME). An example instance of the signature syntax shown in Listing 4-4 is given in Listing 4-5.

Some readers may notice the nature of the data inside the `<SignatureValue>` tag. This is the Base-64 encoded signature value. Base-64 encoding is used pervasively in XML-related applications. Base-64 encoding is a convenient, well-defined encoding mechanism for creating a unique, printable representation of arbitrary binary data. Because of its text representation, Base-64 encoding is a natural solution for use in conjunction with XML. Readers unfamiliar with Base-64 encoding should refer to the primer in Chapter 2.

In Listing 4-5 we have again hidden the complexity of the `<SignedInfo>` element. We ignore the details of this element and just assume that it contains a reference to the original document that we are signing. The Base-64 encoded data shown in Listing 4-5 is simply the result of applying our chosen signature algorithm and hash function to the contents of `<SignedInfo>`.

We now have enough background information to begin comparing our simplified XML Signature to a "raw" digital signature. Consider the problem of signing a piece of text data residing on some local storage device

Listing 4-5

Instance of the vacuous XML Signature

```
<Signature>
  <SignedInfo>
  </SignedInfo>
   <SignatureValue>
   MI6rfG4XwOjzIpeDDDZB2B2G8FcBYbeYvxMrO/
   Ta7nm5ShQ26KxK71Ch+4wHCMyxEkBxx2HP0/7J
   tPiZTwCVEZ1F5J4vHtFTCVB8X5eEP8nmi3ksdT
   Q+zMtKjQII9AbCNxdA6ZtXfaOV4euO7UtRHyK1
   7Exbd9PNFxnq46b/f8I=
  </SignatureValue>
</Signature>
```

using a "raw" signature. Listing 4-6 shows the piece of data we are going to sign. We can assume that it is an electronic check in a simple, fictitious format.

Let us assume we already have a private signing key and that we are going to perform an RSA signature using the SHA-1 hash function. The output from the signature operation using a 512 bit key might look something like Listing 4-7. An RSA signature operation is just an RSA private key operation applied to a hash of the original document.

This signature value is not very interoperable and does not carry with it much context. The most we could discern is that it is 64 bytes of data. To solve this problem, we need to send along the binary algorithm identifier. This is the same binary data that is shown in Listing 4-2. In Listing 4-8 we will show the same algorithm identifier as interpreted by an ASN.1 parser. The text shown is generated from an ASN.1 interpreter; the actual value that needs to be sent must still be encoded in binary.

This `AlgorithmIdentifier` will give a recipient some information about how the signature was generated so it can be properly verified.

Listing 4-6

Example electronic check

```
check.txt
I authorize a payment of $2 from my checking account to the paperboy.
L. Meyer
```

Listing 4-7

Binary RSA digital signature (512 bit key)

```
92 F4 10 8C BD 29 98 C8 54 59 9D CD 62 F0 18 BE
75 69 4D 64 1A ED E7 7E 6D BD E9 7C 58 EA DE 3C
5B 4F 03 4B A0 F1 6A 1F DC 30 B4 8E 91 82 00 29
72 B6 86 0A B6 CA 3C 80 18 32 55 46 69 57 6D A8
```

Listing 4-8

ASN.1 interpretation of the RSA with SHA-1 algorithm identifier

```
SEQUENCE {
  OBJECT IDENTIFIER sha1withRSAEncryption (1 2 840 113549 1 1 5)
  NULL }
```

Finally, we also need to send the original document. The original document, which is in a text format, is required to determine if the signature verifies. At this point we have three pieces of data that need to be sent to a third party: the signature value, algorithm identifier, and original message. The physical representation of the three entities differs. Two pieces of the data are in binary format and the third is encoded in text. We can solve this problem by applying Base-64 encoding to the two binary pieces, which results in three pieces of data in a printable format. We now have homogeneous data, but we still have no context or header information that gives us clear semantics for the three pieces. A crude attempt at packaging this raw type of signature appears in Listing 4-9.

In our contrived format above, the first line contains the algorithm identifier, the next two lines contain the signature value, and the remainder is the original document. The problem with this type of crude format is that there is no context or structure for the different pieces of the signature. The recipient of such a signature would have to know about our proprietary format in advance. This may be acceptable if we are dealing with a single recipient, but as the number of recipients grows, this type of format quickly becomes unworkable.

This is where the power of XML as a portable data format begins to show some advantages. In Listing 4-10 we will expand on our simple XML Signature syntax and show how two new elements, `<SignatureMethod>` and `<Reference>`, are used to identify the signature algorithm and actual file pointed to. Note that Listing 4-10 still omits additional syntax and features.

We have added the new elements (shown in bold) as children of `<SignedInfo>`. Notice that the `<Reference>` element has an attribute called URI that identifies the file that we are signing, as well as two addi-

Listing 4-9

Packaging a raw digital signature

```
MA0GCSqGSIb3DQEBBQUA
kvQQjL0pmMhUWZ3NYvAYvnVpTWQa7ed+bb3pfFjq3jxbTwNLoPFqH9wwtI6Rgg
ApcraGCrbKPIAYMlVGaVdtqA==
I authorize a payment of $2 from my checking account to the paperboy.
L. Meyer
```

Listing 4-10

Expanded XML Signature syntax

```
<Signature>
  <SignedInfo>
    <SignatureMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="file:///C:\check.txt">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/
       xmldsig#sha1"/>
      <DigestValue>aZh8Eo2alIke1D5NNW+q3iHrRPQ=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
   MI6rfG4XwPFASDFfgAFsdAdASfasdFBVWxMrO/
   Ta7nm5SfQ26KxK71Ch+4wHCMyxEkBxx2HP0/7J
   tPiZTHNYTEWFtgWRvfwrfbvRFWvRWVnmi3ksdT
   Q+zMtKjQAsdfJHyheAWErHtw3qweavfwtRHyK1
   9ExbdFWQAEDafsf/f8I=
  </SignatureValue>
</Signature>
```

tional child elements that identify a digest value and a digest algorithm. The signature operation used in XML Signatures never signs resources directly, only hashes of resources. This not only speeds up single signature operations, but also provides an easy way to sign multiple resources. Multiple `<Reference>` elements can be added to the `<SignedInfo>` element. Only one is shown here. Lastly, the included `<SignatureMethod>` element is an empty element that contains only a single attribute. The attribute is called `Algorithm` and is a URI that describes the type of signature operation used (in this case, RSA with SHA-1).

Consider the differences between the XML Signature shown in Listing 4-10 and the "raw" digital signature shown in Listing 4-9. We might describe Listing 4-10 with words like *structured, context-specific,* or *extensible*, whereas Listing 4-9 might be described as *fragmented, context-free,* or *rigid*. These adjectives encompass the nature of XML data in just about any context, and digital signatures are no different. In fact, we have barely touched on the different facets and features and syntax of XML Signatures. What is shown in Listing 4-10 is a degenerate case that will be used only in simple situations, if at all. In the next section we will examine the additional features of XML Signatures and see how they can be adapted to almost any digital signing situation.

XML Signature Types

Before we concentrate our efforts on the syntax of XML Signatures, it may be useful to examine the three basic types of signatures in terms of their parent-child relationships. Different applications require signature delivery in certain ways, with preferred *signature types*. Certain applications require that an XML Signature be modeled as closely as possible to a real, handwritten contract that includes embedded signatures in certain parts within the original document. Other applications may process the original data separate from the signature and may require that the original data be removed from the signature itself. The original document tightly coupled with the parent `<Signature>` element (the original document is parent or child to `<Signature>`) is an *enveloped* or *enveloping* signature. The original document kept apart from the `<Signature>` element (the original document has no parent-child relationship to `<Signature>`) is a *detached* signature. Intricate pictures of these types of signatures can be drawn, but a simple way of looking at them is in terms of their XML structure and parent-child relationships. Listing 4-11 shows the XML structure of these three types. An enveloped signature must be child to the data being signed. An enveloping signature must be parent to the data being signed. A detached signature is neither parent nor child to the data being signed.

