RSA PRESS

XML Security

Develop and deploy a solid XML
security strategy

Learn about the structure and syntax
of XML signatures

Counteract XML-specific security
breaches effectively

Blake Dournaee

XML Security

This page intentionally left blank.

XML Security

Blake Dournaee

McGraw-Hill/Osborne

A Division of The McGraw-Hill Companies

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please contact George Hoare, Special Sales, at george_hoare@mcgraw-hill.com or (212) 904-4069.

TERMS OF USE

Dedication

To my family, who gave me endless inspiration and support

About the Author

Blake Dournaee joined the developer support team of RSA Security in 1999, specializing in support and training for the BSAFE cryptography toolkits. He has a B.S. in computer science from California Polytechnic State University in San Luis Obispo and is currently a graduate student at the University of Massachusetts.

Contents at a Glance

This page intentionally left blank.

Contents

This page intentionally left blank.

Preface

The inspiration for this book comes from an innocuous personal story. It is hard to believe that a few simple events in the course of my day translated into a complete book.

There was a time when I knew even less about XML Security than I do now (I should hope to convince the reader that I know *something* about the subject). We shall see. In particular, I remember researching something brand new called an *XML Signature*. At the time, I knew nothing about XML, but knew a few things about security. Needless to say, after staring blankly at the XML Signature Candidate Recommendation, I instantly knew that the marriage of XML and Security is a world all its own, with its own personality traits, which are separate and distinct from "traditional" applied security. Although XML Security shares common bonds with traditional applied security, a whole new viewpoint and set of conceptual tools are needed to understand the subject in its entirety. At the moment I made this realization, I realized the scope of the research and work ahead of me. I only wished that I had a guide that would explain everything to me in an easy, understandable manner. The piece of inert wood byproducts in your hands represents my best attempt at this vision.

This page intentionally left blank.

Acknowledgments

The first person I would like to thank is my father, whose sacrifice and love gave me endless inspiration. I would also like to thank Ilan Zohar. He has thoroughly reviewed this book and been an inspiration to my professional and academic career. He has caught some of my conceptual mistakes and helped ensure the technical precision of the material presented.

Jason Gillis did a great job of reviewing a few of the chapters, and I consider him to be one of my mentors, along with Clint Chan, who also did a great job of providing feedback. This book would surely not exist if it were not for the collective decision of Jason Gillis, Clint Chan, Catherine Huang, Patrick Lee, and Eleanor Huie to hire me as part of the BSAFE Developer Support group in 1999.

I want to give a special thanks to Daisy Wise, who helped me with my algorithms class. I would probably be retaking the class instead of writing this book without her patience and dedication. David Rutstein helped keep my energy going with his dedication as a workout partner.

Dale Gundersen deserves special thanks for the artwork found in Chapter 2; he has provided the most elaborate cryptographic machine ever seen! Bryan Reed also deserves special thanks for providing me with support and patience during the course of this work.

Steven Elliot deserves a special thank you for providing me with the opportunity to write, as do Tracy Dunkelberger, Alexander Corona, and Beth Brown, who helped make the process as painless as possible.

Other individuals that deserve special recognition include Rosie M. Faifua, who supported me throughout the bulk of this work. Stephanie Blossom and Chris Jones also deserve a special thank you because many of the chapters were written in the confines of their apartment. In addition, frequent visitors to 720 Foothill Boulevard in San Luis Obispo deserve a special thank you for providing the necessary distractions at the proper time.

Above all, however, was the collaboration and encouragement of Jeremy Crisp, who did absolutely nothing, but still gets his name in a published work.

This page intentionally left blank.

About the Reviewer

Ilan Zohar, currently with Hewlett Packard, has been deeply involved with various security projects, but most recently, the implementation of XML security standards. Previously he worked with the RSA BSAFE™ CryptoC team for RSA Security Inc. Ilan graduated as an MS Electrical Engineering from Stanford University in 1999, where he also specialized in cryptography and its applications to information security.Ilan holds a M.S. in Mathematics from the Technion Israel Institute of Technology, Haifa, Israel, a B.S in Electrical Engineering Cum Laude, and a B.A. in Mathematics Summa Cum Laude. You can reach Ilan at ilan@stanfordalumni.org.

This page intentionally left blank.

Introduction

This book is an introduction to XML Security. You should be excited because a lot of learning is about to take place. This book takes the huge field of applied security and equally huge field of XML and smashes them together into something small enough to hold in your hand.

This book is incomplete and bears resemblance to a split-second in the evolutionary time-line of XML Security. It is the nature of technology to spin far beyond the boundaries of a single book. Because of this, we pay special attention to understanding concepts and shy away from fluid details. The purpose of this book is to communicate concepts and often this is done at the expense of some details. You should finish this book and be able to explain XML Security basics in a way that can be easily understood, rather than display encyclopedic knowledge of XML Security.

Before we take a swim in the sea of XML Security, we need to define what it is we are talking about. What is XML Security? What is the scope of this book? XML is an enabling technology for the portability of data. XML Security is the application of *applied security* to XML structures. One naïve way to break apart the subject of applied security is to make a division between data privacy and authentication. Taking our definition further, we can make the substitution and arrive at the following definition: XML Security is the application of data privacy and authentication to XML structures.

Not all of my readers will have a good knowledge of data privacy or authentication, and because of this, we begin the book with a primer on security concepts. This is the focus of Chapter 2. The primer offers only the bare minimum for the voyage through the high-country of cryptography. The journey can be made with the material provided here, but it is recommended that the reader visit the references section and travel with another security text. Chapter 2 omits many details regarding specific cryptographic algorithms. For example, we cover Triple-DES (a specific cryptographic algorithm) and how it works from a high-level perspective, but leave the actual algorithm as a black box and defer the particulars.

Similarly, not all readers will have a working knowledge of XML. This topic rivals applied security in its depth and breadth and is full of conceptual prerequisites. The reader simply cannot process the information in this book without understanding some basic things about XML. Fortunately, a primer on XML is provided, which is the subject of Chapter 3. Chapter 3 covers the details and basic concepts behind XML as it is related to XML Security. Chapter 3 is written to communicate the bare minimum XML knowledge required to understand most of XML Security.

Additional XML resources may be useful for pinpointing exact details, but they are less of a requirement. The XML primer contains all the reader needs to know about XML to understand XML Security.

Once the primer prerequisites have been met, the real learning begins in Chapter 4 where we begin our coverage of the XML Signature Proposed Recommendation. All of the XML technologies we cover in this book originate in a standard that is in development by the World Wide Web Consortium (W3C). You may wonder about the story behind the W3C. Instead of giving them an introduction, I will let the W3C make their own introduction (from www.w3.org):

> The World Wide Web Consortium (W3C) develops interoperable technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential as a forum for information, commerce, communication, and collective understanding.

At least three basic W3C activities relate directly to XML Security. These include the XML Signature, XML Encryption, and the XML Key Management Specification (XKMS). Some will argue that the list of activities related to XML Security doesn't end here, but also includes other technologies such as Trust Assertions (XTASSs) or transport protocols (SOAP). While this can be argued, the three aforementioned activities are certainly the most fundamental because they define the mechanisms for

basic cryptographic operations, as well as provide for the establishment of trust.

Chapter 4 begins our adventure into understanding the XML Signature and how it works. Of the three core XML Security technologies, the XML Signature Proposed Recommendation is the most mature. It is highest-ranking in its status. W3C specifications follow a process toward the pinnacle of becoming a W3C Recommendation, which is the highest tier for a W3C specification. As of the time of writing, the XML Signature specification is at the Proposed Recommendation status, which is one level prior to reaching the blessing of becoming a W3C Recommendation. Throughout our discussion of the XML Signature, I use the shortened phrase the XML Signature Recommendation instead of the more copious the XML Signature Proposed Recommendation. I am doing this only for the sake of brevity of discussion. The XML Signature Proposed Recommendation is simply too wordy to provide for readable prose. You should understand, however, that the XML Signature specification is only a *proposed recommendation*, regardless of my potentially confusing usage.

Because of the maturity of XML Signatures, the discussion spreads across three chapters. Chapter 4 is devoted to the syntax of an XML Signature; Chapter 5 is devoted to the processing rules for the XML Signature; and Chapter 6 is a summary of proposed scenarios and frequently asked questions. We will find that the complexity of the XML Signature alone is enough to stun even the sharpest minds, and in keeping with our goal of conceptual understanding, it is best to provide a playpen to try on the new ideas.

Once we have fully understood how the XML Signature works, we make the transition into XML Encryption in Chapter 7. XML Encryption is a younger technology and only boasts the rank of Last Call Working Draft, a full two steps below the Proposed Recommendation status of the XML Signature. The XML Encryption draft is younger, and because of this, it is subject to change. To prevent myself from executing embarrassing errors, I have left out some of the details of XML Encryption, although the foundation and ideas on which it is built remain strong. Our study of XML Encryption will show how data privacy can be applied to XML structures.

Chapter 8 transitions from a more conceptual discussion to a look at how a practical toolkit implementation of XML Signatures works. The product chosen is RSA Security's BSAFE Cert-J toolkit, which has support for producing signed XML structures. You should flip to the back of this book and discover how you might obtain a free evaluation copy of

Cert-J to play with. Current export laws prevent the inclusion of a CD directly with this book. The sample code provided in Chapter 8 can be obtained at this book's web site, located at www.rsasecurity.com/go/xmlsecurity.

Chapter 9 looks at the XKMS, which is the youngest XML Security technology covered in this book. The XKMS is at the lowest level of W3C maturity, currently a Working Draft. Because of this, a full discussion of how XKMS works is not covered. The reader is instead left with a good high-level conceptual understanding of XKMS.

The book concludes with an end section called "Additional Resources." This section includes a references section, information on exclusive canonicalization, and some information about template signing for RSA BSAFE Cert-J. I urge you to visit all of the web links provided in the back to complete your knowledge of the XML Security playing field. With a preview fresh in your mind, the excitement and pursuit of understanding is about to begin. Enter XML Security!

Security Primer

The study of cryptography and its related fields such as algorithms and mathematics is by nature theoretical and experimental; the application of these ideas to the real world represents the domain of applied cryptography and applied security. This chapter seeks out some fundamental ideas in the domain of applied cryptography and applied security that relate to XML Security. Very little time will be spent on the details of specific cryptographic algorithms. The focus is instead on basic cryptographic concepts and enabling technologies that bridge the important gap between applied security and XML. Special care is taken to cover all of the security concepts that are related to XML Signatures, XML Encryption, and XML Key Management (XKMS).

Because of the limited nature of a single chapter, there is not enough room for a verbose study of cryptography from its theoretical roots up through its real-world application. For those readers interested in a verbose study of cryptography, there are a number of outstanding books on the subject. Visit the references section at the end of this book for a recommended list. Further, many details about algorithms and processes have been omitted from this chapter as they are better explained in other resources. The goal of this chapter is to provide the reader with enough background in applied security to be able to understand XML Security and its importance.

Encryption

Finding a point to jumpstart a discussion of the immense topic of applied security is a daunting task. There are many places to begin, and it would be difficult to argue that one entry point is more correct than another. The general concept of encryption, however, provides a broad launching point and has the added benefit of being extremely visible. That is, almost all readers should have some sort of previous encounter with the topic, either formally or casually.

The basic idea behind encrypting data with a single key is quite simple: provide a piece of plaintext and an encryption key to an encryption method and turn the crank. Once the crank rests, the outcome should be some sort of apparently random permutation of the plaintext. The idea, of course, is that the output is undecipherable to anyone who doesn't possess the decryption key or can't otherwise guess the answer (in this case, the encryption key and decryption key are the same key). This familiar picture is shown in Figure 2-1 and is known as symmetric key encryption.

Figure 2-1 shows an example of encryption with a single key (symmetric encryption). The plaintext is fed to the encryption algorithm (crank) in blocks. These blocks combine with the encryption key and produce the final encrypted output, which is called the cipher text.

The added complexity and confusion behind encryption lies not in the basic idea, but instead in the plethora of additional enabling technologies required to make encryption practical and usable. We can define some usability constraints for encryption, based on the fundamental components shown in Figure 2-1. In particular, it is wise to start asking questions of Figure 2-1. These questions represent the proverbial bridge that brings mathematical ideas into the realm of reality.

The following questions apply to Figure 2-1:

- What is the nature of crank? Does the encryption method meet my requirements in terms of speed and strength? How does the encryption method operate on the plaintext?

- What is the nature of the plaintext? Is the data compatible in size with the encryption function? Are alterations to the input data necessary?

- What is the nature of the encryption key? How is it transported? Is it of appropriate size? How was the key generated?

Figure 2-1

Encrypting
arbitrary data

Encryption Key (E_k)

Plaintext

Ciphertext

Encryption Algorithm
("Crank")
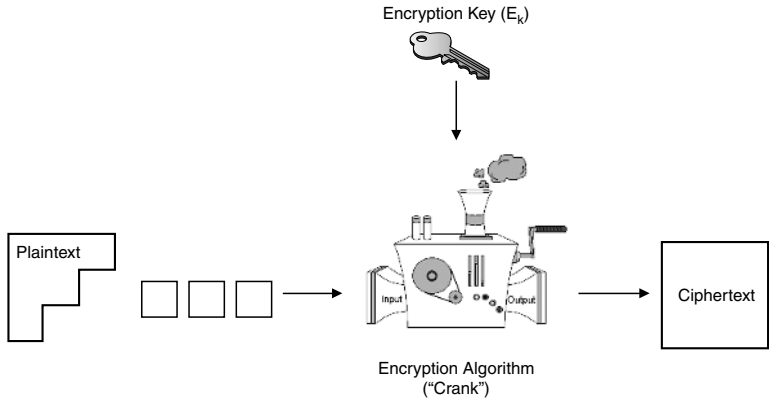
■ What is the nature of the entire process? Has it been done in accordance with an accepted or proposed standard? Is the process secure? Is the process usable?

The previous questions cover a lot of ground and the answers comprise most of the necessary knowledge needed to understand encryption for the purposes of XML Security.

Symmetric Ciphers (The Nature of the Crank)

A *symmetric cipher* is an encryption algorithm that uses the same key for both encryption and decryption. The symmetric ciphers that we are most concerned with include two variations: block ciphers and stream ciphers. A block cipher encrypts plaintext in blocks of a fixed size. The block size is related to the specific symmetric cipher and key size. Block ciphers take the center stage for our discussion because they are common and extensively supported in XML Encryption.

A stream cipher is a slightly different animal that relies on a key derivation function to generate a key stream. The XOR (exclusive-OR) operation can then be used between each byte of the plaintext and each

byte of the key stream to produce the cipher text. Stream ciphers are usually faster and smaller to implement than block ciphers, but have an important security trade off. If the same key stream is reused, certain types of attacks using only the cipher text can reveal information about the plain text. Although XML Encryption has support for stream ciphers, none are currently specified. An example of a stream cipher is the RC4 algorithm developed by RSA Data Security. More information on RC4 can be found in one of the recommended cryptography books found in the references section at the end of this book.

Two important block ciphers used in XML Encryption are Triple-DES and the AES. Triple-DES is a variation of the Data Encryption Standard (DES) algorithm and AES is the Advanced Encryption Standard. The goal here isn't an extensive mathematical discussion or cryptanalysis of either algorithm. Instead, the reader should learn how each algorithm works in terms of its required parameters and supported key sizes. Readers who want more detail about either algorithm should refer to the references section at the end of this book.

Triple-DES

Triple-DES is an interesting cipher because it is actually a reincarnation of another cipher called DES, the Data Encryption Standard. DES uses a 64-bit key consisting of 56 effective key bits and 8 parity bits. The size of a DES key never changes—it is always 64 bits long. The block size for Triple-DES is 8 bytes, which means it encrypts data in 8-byte chunks.

The original DES cipher proved to be susceptible to a brute force cipher text attack. A brute force cipher text attack consists of trying every possible key in the DES key space (2^{56} possible keys) against the cipher text until the correct plaintext is found. To thwart this type of attack, a larger key space is needed. This requirement is fulfilled by Triple-DES, which executes the DES algorithm three times. The effect of executing DES three times raises the key size to 192 bits. Of these 192 bits, only 168 bits are effective. (This is because for each 64-bit DES key, only 56 bits are useful.) The size of the key space is therefore raised to 2^{168}, which is a much larger key space. Current cryptanalysis of Triple-DES estimates the cipher effective to only 108 bits, which is still considered secure.

Triple-DES is often referred to as Triple-DES EDE, meaning Encrypt, Decrypt, Encrypt. When we say Encrypt, Decrypt, Encrypt, we are refer-

ring to the precise operations that comprise one pass of the Triple-DES algorithm on a single block of data. A high-level diagram of how Triple-DES works on each block of plaintext is shown in Figure 2-2.

In Figure 2-2 the plaintext block enters the cipher and is encrypted using DES with the first third of the 192-bit Triple-DES key (the first 8 bytes). Next, this cipher text is decrypted with the middle third bytes of the 192-bit Triple-DES key. Note that this decryption operation doesn't actually decrypt the block previously encrypted. The reason is because we are decrypting using the wrong key. Remember, the decrypt operation occurs with the middle 8 bytes of the initial 192-bit key (not the first 8 bytes, which *would* decrypt the data). The decrypt operation actually ends up jumbling the data further. The cipher ends when the block is encrypted with the last 8 bytes of the initial Triple-DES key. In general, Triple-DES is a slow cipher because it is comprised of three executions of the DES cipher on *each* block of the plaintext.

Padding and Feedback Modes

Two concepts that the reader should understand about block ciphers are *padding* and *feedback modes*. Bringing up these topics in the context of Triple-DES is useful because the reader will immediately have some sort of frame of reference to base examples on.

Figure 2-2

Triple-DES operation

Triple-DES Encryption Key (E_k)(192 bits)

First 64 bits   Middle 64 bits   Last 64 bits

Plaintext Block
8 bytes
DES Encryption

DES Decryption

DES Encryption
8 bytes
Ciphertext Block
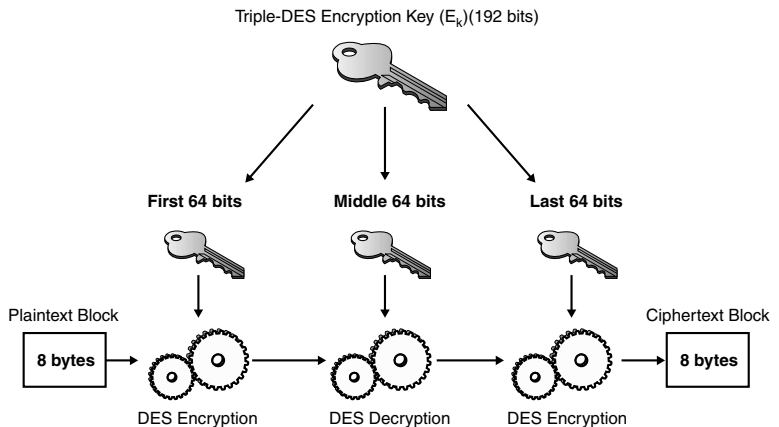
The first concept, padding, is rather simple and will be introduced first. The second concept of a feedback mode is slightly more complex, although most of the complexity and details can be safely considered out of scope for our purposes here.

Padding (The Nature of the Plaintext)

Padding is used pervasively in both symmetric key encryption and asymmetric key encryption (discussed in the next section). Consider the problem of encrypting a 6-byte chunk of data using Triple-DES. Triple-DES only operates on blocks of 8 bytes. At first glance it appears impossible to encrypt our 6-byte piece of data with Triple-DES and in fact, this is correct. It *is* impossible to encrypt a 6-byte block of data with Triple-DES unless padding is added, which forces the block to be 8 bytes.

One crude method of padding is to simply add random bytes to the plaintext until it is 8 bytes long. If one were to do this, however, the original size of the data would need to be communicated to the recipient and this would introduce extra complexity. Similarly, one could simply pad the data with 0's. This might work, but will cause problems if there are trailing 0's in the plaintext. How will the verifier know the boundary between the plaintext and the padding?

The solution is to use a padding scheme that has built-in support for marking the plaintext with an identifier that tells the verifying application how many bytes comprise the padding (and by implication, which bytes to strip). A padding scheme also has the property that it doesn't give an attacker extra information that might compromise the cipher text. A padding scheme should work like magic. A user of Triple-DES encryption should think of passing an arbitrary amount of data to the cipher and the padding scheme should add the necessary pad bytes upon encryption and strip the necessary pad bytes upon decryption. This entire discussion implies that Triple-DES alone is not very usable unless it is paired with some sort of padding scheme. The most common padding scheme for symmetric ciphers is defined in PKCS#5 (Public Key Cryptography Standards). Padding is generally "auto-magic" for symmetric ciphers, but some additional complexity and constraints arise when padding is used for an asymmetric cipher such as RSA. The two padding schemes for the RSA algorithm are discussed in the upcoming section on asymmetric ciphers.

The XML Encryption draft defines a padding scheme similar to PKCS#5 for symmetric ciphers. This scheme is described in the following section with some examples.

XML Encryption Padding Scheme (Symmetric Ciphers)

Given an n byte plaintext block and a b byte block size where $n \leq b$:
If n is less than b:

    Let $k = b - n - 1$ arbitrary pad bytes and append these bytes to n

If n is equal b:

    Let $k = b - 1$ arbitrary pad bytes and append these bytes to n

    In both cases ($n < b$ and $n = b$), append a final byte whose value
is $k + 1$.

Suppose that we have a 6-byte block (ABCDEF) that we wish to encrypt with Triple-DES. The block size that Triple-DES operates on is 8 bytes. This means that $n = 6$ and $b = 8$. We therefore need $8 - 6 - 1 = 1$ byte of arbitrary padding. Suppose our one byte of arbitrary padding is Z. Our plaintext block now looks something like this: ABCDEFZ. The final step is to append a final byte with the value $k + 1$. The value $k = 1$ and $k + 1 = 2$. Therefore, our padded plaintext block appears as follows: ABCDEFZ2.

We have now padded the plaintext in such a way as to communicate to the recipient the number of pad bytes actually added. Let's try another example that exercises the second case for the padding scheme. Suppose that we have an 8-byte block (ABCDEFGH) that we wish to encrypt with Triple-DES. The reader may claim that we don't need to pad the 8-byte block for Triple-DES. In this case the reader is technically correct. However, the padding scheme always assumes the existence of a pad byte at the end. This means the recipient is expecting the last byte to communicate a padding value. If we don't pad it properly, the recipient will try to convert the H into a number and strip padding that doesn't exist. The recipient has no way to determine which blocks originally required padding and which were the perfect size. Because of this, everything is padded—even in the cases where it is unnecessary.

For the case of an 8-byte block size and an 8-byte block, $k = 7$. This means that after adding 7 arbitrary pad bytes, the block becomes: ABCDEFGHQWERTYU. The block is 15 bytes long, and adding the last byte, which is an 8 ($k + 1$) makes two perfect blocks. The final plaintext looks as follows: ABCDEFGHQWERTYU8. The recipient will strip off the

last 8 bytes (after decryption) and the plaintext now becomes ABCDE-FGH, which is the original plaintext.

Feedback Modes

For a block cipher such as Triple-DES, a one-to-one relationship exists between a single plaintext block and the corresponding cipher text for a given key. In other words, repeated plaintext blocks always encrypt to the exact same cipher text if the key is the same. If a piece of plaintext contains repeated blocks, the cipher text will also contain matching patterns that may aid an attacker.

Furthermore, a block cipher lacks an inherent relationship between each cipher text block. This means that a possible attacker monitoring encrypted blocks can permute or remove blocks at will. The ability to remove cipher text blocks undetected can be dangerous as it may alter messages completely. For example, consider the following plaintext message:

```
*Don't* Attack at Dawn.
```

The first plaintext block (8 bytes) is `*Don't*`, the second plaintext block is `Attack  a`, and the third plaintext block is `t  Dawn`. Once encrypted, the message becomes three cipher text blocks. Let's assume the first cipher text block is ABCDEFGH, the second block is IJKLMNOP, and the third block is QRSTUVW. If these three blocks are sent in succession, and an attacker happens to remove the first block (or otherwise block it from being sent), the remaining two blocks would arrive and be decrypted. Upon decryption the receiver would have the message, `Attack at Dawn`, instead of the intended message, `*Don't* Attack at Dawn`.

A feedback mode is a way of creating a strong relationship between output cipher text blocks as well as preventing patterns from appearing in the cipher text. There are several feedback modes, but the most common mode is called *cipher block chaining* mode and is abbreviated CBC mode. When CBC mode is used, the current plaintext block is XOR-ed with the previous cipher text block. The XOR operation has the effect of altering the plaintext block in such a way that it is highly unlikely to produce some sort of repeated cipher text. Furthermore, there is now a strong relationship between all of the cipher text blocks. The message cannot be fully decrypted unless the all the cipher text blocks are present.

The careful reader may ask about the first plaintext block. If each plaintext block is XOR-ed with the previous cipher text block, what hap-

pens in the case of the first plaintext block? There is no previous cipher text block to XOR with. This special case requires something called an *initialization vector* (IV), which is used to kick-start the cipher block chaining. The initialization vector is shared, nonsecret information that must be communicated to the recipient in order to successfully decrypt the first block of the message.

At this point, the reader should have enough preliminary information to completely understand how Triple-DES works from a conceptual point of view. To test this understanding, let's look at the *URI identifier* for Triple-DES as it appears in the XML Encryption draft. This is shown in Listing 2-1, which is only a single line. A URI identifier is the name given to URI strings that are chiefly used as string identifiers. More information about URIs in general is given in Chapter 3. In any case, it's simply an identifier that tells a decrypting application that Triple-DES will be used. The details of how and why are discussed in Chapter 7, and are not important here. What is important here is to dissect the identifier and match it against concepts learned so far.

The term URI identifier may appear redundant in and of itself. After all, if we expand the definition of URI we will arrive at something like this: "Uniform Resource Identifier identifier." On the surface, this doesn't seem to make sense. The insight here is that these URI's are not meant to be de-referenced—they are simply string identifiers, hence the term *URI identifier*. The term *identifier* here is borrowed from the equivalent "identifier" concept found in a programming language. The recognition of the *URI identifier* is necessary because we will also see URIs that *are* meant to be de-referenced and we need an intellectual way of differentiating between the two.

The important element of Listing 2-1 is the string after the # sign. We know the key size used is 192 bits, the algorithm is Triple-DES and the feedback mode is CBC mode. What isn't explicit is the padding scheme used. Fortunately, this is explicit in the XML Encryption draft and is universal for all block ciphers specified therein. The padding algorithm used is the same scheme defined in the previous section on symmetric cipher padding.

Listing 2-1 Triple-DES URI identifier	`http://www.w3.org/2001/04/xmlenc#`**`tripledes-cbc`**

The reader should now understand some fundamentals about block ciphers such as the block size, padding scheme, and feedback mode. The reader is now in great shape to tackle another supported algorithm in XML Encryption—the Advanced Encryption Standard.

AES

The Advanced Encryption Standard (AES) replaces DES as the new government-sponsored block cipher. The announcement of the AES by the Commerce Department was historic because it was the first time that a global, open process was used to choose a government-sponsored encryption algorithm. The National Institute of Standards and Technology (NIST) held an open competition among 15 candidate ciphers and chose a cipher called Rijndael, developed by Joan Daemen and Vincent Rijmen. The AES *is* the Rijndael block cipher and for practical purposes these two terms can be considered equivalent.

Rijndael is a *variable key size* cipher. This contrasts Triple-DES, which has a fixed key size. The AES specification specifies a 16-byte block size and three choices for key size: 128-, 192-, and 256-bit. The AES specification names these three flavors as AES-128, AES-192, and AES-256. These size constraints simply mean that while it is possible for someone to implement Rijndael with a 64-bit key size, this particular implementation can't be considered AES-compliant.

The details of the AES cipher are out of scope for this book. The reader should refer to the references section for more information on the inner workings of this algorithm. In short, there are three variations defined by the AES specification and there are three matching URI identifiers in the XML Encryption draft that match these variations. These are shown in Listing 2-2.

After examining the identifiers listed in Listing 2-2, the reader should notice that the AES as used by XML Encryption also uses the CBC feedback mode. It's possible that other feedback modes can be used with the

Listing 2-2

XML Encryption URI identifiers for AES

```
http://www.w3.org/2001/04/xmlenc#aes128-cbc
http://www.w3.org/2001/04/xmlenc#aes192-cbc
http://www.w3.org/2001/04/xmlenc#aes256-cbc
```

AES (or none at all), but CBC mode is the most common feedback mode and currently the only one defined by the XML Encryption draft.

Symmetric Key Generation (The Nature of the Key)

The concept of a symmetric key can be reduced to a single byte string of the appropriate size. Any string of bytes is a legal symmetric key from a mathematical point of view. This means that one can use a symmetric key consisting of a single string of matching bits, or all zeroes; the nature of a symmetric encryption algorithm does not place special restrictions on the semantics of these bytes. Some symmetric algorithms, however, are subject to the presence of weak keys and care must be taken that when a random string of bytes is generated, a weak key is not chosen. For more information on weak symmetric keys, see one of the cryptography books listed in the reference section at the end of this book.

This is not the case for an asymmetric cipher such as RSA, which is discussed in following sections. What *is* important, however, is the choosing of these symmetric key bytes. Symmetric keys should be derived from a random source of information. If an attacker can reproduce or re-enact symmetric key generation, the protection afforded by the cipher is reduced to zero because the key is known. It is important to understand that numbers produced by a computer program are never random, but pseudo-random. The reason is because a computer (by definition) is simply a large state machine, and by returning the computer to the proper state, the corresponding random number can be reproduced. This violates a fundamental definition of randomness (reproducibility) and is the chief reason why the term *pseudo-random* is used.

Symmetric Encryption

The reader should now be able to answer some of the initial questions posed at the beginning of this chapter that relate to symmetric encryption. For example, we have discussed two types of cranks used for symmetric key encryption: Triple-DES and the AES. The AES is faster than

Triple-DES and has flavors with longer key sizes (which make these versions more resistant to a brute force attack). Moreover, we have also discussed the CBC feedback mode, which strengthens the crank by providing strong relationships between each cipher text block as well as obfuscating possible patterns in the cipher text. CBC feedback mode is not an add-on feature and is used to prevent attacks against the cipher text.