Interestingly, a single `<Signature>` instance may be described as a combination of the above types. It is possible for a `<Signature>` to have multiple `<Reference>` elements, each of which may point to data local to the `<Signature>` block and kept remotely. Listing 4-12 shows an example of a `<Signature>` block that is both *enveloping* and *detached*.

The two `<Reference>` elements shown in bold point to source data that is located in different places. Because one piece of data (the `<original_document>` element, also shown in bold and referenced via

Listing 4-11

Enveloped, enveloping, and detached XML signatures

```
<!-- Enveloped Signature -->
<original_document>
  <Signature> ... </Signature>
</original_document>
<!-- Enveloping Signature -->
<Signature>
  <original_document>
  </original_document>
</Signature>

<!-- Detached Signature -->
<Signature> ... </Signature>
```

04_CH04/DournaeeX 1/24/02 10:30 AM Page 121

Listing 4-12 Enveloping and detached XML Signature

```
<Signature>
  <SignedInfo>
    <SignatureMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="http://www.myserver.com/importantFile.xml">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>aZh8Eo2alIke1D5NNW+q3iHrRPQ=</DigestValue>
    </Reference>
    <Reference URI="#ImportantMessage">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>qGh8Eo2alJke1D7NNW+z3iHhRPF=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
   MI6rfG4dwPFASDFfgAFsdAdASfasdFBVWxMrO/
   Ta7nFCSDAhnTRhy45vJSDcvadrtrEQW2HP0/7J
   tPQTBFfwGVwfqewrfVfewgrtgvfbwdj7jujYdT
   Q+zMtKRQGRE1grewrfht32rwhnbtygrwtRHyK1
   3EFfdasreEDafsfgf8I=
  </SignatureValue>
  <Object>
    <original_document>
      <very_important_element id="ImportantMessage">
        Milk Chocolate is better than Dark Chocolate!
      </very_important_element>
    </original_document>
  </Object>
</Signature>
```

its attribute) is inside the document, and one piece of data is external (the reference to `importantFile.xml`) this signature has the dual properties of being both *enveloping* and *detached*.

XML Signature Syntax and Examples

Listing 4-3 gives the core structure of an XML Signature. XML Schema definitions and Document Type Definitions (DTDs) give the formal syntax and grammar of all child elements of `<Signature>` as specified in the XML Signature Recommendation. Rather than repeat the formal syntax given in the recommendation, we will give informal descriptions that attempt to document the nature and intent of each element. We have

04_CH04/DournaeeX 1/24/02 10:30 AM Page 122

already seen examples of how some of the elements are used in the creation of a basic XML Signature. Here we will expand our examples and discussion to cover all of the components of the XML Signature syntax.

XML Signature Syntax

The following section lists and describes the elements that comprise the XML Signature Syntax.

The <Signature> Element

The parent element of an XML Signature is, of course, the `<Signature>` element. This element identifies a complete XML Signature within a given context. This parent element can contain a sequence of children as follows: `<SignedInfo>`, `<SignatureValue>`, `<KeyInfo>`, and `<Object>`. Note that the last two elements are optional. Two things are important about the `<Signature>` element. First, an optional `Id` attribute can be added as an identifier. This is useful in the case of multiple `<Signature>` instances within a single file or context. Secondly, the `<Signature>` element must be *laxly-schema valid* to its constraining schema definition. This type of validity is related to a best-effort attempt at schema validation.

The <SignedInfo> Element

The next element in the sequence is the `<SignedInfo>` element. This element is the most complex element (it has the most children) and ultimately contains a reference to every data object that is to be included in the signature. As the name implies, `<SignedInfo>` encompasses all the information that is actually signed; that is, the *signed information.* The contents of `<SignedInfo>` includes a sequence of the following elements: `<CanonicalizationMethod>`, `<SignatureMethod>`, and one or more `<Reference>` elements. The `<CanonicalizationMethod>` and `<SignatureMethod>` elements describe the type of canonicalization algorithm and signature algorithm used in the generation of the `<SignatureValue>`. These two elements simply contain identifiers; they do not actually point to any data used in signature generation. These identifiers must be included as part of the `<SignedInfo>` to prevent against substitution attacks. For example, if the `<SignatureMethod>` element were explicitly defined outside the `<SignedInfo>` element, an adversary could modify the signature method identifier and wreak havoc with someone trying to properly validate the signature.

Another interesting and important element is the `<Reference>` element. References define the actual data that we are signing. Most of the added features of XML Signatures show up in the definition and usage of `<Reference>` elements. Because of their importance, they are treated separately in Chapter 5. For now it is enough to know that they define a data stream that will eventually be hashed and possibly transformed. The actual data stream is referenced by a URI. URIs are a universal mechanism for referencing data locally or remotely. It is possible to omit the URI identifier on, at most, one `<Reference>` element if the application can determine the source data from another context. More discussion on URIs can be found in Chapter 3.

Discussion of hierarchy can be confusing; a visual example often helps. Listing 4-13 shows an example of the structure that we have been piecing together so far.

Listing 4-13 focuses on three elements: `<CanonicalizationMethod>`, `<SignatureMethod>`, and `<Reference>`. The `<Canonicalization Method>` points to the canonicalization method required by the XML Signature Recommendation. This specific method is called *Canonical XML Without Comments*. A more thorough discussion of Canonical XML is given in Chapter 5. The URI used here (**http://www.w3.org/TR/2001/REC-xml-c14n-20010315**) is merely an identifier, not a source of data or an algorithm source. This can be quite confusing at first; URIs are used both as identifiers and as data streams. The two URIs specified in `<CanonicalizationMethod>` and `<SignatureMethod>` are used as

Listing 4-13

The `<SignedInfo>` element and its children

```
<Signature>
  <SignedInfo>
     <CanonicalizationMethod Algorithm=
      "http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
      <SignatureMethod Algorithm=
      "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <Reference URI="http://www.rsasecurity.com">
        <DigestMethod Algorithm=
        "http://www.w3.org/2000/07/xmldsig#sha1"/>
        <DigestValue>aZh8Eo2alIke1D5NNW+q3iHrRPQ=</DigestValue>
      </Reference>
   </SignedInfo>
   ...
</Signature>
```

identifiers, whereas the URI specified in the `<Reference>` element is an actual data stream that is digested and then subsequently signed.

In addition to a public-key signature scheme, the XML Signature recommendation requires that HMAC be implemented as an option for the `<SignatureMethod>`. An HMAC is an authentication code based on a shared secret key. For cases where a shared secret exists between two parties, an HMAC might be a better choice for signature authentication. The computation of an HMAC is considerably faster than an expensive RSA or DSA signing operation. Listing 4-14 shows an example of a `<SignedInfo>` that utilizes HMAC as its `<SignatureMethod>`. In addition to the identifier that describes the HMAC algorithm used (in this case the referent is HMAC-SHA1), the `<SignatureMethod>` element specifies an additional child element called `<HMACOutputLength>`. This element allows for modification of the HMAC output. Additional cryptographic tradeoffs are also possible by truncating the output of the HMAC. More information can be found in RFC 2104, or the HMAC primer, in Chapter 2.