We also discussed the nature of the plaintext and pondered whether alterations are necessary. We realized that padding must be used to ensure that arbitrarily sized blocks of plaintext are compatible with a given block cipher. The questions still unanswered relate to the management of the encryption key (the nature of the key) as well as the nature of the entire process. Key management is discussed in the next section and the section on cryptography standards helps answer questions about the entire process.

Asymmetric Ciphers

An *asymmetric cipher* is an encryption algorithm that uses non-matching keys for encryption and decryption. This contrasts symmetric key cryptography, which uses the same key for both the encryption and decryption operation. With an asymmetric cipher, one key is used for encryption but is completely useless for decryption. Likewise, only one key can be used for decryption and is useless for encryption.

The most popular asymmetric encryption algorithm is the RSA algorithm, which was developed by Rivest, Shamir, and Adleman. While there are other asymmetric encryption schemes, the RSA algorithm is the only one I will cover or discuss in this book because it is the only such encryption scheme currently specified in any of the XML Security Standards.

We must be careful to distinguish imposter algorithms from the central idea in this particular section. For example, the XML Signature Recommendation has support for the DSA signature algorithm, which uses asymmetric signature keys. This algorithm, however, cannot be used for encryption and is discussed instead in the section on digital signatures. Similarly, the XML Encryption draft specifies support for the Diffie-Hellman key exchange algorithm. While this algorithm is a foundational algorithm for public-key cryptography, it is not an asymmetric encryption scheme even though it uses asymmetric keys. To add to the confusion, the RSA algorithm can also be used for digital signatures and it too uses

asymmetric keys. Table 2-1 shows a summary of the various asymmetric algorithms as used in the context of XML Security.

Table 2-1 lists algorithms that somehow rely on asymmetric keys to perform an encryption, signature, or key exchange operation. The reader should be certain to distinguish between the algorithms listed in Table 2-1 and understand where each one is used. We begin with an overview of the RSA encryption algorithm and the logistics of asymmetric encryption.

Introduction to the RSA Algorithm

The RSA algorithm is arguably one of the most famous asymmetric encryption algorithms—it is conceptually simple to understand, based on a well-known mathematical problem, and is robust and usable in many different environments. It has been highly studied and one can estimate that there are thousands of descriptions of the RSA algorithm floating around on the Web. It almost seems as if adding to this plethora of prose on the subject will accomplish little, as the details can be found with a simple Web search. Nonetheless, I am presenting a short summary and tour of the conceptual highpoints of the RSA algorithm here for those readers currently offline.

My reader may argue that we have not given asymmetric encryption a proper introduction. Most descriptions of this material begin with formulating asymmetric encryption as a solution or remedy to certain problems caused by symmetric key encryption.

Symmetric encryption as a means of data protection works just fine, except for one detail: we need a way of distributing the actual key to the intended recipient. In other words, we still haven't solved our entire problem. We have merely created another, arguably smaller instance of the

Table 2-1	Algorithm Name	Used in XML Encryption?	Used in XML Signatures?
Asymmetric Algorithms in XML Security	RSA Encryption Algorithm	Yes	No
	RSA Signature Algorithm	No	Yes
	DSA Signature Scheme	No	Yes
	Diffie-Hellman Key Exchange	Yes	No

problem. This book (and others like it) might not exist if it were possible to mentally connect with a recipient (via ESP or other extra-sensory phenomenon) and transmit the key securely. Solving this key transport problem actually solves part of the larger, "I want to send data securely" problem.

The argument goes like this: If we could meet and exchange the key, why even bother with all of the complexity of encryption—let's just exchange the message and be on our way. The involved reader may even argue further: Well, why don't we meet ahead of time with the intended recipient to exchange the key, and *then* exchange the messages? This is, again, another simple solution. While this is a good solution for communicating with a small number of people, it falls outside the realm of practical use when access to a secure channel is needed for an increasing magnitude of people, not to mention recipients that may be many thousands of miles away. Unfortunately, symmetric key encryption also garners a host of other problems, such as repudiation. For example, one of the members in a symmetric key exchange can simply deny that he or she wrote a certain encrypted message. While this doesn't seem like it would be much of a problem, trust is a central issue when sending any type of secure message. The receiver must be absolutely certain that any secure message received was indeed sent by the claimed sender; a false message thought to be true may be just as much of a menace as a compromised symmetric key.

A symmetric cipher also makes no provisions for data integrity. The receiver of a message has absolutely no way to verify that a message has not been altered. Thus far, we have generated three major complaints, or objections regarding symmetric key encryption:

- Symmetric key encryption fails to resolve the issue of scalable key distribution.
- Symmetric key encryption fails to resolve the issue of repudiation.
- Symmetric key encryption fails to resolve the issue of data integrity.

We can clearly see that the strength of the crank or cipher is only a small part of the story regarding the development of a secure infrastructure. What is needed here is a completely new encryption paradigm, one that somehow avoids these three problems, without introducing too many new problems.

Asymmetric Encryption with RSA

Allow me to begin with a disclaimer: the concepts surrounding asymmetric encryption are not substantially difficult to understand; however, some readers may be conditioned to believe in some innate cloud of confusion that regularly surrounds this issue. Often, the logistics of asymmetric encryption are shrouded in a mystery of terms (public key, private key, signing key, decryption key, and so forth). It is my desire to end this confusion and explain, in the most simple of terms, the logistics of an asymmetric cipher as it relates to RSA. We will not spend time on the mathematics of RSA, as this falls slightly out of scope for this particular book.

Meet Alice, Bob, and Eve

Traditionally, three basic characters describe the logistics of asymmetric encryption: Alice, Bob, and Eve. The logic is simple: Alice wishes to send a message to Bob, but wishes to do so in such a way that a third party (Eve) cannot intercept or decipher the message. The question to ask here is as follows: How do we perform this transaction without falling prey to the same ailments present in the older method of symmetric encryption? The trick is a simple, but clever maneuver: Each party (Alice and Bob only) must generate two keys. When we use the word key, think simply of a string of bits. This simplification will deteriorate when we look at some actual RSA keys, but it is close enough for now.

Three important points to remember regarding these generated keys are as follows:

- These keys are mathematically related somehow.
- One key is used to encrypt messages and one key is used to decrypt messages.
- The keys are generated using random data. The names given to these keys are public key and private key. The public key is used for encryption and the private key is used for decryption.

Alice and Bob now both have a key-pair or set of keys: one for encrypting messages, and one for decrypting messages. If Alice wishes to send a message to Bob, Alice must simply obtain Bob's public key. This public key need not be kept secret; it is only used for encrypting data. Public keys can be stored anywhere they are publicly accessible. The most common place to find a public key is inside an X.509 certificate. More information on

X.509 certificates is found in the section entitled "Trust, Certificates, and Path Validation."

Once Alice has obtained Bob's public key, she must use an encryption algorithm compatible with the generated keypair. With the encryption algorithm, Alice encrypts the message. Keep in mind these two points: She has used Bob's public key to encrypt the data, and she has used a publicly known encryption algorithm.

Only Bob can access the resulting encrypted message. Why? Bob is the only one who has the corresponding decryption key, or private key. This is another defining property of the private key—it is assumed that Bob has not given his private key to random people or otherwise divulged it publicly in any way. Remember that the keys are mathematically related in such a way that a message encrypted with a public key can only be decrypted with the corresponding private key. The term *only* is loaded—there is another way to decrypt the message, but this would involve searching for the key itself or attacking the RSA algorithm directly. Given an adequate key size, both attacks are currently unfeasible.

Highlights of the mathematical details for the RSA algorithm are given in the next section. We will not spend time on these details, as their proper place is in a cryptography book. Instead, I will note a few interesting things about the RSA key generation process, the actual key values (which is important for understanding how RSA keys are packaged), and the nature of the RSA algorithm itself.

RSA Algorithm Details

The RSA algorithm is based on the difficulty of factoring numbers. We will come to grips with what this means and how this system works shortly. We will answer two broad questions in this section. The first concerns the *logistics* of the RSA algorithm and the second concerns the *security* of the algorithm itself. We can skirt a theoretical discussion about algorithms if we make an intractability claim. That is, we will assume that a desired property of a secure asymmetric algorithm is intractability of the fundamental problem on which it is based. For the RSA algorithm, the fundamental problem is factoring. When we say the problem of factoring, we are referring to the fact that it is computationally expensive to factor a number into its prime factors (if the number is large enough).

The involved reader may wonder what intractability means from a practical standpoint. A problem is intractable for our purposes here if it presents a potential attacker with such an obstacle that he or she might as well walk around in a futile attempt to get hit by lightning instead. An attacker would have a better chance of being struck with an errant bolt of lightning than finding a solution to an intractable problem. This is the intuition about intractability that we are aiming for.

This small discussion of intractability is by no means complete without the introduction of the skeptic, the man or woman who relies on the inherent weakness of this largely inductive argument concerning intractability. The skeptic might assert a claim that empirical evidence concerning the intractability of such an algorithm has not yet been proven to exist, or more concisely: just because no one until now has found an easy solution to intractable problems, it does not necessarily follow that such a solution doesn't exist, or can't later be discovered.

What does this mean to our discussion and how might we better understand this philosophic question? We might begin by making a division in viewpoints by assigning the argument of the theoretician on one side and positioning the argument of the engineer on the other. Before we pursue this dualism, I will refresh our definition of concern.

Intractable Problem:
A problem in which no easy solution has yet been found (that is, that of polynomial time or better). Another way to phrase this is as follows: An intractable problem is one where no solution significantly better than an exponential solution has yet been found.

Case 1: "The Engineer"

The engineer is making a case for practicality, a case for how the world works. The assumption here is that our computing power will not increase substantially in the years to come, and even substantial increases in computing power will not yield faster search times for it is trivial to increase the length of an RSA key to thwart vast increases in computing power. Increasing the key length of an RSA key has the effect of exponentially increasing the difficulty of cracking that particular problem. The particular assumption the engineer is banking on is the assertion that computing power cannot increase exponentially.

Case 2: "The Theoretician"

The theoretician or philosopher is making a case for possibility. It is extremely difficult to prove intractability for a given mathematical problem; the distinct possibility exists that a fast, easy solution to an intractable problem exists and can be found. Perhaps a new technique or insight into the problem is possible. The engineer largely ignores the fact that new ways of computing—quantum computation, biological computation, or molecular computation, for example—rapidly increase the computing power in an exponential way that, in theory, may provide easy solutions for some intractable problems.

So now the question arises: Is the RSA cryptosystem secure? What seemed at the outset as a relatively easy question to answer has now opened up a deep channel of philosophical quandary. Apparently, the answer to the question is decidedly dependent upon which vision of reality one has. In practice, little headway has been made in effectively breaking RSA. In December 2000, a 512-bit RSA modulus was successfully factored. This effort required around 300 computers and about 5 months of time. For every bit added to an RSA key, however, the time to factor the number increases by a measure of 1.035. Assuming this sort of constant increase in time, the time needed to factor a 1024-bit modulus lies somewhere in the range of 3 to 30 million years.

RSA Logistics

Thus far we have learned that asymmetric encryption systems are based on some fundamentally intractable problem. In the case of RSA, the intractable problem at hand is the factorization of a large number. To turn this intractable problem into a practical solution for public-key encryption requires only a handful of mathematical steps. The mathematical processes can be divided up into two processes: key generation and encryption/decryption. The key generation process and encryption/decryption process is shown in pseudo-code format in two boxes as follows:

Key Generation:

1. We begin by choosing two large prime numbers. Traditionally, these numbers are called p and q.

2. We compute the product $n = pq$. The value n is referred to as the modulus.

3. Next we must choose two more numbers. These are referred to as the public exponent (e) and the private exponent (d).

4. The value e must be chosen less than n and relatively prime to $(p - 1)(q - 1)$. The term *relatively prime* means that $(p - 1)(q - 1)$ and e have no common factors except 1.

5. The value d must be chosen such that $(ed - 1)$ is divisible by $(p - 1)(q - 1)$.

6. We now have two key values:

   The public key: (n,e)

   The private key: (n,d)

RSA Encryption andDecryption:

1. To encrypt a message m so it results in ciphertext c we use the following:

   $c = m^e$ mod n (remember, encryption is done with the public-key)

2. To decrypt a message c so it results in plaintext m we use the following:

   $m = c^d$ mod n (remember, decryption is done with the private-key)

Now that we are armed with the basic knowledge needed to execute an RSA encrypted exchange, let us practice and see how the system works in an intimate manner. We can scale down our example to such a degree as to facilitate the simplicity of calculations. We will show the example of key generation and leave the example of encryption/decryption as an exercise for the reader.

RSA Key Generation Example

1. Take two large primes:

   let $p = 37$

   let $q = 23$

   let $n = pq = 37 \times 23 = 851$

   find $(p - 1)(q - 1) = 792$

   let $(p - 1)(q - 1) = r$

2. Find the number e:

   let e be a number x such that $e < n$ and [gcd(e,r)] is equal to 1.

   let $x = 5$

   let $e = 5$

3. Find the number d:

   let d be a number y such that [$(ed - 1)$ mod $(p - 1)(q - 1)$] is equal to 0.

   [$(ed - 1)$ mod 792] = 0

   [$(5d - 1)$ mod 792] = 0

   [$(5 \times 317) - 1$ mod 792] = 0

   let $y = 317$

   let $d = 317$

   To conclude, we have now established two values:

   The public key : $(n,e) = (851,5)$

   The private key: $(n,d) = (851,317)$

The keys generated in the previous key generation sample are of course, trivial and useful only for explicative purposes. The important thing to note and remember, however, is that the public key and private key are represented not by a single string of bits, but by a tuple. Two data items comprise an RSA key: the modulus and the public or private exponent.

RSA Problems

No encryption scheme, despite how clever or ingenious, is going to be perfect in every way. The RSA encryption algorithm suffers from at least two drawbacks:

- Key generation can be very slow.
- RSA operations are much slower than similar symmetric key operations and require special padding schemes to be usable.

RSA Key Generation Issues

RSA key generation involves choosing random prime numbers that fit the specific constraints described in the previous key generation process. One aspect that is glossed over in the example is the choosing of the primes. In the previous example, we picked 37 and 23 out of the blue. These numbers are hardly large at all and are trivial. When real RSA key generation occurs, the numbers chosen will be hundreds of bits in size. Furthermore, not every appropriately sized number chosen will be prime. What must be done during RSA key generation is *primality testing*, which is an expensive operation by itself. Finding appropriately sized prime numbers that fit the constraints of the key generation process can be time consuming, even on fast processors. Because of this, key generation has the potential to take a significant amount of time if the proper primes aren't immediately found. The key generation process is a one-time performance hit and need not be repeated unless a new key pair is generated.

RSA Operations and RSA Padding Issues

The act of encrypting or decrypting with the RSA algorithm utilizes a mathematical operation called *modular exponentiation*. This operation contrasts a symmetric cipher, which usually relies on faster bit manipulation techniques. In particular, operations done with the private key for

RSA are usually much slower than the corresponding operations done with the public key. The reason for this is because the private exponent is usually much larger than the public exponent and modular exponentiation takes longer. In the key generation example, the public exponent is the value 3 while the private exponent is the value 317. This factor of a 100 translates into private key operations that are on average 13 times slower, even given current optimization techniques.

Furthermore, because of the modular exponentiation operation, the input to RSA encryption or decryption must be interpreted as a number. This constraint simply means that special padding techniques must be used to ensure that the input data is compatible with the *modular exponentiation* operation.

When no padding scheme is used with the RSA algorithm, there are two constraints that must be met for encryption to work: The total length of the data must be a multiple of the modulus size, and the data must be numerically less than the modulus. For example, consider a 1024-bit RSA key. Such an RSA key has a 128-byte modulus. Suppose next that we wish to encrypt 203 bytes of data. We would need to encrypt the data in blocks of 128 bytes and add the appropriate padding. For 203 bytes of data we would split the data into a 128-byte block and a 75-byte block with 53 bytes of padding. This, however, only fulfills the first constraint. The second constraint means that we have to prepend a 0 (zero) to the beginning of each block (and shift the pad bytes accordingly) to ensure that the numerical value of the data is smaller than the modulus size. This sort of bit manipulation is tedious and error prone, but it is the only way to encrypt large amounts of data with the RSA algorithm. This approach is sometimes referred to as *raw RSA*.

Fortunately, padding schemes exist to handle this sort of bit manipulation. The padding schemes for RSA work like magic; all that is needed is data to be encrypted or decrypted. The padding scheme ensures the data is compatible in size and numerical construction. Two important padding schemes commonly used with the RSA algorithm are PKCS#1 v1.5 padding and OAEP (PKCS #1 v2.0) padding. Both of these padding schemes are used in the XML Encryption draft. The PKCS#1 draft specifies two flavors of padding. One is called PKCS#1 Block 02 and the other is called PKCS#1 Block 01. The former is used for encryption and the latter is used for an RSA Signature.

In 1998 an attack called the Bleichenbacher attack was formulated against PKCS#1 Block 02 padding. The attack is based on well-defined checking done by the padding scheme as it checks for specific bytes in specific locations upon decryption. The attack is based on the sender sending fake RSA messages and waiting for a specific error response from the recipient. If enough fake messages (about 1 million) are sent, it might be possible to recover one encrypted message. Using a different padding scheme called Optimal Asymmetric Encryption Padding (OAEP), which was developed in 1995 by Bellare and Rogaway, can thwart this attack. We will *not* go into the details of OAEP padding in this book. Instead, the reader should visit the references section and read the standard to learn about implementation details.

The most important thing to understand about OAEP padding is that it relies on a hash function (discussed in the section on digital signatures), a mask generating function, and some seed bytes to make the padding work. These additional inputs are evidenced in the way that padding is specified in the XML Encryption draft, shown in Listing 2-3.

The reader doesn't really have to know much about XML or XML Security just yet to see how these RSA padding schemes are specified. In fact, the reader should only concentrate on the snippets in bold. The first `<EncryptionMethod>` element makes an obscure reference to rsa-1_5. This in fact specifies the RSA algorithm for encryption using the padding scheme specified in PKCS#1 version 1.5.

The second `<EncryptionMethod>` element makes the reference to `rsa-oaep-mgf1p`. This convoluted string specifies RSA encryption using the OAEP padding scheme and also specifies the *mask generating function* (mgf1p). Furthermore, the references to `sha1` and the `<OAEPparams>` element give the padding scheme a hash function to rely on as well as the necessary seed value. The thing to take away from this example is the fact that different padding schemes for the same algorithm can complicate the amount of necessary details.

Listing 2-3

RSA padding schemes in XML encryption

```
<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1 5"/>

<EncryptionMethod
  Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <OAEPparams> 65cVf9M2x </OAEPparams>
<EncryptionMethod>
```

Digital Envelopes

PKCS#1 padding for RSA has an important restriction. This padding scheme places an upper limit on the size of the data that can be encrypted. The limit is determined by the simple formula $s = k - 11$. The value s is the number of bytes that can be encrypted and the value k is the size of the modulus in bytes. This information is shown in Table 2-2.

The news of these constraints with the PKCS#1 padding scheme may alarm new readers. Some may protest, "I can't encrypt more than 245 bytes, even with a 2048 bit RSA key?" Unless raw RSA is used and the blocks are managed by the implementer, the RSA algorithm with PKCS#1 padding cannot exceed the $k - 11$ constraint. The good news, however, is that this constraint is almost never met because of something called a *digital envelope*.

A digital envelope works as follows. Instead of a recipient encrypting a large file with the RSA algorithm, the recipient uses a symmetric cipher (such as AES) for the large file and instead encrypts the symmetric key with the RSA algorithm using PKCS#1 padding. This technique always works because symmetric keys are very small. Even strong symmetric keys are only around 200 bits long, and this fits nicely inside of any of byte constraints given in Table 2-2. The XML Encryption draft calls this concept *key transport* and a diagram is shown in Figure 2-3.

In Figure 2-3 the main idea includes the encryption of bulk data with a symmetric key, and further, the encryption of this symmetric key with the recipient's public key. In the end, both data items are sent to the intended recipient. This technique of digital enveloping has the advantage of minimizing the amount of slower RSA operations and providing a more efficient means of encryption while at the same time solving the key distribution problem.

Upon receipt of the two encrypted items, the recipient first decrypts the symmetric key using the proper private key and then decrypts the bulk

Table 2-2	**RSA Key Size**	**Upper Limit for Encryption using PKCS#1 v1.5**
PKCS#1 Padding Size Constraints	512 bit	53 bytes
	768 bit	85 bytes
	1024 bit	117 bytes
	2048 bit	245 bytes

Encryption Key (E$_k$) ⟶ RSA Encryption

Bulk Data ⟶ Symmetric Cipher ⟶ To Recipient...
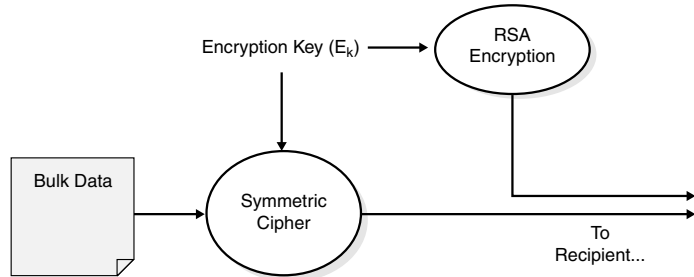
data using the appropriate symmetric algorithm. When a digital envelope is used, the RSA algorithm effectively transports the decryption key from sender to recipient, earning the name key transport. In Chapter 7 we will see how a digital envelope is packaged using the syntax and semantics of XML Encryption. The reader should take care to study the digital envelope diagram thoroughly and be sure the logisticsare understood. This idea is assumed knowledge in Chapter 7.

Key Agreement

Asymmetric encryption solves the symmetric key transport problem with the use of a digital envelope. Another way of managing symmetric keys is to rely on a *key agreement* algorithm. A key agreement algorithm is an algorithm that produces some sort of shared secret value without any prior shared secret information. In some ways, this works like magic. Once the shared secret has been decided, a symmetric cipher can be chosen and used to for encrypted communication.

Key agreement algorithms come in two flavors: synchronous key agreement and asynchronous key agreement. Synchronous key agreement refers to the act of agreeing upon a key in real time. Synchronous key agreement is ideal for a protocol situation where both parties want to immediately begin exchanging encrypted communication. During a synchronous key agreement both parties generate the shared secret at approximately the same time. This contrasts asynchronous key agreement, which assumes that the shared secret is generated at different times.

It is difficult to understand key agreement without a working example of an actual key agreement algorithm. The XML Encryption draft has support for the Diffie-Hellman (DH) key agreement protocol. This key exchange algorithm is very famous and is considered foundational for the study of public-key cryptography. Unfortunately, I will skirt the mathematical details of the algorithm and focus instead on the logistics. The details of the DH algorithm can be found inside one of the cryptography reference books listed in the references section at the end of this book.

Diffie-Hellman Key Agreement Logistics

The DH algorithm does not encrypt data or make a digital signature. The algorithm is solely used for the generation of a shared secret. To describe the DH algorithm, we will assume the existence of two parties: an originator and a recipient. These terms are used to match the same nouns used in the XML Encryption draft. I have used the term sender in previous sections to mean the same thing as originator.

The logistics of a DH key agreement can be split into three parts: Parameter Generation, Phase 1, and Phase 2. Parameter generation is the generation of a nonsecret public value called p. This value must be pre-shared, but it need not be kept secret. The parameter generation process is expensive computationally because it involves searching for a prime number in a method similar to RSA key pair generation.

The next phase of the DH algorithm is called Phase 1 and requires that the parties involved in the key exchange generate two values each: a public value and a private value. The DH private value, called x, is generated first and is a randomly chosen given a few constraints based on p. The public value, called y is generated based on x using modular exponentiation. The values x and p are mathematically related in such a way that it is intractable to determine x given p. Once each party generates their x and p, the public values are exchanged. We can be more careful with notation and use x_1, p_1 to denote the pair from the originator and x_2, p_2 to denote the pair from the recipient. When we say the public values are exchanged, we mean that the originator trades p_1 for p_2 and the recipient (by definition) makes the same trade (p_2 for p_1). This concludes Phase 1 of the DH algorithm.

Phase 2 of the DH algorithm is quite simple. The originator and recipient simply compute the following function: $z = (y')^x$ mod p. The value y' is replaced with either y_1 or y_2. That is, the originator computes $z = (y_2)^x$ mod p and the recipient computes $z = (y_1)^x$ mod p. The shared secret is the value z. Both the sender and the recipient now share a secret value that can be used as a symmetric key (or be used as a seed value to generate a symmetric key). In fact, the XML Encryption draft specifies additional computations using a hash function that mixes in other information in the generation of the shared secret. This is mentioned again in Chapter 7 and explained in the XML Encryption draft.

In any event, this is a minor implementation detail. The logistics of the key agreement remain of supreme importance here. Once the reader grasps these logistics, the details fall into place. The reader is *urged* to research the DH algorithm further, as it is rather fascinating and it almost appears magical in its operation.

The DH algorithm as just described doesn't specify any notion of time, which is important for determining if an asynchronous key agreement or real-time synchronous key agreement is occurring. Let's examine the differences with these two variations with some pictures. Consider Figure 2-4, which shows a synchronous DH key agreement.

In Figure 2-4 the steps for key agreement have been placed in temporal order. This means that the originator and recipient perform the agreement in real-time simultaneously. The process begins with parameter generation, which temporally precedes Phase 1 and can be done by the originator, recipient, or other third party. Once parameter generation has occurred, both the originator and the recipient must have access to the value p, but it need not be kept secret.

This synchronous is notion of the DH algorithm is *not* the same notion supported by the XML Encryption draft, which only supports asynchronous key agreement. The process begins by assuming a shared p value. The originator begins by fetching the recipient's public value (who may be offline or otherwise absent) and proceeds to generate the shared secret z based on the recipient's public value. From here, the originator encrypts data using z and then may send the encrypted data, and the originator's public value to the recipient. The recipient may then use the originator's public value to generate z and decrypt the message. This process is explained in more detail in Chapter 7.

Figure 2-4

Synchronous DH
key agreement
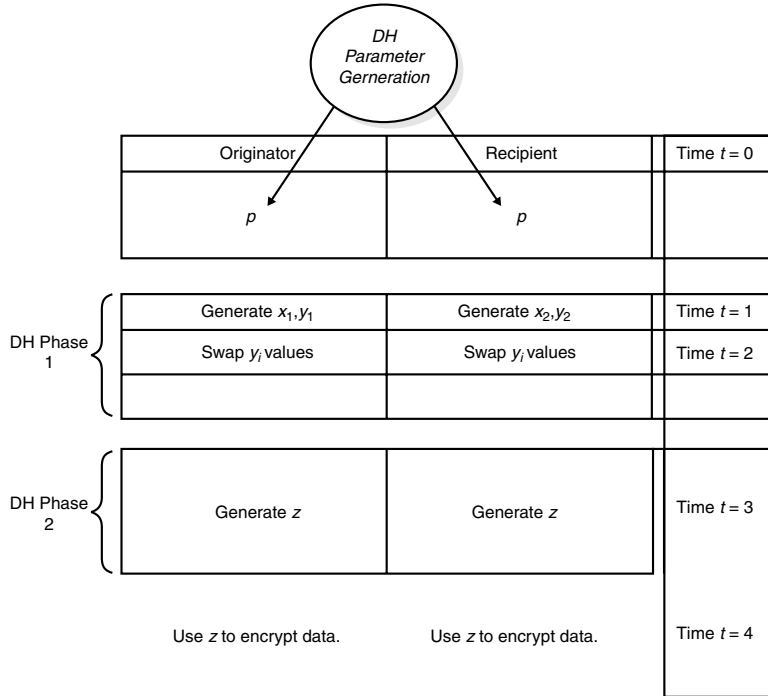
Digital Signature Basics

When we transitioned from discussing symmetric encryption to asymmetric encryption we postulated that symmetric encryption has three major usability problems. The first is the *key distribution problem,* the second is the *repudiation problem*, and the final problem is that of *data integrity*. We saw in the previous section how the key distribution problem is solved using RSA encryption or the DH key agreement algorithm. The final problems of repudiation and data integrity are solved with something called *a digital signature*. If a digital signature is combined with a key distribution solution such as RSA key transport or DH key agreement, then all three problems are effectively solved. The three signature methods used in the XML Signature draft are RSA signatures, DSA signatures, and signatures based on a hash based method authentication

code (HMAC). As we will see, HMAC is more of an authenticator than an actual signature mechanism.

All three schemes will be explained in this section. Before we can discuss the concept of a digital signature, we need to understand an underlying piece of mathematical machinery called a hash function.

Hash Functions

A *hash function* is a mathematical function that maps an input value from a domain to range, where the output range is smaller than the input domain. For example, a function that maps real numbers to integers would be a hash function. Another example of a hash function is the mod operation as used in a simple function such as $f(x) = $ x mod 5. If we assume that x is in the domain of positive integers then the only possible values for this function are 0, 1, 2, 3, and 4. The input value is the infinite domain of positive integers and the output range is 0, 1, 2, 3, and 4.

Hash functions used in digital signatures have this basic defining property, but also have two other important properties. Hash functions used in a digital signature must be collision resistant and noninvertible. A hash function that is collision resistant is one where it is intractable to find two different inputs that produce the same output value. For example, the function $f(x) = x$ mod 5 is *not* collision resistant because we can easily pick two values (*let x_1 = 20 and let x_2 = 40*) that produce the same output value (0). Further, this function is not invertible. That is, given the output value 0, there is no way to go back and compute the original input value (although guesses can be made).

There are a few hash functions specified in the XML Signature and XML Encryption drafts. The most pervasive and common hash function used in digital signatures is the SHA-1 function. The acronym stands for *Secure Hash Algorithm 1*. The function operates on an arbitrarily sized byte array and produces a fixed 20-byte value called the digest. The execution of SHA-1 has the effect of mathematically summarizing any piece of binary data into a compact 20-byte representation. SHA-1 is sophisticated, and because it is unfeasible to find a collision, we can safely assume that two byte arrays b_1 and b_2 are exactly equivalent if they have the same SHA-1 20-byte digest. Other variations of SHA-1 include SHA-256 and SHA-512. SHA-256 produces a 32-byte digest and SHA-512 produces a 64-byte digest. These hash functions can be considered more secure than SHA-1 simply because they have a smaller chance of producing a collision

(although no collisions have yet to be found in SHA-1). Those readers interested in the technical details of SHA-1 should visit one of the recommended cryptography books listed in the references section at the end of this book.

Now that the reader knows how a hash function operates, we can begin building the concept of a digital signature. The first signature scheme we will look at is the RSA Signature scheme. This is a good starting point because it uses the same previously discussed RSA algorithm, but with slightly different inputs.

RSA Signature Scheme

Suppose first we have a piece of data to be digitally signed. The signer begins by hashing the data to produce a 20-byte digest using SHA-1. This digest is then encrypted using the signer's private key.[1] The resulting RSA output is sent along with the original document. The recipient verifies the signature by decrypting the signed digest with the sender's public key. If this is successful, the original document is then hashed using SHA-1 and compared against the decrypted digest. If the digests match, the signature is valid. If there is a problem with the digests or with the decrypting of the signed digest, the signature fails. The logistics of signing are shown in Figure 2-5 and the logistics of verifying are shown in Figure 2-6.

An RSA signature provides repudiation and data integrity in two ways. The signer is the only person who can create the signature because only they have the private key. This means that a signer cannot deny that a message was signed with their private key; this is called non-repudiation and provides authenticity of the signer provided that there is an accurate

Figure 2-5

RSA Signature:
signing operation
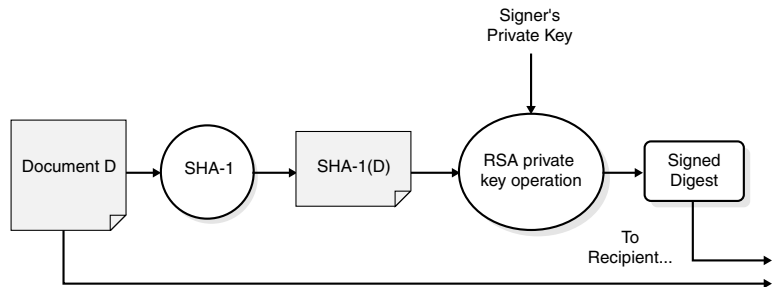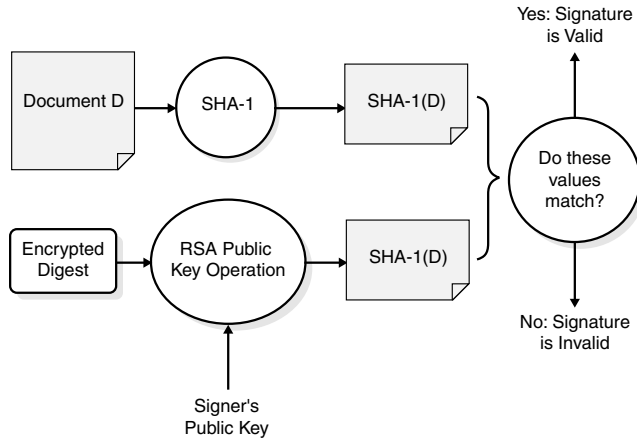
binding between the signing key and an actual individual (this is explained more in the section on certificates later in this chapter).

Data integrity for the document is provided by the combination of the RSA algorithm and the SHA-1 hash function. Because SHA-1 is sensitive to one-bit differences in an input document, even a single bit out of place will cause the signature to fail. Further, because SHA-1 is collision resistant, the chance of finding two documents that produce the same digest value is infinitesimal. Often digital signatures and encryption can be combined. One way of packaging a signed and encrypted document is to sign the document first and then package the signature and original document and then encrypt both of them. This entire package solves the key distribution problem, the repudiation problem, and the data integrity problem. This sort of mixing of technologies is seen in Chapter 7 with examples of how an XML Signature is used in conjunction with XML encryption.

An RSA signature utilizes an RSA private key operation for signing and an RSA public key operation for verifying. This means that a padding scheme must be used to ensure the input data is compatible with the mathematical constraints of the RSA algorithm. RSA signatures employ PKCS#1 Block 01 padding. This padding scheme is very similar to PKCS#1 Block 02 padding. The important thing to note here is that Block 01 padding is used for signatures and Block 02 padding is used for encryption. For details on PKCS#1 Block 01 padding, see the PKCS references at the end of this book in the references section.

The XML Signature Recommendation supports the use of an RSA signature as denoted by the following URI identifier:

```
<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
```

The reader should simply ignore the rest of the string and focus on the fragment shown in bold. This part of the URI identifier signifies that an RSA signature with the SHA-1 hash algorithm should be employed. PKCS#1 Block 01 padding is implied in the preceding algorithm specification as stated in the XML Signature Recommendation. An RSA signature works in a similar manner to DSA, which is the government approved digital signature algorithm. Both are supported in the XML Signature Recommendation and DSA is the next topic of discussion.

DSA Signature Scheme

The DSA signature scheme can only be used for signing. The DSA algorithm cannot perform encryption or decryption. There are a few mathematical logistics that must be understood to see how DSA works. In particular, there are three phases for the practical use of DSA. These phases include parameter generation, key generation, and then the signing and verifying operations.

DSA uses a public key and private key in much the same way as an RSA signature. The private key is used for generating a signature and the public key is used for verifying a signature. These keys, however, are generated from parameter values called *p, q,* and *g.* The value *p* is the prime, the value *q* is the subprime, and the value *g* is the base. Parameter generation relies on primality testing and modular exponentiation. Because of these computationally expensive tasks, parameter generation is the most time-consuming DSA operation. Once the parameters have been generated, the key generation is relatively quick. Most applications in the real world use pre-generated DSA parameters.

DSA key generation involves the choosing of a random private key called *x* and a public key (derived from *x*) called *y.* The private key is used to sign the SHA-1 hash of the input file using a procedure specified by the DSA. The output of the signature operation is a pair of numbers called *r* and *s*. These numbers represent the digital signature over the input file.

To verify a DSA signature, one must take the pair (*r,s*) and perform the verification procedure specified by the DSA. The verification procedure requires the parameters to be accessible and also relies on the 20-byte

SHA-1 hash of the input file. The specific mathematical steps are skipped in this description, but can be found in one of the cryptography books in the references section at the end of this book.

There are only a few basic things to remember about DSA. First off, because it is a government mandated signature scheme, it is often used when performing transactions with the government. Secondly, DSA parameter generation is extremely slow, but signing and verifying are quick in comparison. Finally, DSA cannot be used to encrypt or decrypt data—it is a pure signature algorithm. The XML Signature Recommendation has support for DSA via the following URI identifier:

```
<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
```

Note the fragment in bold that specifies the DSA signature scheme with the SHA-1 hash function. One can argue that this string is too verbose because the DSA signature scheme is defined to use SHA-1 by default.

The final signature scheme that will be discussed is based on a symmetric key and is called HMAC. This is another type of signature supported by the XML Signature Recommendation and is discussed in the next section.

HMAC Authentication

The term HMAC stands for *hash-based message authentication code*. An HMAC is a way of authenticating a message using a symmetric key. We are using the term *authenticate* instead of *sign*. There is an important but subtle difference between these two ideas. A signature implies that the signed document is uniquely bound to the signer in some way. For example, a private key is used to create an RSA or DSA signature and we assume that this private key by definition only belongs to one person. Consider next a symmetric key used to create a signature. A symmetric key is usually shared by at least two people. Because of the shared nature of a symmetric key, the key isn't necessarily bound to one person, but both. That is, given a message authenticated with a symmetric key, we don't immediately know (without some other context) if the sender or recipient created the message. From this point of view, a symmetric key used in the authentication of a message cannot be properly called a signature.

Now that the reader knows the difference between an authenticator and a signature, we can describe the actual logistics of HMAC.

The HMAC specified in the XML Signature Recommendation uses the SHA-1 hash function to create a 20-byte hash based on a symmetric key k and an original message M. The actual procedure is specified in RFC 2104 and involves extending k to 64 bytes and computing the SHA-1 hash (along with some other fixed values) of the key and the message. The end result is a digest value that is a function of the symmetric key and the original message. The security of HMAC comes from the fact that only the person with the correct symmetric key and unaltered message can re-create the proper authentication code. A diagram of an HMAC exchange is shown in Figure 2-7.

The curious reader may wonder why only one direction of the HMAC is shown in Figure 2-7. The answer to this question lies in the fact that the HMAC operation is the same for both the sender and recipient. HMAC is a one-way operation intended to authenticate a given message. This means that the recipient will re-create the HMAC authentication code (20-byte) value and compare this against the HMAC sent along with the original message. HMAC doesn't have a reverse verification operation—it is a one-way transformation. The XML Signature Recommendation defines the following URI identifier specifying HMAC as used in an XML Signature:

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

The reader should once again concentrate on the snippet shown in bold that denotes HMAC with SHA-1. More examples of how an XML Signature works with HMAC are given in Chapters 4 and 8.

Figure 2-7

HMAC diagram

Symmetric key *k*

20-byte Digest Value

HMAC

ABFF345609AE9254CDEF..

M

Original Message *M*
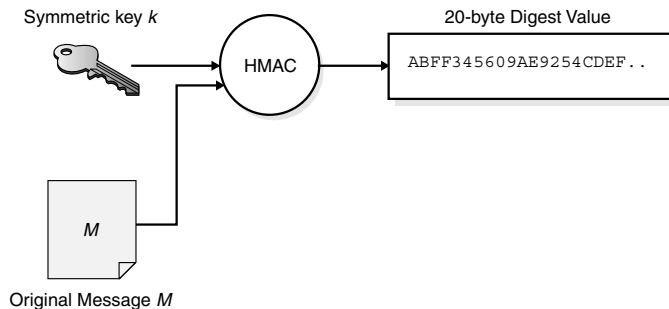
Prelude to Trust and Standardization

The reader now has a bird's eye view of cryptography as seen by the XML Security standards. The previous coverage of encryption and digital signatures represent the core technologies used by the XML Signature Recommendation and the XML Encryption draft. It is important to note that both of these standards are extensible and will likely add algorithms and features as they mature. The third and youngest technology covered in this book is the XML Key Management Specification or XKMS (see Chapter 9). As we will see, this technology brings about a certain degree of real-world usability to XML Signatures and XML Encryption. This section on digital certificates does the analogous job of XKMS for cryptography in general; that is, it brings usability to public key cryptography.

In addition to covering digital certificates, this section also answers the last question of those posed at the beginning of the chapter, which is: What is the nature of the entire process? Digital certificates are cumbersome to understand without referring directly to standards or processes, and these two topics are nicely intertwined and will be presented in lockstep.

Cryptographic processes and objects are highly standardized to ensure clarity of implementation across disparate platforms. We will take a look at the existing cryptography standards and see how they compare to the analogous emerging XML Security standards. Special focus will be on first class cryptographic objects such as keys and certificates.

Raw Cryptographic Objects

Digital certificates and cryptographic standards add usability to raw mathematical ideas. A digital signature or encrypted blob of data is completely useless without the proper context and auxiliary information. For example, suppose we want to sign an important document D using an RSA signature. What practical steps do we have to follow in order to actual create the signature? A sample outline follows:

1. Obtain the document *D* to sign.
2. Represent *D* in binary.
3. Obtain programming tools to perform RSA key pair generation.
4. Generate an RSA key pair.
5. Represent the public key and private key in binary.
6. Perform the mathematical signing operation.
7. Represent the output signature in binary.

The previous steps produce something we call a *raw digital signature*. The word *raw* is not an understatement—there is absolutely no additional context involved. We're talking pure bytes. An example of a raw RSA signature is shown in Listing 2-4.

Listing 2-4 shows a 64-byte PKCS#1 padded RSA signature. The key used to create the signature is a 512-bit RSA key. The obvious problem with Listing 2-4 is that it carries with it no additional context or information. How on earth do we know where it belongs, who created it, which document it represents, or which algorithm is used? In fact, Listing 2-4 can be almost *anything* at first glance; it's just binary at this point.

Cryptographic Standards

This is where cryptographic standards and processes make their move. Cryptographic standards are used to package and represent raw mathematical ideas such as a raw digital signature or raw encrypted data, including cryptographic keys and digital certificates. The motivation for these is obvious—how can we use cryptographic objects without some sort of standardization? At the very least we need a standard way of packaging cryptographic objects and a standard way of encoding objects to survive different computing environments. XML Security is simply the

---

Listing 2-4

A raw digital signature

```
63 EF 96 B2 A9 29 38 73 70 22 BD F3 89 DB C3 43
DD 97 CC F1 8B 86 9B 7F B4 39 3D 37 D4 44 E3 12
79 40 9A 3E F5 F2 A6 51 75 5F CE 44 DD 71 7A 9C
97 43 A6 44 83 78 20 C4 9D 17 5B 0B F0 BC 54 F8
```

introduction of XML-based standards for the XML representation of these same cryptographic objects.

ASN.1 and Binary Encoding

Applied cryptography is based on binary format objects that are represented using something called ASN.1. ASN.1 stands for *Abstract Syntax Notation One*. We will not give details about ASN.1 in this book, but it is fundamental for understanding applied security. The reader is urged to visit the references section for more information about ASN.1.

A commonly posed question is: So what is ASN.1? The short answer is that ASN.1 is a language used to describe transfer syntaxes. The phrase *transfer syntax* is used to describe some intermediate language used between disparate computing environments. For example, consider a computer *A* that uses the EBCDIC character encoding scheme and a computer *B* that uses ASCII. If we were to transfer a raw digital signature from machine *A* to machine *B*, the signature would not verify on machine *B* because the encoding is different and the ASCII representation will correspond to a different binary representation.

This problem of disparate computing environments affords the general solution of a transfer syntax. The idea works like this: Machine *A* encodes the raw digital signature into an intermediate format *C* and then sends this format *C* to machine *B*. Machine *B* is expecting format *C* and decodes this properly into the appropriate binary without wrecking the signature. This intermediate format *C* is defined using the language of ASN.1. In some respects, ASN.1 is similar to XML because it provides a way to send data in a portable manner.

Let's see what ASN.1 looks like. Listing 2-5 shows an RSA public key encoded using ASN.1.

Listing 2-5

An example of an ASN.1 encoded public key

```
SEQUENCE {
  SEQUENCE {
    OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
    NULL
    }
  BIT STRING 0 unused bits
    30 46 02 41 00 DF 02 C3 6D 8A 55 8E FD FC 24 C8
    41 42 45 9E 62 9D D0 54 30 13 89 66 14 F6 C2 95
    AB DF 87 F8 F3 57 7D 5B 4E F6 BB 8A A5 98 F3 F1
    8F C2 0B 16 18 24 79 17 24 27 B4 51 39 9D 26 7F
    AB 45 98 EA 1F 02 01 11
  }
```

Listing 2-5 uses the syntax of ASN.1 to create a type for an RSA public key, which is called the `SubjectPublicKeyInfo`. This particular structure is defined in RFC 2459 and is a nested structure. The outer context gives an object identifier. This is the string `rsaEncryption` and the corresponding number 1.2.840.113549.1.1.1. This tells us that the key is an RSA public key and not some other sort of key. The BIT STRING contains the actual key value, which is actually two values: the public exponent and the modulus. (See the section on key generation for a refresher on the mathematical components of an RSA public key.)

The structure shown in Listing 2-5 doesn't answer all of the questions about ASN.1. The most compelling question is: How is Listing 2-5 encoded? That is, there must be a way to take the structure shown and transform it into binary. This ASN.1-to-binary transformation is done with something called BER, which stands for *Basic Encoding Rules*. If one were to apply BER encoding to Listing 2-2, it would produce the output shown in Listing 2-6.

Listing 2-6 represents a public key that is ready to be transferred across disparate platforms. The idea here is that this binary blob is sent to anyone who can decode BER and that individual will obtain the exact same byte representation once the ASN.1 structure is decoded. Many objects are encoded using ASN.1 and BER. For example, Listing 2-7 shows a BER encoded private key.

Listing 2-7 shows an encrypted private key. This particular structure is called an `EncryptedPrivateKeyInfo`. This type of structure is used to protect a private key that may be on disk or in transit. Private keys are considered to be extremely sensitive and are usually encrypted to prevent tampering and discovery. We will see in Chapter 8 how actual encrypted private keys are used in the Cert-J toolkit.

Base-64 Printable Encoding

BER encoding provides a useful portable data type for cryptographic objects, but often times binary is difficult to deal with. Text-based proto-

---

Listing 2-6

BER encoded public key

```
30 5A 30 0D 06 09 2A 86 48 86 F7 0D 01 01 01 05
00 03 49 00 30 46 02 41 00 DF 02 C3 6D 8A 55 8E
FD FC 24 C8 41 42 45 9E 62 9D D0 54 30 13 89 66
14 F6 C2 95 AB DF 87 F8 F3 57 7D 5B 4E F6 BB 8A
A5 98 F3 F1 8F C2 0B 16 18 24 79 17 24 27 B4 51
39 9D 26 7F AB 45 98 EA 1F 02 01 11
```

Listing 2-7

BER encoded
RSA private key

```
SEQUENCE {
  SEQUENCE {
    OBJECT IDENTIFIER
      pbeWithSHAAndDES-CBC (1 2 840 113549 1 5 10)
    SEQUENCE {
      OCTET STRING
        00 11 22 33 44 55 66 77
      INTEGER 5
      }
    }
  OCTET STRING
    03 DB BF 95 98 61 B5 1A 64 68 DC C9 FF 87 08 0B
    D1 3C 67 FA 43 A6 67 72 95 C5 B3 1A 35 C5 03 93
    0B 10 DC D7 3E 17 20 EF 6C 02 7C C2 A8 D8 9D 0C
    BE AF 41 53 46 41 26 1E 3E E4 89 69 D6 90 08 8D
    E7 B9 6A 3E 9F FE D4 1A 54 FE 66 1E 08 A6 11 95
    F8 F8 21 41 93 3F EB AC 2C D9 C6 D0 1E 3B 9A 5A
    9E 71 87 A8 E9 3E AD B5 3F 00 E4 F9 11 AE CD 5F
    3E 38 35 61 FC 16 A9 B1 7B C4 2E 03 FE B6 48 F5
    ...
  }
```

cols such as MIME skirt the problem of printable binary representation with a printable encoding. The most popular printable encoding scheme is called Base-64 encoding and has been defined in RFC 1421, RFC 1521, and RFC 2045—the latter being the RFC that describes part of the MIME protocol. The encoding algorithm specified in these RFCs represents binary data as a subset of 64 printable ASCII characters. This encoding is useful for security applications that have a need to transport some sort of binary cryptographic object via a text-based protocol. We can encode any sort of binary data into printable characters using Base-64 encoding. For example, Listing 2-4 encoded with Base-64 is shown in Listing 2-8.

For a recipient to verify this signature, they must first Base-64 decode the string to obtain the raw digital signature. We can even take this a step further and encode BER encoded binary cryptographic objects. This will add another layer of encoding and force the BER encoded binary into a printable representation. A Base-64 encoded `SubjectPublicKeyInfo` is shown in Listing 2-9.

Listing 2-8

Base-64 encoded
signature value

```
Y++WsqkpOHNwIr3zidvDQ92XzPGLhpt/tDk9N9RE4xJ5QJo+9fKmUXVfzkTdcXqcl0
OmRIN4IMSdF1sL8LxU+A==
```

```
MFowDQYJKoZIhvcNAQEBBQADSQAwRgJBAOS6CHiE6r/38yblxc/3lnk8lZbHcaQE
MY5yNqnZSdAIWeJz8ryn/ymrY/xbqtarGmwbxGBUd4R3jXWBCrhn44kCARE=
```

The summary of the conceptual transformations undergone by the public key is outlined in Figure 2-8.

The most important thing to understand about Figure 2-8 is the fact that the actual mathematical object (public key) has undergone drastic changes. *Most of the usability problems of applied security come from a misunderstanding of how cryptographic objects are represented.* Once this public key leaves Computing Environment A, it must be properly decoded in order to arrive at the original public key value. As the reader may have guessed, there are ASN.1 representations of many cryptographic objects, including digital signatures, signed data, keys, and digital certificates. These various cryptographic objects are defined in RFCs and the PKCS standards. PKCS standards are denoted with the name PKCS#? where the ? is a number identifying a particular standard. To date there are 12 published PKCS standards, most of which specify cryptographic algorithms, processes, and cryptographic objects. Some of the most useful PKCS standards are listed in Table 2-3.

Figure 2-8

Encoding lifecycle of an RSA public key


```
MFowDQYJKoZIhvcNAQEBBQADSQAw
RgJBAOS6CHiE6r/38ybl
xc/3lnk8lZbHcaQEMY5yNqnZ
SdAIWeJz8ryn/ymrY/xbqtar
GmwbxGBUd4R3jXWBCrhn44kCARE=
```
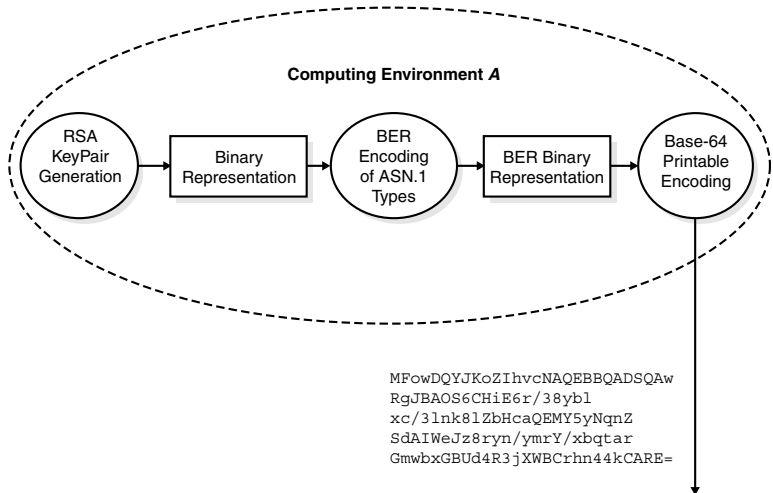
	PKCS Standard	Description
Table 2-3	PKCS#1	RSA encryption and signing processes, RSA key formats, and cryptographic primitives
Some Important PKCS Standards	PKCS#7	Extensible cryptographic messaging syntax used for packaging digital signatures, digital envelopes, encrypted data, and data that is signed and encrypted.
	PKCS#8	Private Key Storage
	PKCS#12	A portable format for transporting private keys, certificates and other miscellaneous cryptographic objects.

Of the standards listed in Table 2-3, PKCS#7, is important to mention because some may argue that it competes directly with the XML Signature Recommendation and the XML Encryption draft. PKCS#7 defines an extensible structure that enables cryptographic operations on arbitrary data, such as a piece of data that is signed and encrypted. In addition, it also has the mechanism to transport keys and information about the signer. We will see these various aspects and features present in an XML Signature in Chapter 4. Two important differences between PKCS#7 and XML Signatures or XML Encryption are as follows:

- PKCS#7 is a binary format whereas XML Signatures and XML Encryption use XML.
- PKCS#7 is one standard that provides support for both signed and encrypted data simultaneously.

XML Signatures and XML Encryption support signing and encryption separately, but can be combined to provide support for both. The details of the PKCS standards are not the subject of this book, but are covered in other security books. See the references section at the end of this book for more information on the PKCS standards.

The last major subject of this chapter concerns digital certificates and the notion of public key binding. This subject is important to understand for Chapter 9, when the XML Key Management Specification is discussed.

Trust, Certificates, and Path Validation

When considering whether to validate a digital signature or send an encrypted message to a recipient, there must be absolute certainty regarding the binding of the public key to an identity. If this binding cannot be trusted, then the action itself is meaningless. For example, suppose that we would like to validate a signed document that claims to be signed by John Doe. The type of signature used is irrelevant at this point—it could be an XML Signature or a raw digital signature. The act of verifying trust is independent of the signature packaging mechanism.

Upon inspecting the signed message, the first step is to obtain the signer's public key (here we are assuming that the signer is using a public key cryptography system) if not already included. Most signature packaging standards enable the inclusion of a public verification key within the signed message itself. We will see more of this when we examine XML Signatures in Chapter 4 and Chapter 5. In our fictional scenario here, let us assume that John Doe's public key is included within the signed message.

Once we obtain the public key, we can repeat the signing algorithm using this public key. At this point we are merely verifying the cryptographic binding of the public key and nothing further. This test, however, is not meaningful if we don't know for certain that the public key we are using actually belongs to the purported signer John Doe. The signature is only meaningful if there is a verifiable link between the name John Doe and the public key that is used to successfully verify the signed message. To make this example clear, consider a rogue signer named Sally who wishes to impersonate John Doe. Sally generates an RSA key pair and signs a message using her private key. She then packages the message with the name John Doe along with her public key. The recipient will receive the message and attempt to perform cryptographic verification. This cryptographic verification will be successful because the proper public key has been included (even though it belongs to Sally, and not John). In short, if the recipient doesn't verify the *binding* between the name John Doe and the included public key, the signature verification operation is meaningless.

In a traditional PKI, the binding between a public key and a recipient is represented with a digital certificate. At the very least, a certificate is a signed data structure that contains someone's name and public key and asserts that these entities are bound together. In addition, it contains a

validity interval that states how long the certificate is valid for. The validity interval is used during path validation.

Like other cryptographic objects, a digital certificate is represented in ASN.1 and then Distinguished Encoding Rules (DER) encoded. DER is a subset of BER encoding which ensures a unique encoding for certain ASN.1 types. There is only one certificate format that we will concern ourselves with in this book, which is the ASN.1 structure defined in RFC2459. Sometimes, a certificate will be placed inside a PKCS#12 message and this is considered a digital certificate. This, however, is incorrect. The PKCS#12 format is simply a container for certificates and other objects.

Some may wonder how this binding is proven. For example, a certificate can be fabricated which claims to bind the name John Doe to Sally's public key. Nothing we have said thus far will stop anyone from making this assertion. The reason this can't be done is because some trusted third party known as the certificate authority (CA), must verify the binding between a name and a public key. Once the trusted CA verifies that the binding is indeed correct, the certificate is digitally signed. The verification of the binding of the name and public key isn't a cryptographic operation, but usually a physical operation where the CA actually verifies that a physical individual is bound to a public key. Such verification is done through public records or a face-to-face interaction.

From this point on, anyone who wishes to verify the binding between the name and the public key in a certificate must first believe that the trusted CA has done its job correctly. If the CA who signed a given certificate is trusted, it then follows that the binding between the name and the public key in the certificate is also trusted. When this scenario occurs, we can say that trust *emanates* from the trusted CA. This simple case is shown in Figure 2-9.

Figure 2-9 shows the high-level process of what happens when a digital certificate is created. At first, the binding between the name John Doe and the public key is only a purported binding. This is shown with the dashed line. Our fictional certificate authority, called Trusted CA 1, verifies the binding and creates a digital certificate. Notice that the arrow between the name and public key is solid. The assumption here is that Trusted CA 1 actually went through the trouble of making sure that the name John Doe is actually bound to the correct public key. The digital certificate contains an issuer name (Trusted CA 1) and a subject name John Doe which indicates that this issuer is the one responsible for asserting
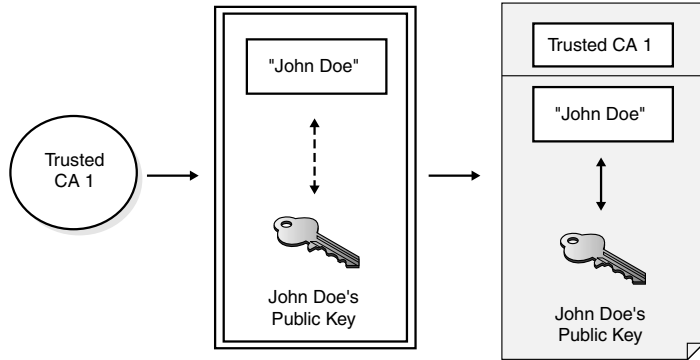
the binding between John Doe and John Doe's public key. Finally, the entire entity is digitally signed with the signing key belonging to Trusted CA 1.

The scenario shown in Figure 2-9 is vacuous and is almost never seen in the real world. Instead of a single trusted CA, it is often the case that many intermediate certificate authorities form what is called a *certificate chain*. This picture is similar to Figure 2-9, except for the fact that John Doe's certificate isn't signed by Trusted CA 1, but is instead signed by some other certificate authority. The implication here is that if we trust the top of the chain, we also trust all of the intermediate certificate authorities and finally trust the name and key binding of the end-entity. This is shown in Figure 2-10.

In Figure 2-10 we don't have to explicitly trust every intermediate certificate authority. Instead, trust emanates from Trusted CA 1 and by implication the end of the chain is trusted. That is, we can be confident of the binding between John Doe and his public key simply by trusting the actions of Trusted CA 1. In Figure 2-10, the *issuer name* is the top name in each certificate and the *subject name* is the bottom name in each certificate.

The trust concept is similar to a prime mover idea, where only one entity begins a chain of trust. The concept of a certificate chain is a repeated theme when dealing with digital certificates and signed documents. It is often the case that a signer will have its identity verified with a chain of certificates rather than just a single certificate and single trusted certificate authority. In addition to asserting trust, a mechanism called a *Certificate Revocation List* (CRL) lists certificates that are no

Figure 2-10

A conceptual certificate chain
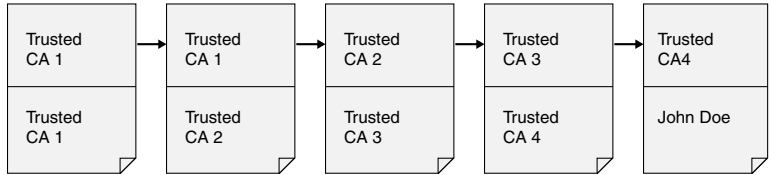
longer trusted (revoked). The trusted CA signs the CRL, and in this sense the trusted CA can also remove the public key binding with the CRL mechanism, therefore negating trust. The mechanics of a CRL are usually not timely. A CRL cannot be published in an instantaneous manner, and it is possible for a revoked certificate to be used if its use predates the appearance of the next CRL.