Notice in Listing 4-14 the use of a local reference for the source file to sign. While it is possible to sign a file that is kept locally, this may cause problems when the recipient tries to verify the signature. When signature verification occurs, the `<Reference>` elements determine where the data to verify comes from. A remote recipient is unlikely to have access to the same file resource kept on a local machine. With XML Signatures, it is possible to package the original data inside the `<Signature>` element with an *enveloping* signature (not shown in Listing 4-14; the signature shown is a *detached* signature) to avoid this problem.

Listing 4-14

Using HMAC for the `<Signature Method>`

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod Algorithm=
    "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
    <SignatureMethod Algorithm=
    "http://www.w3.org/2000/09/xmldsig#hmacsha1">
      <HMACOutputLength>80</HMACOutputLength>
    </SignatureMethod>
    <Reference URI="file:///C:\signme.xml">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/
       xmldsig#sha1"/>
      <DigestValue>lsn6Q7VlZGRt1norERfoIelQHJA=</DigestValue>
    </Reference>
  </SignedInfo>
...
</Signature>
```

Finally, much like its parent element `<Signature>`, the `<Signed-Info>` element also has a provision for an `Id` attribute. This attribute can be used as an identifier and may be referenced from other `<Signature>` elements. Following the `<SignedInfo>` element is the `<Signature-Value>` element. We have already seen examples of this element. It is little more than a container to hold an encoded binary signature value. The encoding is Base-64 ASCII encoding as specified in RFC 2045.

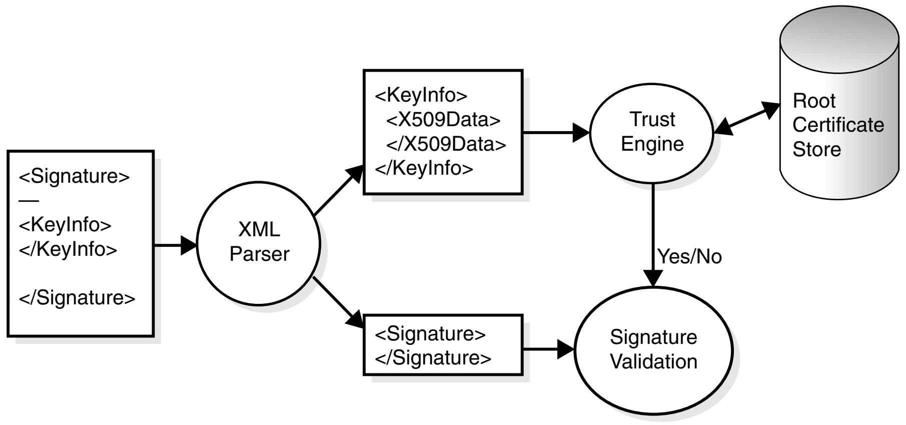
The <KeyInfo> Element

Following `<SignatureValue>` is the optional `<KeyInfo>` element. The `<KeyInfo>` element is a powerful element that allows for the integration of trust semantics within an application that utilizes XML Signatures. Simply put, a `<KeyInfo>` element contains specific information used to verify an XML Signature. The information can be explicit, such as a raw public key or an X.509 certificate, or the information can be indirect and specify a remote public-key information source. `<KeyInfo>` is a powerful element because it allows a recipient to verify the signature without having to explicitly hunt for the verification key. This feature is useful for automating signature verification, but this type of element can also be dangerous. This element moves the problem of trust away from the signature syntax and into the domain of the application. An application that is receiving the signature must know how to make proper trust decisions based on any included `<KeyInfo>` material. A receiving application *must* know when to trust material inside `<KeyInfo>` and when to discard it. Without explicit trust semantics, any XML Signature with a proper `<KeyInfo>` element will successfully verify, giving the recipient little reason to trust the sender.

One way to manage trust in an application that relies on XML Signatures is to delegate to a trust engine that takes as input a `<KeyInfo>` element and makes a trust decision based on its contents. Figure 4-1 shows how an input XML document that contains a `<Signature>` element can be parsed to retrieve the `<KeyInfo>` element. The `<KeyInfo>` element in this example contains an X.509 certificate that is subsequently passed off to a trust engine that conveys a binary trust decision to the signature verification component. The example is simple certificate path validation; the certificate inside `<KeyInfo>` is checked against a store of trusted root certificates. This trust engine concept is one of the defining facets of XKMS.

Figure 4-1

A simple Trust Service



Certificate path validation makes for a convenient example, but it is *not* the only way of asserting trust over public key material. XML Signatures allow for a wide array of components within `<KeyInfo>`. Table 4-2 describes the various element choices for `<KeyInfo>` as defined by the current XML Signature Recommendation.

Multiple child elements included within a single `<KeyInfo>` must all refer to the same verification key (with the exception of a certificate chain). This restriction prevents ambiguities during signature verification. The host of available child elements for `<KeyInfo>` allows for a high degree of application-specific trust processing. Furthermore, it is permissible for an application to add its own custom elements, provided they reside within a nonconflicting namespace and do not break the compatibility of the existing elements. Not all elements are required in compliant implementations of XML Signatures. Only `<KeyValue>` is required, whereas `<RetrievalMethod>` is recommended. The `<KeyValue>` element is designed to hold a raw RSA or DSA public key with child elements, `<RSAKeyValue>` and `<DSAKeyValue>`, respectively. Public keys inside `<KeyValue>` are represented by their Base-64 encoded raw numerical components. Well-defined BER encoded formats already exist for RSA and DSA keys. These are *not* explicitly used in conjunction with the `<KeyValue>` element, although they might be used in the context of an application specific, custom element. Listing 4-15 shows an example of a standard public key format as defined by X.509.

To contrast the binary format above, Listing 4-16 shows how a similar RSA public key would be represented as part of a `<KeyValue>` element.

Table 4-2

`<KeyInfo>` Child Element Choices

Element Name	Description
`<KeyName>`	A simple text-identifier for a key name.
`<KeyValue>`	Either an RSA or DSA public key.
`<RetrievalMethod>`	Allows for the remote reference of key information.
`<X509Data>`	X.509 certificates, names, or other related data.
`<PGPData>`	PGP related keys and identifiers.
`<SPKIData>`	SPKI keys, certificates, or other SPKI-related data.
`<MgmtData>`	Key agreement parameters (such as Diffie-Hellman parameters).

Listing 4-15

ASN.1 interpretation of an RSA public key as defined by X.509

```
  0 30   90: SEQUENCE {
  2 30   13:   SEQUENCE {
  4 06    9:     OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1
               1)
 15 05    0:     NULL
           :     }
 17 03   73:   BIT STRING 0 unused bits
           :     30 46 02 41 00 BA EA 11 7D D0 8D 35 7D 69 9D 5D
           :     F7 2F 5C CE 7A 1D 5E 75 52 E8 F4 4A 02 67 D5 59
           :     6A 43 E9 AF 4D 3E 1E 2E 42 0C 09 32 CA 5C 0E 21
           :     4C 44 97 86 EC 47 6D 6F D0 21 AB DA 54 FA 22 DC
           :     2F A3 E5 AD F7 02 01 11
           :   }
```

Listing 4-16

`<KeyValue>` element that contains an RSA key