Most of the details about certificate path validation, certificates, and CRLs are found in RFC 2459. An entire series of books can be written about RFC 2459 and because of this, most of our discussion will focus on conceptual basics. Rigorous definitions can be helpful when discussing trust and certificates. The first definition that we will look at captures much of the previous discussion in a concise matter and comes directly from RFC 2459.

Certification Path

A certification path is a sequence of n certificates such that:

- For every certificate x, in $[1, (n - 1)]$, the subject name of certificate x matches the issuer name of certificate $x + 1$.
- The first certificate in the chain is the trusted certificate authority ($x = 1$).
- The last certificate in the chain is the end-entity certificate ($x = n$).

After reading this definition, re-examine Figure 2-10 and notice that the picture shown matches the three constraints made explicit in our definition of an intractable problem. Figure 2-10 shows a sequence of five certificates, and for each certificate 1 through 4 (1 through $n - 1$), the subject

name and issuer names match. The careful reader may wonder why the last certificate in the chain is not considered in this first constraint. This is because of the way the definition is written. The nth certificate is the last certificate in the sequence, and because of this there is no $x + 1$th certificate to compare to. The next constraint simply tells us which certificate is the trusted certificate authority (this is the first certificate) and the final constraint tells us which certificate is the end-entity certificate. The term end-entity is used to describe any end user or end user system acting as the subject of a given certificate.

Path Validation

Given a certification path, a number of steps must be followed to actually validate and ensure trust over this path. These steps are numerous and can become tedious and complicated. The complexity of path validation is one of the contributing factors to the complexity of PKI in general. Certificate path validation begins with a number of inputs, which are enumerated in the following list.

1. A certification path of length n
2. The current date and time, T_c
3. An arbitrary time T for which to check against
4. A set of acceptable policy identifiers

The first item in the previous list is a certification path. This is the same entity found in the intractable problem definition and can be a path consisting of just two certificates (a trusted CA and an end-entity certificate), or it can be a larger chain. The size of the certification path is irrelevant.

The next two inputs that must be available during path validation are date and time values. The value T_c represents the current date and time whereas the value T is an arbitrary time against which the path should be validated. In most cases $T = T_c$, but it is often useful to roll back the value of T to accomplish a path validation check for some time in the past.

The final input is a set of policy identifiers. A policy identifier is just a number that represents some sort of arbitrary policy or constraint under which a certificate has been issued. For example, some organizations that employ certificates for their users will often have difference flavors or lev-

els of certificates. Certificate policies can be used to differentiate between evaluation certificates, code-signing certificates, or certificates that have stronger semantics, such as hardware-based private key storage.

Once a certificate is issued under a specific policy, the policy identifier and optional qualifier information is appended to the digital certificate as an extension. Certificate policies are entirely fictional and are invented ideas and constraints that map to an arbitrary number. The reason they are used in path validation is to designate acceptable certificate policies so it is possible to provide more control over how path validation occurs. Furthermore, certificate policies may also be mapped to one another. For example, if two organizations create similar policies and wish to equate their policies, they can be mapped to one another so they mean the same thing during path validation. Certificate policies add considerable complexity to the path validation process because of their arbitrary and invented nature; the number of certificate policies or their meanings is potentially limitless.

Path Validation State Machine

The path validation processed is defined as a state machine. Each node in the state machine represents some sort of constraint that needs to be checked. The approach taken here is to simplify the state machine to only hit on the high points of the path validation process. This simplification can be accomplished by combining many constraints into a single state. For example, we will summarize the path validation state machine in three states: state A, which consists of checking basic certificate information; state B, which consists of checking policy information, and state C, which consists of the checks done against the CA certificate.

Each certificate in the certification path travels through the state machine, but only the trusted CA certificate goes through the checking done in state C. A picture of the state machine is shown in Figure 2-11.

In Figure 2-11, the xth certificate is checked by first examining some basic certificate information such as the signature, revocation, and name chaining. If all of the constraints that comprise state A successfully pass, the certificate moves on to state B, where the certificate policy extension is checked against the initial and acceptable certificate policies. If this is successful and the xth certificate is not the top of the chain (that is, not a trusted CA certificate), the process ends for this certificate and the $x + 1^{th}$
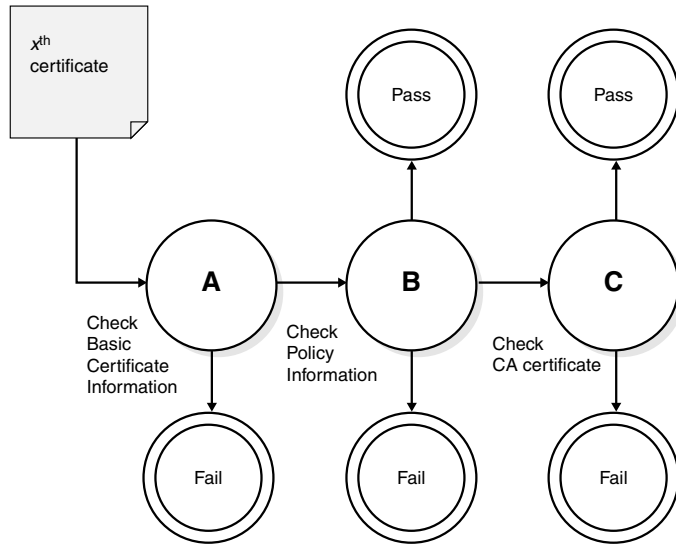
certificate in the certification path is checked. If the certificate *is* a trusted CA, then the constraints that comprise state C are checked. State C will normally be entered when the first certificate in the chain is processed because processing always begins with some sort of trusted certificate.

This high-level process hides numerous details, some of which are described in the explanation of each of the states in the text that follows. Path validation is an intensive task, and as the number of certification paths and trusted CAs grows, the potential exists for a great deal of heavy processing. In each state, the assumption is made that we are operating on the xth certificate.

State A

The first process required in state A is to check and see if the xth certificate was signed using the public key from the previous certificate in the certification path. This is the $x - 1^{\text{th}}$ certificate. If there is no previous certificate (for example, if the first certificate is the trusted CA), then this check is skipped. The check here involves a cryptographic signature verification

operation where the signature on the xth certificate is verified with the appropriate public key. This integrity check prevents the certificate from being altered or corrupted.

Following this, the validity, revocation, and name chaining are all checked. Of these three checks, the revocation check has the potential to be quite expensive. When a certificate is checked for revocation, it is compared to the values in a recent certificate revocation list (CRL), which is simply a signed list of revoked certificates. The CRL can be obtained from a remote server or can be cached locally. CRLs have the potential to grow in size as more certificates are revoked over time. The growth of the CRL, however, won't be unbounded because certificates fall off the CRL when they expire. Real-time status checking is also possible where the xth certificate is passed off to a service that determines the revocation status. This is a typically a more accurate way of determining certificate revocation because the status information obtained is more timely.

The certificate validity is checked when the validity interval in the xth certificate is compared to the current time (T_c) and the name chaining check involves ensuring that the issuer of the xth certificate is the subject of the $x - 1^{th}$ certificate. This must be true for all certificates in the certification path except for the trusted CA that will be self-signed.

State B

This state is where policy constraints are checked and policy mapping is performed. The certificate policies extension is parsed and the policy numbers (OIDs) are compared to the set of initial and acceptable certificate policies. In general, this state is complex due to the policy-mapping algorithm and various checks that must be made. Certificate policies and how they are used falls out of scope for our purposes here. The curious reader who wishes to learn more about certificate policies should visit the references section for more information.

State C

This state is entered for all certificates but the end-entity certificate. This state represents checks to ensure that the certificate is an acceptable CA certificate and meets certain basic properties. Some of things checked include certain certificate extensions as well as additional policy information.

Authorization

Once the certification path is deemed valid and acceptable, additional extensions that may be present in the end-entity certificate are checked. Any other constraints over and above certificate path validation falls into the scope of authorization. Only when the certificate is trusted can it be further examined to see if it meets a specific purpose or use case.

Additional Information

The highlight of this section is to illustrate that path validation and trust is a *complicated* task that involves many steps. Most of the additional details such as certificate extensions and certificate policies can comprise an entire book on their own. There is far too much detail to present here. The complexity of path validation is important to understand when we discuss the XML Key Management Specification (XKMS) in Chapter 9. XKMS offers to move the problem of path validation and trust to a service in an attempt to simplify the processing done by the client application.

Chapter Summary

This chapter begins with a quick introduction to symmetric key encryption. In particular, fundamental questions about the nature of encryption were posed. Understanding the breadth and depth of these fundamental questions provides the reader with a good background for understanding applied security in general. The discussion of symmetric key encryption includes descriptions of padding schemes (which alter the plaintext) and feedback modes (which add security features to a symmetric cipher). The two symmetric ciphers discussed include Triple-DES and AES. These ciphers are currently supported in the XML Encryption draft.

The next topic of discussion for this chapter is asymmetric encryption. We made the distinction between the RSA algorithm for encryption and similar technologies that also rely on asymmetric keys such as RSA signatures, DSA signatures and the Diffie-Hellman key exchange algorithm. Following asymmetric encryption was a discussion of digital signatures including RSA signatures, DSA signatures, and HMAC authenticators.

BER encoding and its relation to ASN.1 is also discussed, including comparisons to raw digital signatures and raw encryption. A discussion of how Base-64 encoding is used to encode binary cryptographic objects is given. The chapter concludes with a lengthy description of public key binding and trust as it relates to digital certificates, including the rationale for path validation.

This page intentionally left blank.

XML Primer

The world of Extensible Markup Language (XML) is an endless ocean of standards and technologies that mimics a living being, constantly changing and ever evolving. Trying to capture the breadth and depth of XML in a single chapter is a hopeless task with no end. The goal here, then, is to focus on a few anchors in the XML ocean that represent *fundamental concepts*. There is little point in repeating details that can be found in the text of a published standard. Instead, the reader will find an explanation of fundamental XML concepts bolstered with real-world examples. Special focus is on building block technologies that provide the groundwork for the sea of XML.

This chapter is divided into two broad topics, an introduction to basic XML syntax and a preliminary discussion of XML processing. The division between syntax and processing is a theme that will be revisited throughout this book. The core XML *syntax* topics discussed include the basics of well-formed documents, markup concepts, some information about namespaces, and numerous examples. XML *processing* is discussed with the presentation of two topics: the Document Object Model (DOM) and the XPath data model. The seasoned reader with previous experience with XML might find this chapter a bit repetitive, but XML syntax and processing is a necessary building block for XML Security.

What Is XML?

This question reminds me of an assignment once given in high school where the task was to answer the broad question: "What is History?" As youngsters we were quick to respond with simple answers such as "History is what happened in the past," or for those of us who were really lazy, we would provide the proctor with Webster's definition (and no doubt receive a poor grade).

The question "What is XML?" is a loaded question that has no simple answer. It almost seems like any simple answer given would be akin to answering the "What is History" question with the same naiveté of someone in high school. For XML, the answer to this question often depends on the audience. There are books on XML for managers, software developers, sales representatives, and marketing people. Here we will take the viewpoint of a software engineer or computer scientist as we attempt to create yet another definition of this technology.

Meta-Language and Paradigm Shift

The topic of XML Security requires viewing XML from a slightly different angle compared to other technologies that leverage XML to accomplish its goals. XML Security is devoid (thankfully) of *presentation semantics*. That is, the current XML Security specifications don't focus on rendering or displaying an XML Signature or encrypted XML element. In this respect, XML Security technologies are more closely related to existing security technologies such as the Public Key Cryptography Standards (PKCS), discussed in Chapter 2. Don't look for any HTML or JavaScript code in this book, because it is simply not the focus of the XML Security specifications.

Two vocabulary words that are especially useful in defining XML from an XML Security standpoint are *meta-language* and *paradigm shift*. The noun *meta-language* best describes the *what* part of XML Security and the verb *paradigm-shift* helps describe the *where* part of XML Security. Thankfully, the *how* part is left to the dedicated authors of the XML Security Recommendations and Drafts.

Meta-Language

XML is not a language. Despite the name Extensible Markup Language, it is easiest to understand XML as a meta-language. What exactly does

this mean? A *meta-language* is a language used to describe other languages. Some would call this a true language. The prefix "meta" used in this sense has its roots in analytic philosophy and loosely means one level up or above and beyond.

So what does this mean for you? In simple terms, XML is a *syntax* used to describe other markup languages. The XML 1.0 Recommendation as released by the W3C defines no tag set or language keywords. The skeptical reader can check for himself or herself, but simply put, only a syntax and grammar is defined by the XML 1.0 Recommendation. The term *syntax* here refers to a set of constraints about how and where tags can be placed, and the acceptable range of characters that are legal, as well as the rules for markup in general. Moreover, the basic rules and syntax of XML 1.0 are deceptively simple and can be learned in the better part of an hour.

At this point, no examples have been provided, so the assumption is that the reader is as lost as ever. The next question is: "If XML is a syntax used to describe other markup languages, what are these *other markup languages?*" The answer to this question is the sea of XML-related standards and, more importantly, the XML Security Standards. For example, the XML Signature Recommendation defines a *markup language* used to represent a digital signature; the XML Encryption Draft defines a *markup language* used to represent encrypted elements. Similarly, markup languages such as MathML or DocBook are also *other* markup languages that are defined in accordance with the syntax put forth by the XML 1.0 Recommendation. MathML is a markup language for representing mathematics and DocBook is a markup language used for representing articles or books.

The final part of our XML definition relates to how exactly these other markup languages are defined: XML is a syntax used to describe other markup languages *using markup*. This seems like a circular definition— of course markup languages use *markup*. A short example of arbitrary markup is given in Listing 3-1.

The preceding listing is arbitrary data with tags around it. For example, we have a piece of data, `Samuel Adams`, that is marked up with the tag `<Good_Beer>`. The important point here isn't what the tags say or even what they mean; instead, the focus should be on *what the tags can do*. Tags provide *markup* and *markup* can accomplish many different things with regards to *data*. In fact, the list of things that markup can accomplish is so important that it belongs in a box.

Listing 3-1

Example of
arbitrary markup

```
<Food>
  <FrenchFries> Curly Fries </FrenchFries>
  <Beers>
    <Good_Beer> Samuel Adams </Good_Beer>
    <Good_Beer> Guinness </Good_Beer>
    <Bad_Beer>  Budweiser </Bad_Beer>
    <Bad_Beer>  Fosters </Bad_Beer>
  </Beers>
</Food>
```

The Roles of Markup

- Markup can add semantics to data.
- Markup can demarcate data.
- Markup can define roles for data.
- Markup can define containment.
- Markup can define relationships.

With the careful use of *tags* around *arbitrary textual data,* we can accomplish almost any sort of semantics desired. We can invent tags and structures, give various roles to data, and define relationships between tags. The power of *markup* is nearly limitless for carving up any sort of data. This property of markup is especially powerful because the basic concept behind markup is simple. It doesn't take years of practice to begin using markup, nor is the syntax complicated and difficult to understand. It is this combined simplicity and power that makes XML an exciting technology.

Because XML is a meta-language, every use of XML and markup to add semantics to data results in the creation of a markup language. For example, if someone were to look at Listing 3-1 and ask the question: "What language is that?" the correct answer is: "It's a fictional markup language that uses the syntax of XML." In short, Listing 3-1 actually defines its own markup language. It uses the tags `<Food>`, `<FrenchFries>`, `<Beers>`, `<Good_Beer>`, and `<Bad_Beer>`. While the tag set is small and rather useless, it is a valid markup language. Before we move into the specifics of XML syntax, we will examine the other high-level defining component of XML as it relates to XML Security: *paradigm-shift.*

Paradigm-Shift

XML Security represents a clear paradigm-shift from ASN.1-based, binary standards toward more portable, text-based XML solutions. Most of the entities in the security world that relate to *cryptography* and public-key infrastructure (PKI) use ASN.1 types to encode various entities. When use the term *cryptography,* we are really referring to *applied cryptography* that takes a standards-based approach—real cryptographers probably do much of their work with pencil and paper.

Examples of these binary format security standards include X.509 certificates or most of the PKCS standards—all of these use ASN.1 types to encode their pieces and parts. Examples of these formats have already been given in Chapter 2, but the paradigm shift that XML Security promises is a shift from BER encoded ASN.1 "objects" to the analogous XML structures. A clear example of this shift is seen with the way a verification key (usually a public key) is represented in an XML Signature. As discussed in Chapter 4 and Chapter 8, the `<KeyValue>` tag (just more markup) is used to represent a raw public key that can be used for decryption. This is shown in Listing 3-2. This is contrasted with the X.509 `SubjectPublicKeyInfo` introduced in Chapter 2 and discussed again in Chapter 8, shown again in Listing 3-3.

Listing 3-2

The `<KeyValue>` element from an XML signature

```
<KeyValue>
  <RSAKeyValue>
    <Modulus>
      s3mkTQbzxuNFPFDtWd/9jvs8tF5ynBLilbG/sT24OglEol
      1PBvRe+VUJU0eI2SRhN/KtZv4iD2jwT0Sko0eeJw==
    </Modulus>
      <Exponent>EQ==</Exponent>
  </RSAKeyValue>
</KeyValue>
```

Listing 3-3

A binary `SubjectPublic-KeyInfo` interpreted with an ASN.1 parser

```
SEQUENCE {
  SEQUENCE {
    OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
    NULL
    }
  BIT STRING 0 unused bits
    30 46 02 41 00 B3 79 A4 4D 06 F3 C6 E3 45 3C 50
    ED 59 DF FD 8E FB 3C B4 5E 72 9C 12 E2 95 B1 BF
    B1 3D B8 38 69 44 A2 5D 4F 06 F4 5E F9 55 09 53
    47 88 D9 24 61 37 F2 AD 66 FE 22 0F 68 F0 4F 44
    A4 A3 47 9E 27 02 01 11
  }
```

Both Listings 3-2 and 3-3 are showing us the same *datatype,* an RSA public key. Both structures clearly demarcate the type of key as well as the modulus and the exponent. Listing 3-3 does this a bit more covertly, as the modulus and exponent are encoded inside the BIT STRING. Listing 3-2 uses XML syntax and *markup* while Listing 3-3 uses ASN.1 and BER. Both encoding schemes are intended to be extensible and both encoding schemes have tradeoffs. The XML version is certainly user-friendly and one might argue that it is easier for an application to parse text data with XML markup. The binary version, however, is far more compact (once encoded in binary, half as small or smaller than the equivalent XML version), but, as some would argue, harder for an application to parse. Some would even be appalled at the space wasted by the XML version. The difference between Listings 3-2 and 3-3 is the paradigm shift. For better or for worse, emerging XML Security standards have a strong "ASN.1 hate factor" and instead opt for cryptographic objects to take form as semantically clean, easy to read, and easy to parse XML variations.

Now that the reader has some basic high-level notions about what XML is (perhaps fuzzy notions), we can begin our descent from abstract high-level ideas to more concrete, tedious details.

Elements, Attributes, and Documents

Three important terms for describing basic XML syntax are *elements, attributes,* and *documents*. All three of these terms are special and important; they encompass a large portion of the conceptual playing field for XML and provide the foundation for the remainder of this chapter as well as the entire book.

Elements and Attributes

We have spent some previous discussion throwing around the term *markup*. A small example was given, and the reader should have seen some *tags* with stuff inside, but little else should be evident besides the fact that markup is useful for the intellectual carving of data.

Markup is often divided into two separate vocabulary words when we are talking about XML: *elements* and *attributes*. An element is a *start tag* and an *end tag* including the stuff inside of it and an attribute is a simple name-value pair where the value is in single or double quotes. An

attribute cannot stand by itself and must be inside the start tag of a given element. Two short examples follow:

```
<Food> Ice Cream </Food>
<Food Flavor="Chocolate"> Ice Cream </Food>
```

The first line in the previous example is an element called `Food` that has `Ice Cream` as its element content. The second line in the previous example is the same element with a name-value pair added to it (this is an attribute). The name is `Flavor` and the value is `Chocolate`. Elements may also be empty, having no element content. This is shown in the example that follows:

```
<HealthyFood></HealthyFood>
<HealthyFood/>
```

The first line in the previous example is an element called `<Healthy-Food>` that has nothing inside of it. The second line is *shorthand* for the same empty element. Take note of this shorthand notation, because it is used pervasively in many XML documents. This notation can be confusing at first, but in all cases, it simply means an empty element.

Attributes may be used arbitrarily within start tags to add more meaning to the data. In fact, there was a great deal of contention over the inclusion of attributes within the XML syntax. The reason is because any data that can be modeled with elements alone can also be modeled with an attribute-centric approach and vice versa. Consider the following short example. This example contains the same data as Listing 3-1, but it is modeled almost entirely with attributes.

```
<Food FrenchFries = "CurlyFries"
      Good_Beer1  = "Samuel Adams"
      Good_Beer2  = "Guinness"
      Bad_Beer1   = "Budweiser"
      Bad_Beer2   = "Fosters"
</Food>
```

The markup used in the preceding example is certainly clumsier than Listing 3-1, but the point here is that we are *roughly* modeling the same data, but using attributes instead. There are five attributes inside the `Food` element and they provide us with the same basic information as shown in Listing 3-1. So which one is better? Both are legal XML documents; the answer to this question is really an answer to a much more complicated *data-modeling question*. The convention, however, is to use

attributes more sparingly than elements. Elements and their contents usually represent concrete information that will be displayed or rendered, while attributes usually represent information required for processing. This dichotomy, however, isn't strict or formal and there isn't anything written down that says that this is how it must be. This is just convention. XML Security-based standards use attributes heavily for algorithm information and data sources while elements are used for concrete cryptographic objects, such as keys or signature values. For example, consider the short example that follows:

```
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

This `<DigestMethod>` empty element uses an attribute called `Algorithm` with the rather long value `http://www.w3.org/2000/09/xmldsig#sha1`. This element is commonly seen in an XML Signature, and the meaning of the attribute value is intended to be the SHA-1 hash function. Notice that even though the element is empty, it still communicates information via the attribute. There is no requirement that all useful elements must have content—this use of an empty element with an attribute is frequently seen in the XML Security world. This shouldn't mean much to the reader at this point; these details will be hashed out in Chapter 4 and Chapter 5.

XML Documents

Throughout this book and throughout many of the XML Security standards, reference is made to something called an *XML document*. This term has a specific meaning and carries with it some implicit properties, the most notable of which is the *well-formed* property. This property is the most basic set of constraints that can be put on data represented using XML; it defines simple syntax rules for the legal positioning of elements and attributes. The reader is now asking, "So, what does *well-formed* mean? What are these constraints?" In short, the list of constraints for a well-formed document follows—again placed in a box because of their importance.

There is a bit of hidden detail here, but not much. Let's examine these four constraints and go through some examples.

Data represented in XML is well-formed if . . .

- There is exactly one *root element.*
- Every *start tag* has a matching *end tag.*
- No tag overlaps another tag.
- All elements and attributes must obey the naming constraints.

Root Element

The root element constraint is perhaps the easiest to see and understand. Simply put, any data that wants to be well formed must have *exactly* one root element. This means there must be one (and only one) *parent* element. The root element has a synonym called *document element.* Sometimes we use the term *root element* and sometimes we use the term *document element*. These refer to exactly the same thing; sometimes one just sounds better! Listings 3-4 and 3-5 are examples of data that do *not* have a single root element, while Listing 3-6 and Listing 3-7 correct these examples to make them well formed. The additional root elements have been added in bold.

Listing 3-4

Sample XML data without a root element (not well formed)

```
<Dark_Chocolate>
  <Brand1>Hersheys</Brand1>
  <Brand2>Ghiradelli</Brand2>
</Dark_Chocolate>
<Ice Cream>
  <Brand1>Ben and Jerry</Brand1>
  <Brand2>Dryers</Brand2>
</Ice Cream>
```

Listing 3-5

Sample XML data without a root element (not well formed)

```
<Student> Joe </Student>
<Student> Bob </Student>
<Student> Mary </Student>
```

```
<FatteningFoods>
  <Dark_Chocolate>
    <Brand1>Hersheys</Brand1>
    <Brand2>Ghiradelli</Brand2>
  </Dark_Chocolate>
  <Ice Cream>
    <Brand1>Ben and Jerry</Brand1>
    <Brand2>Dryers</Brand2>
  </Ice Cream>
</FatteningFoods>
```

```
<Students>
  <Student> Joe </Student>
  <Student> Bob </Student>
  <Student> Mary </Student>
</Students>
```

Start Tags and End Tags

The next constraints, start tags and end tags, are also simple and easy to see. For every start tag, there must be an associated end tag. Listing 3-8 shows the incorrect data and Listing 3-9 corrects this data by adding the proper end tags.

Again, this constraint is quite simple. It is easy to see by inspection if end tags or start tags are missing—the syntax so far is just not complicated. Let's move along to the last two constraints.

Overlapping Tags

Tags cannot overlap each other in such a way that one tag is closed before another tag. This constraint is difficult to describe with clarity, but sufficiently easy to see in an example. Consider the next small example:

```
<Student>
  <SSN>123-45-6789</Student>
</SSN>
```

The previous example is in a lot of syntactic trouble—not only is there no clear root element, but the <Student> tag is closed before the <SSN> tag. To fix this, you have to be sure that the elements do not overlap. The

Listing 3-8

Sample XML data missing some end-tags (not well formed)

```
<Candy>
  <Good_Candy>
    Milk Chocolate
  <Bad_Candy>
    Dark Chocolate
</Candy>
```

Listing 3-9

Sample XML document (well-formed data)

```
<Candy>
  <Good_Candy>
    Milk Chocolate
  </Good_Candy>
  <Bad_Candy>
    Dark Chocolate
  </Good_Candy>
</Candy>
```

following example shows how you can fix this to make it an XML document that is well formed.

```
<Student>
  <SSN>123-45-6789</SSN>
</Student>
```

Naming Constraints

Most of the tedious details of the well-formed property are contained within the naming constraints for elements and attributes. These constraints are much more broad than the previous constraints because they limit the range of acceptable characters for elements as well as some details on white space. The least you need to know is contained in the box that follows.

Naming Constraints

- Element names must begin with a letter or underscore.
- Element names cannot contain embedded spaces.
- Element names are case sensitive.
- Attribute names must be unique per element (start tag).
- Attribute values must use single or double quotes.
- Attribute values cannot contain a < character.

Again, these naming constraints are still not that complicated. Listing 3-10 shows the gratuitous violation of every rule listed in the previous box. The idea is to get the reader in tune with what the simplest legal XML documents look like, and one way of reinforcing this is to look at the illegal use of XML data.

The reader is challenged to scan Listing 3-10 and make an attempt to find six naming constraint violations. The idea here is that well-formed XML documents are not complex; the syntax is simple to learn and quite intuitive.

If the reader has grasped the previous concepts with regard to XML documents, the well-formed property and markup (elements and attributes), most of the battle is already won. At this point, the reader should have the conceptual tools to build the *simplest possible legal XML documents*. We have made it this far with only three well-defined terms: documents, elements, and attributes. Once the reader can create and use XML documents, it is possible to begin playing with and understanding the pieces that comprise XML Security.

Aside from arbitrary binary data, any sort of XML data that is signed or encrypted must be, at the very least, an XML document. Another point that should be made here is that XML documents can be created using only the simplest of tools; a text editor is all that is required. No browsers or fancy tools are necessary. What we are looking at here is pure data, carved up with markup and limited only by the well-formed constraint.

As a final exercise, consider Listing 3-11 that shows an XML Signature. Using only the basic rules of XML syntax, the reader should have enough information to determine if Listing 3-11 is a legal XML document, despite the fact that the reader should have little or no knowledge of the meaning of any of the elements or attributes used.

Listing 3-10

Sample XML data that violates almost every naming constraint (not well formed)

```
<4Root_Element>
  <Message> This is an invalid element </message>
  <Another Message> This is also an invalid element </Another Message>
  <Note color="blue" color="green"> These are colors </Note>
  <Note color=red> I forgot this color </Note>
  <Note lt="<"> Less than sign </Note>
</4Root_Element>
```

Listing 3-11

XML data that is
not well formed

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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/2000/09/xmldsig#sha1"/>
      <DigestValue>szlrBmSpQUJCO/ykyhS126/xMOM=<DigestValue>
    </Reference>
  </SignedInfo>
<SignatureValue>
  UOmRz+EiYhy5LEsZ+fXBKnHlzWpJ+HFCOQlhWdb
  I/DVlv7Szt11BEfn8fpC4bxG19UDd7MbpRedi
  3qUeVP+GSvMElrPo8u++KsHMKGsaPoqeUOoUI
  bW7biuW1rMSYpESdUeWbmy2p2/P8sulMouHrT
  q+Jv92GQ+itjLimhmHLTs=
</SignatureValue>
</Signature>
```

The solution is, of course, the erroneous `<DigestValue>` element. There are two opening tags for this element but no corresponding closing tag.

The URI

One potential confusing aspect of XML documents is the pervasive use of URIs. A URI is a *Uniform Resource Identifier* and is a short string value intended to identify something on the Web—whether it is a file, a service, or a person. In fact, a URI can identify *any* resource that has identity. An example of a URI is the string value `http://www.rsasecurity.com`.

Listing 3-11 includes five URIs within the markup. At this point, it's not clear what they are used for (other than the fact that they are *attribute* values), and they tend to clutter the markup because of their length.

Most readers have encountered URIs that identify Web resources such as Web pages that use the HTTP *scheme*. This is the same string that is pasted into a browser and used to visit your favorite Web site, and in this case data is retrieved from the location (Web pages, graphics, and so forth). The URIs used in Listing 3-11 differ in that not all of these URIs are used as a *data source*. That is, in the context of Listing 3-11, some of

the URI values are not meant to be retrieved, but instead are meant to be used as *identifiers*. This may seem odd because most URIs seen outside of XML documents are meant to be de-referenced and used as a data source.

As we examine various XML technologies we will see the distinction between those URIs used mainly as identifiers versus those URIs that are simply meant to be sources of data. We will see that any sort of URI can be used as a pure identifier, even if it happens to point to some real data. The first example of URIs used as identifiers occurs in the following section on *Namespaces in XML*.

Namespaces in XML

Another conceptual hurdle is the topic of *namespaces* in XML. The purpose of a namespace when used in an XML document is to prevent the collision of semantically different elements and attributes that happen to have the same name. For example, suppose that an author of an XML document wants to use an element called `<Fans>`. This particular element can refer to the noun *fan* as in a ceiling fan, or it can refer to the noun *fan* as related to an attendee at a sporting event. Still more problems occur if this element, ostensibly created by two different authors, is merged into a single XML document. Clearly, the meaning of the element is ambiguous and it is unreasonable to expect any sort of application to be able to make the distinction between the different elements without some other qualifying information.

This problem is solved with the use of a *namespace*. A namespace in the context of XML is simply a collection of element and attribute names identified by a *URI Reference*. The intention of an XML namespace is to provide a globally unique name for an element or attribute. The previous sentences should mean precious little to the reader without some examples. Before the examples, a point of clarification must be made. The term *URI Reference* may be confusing —what is a URI Reference? A URI Reference is a URI that is used as a *string identifier*; it has no purpose beyond this. Some may argue that this term is redundant and confusing, but it is the nature of the URI in these examples; simply an identifier.

A pertinent example is the namespace used for an XML Signature. The URI Reference is `http://www.w3.org/2000/09/xmldsig#` and the collection of element and attribute names that are associated with this string identifier include the elements and attributes that help define an XML Signature. The specifics of the elements used in an XML Signature

are given in Chapter 4 and Chapter 5. The conceptual picture looks something like Figure 3-1.

The idea behind the namespace is quite simple, but the syntax to use a namespace inside an XML document can get quite confusing because there are multiple ways to accomplish the same thing. Let's work through some short examples. First consider Listing 3-12, which shows an XML document using elements from the XML Signature namespace *without* any sort of namespace qualification. Let's not worry about the contents of

Figure 3-1

The XML Signature namespace and its related elements


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

    <Signature>

<SignatureMethod>                <Transforms>

                    <CanonicalizationMethod>

    <Reference>
                        <SignedInfo>
<DigestMethod>

            <DigestValue>

                <SignatureValue>
```
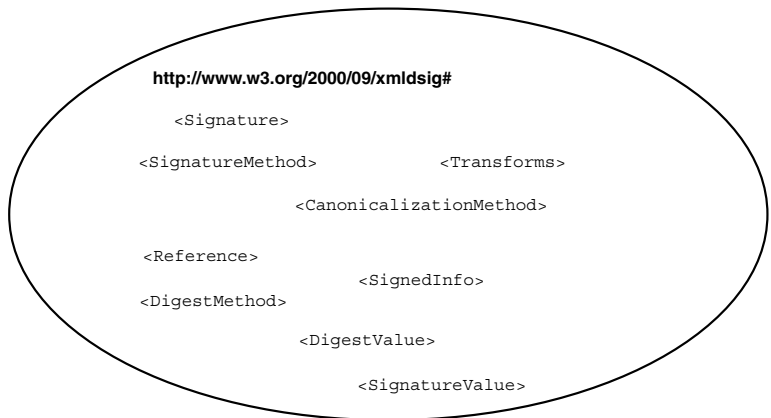
Listing 3-12

An XML document using elements from the XML Signature namespace *without* an explicit namespace qualification

```
<Signature>
  <SignedInfo>
   ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

the elements right now; this is not essential to understanding how the namespace declarations work.

Listing 3-12 is ambiguous as far as the namespace is concerned. This XML document uses elements called `<Signature>`, `<SignedInfo>`, and `<SignatureValue>`, but doesn't tell us where these element names came from. It is possible at this point that these element names belong to another XML document (not an XML signature). Listing 3-12 is akin to using the `<Fans>` element indiscriminately, leaving the semantics ambiguous. An XML namespace is declared with some use of the `xmlns` attribute. This is shown in Listing 3-13.

The syntax shown in bold in Listing 3-13 declares a *default namespace* for all of the elements in the XML document, including the root element. This means that all of the elements in this document (unless otherwise noted) all belong to the `http://www.w3.org/2000/09/xmldsig#` namespace. This syntax for namespaces is perhaps the most straightforward and easiest to see; it shows how to associate a single namespace within an XML document. We can throw a wrench in the example to see how the syntax becomes more complicated. Consider Listing 3-14.

Listing 3-14 poses a problem because it uses two elements, `<Fans>` and `<CeilingFans>`, that are not part of the XML Signature namespace. We need some way of marking these elements as part of a different name-

---

Listing 3-13

An XML document using elements from the XML Signature namespace using a *default namespace declaration*

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
</Signature>
```

Listing 3-14

An XML document with an improper element for the default namespace

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <Fans>
    <CeilingFans> 4 </CeilingFans>
  </Fans>
</Signature>
```

space, such that a processing application can make the proper distinction. This is where the syntax gets a bit more complicated.

What we need to do is declare an additional *namespace prefix*. The namespace prefix is an arbitrary string (usually short) associated with a given namespace. This prefix is declared as another attribute value inside the parent element for which the prefix is to be valid. This is shown in Listing 3-15.

The *namespace prefix* chosen for Listing 3-15 is the short string `foo`. By using the namespace prefix and the colon separator, we can associate a particular element with a given namespace inside the element itself. This is shown in Listing 3-15 when we declare the `<Fans>` element is a member of `http://fans.com` by naming the element `<foo:Fans>`. This idea is difficult to describe with much clarity, but it is easy to see with an example. The string `foo:Fans` is called the *qualified name* and consists of the *namespace prefix* (`foo`) and the *local part* (`Fans`).

XML documents that combine multiple technologies (such as a document that uses elements from the XML Signature syntax and XML Encryption syntax) will have to deal with declaring the appropriate namespaces and qualifying any elements used. For most of the discussion and examples in this book, namespaces are not shown because they add to the syntactic clutter and can cloud the understanding of some of the basic concepts. That being said, they are an *absolutely essential* part of XML and XML Security because without them there is no way to separate different technologies and elements used within a given context. Further, there is still much more to learn about namespaces! The previous discussion is only a primer and gives the reader just enough ammunition to understand the examples and usage within this book. The reader is urged to visit the references section for places to go to read more about namespaces.

Listing 3-15

One way of using two namespaces in an XML document

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#"
           xmlns:foo="http://fans.com">
  <SignedInfo>
    ...
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <Object>
    <foo:Fans>
      <foo:CeilingFans> 4 </foo:CeilingFans>
    </foo:Fans>
  </Object>
</Signature>
```

More Markup

Elements, attributes, and namespaces represent the most fundamental types of XML markup. Additional markup constructs that should be briefly mentioned are *comments, processing instructions,* and *character data sections*. While not too terribly important for the scope of this book, they are fundamental parts of an XML document and are used in the discussion of the Document Object Model (DOM), which is discussed in the second half of this chapter.

Comments

The idea of a comment for a programming language is pervasive for anyone who has compiled even the simplest program. The idea is exactly the same for XML documents; there is a way for document authors to inform others of what is going on inside their twisted minds. The syntax is simple and a brief example follows. The text between the `<!--` and `-->` delimeters is a comment:

```
<!--  Comment on This!  -->
```

Processing Instructions

A processing instruction is intended to be a customized instruction to the processing application. Processing instructions are currently not widely used in the XML Security standards and can be safely ignored for the most part. The syntax of a processing instruction is the two character delimiter `<?` followed by a *target* and then an *arbitrary data string* and finally the closing delimiter `?>` The key idea about processing instructions is that they are predefined and application specific. For example, if you need to enumerate a certain section of markup as significant for a custom application, you might denote this with a processing instruction. An example of a processing instruction follows:

```
<? application_processor_1 do_task ?>
```

The reason processing instructions are even mentioned here is because they show up in the DOM, which is discussed in the second half of this chapter.

Character Data Sections

XML has some restrictions on certain text characters; for example, you can't create an element with the following markup characters:

```
<Expression> 4 < 5 </Expression>
```

The obvious reason for this is because the processing application that parses the XML document can't determine where elements begin and end. With markup delimiters in the markup itself, it can't match up the < and > characters properly. To remedy this, XML defines what are called *predefined character entities* that are used to represent markup characters used in element and attribute names. These are shown in Table 3-1.

While these are useful, they can become cumbersome in practice. To remedy this, the CDATA section is used to add unparsed text to an XML document. The identifier CDATA stands for *character data*. The syntax looks sort of weird, but the idea here is to be able to add arbitrary data to the XML document without having to worry about using the predefined character entities. A CDATA section looks like this:

```
<Expression> <![ CDATA [ 4 < 5 ]] > </Expression
```

The idea is that you place the text in between the second [(left bracket) and first] (right bracket). This example is semantically equivalent to the first. The CDATA section doesn't have to be used for *just* unparsed text. It can be used for any arbitrary text that the parser should ignore.

More Semantics: The Document Prolog

By now the reader should have a fairly good grasp of XML basics. Some familiarity with namespaces as well as XML document basics allows the

Table 3-1	Predefined Character Entity Name	Value
Predefined Character Entities	>	>
	<	<
	"	"
	&	&
	'	'

reader to understand most of the content of any given XML document. There are, unfortunately, more details that must be hashed out. Most of the following details will not be pursued much further in this book, but they are *essential* to understanding and using XML documents outside the scope of this book.

Most of these details come to us as part of something called the *document prolog*. The term used is quite amusing; one definition for the term *prolog* is an introduction to a play or a novel. Unfortunately, XML documents are never as exciting as live entertainment or a good read, but the term is accurate. The document prolog is an optional set of declarations used to add semantics to the current XML document. The document prolog always precedes the root element in an XML document and carries with it some specific syntax constraints. To provide some motivation for the document prolog, let's look at some examples. Consider Listing 3-16.

Listing 3-16 is a portion of an XML Signature document. The names of the elements are unimportant; the only important thing is the structure of the elements and the attributes and attribute values. Notice that Listing 3-16 is littered with URI strings. This doesn't represent anything too odd—most XML documents use URIs as attribute values. The URI strings are valid attribute values and don't do anything sinister other than make Listing 3-16 more difficult to read.

What if we had a way to remove these URI strings from the markup to make the document more readable, but retain the URI values somehow? A properly constructed document prolog with the necessary declarations allows us to accomplish this. Let's give the solution first and then talk about it. See Listing 3-17.

Some might argue that Listing 3-17 is *more* difficult to read than Listing 3-16. This is only because we haven't discussed all the funny syntax yet. Listing 3-16 adds *general entity declarations* for the URI values—simply put, replacement text. To see this, consider the isolated entity declaration shown next:

```
<!ENTITY  dsig  "http://www.w3.org/2000/09/xmldsig#">
```

Listing 3-16

A portion of an XML Signature document

```
<Reference xmlns="http://www.w3.org/2000/09/xmldsig#"
           URI="http://www.rsasecurity.com" >
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
```

Listing 3-17

The use of *general entities* in an XML document

```
<?xml version="1.0"?>
<!DOCTYPE Signature
    [
      <!ENTITY  dsig  "http://www.w3.org/2000/09/xmldsig#">
      <!ENTITY  alg   "http://www.w3.org/2000/09/xmldsig#sha1">
      <!ENTITY  val   "http://www.rsasecurity.com">
    ]
>
<Reference xmlns="&dsig;" URI="&val;" >
  <DigestMethod Algorithm="&alg;"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
```

The value on the left (dsig) is replaced with the string value on the right in the actual markup. To specify the *general entity* within the markup, the & and ; syntax is used to delimit the general entity name. These are marked with bold in Listing 3-17. The use of general entities within the markup can make the XML document much easier to read, especially when there is a large amount of text that must be repeated throughout a document. Some of the XML Security standards make use of general entities to provide for more readable examples.

All three of the general entities shown in Listing 3-17 are inside the *document prolog*. The document prolog consists of some mandatory pieces, which at the very least must include the *XML declaration,* and the *document type declaration*. The XML declaration is the string value <?xml version="1.0"?>. If a document prolog is included in an XML document, it *must* begin with this declaration. The XML declaration does little more than communicate the version number along with some other semantics such as the *document encoding* and whether the XML document refers to external files. The XML declaration is shown in Listing 3-17 on the first line.

Immediately following the XML declaration is the document type declaration. The document type declaration is designed to provide a grammar for the given XML document. It begins with the string <!DOCTYPE, is followed with a string (Signature), and must end with a closing >. Inside the document type declaration is where *markup declarations* such as our *general entities* belong. (The general entities are actually inside the *internal subset*, but we'll ignore this detail for now.) The reader should observe the syntax of both the XML declaration and the document type declaration. These declarations do not represent well-formed XML; that is, the syntax is special and doesn't use the element and attribute syntax shared

by the main markup. We will return to this point later and see what is being done to alleviate this funny syntax.

At this point the reader should have a basic grasp of the syntax of the document prolog. Aside from the *general entity* (replacement text) declarations, the document prolog usually fulfills a more dignified role as the chief mechanism for assigning a *formal grammar* for the current XML document via something called a *document type definition*. This term is deceptively similar to document type declaration, but refers to something a bit different. The *document type definition* (DTD) is the *collection* of internal and external resources (internal to the current XML document) that collectively provide a *formal grammar* for the XML document. This topic is discussed in more detail in the following section.

Document Type Definition (DTD)

This section discusses the document type definition (DTD), which is the set of rules and constraints for providing a *formal grammar* for an XML document. First we will look at a simple case of a DTD with a fictional markup language. Following this we will examine parts of a real DTD and see if we can make our way through it.

The DTD

In an earlier section the reader was challenged to consider Listing 3-1 and answer the question: "What language is that?" The answer was: "It is a *fictional* markup language that uses the syntax of XML."

This fictional language has a fairly well defined syntax described in the well-formed constraints for XML. It is easy for us to construct legal documents because we have no rules other than those imposed by the XML Recommendation. For example, consider Listing 3-18 that uses the same elements as Listing 3-1.

Listing 3-18

A fictional language that uses XML with no additional constraints

```
<Good_Beer>
  <Food> Samuel Adams </Food>
  <FrenchFries> Guinness </FrenchFries>
  <Beers>
    <Bad_Beer> Budweiser </Bad_Beer>
    <Bad_Beer> Fosters </Bad_Beer>
  </Beers>
</Good_Beer>
```

The reader should be staring blankly at Listing 3-18 and probably wondering why the elements don't make any sense. This is intentional—the syntax of XML is extensible. In fact, with only the well-formed constraints, one can argue that the syntax is *too* extensible. Why? Right now we can create semantically meaningless element combinations that amount to gibberish. We have created a fictional markup language, but at this point we have no way to constrain it in any meaningful way. We can't yet tell *valid* XML documents from *invalid* XML documents for our particular language. We don't have a measure for validity yet so we don't know if Listing 3-1 or Listing 3-18 is legal or illegal even though they are both *well-formed*.

The term *valid* has a special meaning in the context of XML. An XML document is said to be valid if it has been compared against a formal grammar and has not violated any parts of this grammar. This grammar is the document type definition and is usually a file somewhere that contains the rules for a particular markup language. Once the document type definition has been created, we can associate it with a given XML document via the document type declaration. To provide some motivation for this, let's create a simple grammar for Listing 3-1 that allows us to discern *valid* and *invalid* instances of our fictional markup language. Let's call this particular markup language the *Food* language, where our three food groups consist of good beers, bad beers, and french fries—a true diet of champions. Listing 3-19 shows an example of a simple document type definition that lives in a file separate from the main markup file.

The set of constraints shown in Listing 3-19 is roughly equivalent to the following English description: A document that uses `<Food>` as its root element must contain exactly two elements, `<FrenchFries>` and `<Beers>`. The `<FrenchFries>` element must contain some character data and the `<Beers>` element must contain at least one `<Good_Beer>` element or `<Bad_Beer>` element. Both a `<Good_Beer>` and `<Bad_Beer>` element must contain character data.

The syntax used for the document type definition can be quite intuitive, even though it looks weird. Without explicitly defining each line, the

Listing 3-19

Some constraints for the *Food* markup language

```
<!ELEMENT Food (FrenchFries, Beers)>
<!ELEMENT FrenchFries (#PCDATA)>
<!ELEMENT Beers (Good_Beer+ | Bad_Beer+)>
<!ELEMENT Good_Beer (#PCDATA)>
<!ELEMENT Bad_Beer (#PCDATA)>
```

reader should be able to follow the given English description and deduce the meaning of most of the notation used. The fancy name for each line in Listing 3-19 is *element declaration*. Each of these element declarations defines constraints for the element name. Let's look at the first element declaration:

```
<!ELEMENT Food (FrenchFries, Beers)>
```

Each element declaration begins with the string `<!ELEMENT` followed by the element name, and then the *content-model* and finally the closing character `>`. In the previous small example the element name is `Food` and the content-model we are declaring for this element is a sequence of child elements in a specific order. The order is denoted by the order of the element names within the parentheses. The first child element must be `FrenchFries` and the second (and final) child element must be `Beers`. The next element declaration is even simpler:

```
<!ELEMENT FrenchFries (#PCDATA)>
```

The previous example simply says that the `FrenchFries` element must contain only character data—it cannot contain any elements. The odd-looking keyword #PCDATA stands for *parsed character data*. The third element declaration gets a bit more complex, but it is still readable:

```
<!ELEMENT Beers (Good_Beer+ | Bad_Beer+)>
```

The content model here says that the `Beers` element must contain a *choice* (denoted by the | character) of *one or more* (denoted by the + symbol) `Good_Beer` or `Bad_Beer` elements. Cardinality operators such as +, ?, and * are used throughout document type definitions and mean *one or more, zero or one,* or *zero or more,* respectively. We will see these cardinality operators again in Chapter 4 when we look at the structure of the XML Signature. The last two element declarations are simply repeats of the second declaration and merely constrain the `Good_Beer` and `Bad_Beer` elements to only contain character data.

The reader should have a basic understanding of how to put together a simple grammar for a custom markup language. The reader is challenged to reconsider Listing 3-1. Is this XML document valid? Does it conform to the grammar set forth in Listing 3-19? The correct answer is no. The reason is because the `<Beers>` element contains more than one child element—it contains two `<Good_Beer>` elements and two `<Bad_Beer>` elements. Our document type definition constrains the `<Beers>` element

to a *choice* of either one, but not both. This is the constraint defined by the third element declaration.

Now that we have a simple document type definition, we need to associate this with an instance of our *Food* markup language. This is done so a processing application can find the formal grammar and perform the check in a seamless way. Luckily, the syntax to accomplish this association is not difficult. A complete, valid, *Food* XML document that points to an external document type definition is shown in Listing 3-20.

Listing 3-20 uses what is called a *system identifier* (denoted by the SYSTEM keyword) to inform the processing application of the document type definition. In this case, the DTD file lives somewhere on a server called food.com. While a remote URI is shown in Listing 3-20, any valid URI can be used for the system identifier value. A filename by itself usually signifies that the DTD file is in the same local directory as the XML document.

A Real DTD

This section looks at pieces of the DTD for an XML Signature (the entire DTD is boring). The reader doesn't have to know much about an XML Signature yet—the details will be covered in Chapter 4. This section is intended to give the reader some practice reading through a real DTD instead of a fake markup language about beer and french fries.

The first piece we are going to look at is the element declaration for the parent element, which is the `<Signature>` element, as follows:

```
<!ELEMENT Signature (SignedInfo, SignatureValue, KeyInfo?, Object*)>
```

Luckily, this element declaration is simple and is similar to a declaration made in the DTD for the *Food* markup language. The declaration says that a `<Signature>` element must contain a `<SignedInfo>` element, `<SignatureValue>` element, zero or one `<KeyInfo>` element, and zero or more `<Object>` elements. The beauty of this declaration is that it is

---

Listing 3-20

A valid instance of the *Food* markup language

```
<?xml version="1.0"?>
<!DOCTYPE Food SYSTEM "http://food.com/food.dtd">
<Food>
  <FrenchFries> Curly Fries </FrenchFries>
  <Beers>
    <Good_Beer> Samuel Adams </Good_Beer>
  </Beers>
</Food>
```

fairly easy to read and we don't yet have to know a thing about an XML Signature to understand the constraints. Here is another piece, which puts a constraint on the contents of the `<SignatureValue>` element such that it only contains character data and no elements:

```
<!ELEMENT SignatureValue (#PCDATA)>
```

Here is another piece of the XML Signature DTD that gives constraints for the `<SignedInfo>` element:

```
<!ELEMENT SignedInfo (CanonicalizationMethod,SignatureMethod,Reference+)>
```

The constraint simply says that a `<SignedInfo>` element must contain a `<CanonicalizationMethod>` element, a `<SignatureMethod>` element, and one or more `<Reference>` elements.

Learning More about DTDs

These small sections only give the reader the absolute basics with regard to DTDs, and the reader is urged to consult the references section at the end of this book for more information. Some might argue that we made the discussion easier than it really is, and in some cases we have simplified a few things. We omitted *attribute declarations* and *parameter entities*, both of which are found in the actual XML Signature DTD. An attribute declaration is similar to an entity declaration, but it provides constraints for attributes instead of entities and a parameter entity is similar to a general entity, but is intended for use inside a DTD.

XML Schema

Like document type definitions, the XML Schema definition language is used to describe the structure of XML. One view of XML Schema is an overhaul or upgrade of DTDs. Newcomers to XML often get frustrated by the fact that there are two tools for accomplishing roughly the same task. The frustration continues when it is also learned that DTDs are being replaced with corresponding XML Schema definitions.

The DTD faces two major problems:

- DTDs aren't powerful enough to provide sophisticated constraints on an XML document.
- DTDs don't *use* XML.

The first issue sounds plausible, but the second needs a bit of explanation. Reconsider the DTD syntax, specifically any of the previously discussed element declarations. The careful reader should notice that the syntax doesn't correspond to well-formed XML (that is, syntax in which element declarations are made in between an opening < ! and closing > character). There is no end tag for the element declaration. The syntax of DTDs is a bit ironic; it can't be processed in the same way as the document it constrains. There is really no reason for this. The XML syntax is powerful and general enough to model constraints on instance documents that use XML. Switching to an XML-based markup language that constrains XML document instances is inevitable and makes processing the formal grammar easier. In addition, XML Schema is *extensible* in contrast to the DTD language, which is fixed and part of the XML 1.0 Recommendation.

We will not go into any specifics of XML Schema in this book; the topic is simply too large and complex to fit inside a single chapter, let alone a single section. To give the reader a flavor for what XML Schema looks like, we will present part of the Schema definition from the XML Signature Recommendation for a quick tour. The reader should visit the references section for more information on XML Schema. Consider Listing 3-21.

Fortunately, XML Schema is quite intuitive and extensive knowledge of the schema definition language is not required for discerning some basic constraints. XML Schema is much more general in its scope than DTDs and has the features of a programming language.

The two basic *types* in XML Schema are *simple types* and *complex types*. In general, complex types are types that contain other elements while

Listing 3-21

The XML Schema Definition for the `<Signature>` element

```
<element name="Signature">
  <complexType>
    <sequence>
      <element ref="ds:SignedInfo"/>
      <element ref="ds:SignatureValue"/>
      <element ref="ds:KeyInfo" minOccurs="0"/>
      <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
      <attribute name="Id" type="ID" use="optional"/>
  </complexType>
</element>
```

simple types cannot. In Listing 3-21, the element `<Signature>` is defined to be a complex type that contains a sequence of four elements: `<Signed-Info>`, `<SignatureValue>`, `<KeyInfo>`, and `<Object>`. The last two elements, `<KeyInfo>` and `<Object>`, have additional constraints declared using attributes that provide the same functionality as the DTD cardinality operators. Both of these elements are optional and `<KeyInfo>` is constrained to a single instance while `<Object>` is unbounded. Finally, there is also a provision for an attribute value called `Id`, which is also optional.

Processing XML

The reader should now have a fairly good idea of how XML documents are structured and validated, and should understand the difference between a markup language and a meta-language. Given a document that uses XML markup, the reader should be able to tell if it is well formed and should be able to discern some basic validity constraints with the help of a DTD. This section of the primer marks a shift from examining the syntax of XML to understanding how XML is *processed*. XML documents are cool to look at and fun to create, but unless we understand how they are processed and dealt with, the previous section is more a thumb-twiddling exercise rather than something practical.

The next few topics on our plate include understanding the Document Object Model (DOM), which is an API for doing practical things with structured documents, as well as the XPath data model, which is used in the XML Security standards. We will also look at some source code in Java that shows how to use the Apache Xerces toolkit to do a few practical things with XML documents such as parse them and output basic information.

The Document Object Model (DOM)

The DOM is an Application Programming Interface (API) meant for *structured documents*. The reader may be wondering why we are discussing an API at this stage in the book and may also wonder about the relevance of this section altogether. The reason the DOM is important is because it is highly standardized and represents a widely used and accepted program-

ming model for structured documents. This makes the API important because most XML Security implementations (XML Signature or XML Encryption) will have support in some way or another for the DOM; in essence, these implementations are usually written directly on top of the DOM and rely on its functionality and semantics. Because of this, it is an important building block in XML Security.

To support standardization, there are various levels of the DOM. The term *level* is akin to a version number for the DOM, where higher levels represent increased functionality. Our discussion here will include the features and behavior found only in DOM Level 1, and more specifically DOM Level 1 Core, which is the smaller subset of DOM Level 1.

Structured Documents vs. Structured Data

The reader may notice that we have not explicitly mentioned XML documents, but instead began this section using the more general term *structured documents*. The reason for this change in terminology is related to the definition of the *Document Object Model*. The DOM is an API designed for structured documents in general and isn't an API exclusive to XML documents. For example, the DOM can also model HTML documents using the same interfaces.

Our focus with the DOM will be XML documents, and because of this, the more specific term *structured data* is slightly more appropriate. The term *document* can be confusing because a document as such usually implies some sort of presentation coupling such as fonts, colors, graphics, or multimedia. These types of additional presentation semantics are out of scope for XML documents that represent security objects such as an XML Signature or encrypted XML element. We will look at the DOM in terms of structured data, instead of its wider scope of structured documents.

DOM Interfaces

DOM is a collection of interfaces for manipulating structured data in memory using *objects*. We are not throwing out a new term; the term *object* is the same as that found in an object-oriented programming language such as C++ or Java. Simply put, given arbitrary structured data, the DOM specifies interfaces that can be used to access and manipulate this data at a programming language level. A picture of this process is shown in Figure 3-2.

Figure 3-2 shows a *simplified* view of how the DOM looks at structured data. The DOM uses a tree-like structure to model the relationships

Structured Data

```
<Food>
  <FrenchFries>
    Curly Fries
  </FrenchFries>
  <Beers>
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
```

DOM Document
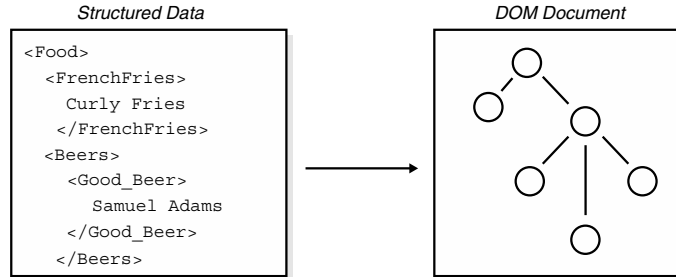
between its interfaces, but does not constrain the actual implementation *to* a tree data structure. Put another way, the DOM appears to act like a tree from the outside, but a particular *implementation* of the data structures is not constrained to a tree structure. A tree structure is an obvious model for structured data such as an XML document. The parent element represents the root of the tree and each child element represents child nodes of the root and so on. A better picture of the scope of the DOM is shown in Figure 3-3.

In Figure 3-3 a new box appears with a question mark inside of it. This signifies that the actual implementation of the DOM is out of scope. A direct relationship between the tree-like interface provided by the DOM and the underlying implementation isn't necessary. DOM merely specifies interfaces; the underlying implementation is up to the vendor that takes on the task of creating a usable DOM API. The outside of the DOM—the user-accessible API—is fixed and has a logical structure that matches that of a tree. The inside—the actual implementation of the DOM interfaces—is not constrained and may or may not match the outer tree view. The idea here is that the DOM be a *portable* interface. For example, it may be desirable to add a DOM interface over some other legacy API that deals with structured data. Because of this, the DOM explicitly separates itself from the implementation of its objects.

This key idea is important to remember; often the DOM will appear clunky and obtuse for basic tasks. The reason is because it is not designed with any one document format in mind, but instead for *arbitrary structured documents*, which may or may not include XML documents.

Inheritance View vs. Flattened View

We will examine the DOM as it is specified for an object-oriented programming language such as Java. DOM APIs exist for many scripting lan-

Figure 3-3

The scope of the DOM

Structured Data	DOM Implementation	DOM Document


```
<Food>
  <FrenchFries>
    Curly Fries
  </FrenchFries>
  <Beers>
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
```
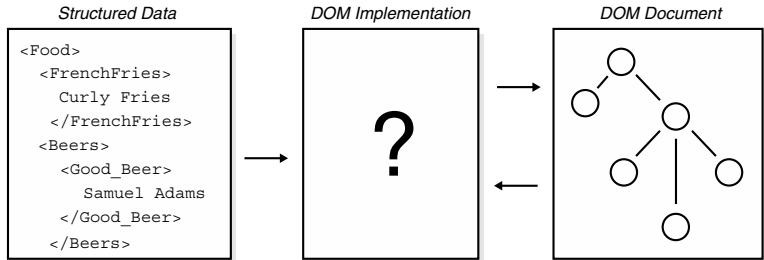
guages and as such exist in two different views: a *flattened view* and an *inheritance view*. Not all languages support object-oriented features such as inheritance and this creates some redundancy in the API. This redundancy adds to the feature set in terms of extra functions and can cause confusion. The flattened view is out of scope for our discussion and the reader should visit the references section at the end of this book for more information on the DOM. All of the upcoming code examples will be given in Java, and, because of this, our primary view of the DOM will be the *inheritance view*.

Complete understanding of the inheritance view of the DOM comes from understanding just two objects: Node and Document. It is difficult to say which object is more fundamental; a clear understanding of both is a necessity for doing anything useful with the DOM. The logical structure of the DOM is a tree, and every object in this conceptual tree is some type of Node object. Consider Figure 3-4.

The first thing to notice about Figure 3-4 is that every node in the conceptual tree has been replaced with a concrete interface called org.w3c.dom.Node. The name of the object comes from the *Java language binding* for the DOM. A given *language binding* for the DOM is defined by the W3C and doesn't refer to a given DOM implementation or API. All compliant DOM implementations for Java *must* use the org.w3c.dom.* packages that define the DOM interfaces. Our focus with the DOM will be Java and it is appropriate the use the fully qualified interface names at this point in time.

The second thing to notice about Figure 3-4 is that all of the objects that represent the structured document in the figure are identical. All nodes are org.w3c.dom.Node objects. This is only true from an object-oriented *subtype* relationship. This is difficult to describe, but easy to show

Figure 3-4

Node objects in
the DOM tree

Structured Data