```
<KeyValue>
  <RSAKeyValue>
    <Modulus>uuoRfdCNNX1pnV33L1zOeh1edVLo9EoCZ9VZakPpr00
    +Hi5CDAkyylwOIUxEl4bsR21v0CGr2lT6Itwvo+Wt9w==
    </Modulus>
    <Exponent>EQ==</Exponent>
  </RSAKeyValue>
</KeyValue>
```

You may wonder why the standard public key format defined by X.509 *is not* used by XML Signatures. After all, X.509 is a widely deployed standard, and many existing applications can already handle the BER encoded raw binary public key. The response falls within the scope of extensibility. A rather heavyweight ASN.1 parser must be used to decode the standard X.509 public key format. This is not the case with the XML markup. Because of its portable nature, any XML parser can successfully parse the `<KeyValue>` element, even if it does not have an XML Signature implementation to rely on. The extensible nature of XML Signatures allows for the addition of a custom element for those applications that wish to use the binary RSA key format.

Another useful `<KeyInfo>` child element is the `<X509Data>` element. This element can bear a host of child elements that all relate to X.509 certificates. The selections of elements for this type reflect common methods of uniquely identifying a certificate. Table 4-3 lists the possible child elements for `<X509Data>`. Any `<X509Data>` element must contain one or more of the first four child elements: `<X509IssuerSerial>`, `<X509SKI>`, `<X509SubjectName>`, `<X509Certificate>`, *or* a single `<X509CRL>` element.

When a Certificate Authority issues a certificate, the certificate must be given a unique serial number.

This uniqueness constraint is not shared across distinct Certificate Authorities. For example, two separate Certificate Authorities may issue two different certificates with matching serial numbers. Consequently, a proper *primary key* for a certificate must include not only a serial number but also an issuer name. This is the purpose of the `<X509IssuerSerial>` element—it is simply an element containing an issuer distinguished

Table 4-3
`<X509Data>` Child Element Choices

Element Name	**Description**
`<X509IssuerSerial>`	X.509 issuer distinguished name and associated serial number
`<X509SKI>`	X.509 SubjectKeyIdentifier extension
`<X509SubjectName>`	X.509 subject distinguished name
`<X509Certificate>`	X.509v3 certificate
`<X509CRL>`	X.509 certificate revocation List

name and serial number pair that uniquely identifies the certificate containing the public verification key. Other methods of uniquely identifying a signer's certificate include the use of the `<X509SubjectName>` element and the `<X509SKI>` element. A subject name uniquely identifies a particular end-entity, but a given end-entity might have been issued multiple certificates from different Certificate Authorities, or may have several different *types* of certificates altogether. These possibilities imply that different public keys may exist among an end-entities possessive certificate collection. To resolve the proper public key within the scope of a given subject name, the use of the `<X509SKI>` element may prove useful. This element is the `SubjectKeyIdentifier` extension as defined by RFC 2459. It is intended to be a unique identifier for a specific public key within an application context. An `<X509SKI>` element is generated by applying a SHA-1 hash directly to the encoded `subjectPublicKey` bit string. This technique creates a unique identifier out of the public key itself. A more concise hash is also specified; the shorter version uses a fixed, 4-bit value with the last 60 bits of the SHA-1 hash of `subjectPublicKey`.

Finally, instead of specifying unique identifiers or pointers to certificates that need to be looked up in an X.500 directory, the verification certificate can be included with the use of the `<X509Certificate>` element.

You may ask how these *binary format* certificate components (distinguished names) are stored and encoded within the text-based XML Signature elements and tags. We have already argued against the use of a heavyweight ASN.1 parser that would be required to process these components during signature verification. Rather than dealing with the DER encoded form of the certificate components directly, the XML Signature Recommendation relies on the ASN.1 to string conversion as specified by RFC2253. This particular RFC defines an algorithm and format for converting ASN.1 distinguished names to UTF-8 string values. For example, Listings 4-17 and 4-18 show the ASN.1 interpretation of a distinguished name followed by its string representation as defined by RFC2253.

Distinguished names are intended to be unique identifiers. The string representation in Listing 4-18 is much more compact and ideal for an XML application, but it is not necessarily unique. This string representation does not absorb all of the information contained within the binary format. For example, if we were to try to reverse the transformation and encode the string in binary, we would lose information such as object identifiers (OIDs) as well as the ASN.1 types used to encode the values (such as, `PrintableString`). Because of this uniqueness constraint, a single

Listing 4-17

ASN.1 interpretation of a name object

```
SEQUENCE {
  SET {
    SEQUENCE {
      OBJECT IDENTIFIER countryName (2 5 4 6)
      PrintableString 'GB'
    }
    SEQUENCE {
      OBJECT IDENTIFIER organizationName (2 5 4 10)
      PrintableString 'Sceptics'
    }
    SEQUENCE {
      OBJECT IDENTIFIER commonName (2 5 4 3)
      PrintableString 'David Hume'
    }
  }
}
```

Listing 4-18

String representation as defined by RFC2253

```
CN=David Hume+O=Sceptics+C=GB
```

`<X509SubjectName>` element used within an `<X509Data>` element may not identify the proper verification key in all circumstances.

Some other important features and restrictions need to be recognized when using the `<X509Data>` element. First, this element is explicitly extensible. It is possible to add custom types from an external namespace for use within `<X509Data>`. For example, the `<X509Data>` element does not include a provision for rigorous certificate messaging standards such as PKCS#7 or PKCS#12. Support for these can be added as a custom element. Secondly, the use of the possible child elements is somewhat restrictive. Care was taken to prevent situations in which two different public keys are referenced from within a single `<X509Data>` element. Whereas only a single `<KeyInfo>` element is allowed in an XML Signature, the number of `<X509Data>` elements is unbounded. This added extensibility demands restrictions to prevent references to different public keys and processing redundancy. The first point regarding restrictions on the `<X509Data>` element is that it is quite possible to have different certificates that contain the same public key. If any combination of `<X509IssuerSerial>`, `<X509SKI>`, and `<X509SubjectName>` appear within a single `<X509Data>` element, they must refer to the same certificate or set of certificates that contain the proper verification key. Furthermore, all elements that refer to a *particular individual certificate* must be grouped together inside a single `<X509Data>` element. If the actual certificate is also present, it must be in the same `<X509Data>` element. If any such elements (`<X509IssuerSerial>`, `<X509SKI>`, and `<X509SubjectName>`) refer to a particular verification key but *different* certificate(s), they may be split into multiple `<X509Data>` elements. Finally, any `<X509Data>` element may also include a Certificate Revocation List (CRL). The format of the CRL is simply a standard X.509 CRL that has been Base-64 encoded for text-based XML element compatibility. CRLs can be used as additional semantics for determining trust. Readers unfamiliar with CRLs can refer to the primer in Chapter 2.

Listing 4-19 shows an example of a `<KeyInfo>` element containing a single `<X509Data>` element that uses a Base-64 encoded X.509 certificate for an explicit verification key.

The final `<KeyInfo>` child that will be discussed is the `<RetrievalMethod>` child element. This element is similar to a `<Reference>` element in that it uses URI syntax to identify a remote resource. In this case,

Listing 4-19

An example `<KeyInfo>` element