```
<Food>
  <FrenchFries>
    Curly Fries
  </FrenchFries>
  <Beers>
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
</Food>
```

DOM Document

Figure 3-5

DOM objects

Structured Data

```
<Food>
  <FrenchFries>
    Curly Fries
  </FrenchFries>
  <Beers>
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
</Food>
```
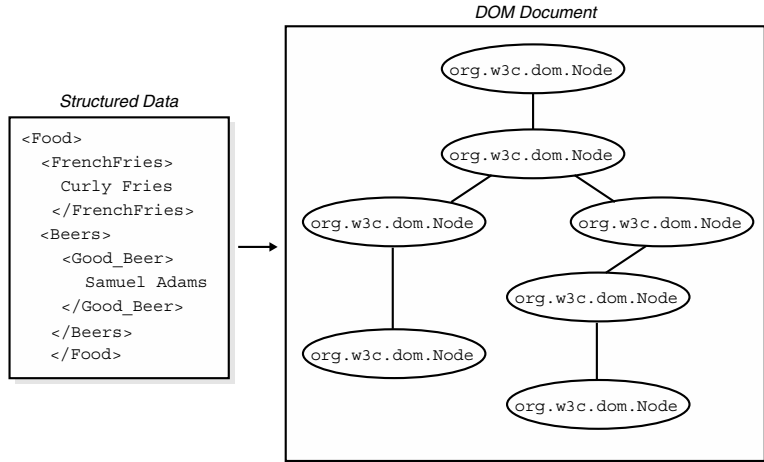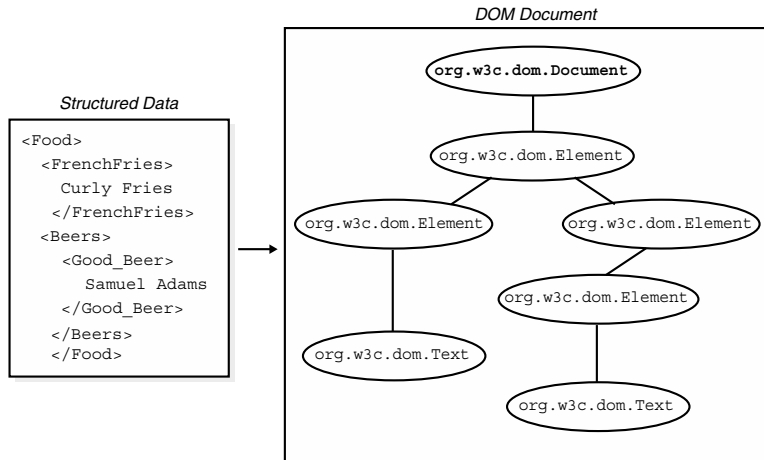
DOM Document

with a picture. Figure 3-5 shows the true objects for the sample XML document, and Figure 3-6 shows the parent-child relationships of the objects.

Figure 3-6 shows the parent child relationships of some common org.w3c.dom.Node subtypes. Not all possible subtypes are shown, and the most common ones are marked in bold. The most important thing to
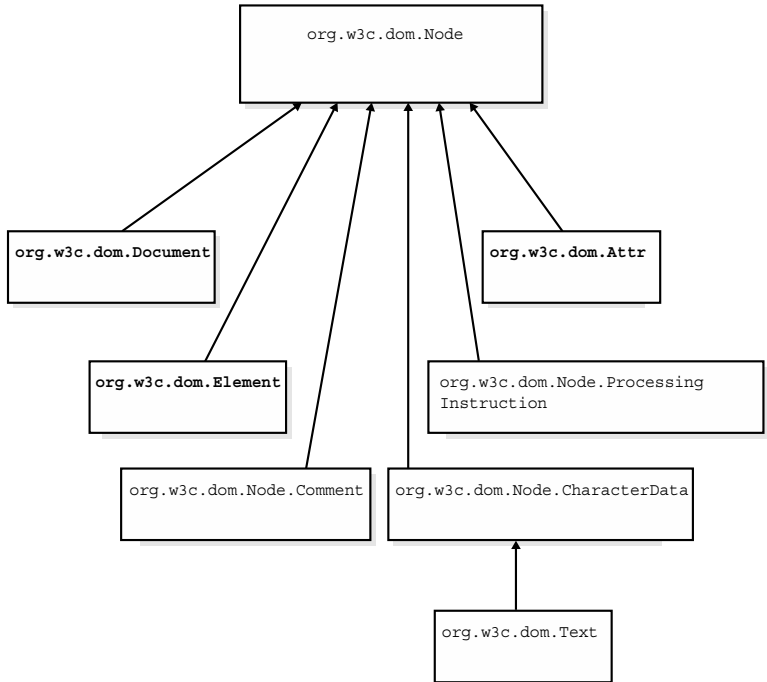
notice about Figure 3-6 is that all of the classes shown are subclasses of
org.w3c.dom.Node and because of this they properly fulfill the subtype
relationship. Each subtype of org.w3c.dom.Node is designed to repre-
sent something in the structured document. For example, the
org.w3c.dom.Element type represents *elements* in an XML document
and the org.w3c.dom.Attr type represents *attributes* in an XML docu-
ment. The list goes on; there are over 15 different subtypes of
org.w3c.dom.Node. We will not cover them all here, because only a few
have immediate interest to us. The reader should refer to the references
section for more complete information on the DOM.

The careful reader should notice that there appears to be a mismatch
between Figure 3-5 and the structured data shown. That is, there is an
extra org.w3c.dom.Document node at the root of the conceptual tree
that has no obvious match to anything in the XML document shown in
the figure. This extra node is the main entry point into the structured

data and the real root of the XML document is actually the first child in the DOM tree.

This point showcases the importance of the `org.w3c.dom.Document` object, which represents the *entire* structured document. Once an `org.w3c.dom.Document` object is obtained for a given structured document, the user can access various `org.w3c.dom.Node` objects that comprise the document tree. Any use of the DOM to model an existing structured document begins with the creation of a `Document` object. Ironically, the actual bootstrapping of the `Document` object is left out of scope —it is the responsibility of the specific DOM *implementation* to provide the user with the necessary methods to create an instance of `org.w3c.dom.Document`. The next section shows how this bootstrapping process works with the Xerces XML Parser.

Bootstrapping with Xerces

Our first goal in this section is to use the Xerces XML Parser[1] (which has support for the DOM language bindings) to create an `org.w3c.dom.Document` object from some sort of real XML data. Once we create the `org.w3c.dom.Document` object, we can traverse the logical tree structure and get information about the XML document. Consider Listing 3-22.

The first things to note about Listing 3-22 are the import statements at the top. The previously discussed *DOM Java language bindings* and the Xerces DOM parser implementation are both added here. The two sepa-

---

Listing 3-22

Using Xerces

```
// Import the DOM Java language bindings
import org.w3c.dom.*;
// Import the DOM parser implementation
import org.apache.xerces.parsers.DOMParser;
class CodeListing31 {

  public static void main (String args[]) throws Exception {
    // Make a new DOM Parser
    DOMParser domParser = new DOMParser();
    // Parse an input document
    domParser.parse("food.xml");
    // Get the org.w3c.dom.Document node
    Document documentNode = domParser.getDocument();
    // Get the first child
    Element rootNode = (Element)documentNode.getFirstChild();
    // What is the name?
    System.out.println("Root element: " + "<"+ rootNode.getTagName()+">");
  }
}
```

---

[1]Xerces can be downloaded for free at http://xml.apache.og.

rate `import` statements showcase the separation of the parser implementation from the DOM interfaces that are specified in the language binding.

Once the proper DOM interfaces and the parser implementation have been imported, the actual constructor is called for the `DOMParser` class. This call and the next two calls are not relying on the DOM API; they are calls that are proprietary to the Xerces processor. In fact, the only calls in the code listing that use the DOM API are the second to last and last function calls.

Once an instance of the `DOMParser` class has been created, a call to the `parse()` function occurs. This is a *blocking call*. In other words, the program is halted while the DOM Parser reads from the specified XML file. This may not seem like a big deal, but when the size of the XML file grows to hundreds or thousands of lines this call has the potential to take a great deal of time.

Once the parsing is complete we are ready to obtain an `org.w3c.dom.Document` object with a call to `getDocument()`. This call returns the `org.w3c.dom.Document` object as specified by the DOM. The reader may think of this call as the transition point from the proprietary Xerces parser to the standard DOM APIs. From here the idea is to use only function calls that are specified by the DOM.

The first call obtains the actual root element in the XML document. In Listing 3-22 we are assuming that the input is the food XML document shown in Figure 3-5; the call to `getFirstChild()` actually obtains the `org.w3c.dom.Element` object that corresponds to the `<Food>` element. The actual string text in between the tag markup (`<` and `>`) in the input document is printed out with the final `getTagName()` call. The result of Listing 3-22 is the stunning output: `<Food>`.

This is hardly useful, but the sample does showcase how the logical structure of the DOM works. When using the DOM it is often desirable to rely on recursive semantics to traverse a structured document. Consider Listing 3-23, which prints out all of the `org.w3c.dom.Element` nodes and `org.w3c.dom.Text` nodes in the `food.xml` document.

All of the gory details are in the `printNode()` function. The idea here is to give this function an `org.w3c.dom.Node` object and it will determine the *type* of node and then make a printing decision. In Listing 3-23 we pass the first child of the `org.w3c.dom.Document` node (the first child is the document element) directly into the `printNode()` function. The first thing done inside `printNode()` is to determine which type of node we have; this function is trivial and only deals with two types of

Listing 3-23

Some recursion
with the DOM

```java
import org.w3c.dom.*;
// Import the DOM parser implementation
import org.apache.xerces.parsers.DOMParser;
class CodeListing32 {
  public static void main (String args[]) throws Exception {
    // Make a new DOM Parser
    DOMParser domParser = new DOMParser();
    // Parse an input document
    domParser.parse("food.xml");
    // Get the org.w3c.dom.Document node
    Document documentNode = domParser.getDocument();
    // Get the first child
    Element rootNode = (Element)documentNode.getFirstChild();
    // Let's print out all the element names and text nodes
    printNode(rootNode);
  }
  public static void printNode(Node nodeToPrint) {
    int type = nodeToPrint.getNodeType();

    if (type == Node.ELEMENT_NODE) {
      String nodeName = nodeToPrint.getNodeName();
      System.out.println("Element Node Found: <" +nodeName+">");
      NodeList childNodes = nodeToPrint.getChildNodes();
      if (childNodes != null) {
        for (int i=0; i<childNodes.getLength(); i++) {
          printNode(childNodes.item(i));
        }
      }
    }
      if (type == Node.TEXT_NODE) {
      String textValue = nodeToPrint.getNodeValue();
      System.out.println("Text Node Found: " +textValue);
    }
  }
}
```

nodes, so the choice is either Node.TEXT_NODE or Node.ELEMENT_NODE. These static identifiers are simply integers used by the DOM implementation to distinguish between different node types. If our node is an element, we first determine the name using getNodeName() and then print this out.

Once the parent element has been printed, the child nodes are next in line. A simple for loop is used to iterate through these and print them out one by one. The object used to hold the list of nodes is the org.w3c.dom.NodeList object. This object is a bit unique in its semantics. At first glance this object appears to model a list of nodes that can be accessed like an array. For example, in Listing 3-23 we use a for loop to iterate through this org.w3c.dom.NodeList object with an item()

function that appears to give us the contents at each position in the node list. Despite the way this object looks, it does *not* have perfect list-like or array-like semantics. This can be confusing for newcomers to the DOM.

The org.w3c.dom.NodeList is a *linear view* of the document tree. This means that if you *add* a node to an org.w3c.dom.NodeList, you are adding a node to the tree. Similarly, if you remove a node, a node gets removed from the tree. As the tree gets updated, the NodeList changes; there is no need to update the org.w3c.NodeList—it will change by itself. NodeLists are useful when it is desirable to do operations that require sequential access (such as printing out the children of a given node).

The second case is that of a *text node*. Text nodes are pervasive throughout even the simplest XML document (such as ours) because white space is considered significant in XML documents. This means that the printNode() function will be called more times than the visible contents of food.xml. Further, this means that we lied a bit in Figure 3-5. Figure 3-5 shows the document structure not counting white space in the original document. In other words, the tree shown in Figure 3-5 has white space nodes (which are proper org.w3c.dom.Node subclasses) removed for the sake of clarity. For example, the reader can see how the white space is counted by looking at the output of Listing 3-23, shown in Listing 3-24.

There are a total of seven org.w3c.dom.Text nodes found in the food.xml document, even though only two are apparent (Curly Fries and Samuel Adams).

Beyond DOM

The previous two sections represent a whirlwind tour of a common paradigm for processing XML data. The DOM and its tree structure model is only one way of processing structured documents. The tree structure

---

Listing 3-24

The output from Listing 3-23

```
Element Node Found: <Food>
Text Node Found:
Element Node Found: <FrenchFries>
Text Node Found:
    Curly Fries
Text Node Found:
Element Node Found: <Beers>
Text Node Found:
Element Node Found: <Good_Beer>
Text Node Found:
      Samuel Adams
Text Node Found:
Text Node Found:
```

model is easy to understand and straightforward to implement, but it is not always ideal for every practical situation. The previously mentioned *blocking* parse() call in Xerces can present a problem for memory constrained environments; every time a document is parsed, a logical structure is built. Applications that only need access to a single node of the tree must incur a large performance penalty in terms of memory.

To avoid this type of performance tradeoff, another processing paradigm is used that sees the structured document as a stream of events. For example, instead of building a logical structure in memory that matches the document, the document is seen as a continuous stream of pieces and components. Each piece of the structured document (this can be an element or attribute) usually represents some sort of *event* and something important is done upon the receipt of the event. XML documents processed as a stream have a performance benefit because no logical in-memory structure is created and the programmer has the flexibility to store the pieces that are needed as they come. The standard API for this type of processing is called the *Simple API for XML Processing* (SAX). We will not discuss the SAX or its use in this book, but it should be considered for applications that don't want to be locked in to the more memory intensive DOM paradigm. The reader should visit the references at the end of this book for more information on SAX.

The XPath Data Model

This next section marks a shift from the practical DOM API to the more conceptual XPath data model. The DOM structure model is intended to be an API for applications that process XML. The XPath data model, while similar to the DOM in its specification, is intended to be a *conceptual structure model* for an XML document.

There are two potentially confusing things about XPath and the XPath data model. First, the XPath data model appears to be very similar to the DOM structure model. Both data models use a logical tree that relies on nodes to represent pieces of the input document (such as elements and attributes). Further, they also have similar constructs for representing a *collection* of nodes. The key idea about the XPath data model is that it is only conceptual and exists as a standard way of referring to an XML document from an intellectual perspective. This means it gets used a lot in the XML standards and drafts, including the XML Security standards. For example, understanding the XPath data model is useful in under-

standing how the XML Signature Recommendation processes data as XML. The DOM model can't be directly used in this case because it is an actual API and specifying XML standards in terms of the DOM would tightly couple a given standard to a mode of implementation, which is out of scope for most XML-related standards.

The second confusing thing about XPath is that it is also a specification of a *path language* for traversing an XML document. This generally adds to the muddle because most people use XPath to write expressions for transforming and selecting pieces of an XML document. This will *not* be our main focus here; the data model that XPath provides is what is most important because it allows us to understand how XML standards view an XML document. Once the data model is understood, the reader is in good shape for understanding how XML documents are transformed.

XPath Nodes

The main construct in the XPath data model is the concept of a node. A node represents an actual piece of an XML document. The difference between a DOM-based `org.w3c.dom.Node` and an XPath node is the scope of what can be represented. An XPath node has a smaller scope in most respects and contributes to a slightly simplified conceptual view of an XML document.

For example, the DOM has an interface called `org.w3c.dom.Entity-References` that extends `org.w3c.dom.Node` and can be used to model entity references in the input XML document. This enables a user to count the number of entity references in an input document—the point being that entity references show up in the DOM's view of an XML document. As another example, the DOM has an interface called `org.w3c.dom.DocumentType` that also extends `org.w3c.dom.Node`. This interfaces allows for access to information inside the document prolog, specifically the DTD, enabling a user to read parts of the DTD and print them out. Again, the DTD is in DOM's view of the XML document.

The situation is a bit different for the XPath data model; there are only seven conceptual node types. Because there are more than seven possible constructs in an XML document, XPath can't model everything and the view is necessarily simplified. The data model defined by the XPath Recommendation consists of the following seven node types: *root nodes, element nodes, text nodes, attribute nodes, namespace nodes, processing instruction nodes,* and *comment nodes*. Our aim is to eventually describe how a given XML document gets chopped up into these node types. That

is, we are about to describe the process by which the XPath data model is *applied* to a given XML document.

Before we discuss the actual nodes themselves, we need to approach something more fundamental, called *document order,* which is the order in which the XPath data model is applied.

Document Order

The term *document order* refers to the order in which the various node types are created, based on a real, physical XML document. This ordering is actually quite straightforward and simple, and can be best described as a list of rules that provide the "cooked" view of the XML document. Document order is summarized as follows:

Document Order

Nodes are organized in the order in which they appear in the XML representation with these constraints:

1. The root node comes first.
2. Element nodes occur before their children.
3. The attribute and namespace nodes of an element occur before the children of the element.
4. The namespace nodes occur before the attribute nodes.
5. The relative order of namespace nodes and attribute nodes is implementation dependent.

Everything about document order is straightforward except for the third point. First, XPath contrasts the DOM in that it respects namespace nodes, where DOM Level 1 just considers namespaces to be attributes (which they properly are). Further, an element node has an associated set of attribute nodes and namespace nodes that aren't defined to be proper children of the associated element node. Attribute nodes and namespace nodes are considered to be associated with or properties of a given element node. This view makes a lot of sense because visually an element and its attributes are adjacent in the XML document. The last point here is that document order looks at an XML document after general entities have been expanded. This is where some of the simplification occurs. The XPath data model doesn't have an entity node and simply treats replaced

entities as text. Remember, with the XPath data model, whatever was in your original XML document can only be seen as one of seven node types (six really, since the root node is fixed).

Now it is time to examine the seven node types and see how they are created from an example XML document. We will attempt to form the XPath data model view of Listing 3-25, which is an updated version of an earlier food XML document.

In Listing 3-25 we have updated our food XML document with some comments, a default namespace, and some attributes. From here, let's look at the node types in detail and see what we can come up with for an XPath data model.

Root Node

Only one root node represents every XML document in the XPath data model in its entirety (not just the document element). This point is often confusing for readers, many ask: "Why have a root node, when we really want to get to the document element of the XML data?" An easy way to explain this is to realize that other items (such as the comments shown) appear as siblings to the document element. This means that if we are to maintain our logical tree structure, we need a conceptual root node to hold the rest of the nodes that are neither parent nor child to the document element (root element). In Listing 3-25 there is one root node with three children (in document order): a comment node, an element (the document element), and a comment node. There is no xml declaration node, so this piece of information is also lost in the XPath data model view. Our conceptual XPath tree thus far is shown in Figure 3-7.

Listing 3-25

Updated "food" XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Here is a healthy meal -->
<Food xmlns="http://food.com">
  <FrenchFries Size=Large" Salted="True">
    Curly Fries
  </FrenchFries>
  <Beers Size="Pint">
    <Good_Beer>
      Samuel Adams
    </Good_Beer>
  </Beers>
</Food>
<!-- don't forget to always drink good beer -->
```
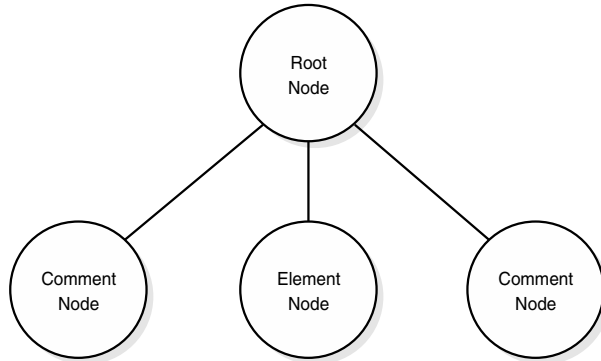
Figure 3-7

The beginnings of
the XPath tree for
Listing 3-25

Element Node

Element nodes represent actual elements in the physical XML document. There is nothing too tricky here. The only other thing to note is that the possible child nodes of an element node includes element nodes, comment nodes, processing instruction nodes, and text nodes. All of these nodes haven't been discussed yet, but we will get to them.

If we add element nodes to our XPath tree, the result looks something like Figure 3-8. A total of three element nodes are added, one for each element in Listing 3-25 (excluding the root element).

Attribute Node

Attribute nodes are slightly more confusing than element nodes. The relationship between an element and its attributes can be thought of as *association*. An element may have attribute nodes associated with it. The confusing part is that the XPath Recommendation defines the element node-bearing attributes to be the parent node of the attributes, but the attributes are not child nodes of the element node. This sentence can be confusing because the term *parent* when used in discussions of a tree structure usually logically implies the presence of *children*. XPath uses the term *set* to describe the attributes associated with a given element—we will expand upon this set idea and use such a notation for our expanded XPath view of Listing 3-25, which now contains attributes. This is shown in Figure 3-9.

The reader should notice that in Figure 3-9 the relationship of the attribute sets to each element. Visually, it makes sense to call the element
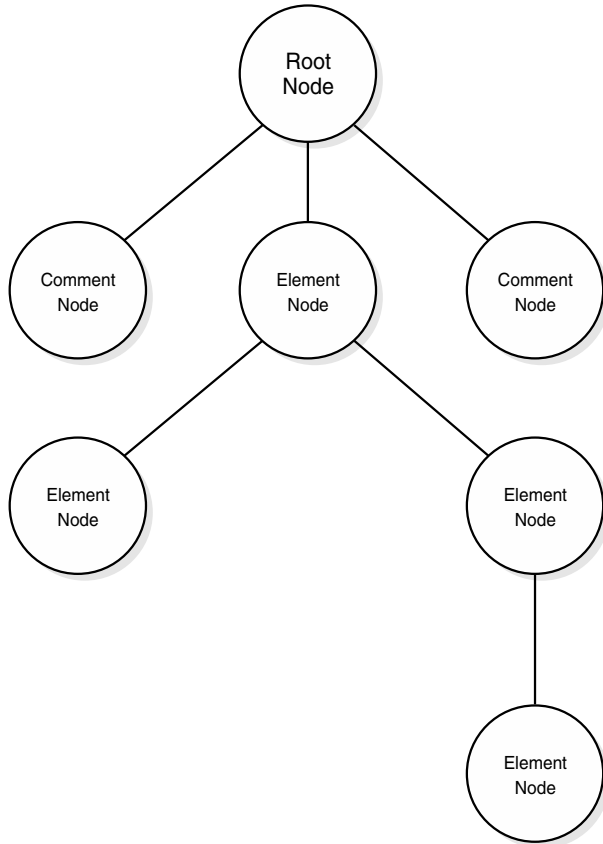
bearing the attributes a parent of the attributes, but it should also be
clear that the attributes are not proper children of the element because
they are not really intermediate nodes in the logical tree.

Another thing to note is that attribute nodes only appear in the XPath
view for attribute nodes explicitly specified in the physical XML document
being modeled (or those specified with default values in the DTD). A DTD
can specify that attributes can take on default values. This isn't some-
thing that was covered earlier, and the reader is urged to visit the refer-
ence section at the end of the book for more information about DTDs.

Figure 3-9

Adding attributes
to the XPath tree
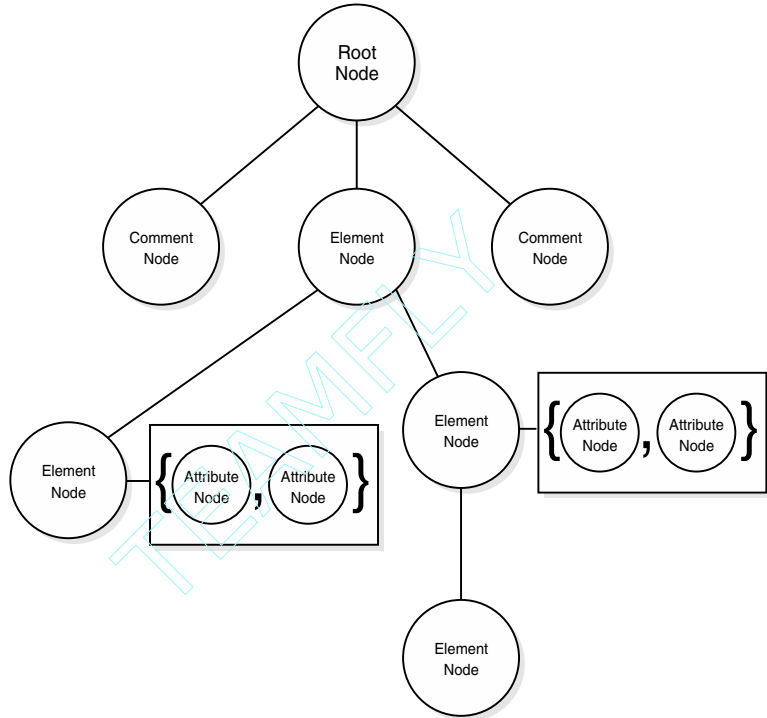
In short, there can be some ambiguities between XPath views of an XML document because XPath doesn't require that the DTD be read (the XML Parser used may not support it). This means that it is quite possible for two identical XML documents to produce different XPath nodes based on the presence or absence of the DTD.

Namespace Nodes

The treatment of a namespace node is similar to an attribute node because namespaces are spiced up attributes from an XML syntax standpoint. Similarly, a given element node is associated with a set of namespace nodes. There is one namespace node for every namespace that is in scope for the current element. For example, if a namespace node was

declared on an ancestor node and is still in scope (it hasn't been overridden or undeclared), then there is a namespace node for this namespace.

Finally, the additional qualifier here is that the namespace nodes should be first in the set; for the picture we will use the same set for both the attribute nodes and namespace nodes. Listing 3-25 only has one namespace node, the default namespace. This namespace, however, is in scope for the entire food XML document. This means that there is a namespace node for *every* element node in our picture. The updated picture is shown in Figure 3-10.

In Figure 3-10 the picture looks a bit skewed as we try to fit everything together. The reader should notice that we have put the namespace nodes first in the associated set. This has to do with document order and the constraint that says namespace nodes must occur before attribute nodes.

Text Node

Text nodes represent any sort of text in the document. There is an XPath text node for all text in the document except for text inside comments, processing instructions, and attribute values. The XPath view of an XML document simplifies text defined using predefined character entities or residing in CDATA sections. That is, an XPath text node doesn't tell you if the text *came from* a predefined character entity or a CDATA section. This information is essentially lost; XPath models all text the same way. Consider the following short example:

```
<Sample>
  <element1> I love XPath! </element1>
  <element2> <![ CDATA [ I love XPath ]]> </element2>
</Sample>
```

XPath sees both <element1> and <element2> as an element node and a text node; the presence of the CDATA section is lost. From this it follows that with the XPath view of an XML document it can't be determined if a given text node had its origins as a CDATA section or simply normal markup. Figure 3-11 shows the final picture for our XPath view of Listing 3-25.

In Figure 3-11 two text nodes were added. One text node corresponds to the Curly Fries text and one text node corresponds to the Samuel Adams text.

Figure 3-10

Adding
namespace nodes
to the XPath tree
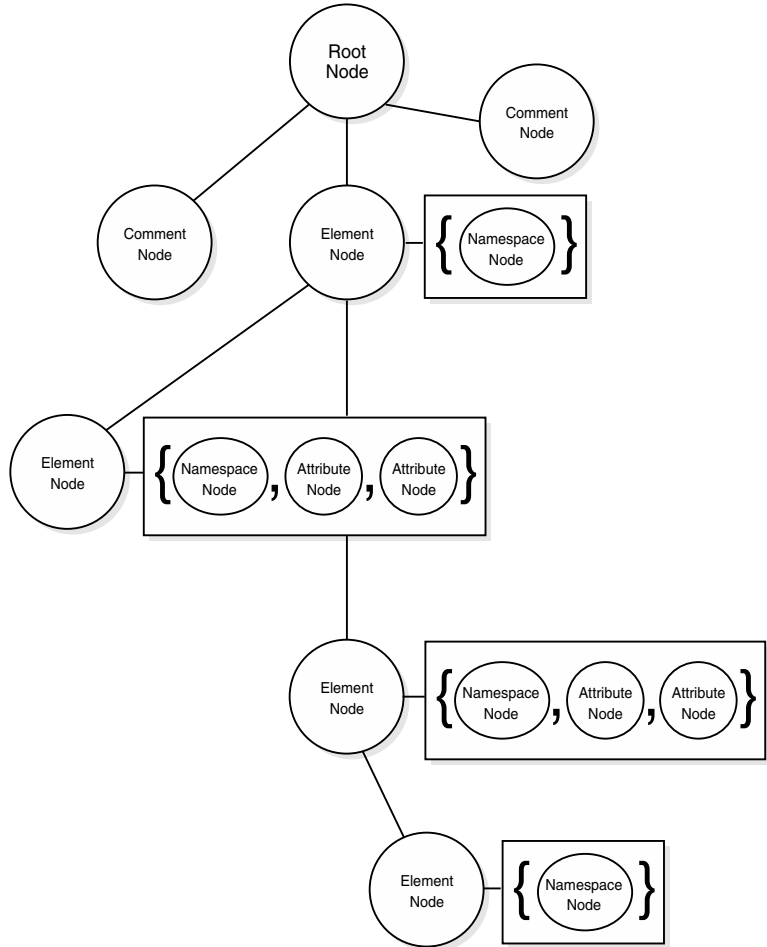
Processing Instruction Node

This node is rather boring for our purposes. In short, XPath can model any
processing instruction in the physical XML document except for process-
ing instructions inside the DTD. There is a single processing instruction
node for every actual processing instruction found inside the body of the
document.

Figure 3-11

The final XPath view for Listing 3-24
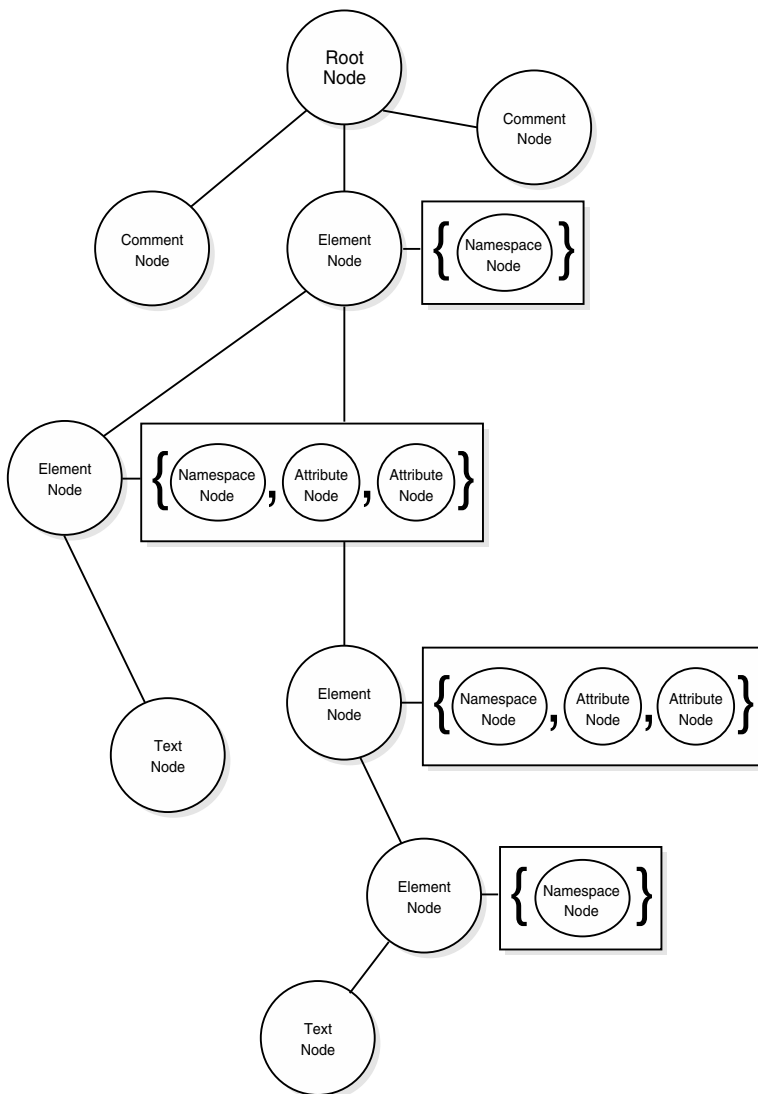
Comment Node

We have already seen this node in action. There is one comment node for every actual comment found in the physical XML document. Again, any comment can be modeled except for comments inside the DTD.

XPath Node Set

Understanding the XPath data model is important because of a single term: *node-set*. This term is used throughout the XML Security standards and is an unordered collection of XPath nodes. Now that you know what the possible node types are, you also now know what a node-set is in this context. The node-set is simply an XPath tree that has been serialized from a tree to a flat list. The reader should also understand that the standard `org.w3c.dom.NodeList` object cannot be used to model an XPath node-set without some changes. A node-set is a proper list, while the `org.w3c.dom.NodeList` is a linear view. This is an important distinction to make because, at first glance, the two concepts appear to be interchangeable.

More on XPath

The main focus of the XPath Recommendation is not the previous data model. The data model is actually the last topic in the XPath Recommendation (some would consider this odd, since it is foundational for XPath). XPath is more concerned with the language and function library for traversing through an XML document and selecting document subsets. This discussion, while interesting, falls out of scope for our goals in this book. The reader will see some simple (largely self-explanatory) XPath expressions in Chapters 5 and 6 that perform some basic selection, but an in-depth tutorial on the matter is left to another resource. The reader should visit the references section for more information on XPath.

Chapter Summary

This chapter has been an XML primer on XML divided into two subsections: syntax and processing. The chapter began with discussing the fun-

damentals of XML syntax including a discussion of elements, attributes, and documents. Well-formed XML was then discussed with some introductory material on XML namespaces and document type definitions, including a small amount of information on XML Schema. The processing section focused on two main topics: The Document Object Model and the XPath data model. Both are fundamental for processing XML as a logical tree structure. The distinction was made between the DOM, which is a practical API, and the XPath data model, which is a conceptual tree model used in XML standards. This distinction was noted as important because both models use a node-based tree structure, but have a different purpose and semantics.

This page intentionally left blank.

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	Operator	Description
Cardinality Operators	*	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 (`Signature-Value`). 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 `<Signed-Info>` 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>
```

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

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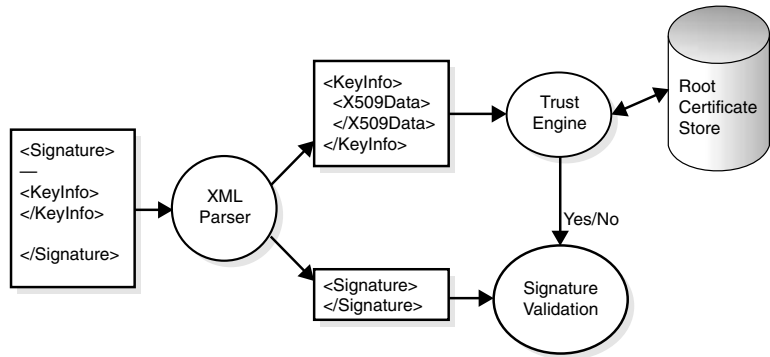
Certificate path validation makes for a convenient example, but it is *not* the only way of asserting trust over public key material. XML Signatures allow for a wide array of components within `<KeyInfo>`. Table 4-2 describes the various element choices for `<KeyInfo>` as defined by the current XML Signature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

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).

Table 4-2

<KeyInfo> Child Element Choices

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	Element Name	Description
`<X509Data>` Child Element Choices	`<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 certifi-

cates 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 `<Retrieval-Method>` 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
      TMBEGA1UEChMKUGhpbG9zb3BoZWTEPMA0GA1UEBxMGQXRoZW5zMQ4
      wDAYDVQQDEwVQbGF0bzEZMEMBcGA1UEDBMQRm91bmRlciBvZiBMb2d
      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 <Key-Info> 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 `<Signature-Properties>` 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
        bmVlcmluZzzEWMBQGA1UEAxMNQmlnIEJhZCBCbGFrZTEcMBoGCSqGSIb3DQEJARYN
        YmJiQGJiYmFzLmNvbTAeFw0wMDA2MjAyMTEzMzVaFw0xMTA2MDMyMTEzMzVaMH4x
        CzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpTb21lLVN0YXRlMQ8wDQYDVQQKEwZTZXJ2
        ZXIxFDASBgNVBAsTC1NlcnZlciBDDZXJ0MRMwEQYDVQQDEwpTZXJ2ZXXJDZXJ0MR4w
        HAYJKoZIhvcNAQkBFg9zZXXJ2ZXJAY2VydC5jb20wgZ8wDQYJKoZIhvcNAQEBBQAD
        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">
      aWcgQmxha2UncyBBdXRoZW50aWNhdGlvbiBTZXJ2aWNlLMRQwEgYDVQQLEwtFbmdp
      bmVlcmluZzEWMBQGA1UEAxMNQmlnIEJhZCBCbGFrZTEcMBoGCSqGSIb3DQEJARYN
      YmJiQGJiYmFzLmNvbTAeFw0wMDA2MjAyMTEzMzVaFw0xMTA2MDMyMTEzMzVaMH4x
      CzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpTb21lLVN0YXRlMQ8wDQYDVQQKEwZTZXJ2
      ZXIxFDASBgNVBAsTC1NlcnZlciBDZXJ0MRMwEQYDVQQDEwpTZXJ2ZXJJDZXJ0MR4w
      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.

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 defini-
tions 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 pro-
vides 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.

CHAPTER 5

Introduction to XML Digital Signatures Part 2

XML Signature Processing

In Chapter 4, we spent most of our time discussing the syntax of XML Signatures. We studied the outer structure and the meaning of the various parent-child relationships that exist in an XML Signature. In addition to the basic syntax, we also made a careful investigation through the defining elements of an XML Signature—we looked at the purpose or idea behind each element and made a note of the required and optional attribute values. The only element that received casual discussion was the `<Reference>` element. Deferring discussion about the `<Reference>` element is necessary because a good understanding of `<Reference>` and its child elements is contingent upon a clear understanding of the XML Signature processing model. This chapter is devoted to introducing the processing model and signature transforms for XML Signatures. The processing model is quite rigid and makes for rather tedious discussion, while the signature transforms are very interesting and are arguably one of the most powerful aspects of XML Signatures.

XML Signatures are created and verified in two steps. Signature generation is referred to as *core generation* and signature validation is referred to as *core validation*. The definitions of these terms are very hierarchical and algorithmic—it is recommended that the reader make

note of the process that is associated with these definitions. A clear understanding of how XML Signatures are created and verified is helpful in grasping the purpose behind many of the additional features used by XML Signatures.

The <Reference> Element

Before we segue into a discussion of the processing model, it is time to elaborate on the role and specifics of the <Reference> element. This element is really at the heart of the XML Signature; it ultimately describes how resources are obtained and possibly transformed to produce the data that is digested and subsequently signed as part of the <SignedInfo> element. The <Reference> element represents a resource of potentially any type or format. A <Reference> can represent *any* arbitrary binary format. This point is especially important because it is often assumed that XML Signatures are signatures over XML data *exclusively*.

Although it is true that many features and tricks can be done with an XML document as a signature source, this does not preclude arbitrary octets from participating in an XML Signature. In fact, an XML document that is referenced as a remote resource is treated as a binary file and processed as octets. XML documents are processed differently only in the case of an *enveloped* or *enveloping* signature, or if one of the transforms used requires the XML data to be processed as an abstract set of nodes (a *node-set*).

A *node-set* is the recommended data model for processing XML within XML Signatures; it is specified in the XML Path Language Recommendation (XPath) and is given a treatment in the XPath primer in Chapter 3. For those readers who are unfamiliar with a *node-set,* one can think if it simply as a set of different *nodes types*, each of which represent some part of an XML document. Seven types of nodes exist: root nodes, element nodes, attribute nodes, processing instruction nodes, comment nodes, text nodes, and namespace nodes. Each of these node-types are part of the XPath data model described in Chapter 3.

The syntax of the <Reference> element is similar to the previously discussed elements—it contains three child elements: <Transforms>, <DigestValue>, and <DigestMethod>, and three optional attributes: Id, Type, and URI. In the simplest case of a <Reference> element, the only required elements are <DigestMethod> and <DigestValue>. All other attributes and elements are purely optional. The Id and Type

attributes represent additional processing semantics that can be placed upon a specific `<Reference>` element. The `Id` attribute is a generic unique identifier that can be used to facilitate remote referencing. Such an `Id` attribute is common among the elements used in XML Signatures —it provides a hook for including `<Reference>` elements that may reside outside of the current application context.

The `Type` attribute specifies the *type* of the `<Reference>` being pointed to. The word *type* is used here in a broad sense, like a data type in a programming language. It does not refer to the actual data or file format of the data referenced by the `<Reference>` element. For example, consider the following `<Reference>` element:

```
<Reference Type="http://www.w3.org/2000/09/xmldsig#Object"
URI="#myobject">
  <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1" />
  <DigestValue>70NvZxcdTB+7UnlLp/J724p8h4zx=</DigestValue>
</Reference>
```

Next, suppose that the referent is an `<Object>` element identified by the attribute `myobject` and contains base64-encoded data as follows:

```
<Object Id="myobject">
  lczCCAbYwggErBgcqhkjOOAQBMIIBHgKBgQDaJjfDTrawMHf8MiUt
  Y54b37hSmYNnR3KpGT10uU1Dqppcju06uN0iGbqf947DjkBC25hKnq
</Object>
```

The example here shows that the `Type` attribute of the `<Reference>` element does not refer to the actual data referenced, but instead to the container in which the data resides. In this case, it is an `<Object>` element.

The XML Signature recommendation specifies two optional values for this `Type` attribute: `http://www.w3.org/2000/09/xmldsig#Object` and `http://www.w3.org/2000/09/xmldsig#Manifest`. These URIs are once again being used as identifiers that may help an XML Processor make better choices about how to parse a given XML document. Such an identifier might be useful in a situation where a processing application wants to pinpoint the location of a `<Manifest>` element without manually parsing multiple `<Object>` elements. For example, it is an easy task to identify an XML element by an attribute value—an XML Processor can look for elements with attribute value `http://www.w3.org/2000/09/xmldsig#Manifest` and it would know that this `<Object>` element probably contains a `<Manifest>` element.

This "data typing" of the elements, however, is not required. No explicit requirement specifies that these `Type` identifiers be used, and the XML

Signature Recommendation requires no validation of their correctness. An application can use the `Type` identifier as it pleases, inventing its own types, or even use it improperly. None of these uses would invalidate the signature or the requirements of the XML Signature Recommendation, although doing so is not recommended due to interoperability concerns.

The most interesting attribute of `<Reference>` is the URI attribute. This attribute describes the data resource that will *eventually* be digested. Notice that we have used the qualifier *eventually*—this is because the data may have additional transformations applied to it before digesting takes place. That is, the data that is actually digested is the octet stream that is the result of optional transforms that are specified (if present) by a `<Transforms>` child element. The `<Transforms>` element is in turn a container for one or more `<Transform>` elements that specify the type of transform that will be performed.

Another way of thinking about this is that the URI element and the `<Transforms>` collectively describe in an abstract sense *how* the data to be digested is arrived at. The `<Transforms>` work in a cascading manner —the output from one transform is fed into the next. One useful mental picture is a waterfall. The data flows from its source, which is ultimately going to be a URI of some type (in most cases), and proceeds to travel down the steps of a waterfall. Each time the data descends a step, a new transformation is applied until the data arrives at the sink, where it is digested and included in the `<DigestValue>` element. Figure 5-1 shows the waterfall view of signature transforms. We will often use the informal term "transform waterfall" to denote transformations that occur in this manner.

The lateral movement of the "water" in Figure 5-1 represents the transformation that is happening to the data, and the vertical falling of the water represents a possible conversion from octets (binary) to an abstract *node-set* or vice versa. Some transformation algorithms (such as XPath) require that the data be processed as an XPath node-set. If the transformation requires a *node-set,* the incoming binary data must be converted; likewise, if the transformation requires *octets*, the node-set must be converted to octets.

After transformations are applied, the resulting data is digested as specified by the `<DigestMethod>` element. This empty element specifies a digest method using an attribute URI identifier. Again, the use of the URI for multiple purposes (as a data *source* when specified in a URI attribute and as a *identifier* when specified in a `<DigestMethod>` element) can be quite confusing, but its use here merely identifies the

Figure 5-1

The
transformation
waterfall

$T_1 \ldots T_n$ Represents the
transformations happening to the
input data.

The notation "*c*" denotes a possible
conversion from node-set to octets
or vice-versa.

Digest
Function
(SHA 1)
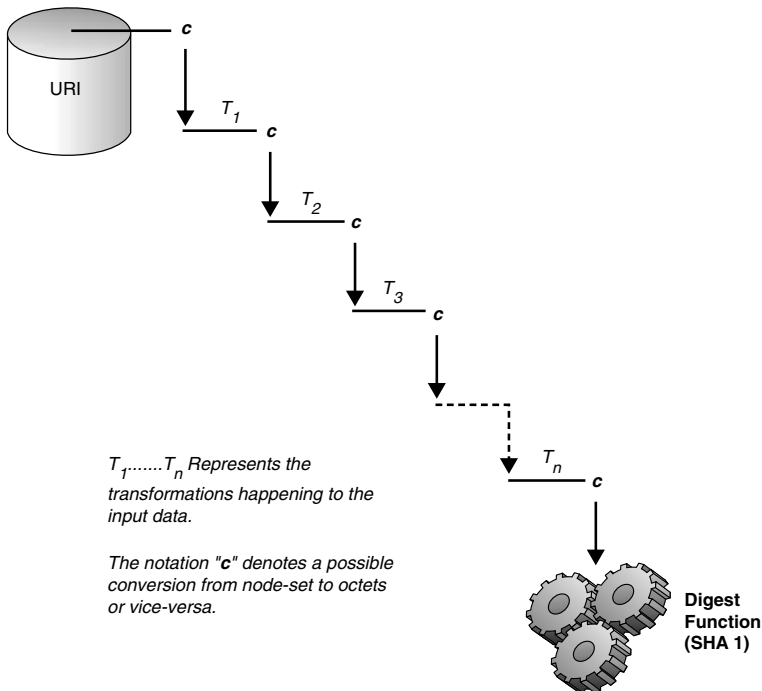
selected digest function. For example, the only digest function that is
supported in the current XML Signature Recommendation is SHA-1.
The identifying URI for SHA-1 is `http://www.w3.org/2000/09/`
`xmldsig#sha1`. This identifier used in a `<DigestMethod>` empty ele-
ment would appear as follows:

```
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

A complete, fully featured `<Reference>` element that uses all of the pos-
sible specified attributes and child elements appears as follows:

```
<Reference
  Id="Full-Reference"
  URI="#EnvelopedText"
  Type="http://www.w3.org/2000/09/xmldsig#Object">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
```

```
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>90NvBvtdTB+7UnlLp/V424p8o5cxs=</DigestValue>
</Reference>
```

In the previous example, notice that this `<Reference>` element is given an `Id` attribute, a `URI` attribute, and a `Type` attribute. In addition, it has a `<Transforms>` element that contains a single transform (base64 decoding). Finally, the two required elements, `<DigestMethod>` and `<DigestValue>,` are specified. They contain the digest value and digest method.

To contrast this example, consider the following example. This is the smallest syntactically valid `<Reference>` element. It contains only the `<DigestMethod>` and `<DigestValue>` elements. Notice that the URI attribute is omitted; in this case, it is assumed that the location of the source data is delegated to the application.

```
<Reference>
  <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"/>
  <DigestValue>GBVZy3Ucg7CyFcPRms9C7bA3qCO=</DigestValue>
</Reference>
```

Thus far, we have given a whirlwind tour of most of the features of the `<Reference>` element. We have expanded on the various attributes and child elements and seen their purposes at a basic level. All that is left is a thorough discussion of the optional `<Transforms>` element—as we have mentioned previously, signature transforms are some of the most powerful features of an XML Signature and must be given a full treatment. For now, we will continue and focus on the XML Signature processing model.

Core Generation

The XML Signature Recommendation defines *core generation* in two steps: *reference generation* and *signature generation*. The first step defines how the digest value of each `<Reference>` element is calculated, and the second step defines how the actual `<SignatureValue>` is calculated. The ultimate goal of *reference generation* is to create actual `<Reference>` elements that are fully featured and have all of their intended pieces (such as transforms, attributes, and digest values). Likewise, the ultimate goal of *signature generation* is to produce the actual signature value and construct the entire parent `<Signature>` block with all of the attributes and

child elements intact. Figures 5-2 and 5-3 show the steps for *reference generation* and *signature generation*.

A few points concerning the description of core generation in the box below may need some extra clarification. In signature generation, it is important to note that in the first step we are not applying a canonicalization method or signature method. This step simply creates an element with the appropriate URI identifiers for each. This is done because both the signature method and the canonicalization method should be signed (such as included in the `<SignedInfo>` element) to resist substitution attacks.

The second step of signature generation is where the actual signature operation takes place. The object that is being signed, namely the `<SignedInfo>`, defines the signature algorithm and canonicalization method used. Remember, in *all* cases the `<SignedInfo>` element *must* be canonicalized before it is signed. The reason for this is because the

---

Core Generation Steps

Reference Generation

For each resource being signed,

1. Apply transforms (if present).
2. Calculate the digest value of the transformed resource.
3. Create a `<Reference>` element. This includes optional attributes, transforms, the identifier for the digest algorithm and the actual digest value.

Signature Generation

1. Using the references created in reference generation, create a `<SignedInfo>` element that specifies a signature method and canonicalization method.
2. Apply the canonicalization method and signature method to the `<SignedInfo>` element.
3. Form the parent `<Signature>` element using the just-created `<SignatureValue>` and the previously created `<SignedInfo>`, as well as all optional elements and attributes.

---

physical representation of the `<SignedInfo>` may change as the XML document that contains this element is sent through XML Processors in various application domains. Canonicalization simply ensures that the same actual octets are signed and compared against the hash value that ensures the signature's validity.

At this point, we have not given a good treatment of canonicalization or its inner workings and purpose. It is, however, an essential part of the core processing rules. Full details on canonicalization are given in the "Signature Transforms" section later in this chapter. For now, it is enough to know that canonicalization is a required step for the core processing rules that helps ensure that the same octet stream is signed and subsequently verified.

It may be interesting to note that the way core generation is defined is a bit odd. The decision to demarcate the core generation steps into two subcomponents with exactly three substeps each seems oddly symmetric and rather arbitrary. Another way of thinking about XML signature generation is to think of it in terms of a raw digital signature. That is, it is useful to think of the creation and formation of the `<SignedInfo>` element as a single operation and therefore a single step.

The second and final step in this alternate processing scheme is to canonicalize and sign the `<SignedInfo>` and create the parent `<Signature>` element. The `<SignedInfo>` is really the data we are signing as part of the signature operation and one might think of it as the data in a raw digital signature if you omit the canonicalization step. This scheme is simply another way of mentally simplifying the core generation process. Most of what makes an XML signature different than a raw digital signature is encapsulated in *how* the `<SignedInfo>` element is formed—inside this element lives the `<Reference>` elements and their child elements along with all of the necessary information (the signature method and canonicalization method) required to produce the signature.

It is often hard to visualize what core generation might look like without a picture. Figure 5-2 shows a pictorial representation of reference generation, and Figure 5-3 shows a pictorial representation of signature generation. Figure 5-2 shows three Web resources that undergo transformations and a message digest to become `<Reference>` elements. What is not shown in the picture is the detail of each reference element. The omitted details include the identifying URI for the data source (if present), the type of transforms used, the type of digest method and the digest value. In Figure 5-3, the three `<Reference>` elements join with a specified `<SignatureMethod>` element and a `<CanonicalizationMethod>`

Figure 5-2

Reference generation
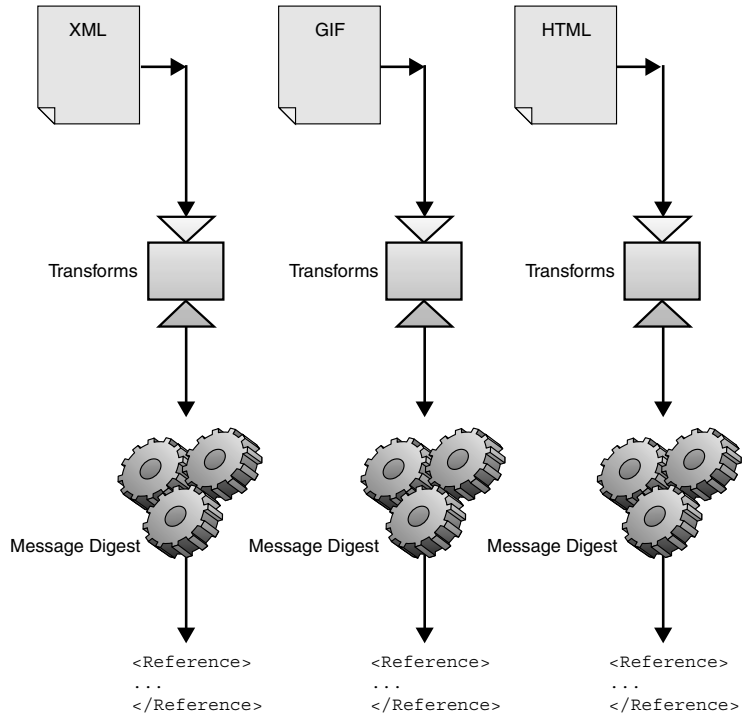
element to form a `<SignedInfo>` element. This `<SignedInfo>` element is subsequently canonicalized and then signed with a private key to yield the `<SignatureValue>`. Finally, the parent `<Signature>` block is formed.

The core generation process is complemented by the reverse process called *core validation*. Core validation is specified in a similar manner as core generation. It is also composed of two substeps: *reference validation* and *signature validation*. The purpose of reference validation is to verify that any data referenced by `<Reference>` elements has not been altered. Digesting the data source and making a comparison against the digest stored in the `<DigestValue>` child element accomplishes this task. Similarly, the purpose of signature validation is to compare the canonical form of the `<SignedInfo>` element against the stored `<SignatureValue>`. The specifics of what happens during core validation are listed in the "Core Validation Steps" box.

Figure 5-3

Signature
generation


```
<Reference>
...
</Reference>
<Reference>
...
</Reference>
<Reference>
...
</Reference>
```

`<SignatureMethod>`   `<CanonicalizationMethod>`

`<SignedInfo>`

Canonicalizer

Signature

```
<SignatureValue>
WWKdFGDkjfnj3x423 ...
</SignatureValue>
```

```
<Signature>
   <SignedInfo>
        …
   </SignedInfo>
   <SignatureValue> … </SignatureValue>
</Signature>
```
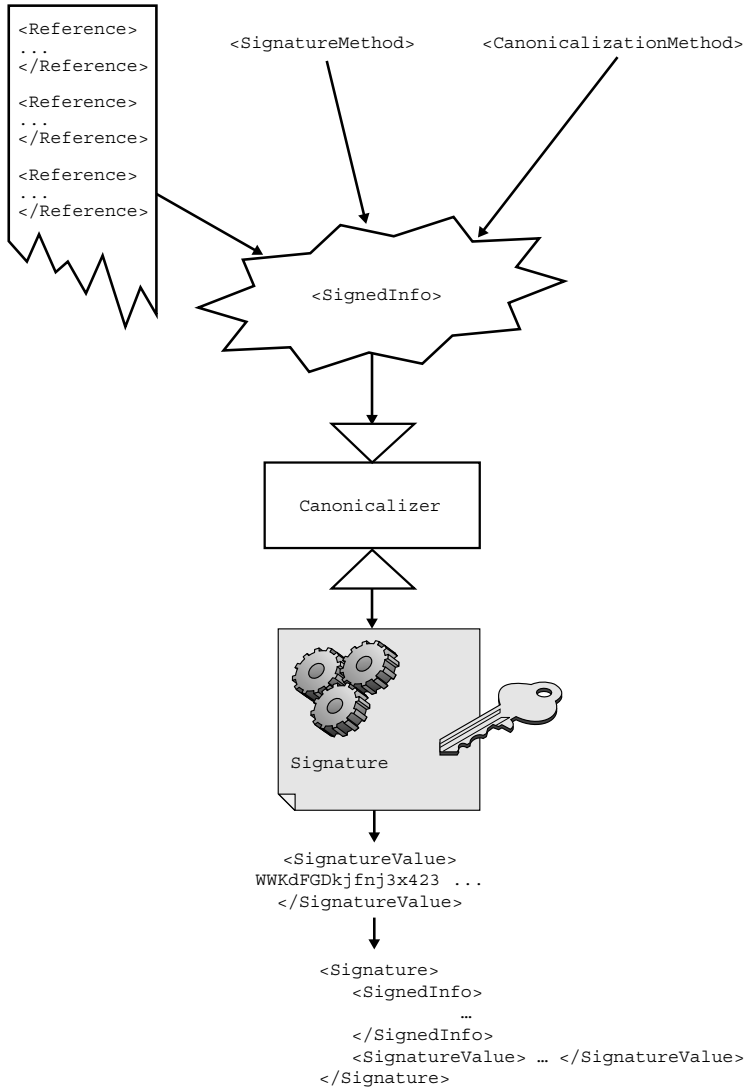
Core Validation Steps

Reference Validation

First, the `<SignedInfo>` element must be canonicalized.

  For each `<Reference>` being validated, perform these steps:

1. The data stream to be digested is obtained by de-referencing the URI attribute of each `<Reference>` element. If no URI attribute is present, the application must know the location of the data source. The final data to be digested is a result of optional transforms that are applied in a cascading manner.

2. The data stream obtained in step 1 must now be digested using the hash function specified in the `<DigestMethod>` element for the current `<Reference>` element being processed.

3. The digest value computed in step 2 is now compared against the `<DigestValue>` element content for the current `<Reference>` element being processed. If these values do not match, reference validation fails.

Signature Validation

1. Retrieve the verification key from a `<KeyInfo>` element or application-specific key source.

2. Using the canonical form of `<SignatureMethod>`, determine the signature algorithm being used and calculate a signature value over the canonical form of the `<SignedInfo>` element. Compare this signature value with the value inside the `<SignatureValue>` element. If these values do not match, signature validation fails.

Core validation is more complex than core generation; a number of issues warrant discussion and further explanation. To begin with, notice that when *reference validation* is performed, we begin by canonicalizing the `<SignedInfo>` element. The careful reader may have noticed that this canonicalization step seems extraneous. The outward syntax of the `<SignedInfo>` element is unrelated to the actual data referenced by a

<Reference> element. That is, canonicalization only alters the syntax of the <SignedInfo> element and does not affect the data stream that flows out of a URI included in a <Reference>. One can make the argument that nothing is gained from a security standpoint by applying canonicalization at this stage, and it may seem more correct to include canonicalization as an operation that is performed as part of the signature validation step instead.

The reason why canonicalization is included in reference generation has to do with an additional requirement that XML Signatures "see" what is signed. When data is included in an XML Signature, it is the *transformed* data that is signed, not the original data. Because of this problem, the possibility exists for a discrepancy between what a user "sees" when a signature is made with the actual signed octets. We will revisit this concept when we give a further treatment of signature transforms. For now, it is enough to know that canonicalization is applied in the first step of reference generation in order to bolster the process requirement that XML Signatures "see" what is signed.