```
<KeyInfo>
    <X509Data>
      <X509Certificate>MIICcjCCAdugAwIBAgIQxo8RBl7oeoBUJR7
      1341R/DANBgkqhkiG9w0BAQUFADBsMQswCQYDVQQGEwJVUzEPMA0
      GA1UECBMGQXRoZW5zMRUwEwYDVQQKEwxQaGlsb3NvcGhlcnMxETA
      PBgNVBAMTCFNvY3JhdGVzMSIwIAYJKoZIhvcNAQkBFhNzb2NyYXR
      lc0BhdGhlbnMuY29tMB4XDTAxMDIxNjIzMjgzNVoXDTAyMDIxNjI
      zMjgzNVowbzELMAkGA1UEBhMCQ0ExDzANBgNVBAgTBkF0aGVuczE
      TMBEGA1UEChMKUGhpbG9zb3BoeTEPMA0GA1UEBxMGQXRoZW5zMQ4
      wDAYDVQQDEwVQbGF0bzEZMBcGA1UEDBMQRm91bmRlciBvZiBMb2d
      pYzCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA1b2CY7+zN4y
      KicJRLgnTLVFXMcw9Xo9jmHPX6h7sTw+W2Ld3PRZSgXlt2vkAUcU
      sA49dGMTPKg/JJjvqu+wWkYbaQ39GbSvmwsO8GTpQERleuGKrptY
      Y/DGU0YFdONyZS7KZ5l1KMKp54PyQNAkE9iQofYhyOfiHZ29kkEF
      VJ30CAwEAAaMSMBAwDgYDVR0PAQH/BAQDAgSQMA0GCSqGSIb3DQE
      BBQUAA4GBACSzFR9DWlrc9sceWaIo4ZSdHF1P3qe5WMyLvCYNyH5
      FmrvKZteJ2QoiPw+aU/QX4d7sMuxGONYW4eiKTVSIfl6uNaMECLp
      Tfg+rZJHVT+2vy+SwfOKMZOFTgh/hGnlNdwtjEku2hIZZlGEF4+n
      6Ss4C/K+gp5K1UmQYvoyXxPK
      </X509Certificate>
    </X509Data>
</KeyInfo>
```

the resource being identified is keying material for use in signature verification. The <RetrievalMethod> element works by specifying a URI, optional type attribute, and an optional set of transforms. We will omit discussion of the transforms for now and return to that topic in Chapter 5, where the details of the <Reference> element are further discussed. When the URI specified in a <RetrievalMethod> is de-referenced, the result is an XML document (except for a single special case) that is any one of the child elements of <KeyInfo>. That is, a <RetrievalMethod> describes the location of any element listed in Table 4-2 (except for <RetrievalMethod> itself). An example usage of <RetrievalMethod> is shown in Listing 4-20.

The example in Listing 4-20 denotes the location of a certificate chain. The URI points to an XML file located on a remote server, and the optional Type element is utilized to add information about what kind of information is inside certChain.xml. Chains of certificates are often necessary to properly identify a verification key. For example, if a given end-entity has a certificate that was signed by an intermediate Certificate Authority (such as, the authority who signed the certificate is itself authorized by another certificate authority), a chain of certificates may be required for a trust engine to properly complete certificate path validation. A trust engine may not have enough information about intermediate certificate authorities that eventually signed the actual verification key. In this case, to complete the path validation, a proper bridge of certificates must be placed between the client's key and the certificate authorities accepted by the trust engine.

The content of certChain.xml is not in a special format; it relies instead on the child elements of <X509Data> as a means to structure a certificate chain. The only restriction given by the <RetrievalMethod> element is that the URI must de-reference to a well-formed XML file with *some* <KeyInfo> child as the root element (again, except for one special

Listing 4-20 A <RetrievalMethod> element that describes the location of <X509Data>

```
<KeyInfo>
    <RetrievalMethod Type="http://www.w3.org/2000/09/xmldsig#X509Data"
     URI="http://www.myserver.com/certChain.xml"/>
</KeyInfo>
```

case). The contents of `certChain.xml` could have been any valid `<KeyInfo>` child. The `<X509Data>` element is shown as a rather arbitrary example. A certificate chain can be modeled as a single `<X509Data>` element that contains multiple `<X509Certificate>` elements. This is shown in Listing 4-21.

For brevity, Listing 4-21 omits the Base-64 encoded certificate content of each `<X509Certificate>` element. The single special case previously noted is the option to have `<RetrievalMethod>` de-reference to a binary X.509 certificate, and not an XML document. This particular type of `<RetrievalMethod>` can be useful for XML-unaware applications that rely exclusively on standard X.509 binary certificates. In this case, the type attribute of `<RetrievalMethod>` could be set to **http://www.w3.org/2000/09/xmldsig#rawX509Certificate**. This URI denotes an *optional* identifier. The type identifier could have been left out if the application already has knowledge about the type of `<KeyInfo>` element that will be sent when the source URI is de-referenced.

One advantage of using `<RetrievalMethod>` to reference a remote certificate chain shows up when multiple `<Signature>` elements require the same verification key, and a certificate chain denotes that verification key. There is no restriction on the number of `<Signature>` elements that may appear within a given file or context. Therefore, a single signer could generate a number of such `<Signature>` elements that rely on a common certificate chain for verification. Listing 4-22 shows how this might be packaged in the case when `<RetrievalMethod>` is *not* used.

Listing 4-22 shows that we have two arbitrary `<Signature>` elements that reference the *same* certificate chain. The entire encoded contents of each `<X509Certificate>` elements are omitted for brevity. A Base-64 encoded certificate typically represents on average about 1500 bytes. For all six such encoded certificates we are using a lot of space in our `<Signature>` elements, around 9KB total, half of which is redundant

Listing 4-21

An example certificate chain using children of `<X509Data>`

```
<!-- certChain.xml
This file represents a certificate chain.
No ordering is explicitly implied. -->
<X509Data>
  <X509Certificate> ... <X509Certificate>
  <X509Certificate> ... <X509Certificate>
  <X509Certificate> ... <X509Certificate>
</X509Data>
```

Listing 4-22

Two <Signature> elements that reference a certificate chain using <X509Data>

```
<Signature Id="Purchase Order 1"  ... >
...
  <KeyInfo>
      <X509Data>
            <X509Certificate> MIIDHzCCAgc ... </X509Certificate>
            <X509Certificate> MIIC2aCWZvc ... </X509Certificate>
            <X509Certificate> MIIDZTEcCCA ... </X509Certificate>
      </X509Data>
  </KeyInfo>
...
</Signature>
<Signature Id="Purchase Order 2"  ... >
...
  <KeyInfo>
      <X509Data>
            <X509Certificate> MIIDHzCCAgc ... </X509Certificate>
            <X509Certificate> MIIC2aCWZvc ... </X509Certificate>
            <X509Certificate> MIIDZTEcCCA ... </X509Certificate>
      </X509Data>
  </KeyInfo>
...
</Signature>
```

information. If we instead rely on `<RetrievalMethod>` to denote the certificate chain, the same `<Signature>` elements can be represented with significant space savings for each `<KeyInfo>` element. Listing 4-23 shows what these `<Signature>` elements might look like.

The `<Signature>` elements shown in Listing 4-23 are more compact than the same elements shown in Listing 4-22. There are many ways to take advantage of the possible child elements offered by `<KeyInfo>`. This element is a rich source of examples because many different methods exist for identifying a verification key and determining trust. The extensible nature of `<KeyInfo>` itself allows for other XML technologies that provide trust semantics to hook into the XML Signature syntax. For now we will leave the additional features of `<KeyInfo>` and proceed to the remaining element in the XML Signature syntax—the `<Object>` element.

The <Object> Element

One way to introduce the `<Object>` element is to discuss some of the additional properties required by the nature of a digital signature. Let us return for a moment to the simple electronic payment authorization

Listing 4-23 Two `<Signature>` elements that reference a certificate chain using `<RetrievalMethod>`

```
<Signature Id="Purchase Order 1"  ... >
...
  <KeyInfo>
    <RetrievalMethod Type="http://www.w3.org/2000/09/xmldsig#X509Data"
    URI="http://www.myserver.com/purchaseOrderChain.xml"/>
  </KeyInfo>
...
</Signature>
<Signature Id="Purchase Order 2"  ... >
...
  <KeyInfo>
    <RetrievalMethod Type="http://www.w3.org/2000/09/xmldsig#X509Data"
    URI="http://www.myserver.com/purchaseOrderChain.xml"/>
  </KeyInfo>
...
</Signature>
```

shown in Listing 4-6. If we assume that L. Meyer signs this electronic check, the paperboy may take the check to a bank and have the bank transfer funds from L. Meyer's account to the paperboy's account. If the paperboy is a particularly malicious character, he may cash the check over and over again by keeping copies of it. He might take it to a different bank, or he may cash the copies slowly over time. The signature will always verify, and the bank will have no way to know if the paperboy is getting new checks or using the same checks repeatedly. A time-stamp is often used to solve this type of problem. If a time-stamp is signed along with the check, the bank can determine if the time-stamp is valid or if a check has already been cashed with that time-stamp. The addition of a time-stamp adds an idempotent property to the check. Repeatedly cashing the check will have the same effect on the paperboy's account as cashing it a single time.

Additional properties *about* a signature can be useful in preventing the fraudulent use of digital signatures. XML Signatures provide a standard way of adding additional semantics in the form of a `<Signature-Properties>` element. XML Signatures *do not* provide a way to interpret these additional properties. For example, there is no provision for an XML Signature to validate the *meaning* of a time-stamp. An application that verifies XML Signatures must know how to understand when a time-stamp is valid and invalid, and what to do when two signatures arrive with the same time-stamp. The use of additional assertions about signatures is useful enough to warrant a specific element for this purpose. Rather than add another child element to `<Signature>`, it is more useful to define a generic container that can hold a plethora of different elements. This is the job of the `<Object>` element. It defines a generic container that may contain other useful elements like `<SignatureProperties>` and `<Manifest>`. The `<Manifest>` element has several interesting uses that will be discussed in the last section. The `<Object>` element can contain data of any type. The only obvious restriction is that if binary data is included within an `<Object>` element, it must be encoded in a printable format suitable for representation in an XML document. This usually means Base-64 encoding, although custom encoding schemes are not forbidden by XML Signatures. The `<Object>` element has three optional attributes: an `Id`, `MimeType`, and `Encoding`. The `Id` is used as a unique way of referencing the `<Object>` element from other places inside the `<Signature>` element. The `MimeType` is an advisory type that indicates to a processing application the type of data that is inside the object, independent of how the data is encoded. The `Encoding` attribute is a URI identifier that describes the type of encoding mechanism used. It may be difficult to see how this fits together without an example. Listing 4-24 shows how one might include a binary GIF file as part of an *enveloping* signature with the use of an `<Object>` element.

Note in Listing 4-24 the use of the `<Reference>` element, shown in bold. This element uses the optional `Type` attribute that identifies the type of object pointed to; in this case, an Object type. This attribute is optional and may be omitted if the application can determine the type through some other means. The URI attribute used in the `<Reference>` element is a mechanism of pointing to the XML resource containing that attribute; in this case, it is the element that has `"ImportantPicture"` as an `Id` attribute value. The `<Object>` element shown uses all three optional attributes. Notice that the `MimeType` does not specify the *content-type* of the information inside the `<Object>` elements, but instead specifies the *type* of data in a broad sense—`MimeType` is used only as a convenient identifier. For more information on MIME and MIME types, the reader should reference RFC2045.

The encoded binary .GIF file resides inside the `<Object>` element and is included in the signature because it is referenced by a `<Reference>` element. The data is not signed directly, but indirectly; a hash of the data inside the `<Object>` is signed (including the `<Object>` tags). The only part of an XML Signature that actually has a signature algorithm applied directly to it is the `<SignedInfo>` element. Because the `<Object>` tags

Listing 4-24 An enveloping signature over a .GIF file

```
<Signature>
  <SignedInfo>
    <Reference Type="http://www.w3.org/2000/09/xmldsig#Object"
     URI="#ImportantPicture">
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
         <DigestValue>HfRNHKuQrDiTy3XABMFbyteg3CG=</DigestValue>
     </Reference>
   </SignedInfo>
   <Object Id="ImportantPicture" MimeType="image/gif"
     Encoding="http://www.w3.org/2000/09/xmldsig#base64">
        aWcgQmxha2UncyBBdXRoZW50aWNhdGlvbiBTZXJ2aWNlMRQwEgYDVQQLEwtFbmdp
        bmVlcmluZzEWMBQGA1UEAxMNQmlnIEJhZCBCbGFrZTEcMBoGCSqGSIb3DQEJARYN
        YmJiQGJiYmFzLmNvbTAeFw0wMDA2MjAyMTEzMzVaFw0xMTA2MDMyMTEzMzVaMH4x
        CzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpTb21lLVN0YXRlMQ8wDQYDVQQKEwZTZXJ2
        ZXIxFDASBgNVBAsTC1NlcnZlciBDZXJ0MRMwEQYDVQQDEwpTZXJ2ZXJDZXJ0MR4w
        HAYJKoZIhvcNAQkBFg9zZXJ2ZXJAY2VydC5jb20wgZ8wDQYJKoZIhvcNAQEBBQAD
        gY0AMIGJAoGBAMg7Y9ZByAKLTf4eOaNo8i5Ttge+fT1ipOpMB7kNip+qZR2XeaJC
        iS7VMetA5ysX7deDUYYkpefxJmhbL2hO+hXj72JCY0LGJEKK4eIf8LTR99LIrctz
    </Object>
...
</Signature>
```

are digested along with the encoded data, a problem with signature validity can result if the data inside the `<Object>` element is moved. For example, assume that the .GIF file we are signing as part of our enveloped signature is moved to a remote location such as a Web server or a distributed file system. If this .GIF file is encoded and then digested, the old digest value will not match because it was created with the inclusion of the `<Object>` tags. This problem can be circumvented with the use of a transformation that removes the `<Object>` tags *before* the signature is created. Signature transformations used to accomplish element filtering are discussed in Chapters 5 and 6. The problem of moving data out of a signature and maintaining signature validity is quite subtle. An objector might make the following claim: if we move the data object *out* of the `<Signature>` element, we must also change the `<Reference>` element that points to this data. If we change this `<Reference>` element, the `<SignatureValue>` will change because the structure and context of each `<Reference>` element is signed directly during core signature generation. Put another way, the movement of data necessitates a change in the `<Reference>` element that points to it, thereby altering the `<SignatureValue>` because every `<Reference>` element is signed directly. This argument is quite convincing, but incorrect. The reason is that the nature of a `<Reference>` element makes it acceptable to omit the URI attribute on at most one `<Reference>` element, if it is assumed that the application knows in advance where the data source resides. Listing 4-25 shows an example of a `<Signature>` that can maintain validity if the data inside the `<Object>` tags is moved elsewhere.

In Listing 4-25, the data that we are signing *happens* to be inside the `<Object>` element. This is arbitrary, and the omission of the URI attribute from the `<Reference>` element implies that the application knows where to get the data. Other elements that can reside inside `<Object>` (other than arbitrary Base-64 encoded data) may avoid this problem with the careful use of attribute identifiers. The use of the `<SignatureProperties>` element within an `<Object>` element is similar to Listing 4-24, but many of the optional attributes can be omitted because the identifying attributes are now stored as part of the `<SignatureProperties>` element. The use of this element is shown in Listing 4-26. The properties listed inside `<SignatureProperties>` are arbitrary and fictional,—any application-defined semantics can be placed inside this element. In Listing 4-26 we will think up a simple arbitrary

Listing 4-25 A `<Signature>` element that omits the URI attribute in the `<Reference>` element

```
<Signature>
  <SignedInfo>
    <Reference>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>HfRNHKuQrDiTy3XABMFbyteg3CG=</DigestValue>
    </Reference>
   </SignedInfo>
   <Object Id="ImportantPicture" MimeType="image/gif"
     Encoding="http://www.w3.org/2000/09/xmldsig#base64">
        aWcgQmxha2UncyBBdXRoZW50aWNhdGlvbiBTZXJ2aWNlMRQwEgYDVQQLEwtFbmdp
        bmVlcmluZzEWMBQGA1UEAxMNQmlnIEJhZCBCbGFrZTEcMBoGCSqGSIb3DQEJARYN
        YmJiQGJiYmFzLmNvbTAeFw0wMDA2MjAyMTEzMzVaFw0xMTA2MDMyMTEzMzVaMH4x
        CzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpTb21lLVN0YXRlMQ8wDQYDVQQKEwZTZXJ2
        ZXIxFDASBgNVBAsTC1NlcnZlciBDZXJ0MRMwEQYDVQQDEwpTZXJ2ZXJDZXJ0MR4w
        HAYJKoZIhvcNAQkBFg9zZXJ2ZXJAY2VydC5jb20wgZ8wDQYJKoZIhvcNAQEBBQAD
        gY0AMIGJAoGBAMg7Y9ZByAKLTf4eOaNo8i5Ttge+fT1ipOpMB7kNip+qZR2XeaJC
        iS7VMetA5ysX7deDUYYkpefxJmhbL2hO+hXj72JCY0LGJEKK4eIf8LTR99LIrctz
    </Object>
...
</Signature>
```

XML format for the electronic check shown in Listing 4-6 and include this in an XML *enveloping* signature along with a `<SignatureProperties>` element. The use of the `<SignatureProperties>` element here is to convey assertions about the electronic check. Note that we could have signed the check in its native format (text file), but casting it as XML makes for a better example because the information inside the check is immediately visible to the reader. We are leaving out additional elements and features

Listing 4-26 Use of `<SignatureProperties>` to convey signature assertions

```
<Signature Id="SignedCheckToPaperBoy">
  <SignedInfo>
      <Reference URI="#CheckToPaperBoy">
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>3846JEYbJymGoDfgMRaH5PYeNQv=</DigestValue>
      </Reference>
      <Reference URI="#FictionalSignatureAssertions"
       Type="http://www.w3.org/2000/09/xmldsig#SignatureProperties">
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>r3653rvQTO0gKtMyu4VfeVu9ns=</DigestValue>
      </Reference>
   </SignedInfo>
     <Object>
        <ElectronicCheck Id="CheckToPaperBoy">
          <RecipientName>PaperBoy</RecipientName>
          <SenderName>L.Meyer </SendName>
          <AccountNumber>765121-2420</AccountNumber>
          <Amount>$2</Amount>
        </ElectronicCheck>
      </Object>
      <Object>
        <SignatureProperties>
          <SignatureProperty Id="FictionalSignatureAssertions"
              Target="#SignedCheckToPaperBoy">
              <Assertion>
                <GenerationTime>
                  Mon Jun 11 19:10:27 UTC 2001
                </GenerationTime>
              </Assertion>
              <Assertion>
                  <Note> Can only be cashed at Bank Foobar </Note>
                </Assertion>
                <Assertion>
                  <ValidityDays> 90 </ValidityDays>
                </Assertion>
            </SignatureProperty>
          </SignatureProperties>
        </Object>
  </Signature>
```

that make Listing 4-26 a proper XML Signature. The intent here is to show how one might use the `<Object>` element.

Notice the use of the two `<Reference>` elements. The first `<Reference>` element points to the electronic check with the use of an attribute identifier, `CheckToPaperBoy`. The digest value appearing in this first `<Reference>` element is the digest of the `<Object>` element that contains the `<ElectronicCheck>` element. More about how this processing is accomplished will be discussed in Chapter 5, when we look at XML Signature processing. Unlike Listing 4-24, when both `<References>` are digested, the `<Object>` tags are *not* included in the digest calculation. This is because the data pointed at is referenced by an XML attribute that points directly at the desired XML element, effectively skipping the `<Object>` tags.

The second `<Reference>` element shown in Listing 4-26 identifies our set of fictional signature assertions with the attribute identifier `FictionalSignatureAssertions`. Notice also that we have used the `Type` attribute to denote the type of object that we are pointing to. This is an optional attribute but may be useful for applications that require additional context during signature processing. A `<SignatureProperties>` element may have an unbounded number of child `<Signature Property>` elements. These child elements provide a natural way to create groups of signature assertions that may be applied to distinct signatures. The `<SignatureProperty>` element has two attributes: an optional `Id` and a required `Target` attribute. If `<SignatureProperty>` is used, the target signature to which it applies *must* be specified. In our example, the `Target` specified is `"SignedCheckToPaperBoy,"` which is the identifying attribute of the parent `<Signature>` element used in Listing 4-26. The `Target` element is required to ensure a strong relation between a set of signature assertions and the actual signature. Mismatching assertions and signatures can be a security risk; if this element were optional, a group of assertions within a file that contained multiple `<Signature>` elements might be ambiguous. It would be difficult to know which assertions were intended for which `<Signature>` elements.

The `<SignatureProperty>` element contains a set of assertions about the electronic check. It is up to the application to process these correctly and make proper trust decisions based on their semantics. The assertions shown are completely fictional.

The <Manifest> Element

The <Manifest> element is another well-defined child element of <Object>. This element is powerful and useful for providing flexible solutions for various signature processing and signature packaging complications. The term "manifest" is used here again and is distinct from the abstract manifest discussed in Definition 4.2. The <Manifest> element used here has a similar meaning—it is simply another collection of <Reference> elements, much like the <SignedInfo> element. The main difference between the two lies in the amount of processing that is required. The <SignedInfo> element is a defining part of the XML Signature and is the actual data that has a signature algorithm applied to it. Consequently, it is also the element that is verified via the signature transformation during the verification process. The <Manifest> element contrasts <SignedInfo> in that its contents are not explicitly verified, only its structure. There is no requirement to actually verify any <Reference> elements specified inside a <Manifest> element. One might think of <SignedInfo> as more constrained in its semantics, while <Manifest> is more relaxed. A <Manifest> element is a *collection* of resources and is also a resource itself. If used in a <Signature> element, it is explicitly specified as a <Reference> inside <SignedInfo>. Listing 4-27 shows how a <Manifest> element might be used inside <Signature>.

The best way to understand Listing 4-27 is to first direct your attention to the <Manifest> element. This element contains a list of <Reference> elements and uses an Id attribute much like previously discussed elements. The number of <Reference> elements allowed in a <Manifest> is unbounded, but the element must contain at least one <Reference> element. In Listing 4-27, the references point to two binary files that reside on a remote server. In this example, we can assume that the files represent some type of important report specified in two formats, GIF format and PDF. When we refer to the <Manifest> from the <Reference> element inside <SignedInfo>, we are really signing the canonical form of the <Manifest> element itself. We are signing the structure of the <Manifest> element and not the binary data referenced by the <Manifest> element. This means that when we verify the signature at a later time, the integrity of the actual data referenced by the <Manifest> element (such as, one of the report files is altered) may be lost, but the

Listing 4-27

An example `<Signature>` element that uses a `<Manifest>`

```
<Signature Id="ManifestExample">
   <SignedInfo>
      <Reference URI="#ReportList"
      Type="http://www.w3.org/2000/09/xmldsig#Manifest">
        <DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>545x3rVEyOWvKfMup9NbeTujUk=</DigestValue>
      </Reference>
    </SignedInfo>
    <Object>
      <Manifest Id="ReportList">
        <Reference URI="http://www.myserver.com/Report.pdf">
          <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
          <DigestValue>20BvZvrVN498RfdUsAfgjk7h4bs=</DigestValue>
        </Reference>
        <Reference URI="http://www.myserver.com/Report.gif">
          <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
          <DigestValue>40NvZfDGFG7jnlLp/HF4p7h7gh=</DigestValue>
        </Reference>
      </Manifest>
    </Object>
...
</Signature>
```

signature will still verify if the `<Manifest>` structure remains intact. In other words, when the `<Reference>` that points to the `<Manifest>` is created, the data that is actually digested is *only* the list of elements inside `<Manifest>` and not the data that comprises these elements.

What this means in practice is that the validation of the data listed in a `<Manifest>` is under application control. Certain circumstances may exist where it is *acceptable* for an application *under certain well-defined circumstances*, to accept as valid a signature with one or more references that fail reference validation. For example, Listing 4-27 references two reports. Let us assume that in the given application context, the receiving application knows that these two reports are semantically equivalent but are in different formats. It may be acceptable for the contents of one of the report files to change and therefore fail reference validation, as long as the other report remains unchanged. In this case, the application still has enough information to continue processing and should not throw an exception or halt due to a single reference validation failure. Other examples of usage for this type of feature include applications that use a large number of `<Reference>` elements—it may be acceptable in this case for a subset of the `<Reference>` elements to fail the digest check if an appropriately large number of these `<Reference>` elements pass the digest check.

The `<Manifest>` element also can provide an efficient means of allowing for the prospect of multiple signers over a set of `<Reference>` elements. Certain application domains need contracts and electronic documents that are disparate in their contents (for example, they contain multiple types of data such as a mixture of text and graphics) and also require multiple signers. To see the problem that `<Manifest>` tries to solve, we can try to solve the problem without the use of the `<Manifest>` element and ponder the results. Listing 4-28 shows the sequence of events and `<Signature>` structures that are formed if three different people attempt to sign three different `<Reference>` elements using three separate signing keys.

The main problem is the redundancy of the `<SignedInfo>` elements. Each `<SignedInfo>` element must be repeated for each XML Signature. In the example, this might not seem like a major issue, but when the `<SignedInfo>` element grows to hundreds or thousands of `<Reference>` elements, potential exists for a lot of wasted space. Employing the `<Manifest>` element can reduce this redundancy. The `<Manifest>` element can be used as a sort of global resource list that can be referenced by any number of `<Signature>` elements. Instead of the signatures signing a duplicate `<SignedInfo>`, each signature signs the contents of a `<Manifest>` element. The only caveat is that because a `<Manifest>` element is usually (but not necessarily) a part of *some* parent `<Signature>` block (it resides inside an `<Object>` element), the signing may not be perfectly symmetric. The resulting structure still implies that the `<Signature>` element that contains the `<Manifest>` element is more significant than the others in some way, but this result is much better than the duplication shown in Listing 4-28. Listing 4-29 shows how the `<Manifest>` element can be used in the creation of a signature with multiple signers and multiple documents.

The resulting signature in Listing 4-29 is more efficient than that shown in Listing 4-28. The signature with the `Id` value of `"EfficientSignature1"` will usually be generated first, because it houses the `<Manifest>` element. The three `<Signature>` elements shown are at the same nesting level and can appear within a single XML document. Each separate `<Signature>` element has an attribute reference to `"ThreeReferences"` that ultimately refers to the `<Manifest>` element inside the first signature element shown.

Listing 4-28

Multiple signers and multiple references without the use of `<Manifest>`

Step 1: The first signer collects the necessary references and signs them.

```
<Signature Id="InefficientSignature1">
  <SignedInfo>
    <Reference URI="#reference1">...</Reference>
    <Reference URI="#reference2">...</Reference>
    <Reference URI="#reference3">...</Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

Step 2: The second signer needs to sign the same information.

```
<Signature Id="InefficientSignature2">
  <SignedInfo>
    <Reference URI="#reference1">...</Reference>
    <Reference URI="#reference2">...</Reference>
    <Reference URI="#reference3">...</Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

Step 3: The third signer needs to sign the same information.

```
<Signature Id="InefficientSignature3">
  <SignedInfo>
    <Reference URI="#reference1">...</Reference>
    <Reference URI="#reference2">...</Reference>
    <Reference URI="#reference3">...</Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

Chapter Summary

At this point, the reader should have a good understanding of the syntax used to express XML Signatures. We started with some abstract definitions to provide a foundation for the nature of XML Signatures, how they are generated, and what they mean. We went through each of the elements in a systematic fashion and showed examples of their use. An XML Signature begins with a parent `<Signature>` element that provides structure and an identifier for the signature. The next element is

Listing 4-29

The use of `<Manifest>` with multiple signers and multiple documents

```
<Signature Id="EfficientSignature1">
  <SignedInfo>
    <Reference URI="#ThreeReferences"
    Type="http://www.w3.org/2000/09/xmldsig#Manifest">
    <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>725x3fVasdfvBGFGjhjyDSFvUk=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>  <!-- From signer #1-->
  <Manifest Id="ThreeReferences">
    <Reference>
     ...
    </Reference>
    <Reference>
     ...
    </Reference>
    <Reference>
     ...
    </Reference>
  </Manifest>
...
</Signature>

<!-- Here comes the second signature -->

<Signature Id="EfficientSignature2">
  <SignedInfo>
    <Reference URI="#ThreeReferences"
    Type="http://www.w3.org/2000/09/xmldsig#Manifest">
    <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>725x3fVasdfvBGFGjhjyDSFvUk=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>  <!-- From signer #2-->
...
</Signature>
<!-- Here comes the third signature -->

<Signature Id="EfficientSignature3">
  <SignedInfo>
    <Reference URI="#ThreeReferences"
    Type="http://www.w3.org/2000/09/xmldsig#Manifest">
    <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>725x3fVasdfvBGFGjhjyDSFvUk=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>  <!-- From signer #3-->
...
</Signature>
```

the `<SignedInfo>` element—the list of things that we are going to sign, the *signed information*. Specific data streams to digest are denoted by `<Reference>` elements, and URI syntax is used to specify this stream. We also saw how the `<KeyInfo>` element can be used to help facilitate the automatic processing of XML Signatures by providing a mechanism for identifying verification key material. Finally, we ended with discussion of the `<Object>` element, a generic container for any type of data object. Two specific types defined by the XML Signature recommendation are useful for inclusion inside an `<Object>` element, `<SignatureProperties>`, and `<Manifest>`. The `<SignatureProperties>` element is a convenient, predefined container for signature assertions. This element contains assertions *about* the signature that it points to. These assertions are useful for determining additional trust semantics over and above what is provided by mere signature validation and data integrity. The `<Manifest>` element is used to solve two problems: it appropriates reference validation to the application domain and provides a convenient means for multiple signers to sign multiple documents. Without the `<Manifest>` element, the resulting signature is larger, has redundant semantics, and incurs a performance penalty during creation and verification.