Previous versions of the XML Signature Recommendation specified that the canonicalization algorithm be executed for each <Reference> element in the <SignedInfo> element. This has the effect of executing the canonicalization algorithm $n-1$ extra times where n is the number of <Reference> elements in the <SignedInfo> element. Canonicalization only needs to happen a single time (provided that the <SignedInfo> element isn't being modified during reference validation).

Secondly, in step two of reference validation, the phrase "de-reference a URI" is being used. This phrase hides a great deal of complexity because a URI can represent almost any abstract resource either locally or remotely. Consequently, the means by which a URI is de-referenced is rather important in making sure signatures validate properly. The most common type of URI that will be used is a URI that specifies HTTP as its protocol scheme, although any valid URI scheme is possible. This being the case, it is also possible to specify a bare hostname for signing, such as **http://www.fictional-site.com**. This use of a data source often baffles XML Signature newcomers who are accustomed to raw digital signatures. The question most often asked here is: "If a hostname is specified as a data source, *what* data is actually signed?" The answer to this question lies in how the URI is de-referenced.

A <Reference> element doesn't look at the outward syntax of the URI; it only cares about the raw data itself, whether it is octets or an XML node-set. The phrase "de-referencing a URI" is not well defined in terms of

the XML Signature recommendation. Instead, the definition comes from the basic concept of de-referencing a pointer and obtaining the data pointed to. In the case of the URI, we are obtaining the data pointed to by following the specified protocol scheme and eventually arriving at the data itself. For HTTP, the XML Signature Recommendation states that when a URI is de-referenced, any HTTP status codes that cause redirection must be followed to obtain the final response data.

With this new information in mind, we can now address the previous question regarding the bare hostname URI. When such a URI is used in a `<Reference>` element, the actual data that is signed is the result of a HTTP request to that hostname. The data that is returned is rather mutable because Web sites may change servers, index pages, or switch from HTML to JavaScript for the response data.

Because of the transient nature of such a Web resource, there is little utility in specifying a bare hostname or Web site as part of an authenticated XML Signature. A simple one-byte change by a pragmatic Webmaster can break the hash value for such a URI reference. One convenient way to *see* the data that is being hashed as part of a `<Reference>` element is to use a simple Java program to de-reference the URI and print out the data. Listing 5-1 shows a simple Java program that takes as input a URI from the command line and prints out the de-referenced data.

It is imperative to note that Listing 5-1 shows how a URI is de-referenced to produce octets. Not all URIs are de-referenced in this manner—for example, some signature transforms expect a node-set as input instead of binary octets. An XML node-set is an abstract representation of the underlying form of an XML document. A URI is de-referenced as a node-set when a specific signature transform calls for this type of input or when the URI is a *same-document* reference. We will discuss these issues further when we expand upon URIs and signature transforms. For now, it is enough to know that most external files used in an XML Signature will be de-referenced as binary in the manner described in Listing 5-1.

The final part of step 2 in the "Core Validation Steps" box for reference generation is concerned with transforms. The data to be digested *may* have transforms applied to it. If any transforms are used, they ultimately produce the actual octets that are digested and verified. If transforms are present, it is not an explicit requirement to apply these in every instance of core validation. Likewise, the core generation process does not ensure that transforms were applied in the recommended cascading manner (if at all).

For example, an XML Signature may be validated based on cached data that has already been transformed, or commutable transforms may be

Listing 5-1   URIDereference.java

```
// import the necessary Java packages
import java.io.*;
import java.util.*;
import java.net.URL;
import java.io.ByteArrayInputStream;
/**
* URIDereference.java
*
* This program takes as input a URI and de-references the URI as octets
* and prints out the octets to the screen via the platform's default
* character encoding.
*
* @author Blake H. Dournaee
*
**/
class URIDereference {

  public static void main (String args[]) {
    if (args.length < 1) {
      System.out.println("Usage: URIDereference <URI>");
      System.exit(1);
    }
    String sourceURI = args[0];
    URL url = null;
    byte[] dereferencedData = null;
    InputStream is = null;

    // Load URI using Java's URL class
    try {
      url = new URL(sourceURI);
    } catch (Exception xp) {
      System.out.println("Caught Exception while initializing URL object");
      p.printStackTrace();
    }

    // Create byte-array representation of URL data

    try {

      is = url.openStream();
      int incomingByte;
      Vector v = new Vector();
      Object[] byteArray = null;

      try {

        while ( (incomingByte = is.read()) != -1) {
          Byte temp = new Byte((byte)incomingByte);
          v.add(temp);
        }
      } catch (Exception xp) {
        xp.printStackTrace();
        System.out.println("Caught Exception while reading bytes");
```

Listing 5-1   URIDereference.java (*continued*)

```
     }

   byteArray = v.toArray();
   dereferencedData = new byte[byteArray.length];

   // Convert Byte objects to primitive types

   for (int i=0; i<byteArray.length; i++) {
     dereferencedData[i] = ((Byte)byteArray[i]).byteValue();
   }

   // Convert the byte array to the default character encoding and
   // store it in a String object.

   String output = new String(dereferencedData);
   System.out.println("De-Referenced Data:\n");

   // Output the String
   System.out.println(output);

    } catch (Exception xp) {
     System.out.println("Caught Exception while reading URL stream");
     xp.printStackTrace();

  }
}
```

substituted. The XML Signature Recommendation merely requires that the application finally obtain the data to be digested. A hyperbole that brings this point home would be a software developer doing a rain-dance and somehow obtaining the octets to be digested. This method would be acceptable and would fulfill the requirements of the XML Signature Recommendation.

Reference validation continues in a looping manner until all `<Reference>` elements inside the `<SignedInfo>` have been processed. Once all of the references have been validated, the actual structure of `<SignedInfo>` is verified using the appropriate `<SignatureMethod>` and verification key.

The first step of signature validation specifies that verification key materials need to be obtained. The XML Signature Recommendation does not normatively specify how to obtain this key information, but it suggests that it may be obtained from a `<KeyInfo>` element. This point is important and should be repeated: XML Signatures themselves *do not* offer

trust semantics. It is up to the processing application to make its own trust decisions. The issue of *trust* is delegated to another proposed XML security standard called XKMS. XKMS stands for *XML Key Management Specification* and attempts to pick up the issue of trust where XML Signatures leave it. XKMS uses the `<KeyInfo>` element as a normative reference and attempts to define a vehicle for trust decisions.

In the final step of signature validation, another detail must be noted. The `<SignatureMethod>` is obtained *from* the `<SignedInfo>` element itself. This means that the name of the signature algorithm used is an integral part of the signature itself and it is signed to prevent substitution attacks.

The entire validation process has many steps, but it is a complete, well-defined process that authenticates the data referenced by the signature. It is important to note that according to the XML Signature Recommendation, a signature is *invalid* unless *both* reference validation and signature validation are successful. There is no further qualifier here —*all* references in the `<SignedInfo>` element must pass the digest comparison.

This rather strict view of signature validity is often too strong for practical use. For example, a `<SignedInfo>` element with hundreds of `<Reference>` elements may be able to tolerate some reference validation failures *within certain application-defined circumstances*. Some applications may decide that it is OK to accept a signature as valid if reference validation or signature validation fails. It is possible for reference generation to fail if the data referred to by a `<Reference>` element changes, and it is also possible for signature validation to fail (but reference validation to succeed) if the structure of the `<SignedInfo>` changes.

These additional weaker semantics represent a hack of the definition provided by the XML Signature Recommendation. The definition of core validation is very strict; it is defined to include both *reference validation* and *signature validation*. Whether or not it is possible to ascribe validity to a signature that only fulfills part of core validation falls outside the scope of the XML Signature Recommendation.

These additional trust semantics concerning the validity of an XML Signature can be quite confusing to those who have only dealt with raw binary signatures. That is, if we were to take a raw digital signature block (such as a binary RSA signature) and verify the signature against an original document, the answer is always going to be *valid* or *invalid*. No gray area exists in terms of the signature validity; there is no additional granularity in assessing validity. If the hashes in a raw digital signature fail to

match, the signature is invalid by definition. For an XML Signature, the answer to the question of validity ideally comes from a positive core generation result, but an application may also take other variables into consideration and ascribe validity based on its own semantics. This can be done because of the added granularity of core validation that is just not present in raw digital signatures.

Many reference implementations of XML Signatures in popular developer toolkits have features for demarcating signature validity from reference validity. Whether or not this feature is a blessing or a burden has yet to be seen. The answer lies in the strength of the additional trust semantics that are commonly used when XML signature validity is assessed.

The URI Attribute: Additional Features

Thus far, we have seen how local or remote resources can be specified by location through the use of a URI. To access data that resides at a distant location, one might use the HTTP scheme, and to access a file locally one could use the FILE scheme (such as `file:///C:\myfile.txt` for a Windows-based machine). Remember, any valid URI scheme is possible, although the method by which the data is obtained is left outside the scope of the XML Signature Recommendation.

The granularity of *what* we can access using these two methods is rather sparse. We can access an entire remote file or an entire local file using the two before-mentioned URI schemes, but nothing more. To someone accustomed to a raw digital signature, this may seem sufficient, but in reality this is largely inadequate because these basic schemes overlook the rich structure of an XML document. More specifically, great utility exists in being able to pinpoint sections of an XML document to participate in a digital signature. Ideally, we would like to be able to include a sub-structure of a large XML document, or even the content of a single element. Listing 5-2 shows an example of the components of an XML document that we might want to isolate.

In Listing 5-2, we can think of this XML structure as the entirety of a given XML document. If this is the case, we already have the means to sign the outermost element (and its children): `<MartialArts>`. This would be equivalent to a URI reference that specifies the entire file. What about signing the child element `<Aikido>` and explicitly *excluding* the parent element `<MartialArts>` and its remaining children from the signature? Or what about focusing solely on the `<Grading3>` element that

Listing 5-2

XML document components: martialarts.xml

```
<MartialArts>
  <Aikido Id="FirstElement">
    <Gradings>
      <Grading1>San Dan</Grading1>
      <Grading2>Ni Dan</Grading2>
      <Grading3>Sho Dan</Grading3>
    </Gradings>
  </Aikido>
  <Karate Id="SecondElement">
    <Gradings>
      <Grading1>Yellow Belt</Grading1>
      <Grading2>Brown Belt</Grading2>
      <Grading3>Black Belt</Grading3>
    </Gradings>
  </Karate>
</MartialArts>
```

resides inside the `<Aikido>` parent? The possibilities here are endless and very application-specific. Most of the processing that is done to isolate a subset of a given XML document is accomplished by *signature transforms,* although some minimal selection processing can be done with URIs alone.

Before we delve into transforms further, we need to examine the specific URI syntax that is used to enable an XML Signature to deal with situations where XML document subsets are going to be processed. That is, the question we are about to answer is concerned with *how* we write URIs in a `<Reference>` element in cases when we are *not* signing an entire file. Listing 5-3 shows three possibilities, each of which attempts to specify some subset of an XML document.

In Listing 5-3(a), we are using a *fragment identifier* to select a specific element inside a remote XML file. This type of fragment identifier denotes the *octet-stream* beginning with the `<Aikido>` element inside martialarts.xml. The data selected is equivalent to the `<Aikido>` element and all of its children (but not the following `<Karate>` element), as shown in Listing 5-2. For those readers familiar with HTML, the behavior shown here resembles an HTML fragment identifier that is based on an *anchor* element. The difference here is that the fragment identifier used in conjunction with an XML document can specify any element, not just an anchor (`<a>`) element, as in HTML.

The XML Signature Recommendation gives us a special caveat with regard to this type of fragment identifier and cautions against it. The semantics of the fragment identifier are based on the type of data refer-

Listing 5-3

URI syntax for an
XML document
subset

Listing 5-3(a)

```
<Reference Id="ExampleA"
URI="http://www.foo.com/martialarts.xml#FirstElement">
 . . .
</Reference>
```

Listing 5-3(b)

```
<Reference Id="ExampleB" URI="#FirstElement">
 . . .
</Reference>
```

Listing 5-3(c)

```
<Reference Id="ExampleC" URI="">
 . . .
</Reference>
```

enced and may produce different results based on how the referent data
is processed. One example of this is the use of a fragment identifier in an
HTML document. If two anchor elements are present that have the same
attribute value, the behavior of fragment processing is undefined. That is,
if an HTML document, `file.html`, has two anchor elements of the form
`<a name="Marcy">` and this element is referenced via a fragment iden-
tifier via `http://www.foo.com/file.html#Marcy`, it is largely unde-
fined which element will be chosen. It is true that we have jumped from
talking about XML documents to HTML documents, but the reasoning is
the same. The XML Recommendation has a cleaner, more standardized
way of selecting XML subcomponents and cautions against fragment
identifiers that follow URIs, as shown in Listing 5-3(a). More information
on this type of fragment identifier is given in Chapter 6.

In Listing 5-3(b), the referent here is to the node-set of the element
(including all its children) that has `"FirstElement"` as its `Id` attribute
value. The potentially confusing point here is that this type of reference
alludes to the element within the *current XML document*. It is a self-
reference (this is why there is no preceding URI scheme or location) to
an element somewhere in the *current* XML document containing the
`<Signature>` element. This type of reference is used most often within
enveloping XML Signatures to include `<Object>` or `<Manifest>`
elements as part of a signature.

The other thing to note is the use of the term *node-set* in this example, as contrasted with *octet-stream* used in the previous example. When we say node-set, this means that the XML document is going to be processed as XML and when we say octet-stream this means that the XML document is going to be processed as binary. This subtle distinction will be explored further when we examine signature transforms. A concrete example of how a bare fragment identifier is used can be seen in Listing 5-4.

A few important things should be noted about Listing 5-4. First off, the signature appears to be an *enveloped signature*. This however, is incorrect. The careful reader will notice that this signature is actually a *detached signature*. The URI reference signs a *sibling* element in the current XML document (in this case, the `<Aikido>` element) and not the entire parent document (which would make it a proper *enveloped signature*). Next, focus your attention on the `<Reference>` element (highlighted in bold). This `<Reference>` element uses a bare fragment identifier `#FirstElement`. This effectively *includes* the contents of the `<Aikido>` element in the signature and *excludes* the remaining child elements of `<MartialArts>`. That is, we can alter any element (except for the `<Aikido>` element, which is signed) of the original document without breaking the signature. This property is especially powerful and exciting because it enables us to pass our signed document to others who can change or add certain pieces without affecting the signature validity. For example, we could make the change shown in Listing 5-5 without affecting the validity of the signature.

The change made in Listing 5-5 is quite significant in terms of changed octets. This change does not affect the validity of the signature because of the property of the *fragment identifier* used. It is also important to note that such a bare fragment identifier can also be used in an *enveloping* signature—we have seen examples of enveloping signatures in Chapter 4 when we discussed the particular syntax of the `<Object>` element.

The last type of URI identifier shown in Listing 5-3(c) is an empty set of double quotes. This is called a *same document reference* and is related to the bare fragment identifier just discussed and shown in Listing 5-4. In fact, the empty URI same document reference is a more general reference that simply includes the entire node-set of the *current* XML document. This sort of document reference is used when arbitrary signature transforms are going to be applied to the document. For example, with the bare fragment identifier, we have the ability to demarcate and sign a single child element and its descendants. What about a situation where we need

Listing 5-4   Example of a bare fragment identifier for a URI reference

```xml
<?xml version="1.0" encoding="UTF-8"?>
<MartialArts>
   <Aikido Id="FirstElement">
     <Gradings>
        <Grading1>San Dan</Grading1>
        <Grading2>Ni Dan</Grading2>
        <Grading3>Sho Dan</Grading3>
     </Gradings>
   </Aikido>
   <Karate Id="SecondElement">
    <Gradings>
        <Grading1>Yellow Belt</Grading1>
        <Grading2>Brown Belt</Grading2>
        <Grading3>Black Belt</Grading3>
    </Gradings>
   </Karate>
   <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
     <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="#FirstElement">
         <Transforms>
           <Transform
          Algorithm="http://www.w3.org/2000/09 /xmldsig#enveloped-signature"/>
           <Transform
          Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
         </Transforms>
         <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"/>
         <DigestValue>ktEp0766IBHkRN8mxV5U9o1FIYs=</DigestValue>
       </Reference>
     </SignedInfo>

     <SignatureValue>
       s5zdhPftVea+3z12CwP0rYn
       CMgBjg03Owa+/9ZZVwo089Bzl74Rjt1Ulf7Z+z2R
       oz0NH75bFrg7XBzoI2gzEwg==
     </SignatureValue>
     <KeyInfo>
       <KeyValue>
         <RSAKeyValue>
           <Modulus>
             1u4kN2qOpGbP2rqvj7t16Pp6dvo7IhuOlb
             dJywBFTFdB0h5eAMyGkrGyi6VpXnGRRxEe
             0S8iPgs3A7NewkQrqw==
           </Modulus>
         <Exponent>EQ==</Exponent>
        </RSAKeyValue>
       </KeyValue>
     </KeyInfo>
   </Signature>
</MartialArts>
```

Listing 5-5 Changing the original document to maintain signature validity

```xml
<?xml version="1.0" encoding="UTF-8"?>
<MartialArts>
  <Aikido Id="FirstElement">
    <Gradings>
       <Grading1>San Dan</Grading1>
       <Grading2>Ni Dan</Grading2>
       <Grading3>Sho Dan</Grading3>
    </Gradings>
  </Aikido>
  <!-- I have deleted the <Karate> element because we ran out of belts. All
  we have left is neon-green and pink belts -->
  <KungFu Id="SecondElement">
  <!-- I need to research gradings for this martial art. -->
    <Gradings/>
  </KungFu>

  <Signature xmlns="http://www.w3.org/2000/09 /xmldsig#">
    <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 URI5"#FirstElement">
        <Transforms>
          <Transform
        Algorithm5"http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
          <Transform
        Algorithm5"http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
        </Transforms>
        <DigestMethod Algorithm5"http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>ktEp0766IBHkRN8mxV5U9o1FIYs5</DigestValue>
      </Reference>
    </SignedInfo>
     <SignatureValue>
       s5zdhPftVea+3z12CwP0rYn
       CMgBjg03Owa+/9ZZVwo089Bzl74Rjt1Ulf7Z+z2R
       oz0NH75bFrg7XBzoI2gzEwg==
     </SignatureValue>
     <KeyInfo>
       <KeyValue>
        <RSAKeyValue>
          <Modulus>
            1u4kN2qOpGbP2rqvj7tl6Pp6dvo7IhuOlb
            dJywBFTFdB0h5eAMyGkrGyi6VpXnGRRxEe
            0S8iPgs3A7NewkQrqw==
          </Modulus>
         <Exponent>EQ==</Exponent>
        </RSAKeyValue>
       </KeyValue>
     </KeyInfo>
   </Signature>
</MartialArts>
```

to sign a more complex subset? Listing 5-6 shows such a requirement. In this example, we would like to sign *just* the elements in bold.

The elements we want to include in the signature (shown in Listing 5-6 in bold) are as follows: The first and third elements of the first `<Gradings>` element in the document, as well as the entire contents of the `<Karate>` element. This sort of complex content selection cannot be done with a bare fragment identifier. The job of the same-document reference helps us obtain sophisticated signature results because it refers to the entire node-set, which is subsequently passed to various signature transforms (an XPath transform would be able to accomplish the selection required previously). In a sense, the same document reference is in itself a simpler reference, but the ultimate utility in this type of reference lies in what happens to this node-set as various signature transforms are applied to it.

Jumping back and forth between remote external documents and self-referring resources can be quite confusing. One point of confusion here lies in the logistical obstacles for the signing to actually happen. For remote references, the way the reference is specified when it is signed is symmetric with the way it is specified when it is validated. For example, in core generation, we think of a picture that shows `<Reference>` elements being formed and then placed into a `<SignedInfo>` element for signing to happen. It is clear that the data to be signed resides at a given URI. When a same-document reference is used, however, we are only specifying an empty set of quotes or a one-word identifier. It is clear that the XML Signature application must still obtain the source document to sign from

Listing 5-6

A complex XML document subset

```
<MartialArts>
  <Aikido Id="FirstElement">
    <Gradings>
      <Grading1>San Dan</Grading1>
      <Grading2>Ni Dan</Grading2>
      <Grading3>Sho Dan</Grading3>
    </Gradings>
  </Aikido>
  <Karate Id="SecondElement">
    <Gradings>
      <Grading1>Yellow Belt</Grading1>
      <Grading2>Brown Belt</Grading2>
      <Grading3>Black Belt</Grading3>
    </Gradings>
  </Karate>
</MartialArts>
```

somewhere. When the signature is verified, the source document is embedded along with the XML Signature when an enveloped or enveloping signature is used. The question remains then, where does the actual source document come from that is filtered for an XML document subset?

This complexity lies outside of the XML Signature-processing rules and is actually an application-specific task. Typically, the source XML document in a same document reference or bare fragment identifier is somehow given to the processing application in a customized manner *beforehand*. That is, an explicit mechanism specifies an actual filename or URI where the location of the source document can be specified. We will see how this practical issue is dealt with when we look at a specific XML Signature implementation in Chapter 8.

Signature Transforms

It can be argued that the true power of XML as a portable data format lies within the vast array of possibilities for transforming XML in a standard way. Most introductory books on XML contain a section on how an XML document can be transformed and filtered to produce various types of formatted output ideal for presentation. XML Signatures have the potential to take transformations to a new level of utility in terms of security applications. For example, it is possible to apply transformations that include or exclude specific portions of a source document.

This type of feature might be useful for electronic contracts in the legal domain that require multiple signers, endorsements, or any manipulation of structure (such as moving or inserting signature blocks) without breaking the digital signature. If digital signatures *act* like normal, handwritten contracts, their adoption and use will eventually become more widespread. The use of signature transforms, however, is not limited to the legal domain exclusively. There have also been applications of XML Signatures to Web forms, online contract signing, and e-commerce—all of which are impossible without the capability to provide a reliable means to be selective about the data being signed while still including additional mutable information in the signature itself.

When we discussed the basics of signature transforms in Figure 5-1, we used the analogy of a waterfall to show how the data flows from a source and goes through a series of transformations to finally become the data

that is provided as input to the digest function. In this sense, a signature transforms plays two distinct roles. The abstract high-level role of a set of signature transforms is to ultimately *describe* the data to be digested. At a more concrete level, a single signature transform is simply an algorithm that operates on at least one input parameter: the result of a URI de-reference or the output of an earlier transform.

To complicate things further, the XML Signature Recommendation enables custom transforms to be used. This will probably hinder interoperability between implementations because an XML Signature that uses a custom transform will be useless to an application that doesn't support or know about this custom transform algorithm. The current XML Signature Recommendation mentions five transforms, four of which are either required or recommended transform algorithms. This list is shown in Table 5-1 and will probably see new additions as XML Signatures mature and new versions of the Recommendation are devised.

Each transform listed in Table 5-1 has a specific purpose and intent. The first potentially confusing point is the appearance of Canonical XML in the list of transform algorithms. In previous discussions, we mentioned that canonicalization was used as a part of *core validation* and *core generation* to ensure that the `<SignedInfo>` element is stripped of syntactic processing changes. It turns out that canonicalization is used in a subtle way inside the transformation waterfall. In short, it is also a transformation itself. The default transformations specified in the XML Signature Recommendation are specified to operate on two basic, broad *data types*: binary octets or a node-set. When a URI is de-referenced or subsequently transformed, the result is either an octet stream or an XPath node-set. For example, Table 5-2 shows common ways of specifying a source URI in a `<Reference>` element and the *data type* that would result from such a reference.

Table 5-1	Transform Name	Keyword
XML Signature Transforms	Canonical XML	Required
	Base64 decoding	Required
	XPath filtering	Recommended
	Enveloped transform	Required
	XSLT transform	Optional

URI Type	Data Type
URI="http://www.fictional-site.com/foo.gif"	Binary octets
URI="http://www.fictional-site.com/foo.xml"	Binary octets
URI="http://www.fictional-site.com/foo.pdf"	Binary octets
URI="file:///C:\files\foo.doc"	Binary octets
URI="#SomeElement"	Node-set
URI=""	Node-set

Canonical XML (canonicalization algorithm) is used when any given transformation requires a conversion from a node-set to binary octets. For example, suppose we are de-referencing the URI, "#SomeElement". According to the XML Signature Recommendation, this type of URI Reference must be treated as a node-set. To treat data as a node-set means that it will be processed as XML. Because of this, the possibility exists for syntactic changes to the data that do not affect its meaning.

For example, if we were to de-reference "#SomeElement" and pass the node-set to a DOM parser for processing, the simple act of reading the XML data and spitting it back out without any explicit changes may result in syntactic changes. Furthermore, the final resting place for the data from a URI is going to be input for a digest function, which operates solely on binary octets. This means that the XML must be canonicalized *before* it is digested to ensure that a one-to-one mapping takes place between the meaning of the XML data and its hash value. Canonicalization *converts* the node-set to binary octets before it is input into the digest function.

If a transformation operates in the other direction, canonicalization is *not* explicitly required, because it is done anyway. To see this, consider what would happen if we had a transform that operated in the opposite direction. Begin with binary octets and use a transform that operates on a node-set. For example, suppose we are de-referencing the URI, file:///C:\foo.xml. According to the XML Signature Recommendation, this type of URI Reference must be treated as binary octets. Suppose next that foo.xml is a well-formed XML document that we would like to process as XML (and perhaps exclude portions of the original document from the signature). This could be done with an XPath transformation using the proper XPath expression (we will visit this in more detail later on).

The problem inherent with our processing thus far is that the XPath transform requires a node-set as input. We can convert the octets to a node-set, but canonicalization here would be redundant because the final digest algorithm operates on binary octets. The node-set would have to be converted to binary (canonicalized) at the end of the transformation waterfall anyway. As long as the digest algorithm always operates on binary octets, canonicalization used in a previous transformation would be redundant. Figure 5-4 shows the transformation lifecycle of a URI that is de-referenced as binary octets, converted into a node-set, and then finally digested.

Notice in Figure 5-4 the source URI is de-referenced as binary octets. These binary octets are subsequently transformed into a node-set (C_1) and then passed to an XPath transform (T_1). Next, the node-set that is output by the XPath transform must be canonicalized and converted to an octet stream. Both the canonicalization and the conversion from node-set to

Figure 5-4

The transformation lifecycle of a binary URI-Reference


```
<DigestValue>ktEp0766IBHkRN8mxV5U9o1FIYs=</DigestValue>
```
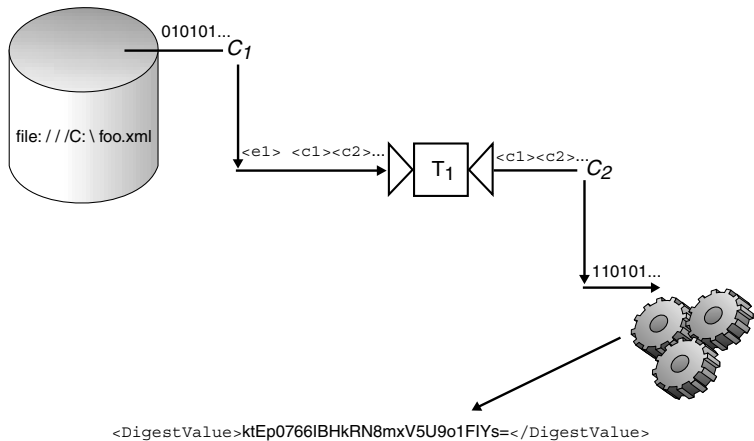
1. The data is de-referenced as octets.
2. The data is then converted (C_1) into an XPath node-set.
3. An XPath transformation occurs (T_1), which requires a node-set as input.
4. The data is then converted (C_2) from an xPath node-set back to octets using the canonicalization algorithm.
5. The data is digested and stored in a `<DigestValue>` element.

octet stream happen in one step; the default canonicalization algorithm used by the XML Signature Recommendation (Canonical XML) outputs its result in octets. Finally, the octets can be passed to a digest function and hashed to become part of a `<DigestValue>` element.

As a general rule, any time XML data in the transformation waterfall is operated on as XML (node-set), it *must* be canonicalized before it is digested. XML canonicalization is arguably one of the most confusing and potentially dangerous (but nonetheless necessary) aspects of XML Signature processing. The next section explores some of the basics about how canonicalization works and we'll dig a bit deeper into why it is such a critical part of XML Signature processing.

Canonicalization

Canonicalization is the first transform that we will examine of the four recommended and required transforms. The canonicalization method is set via an empty element with an attribute identifier. The identifier is once again a URI identifier. Here's an example:

```
<CanonicalizationMethod
 Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
```

This empty element resides inside the `<SignedInfo>` element and is therefore signed to resist replacement attacks. The default canonicalization algorithm is Canonical XML 1.0 (specified previously), but any acceptable canonicalization algorithm can be used. We will only consider Canonical XML in our discussion here, so we might often interchange references to canonicalization algorithm and Canonical XML, even though the first term is far broader than the second. To gain a strong understanding of something as complex as XML canonicalization, it is best to begin with some general truths and definitions that bring about the need for canonicalization.

XML Truths

1. *Syntactic changes are permissible by XML 1.0 that preserve logical equivalence.*

2. *Syntactic changes are permissible by Namespaces in XML that preserve logical equivalence.*

Because of these broad truths, we can define Canonical XML. See Definition 5-1.

Next, we can define the theoretical goal of Canonical XML. See Definition 5-2.

Definition 5-1

Canonical XML   A method for generating a physical representation of an XML document that accounts for permissible changes that preserve logical equivalence.

Definition 5-2

Canonical XML Goal   If two documents, D and D', have the same canonical form, then the two documents are logically equivalent, subject to a specific application context.

Notice that we have made our first reference to XML Namespaces. An XML Namespace is simply a mechanism for grouping element and attribute names. For more information on XML Namespaces, see the primer in Chapter 3.

Definitions are all well and good, but we probably need to look at some examples to get our minds rolling. First, consider the following pair of XML documents:

Document D:

```
<d>Four score and seven years ago</d>
```

Document D':

```
<d>4 score and 7 years ago</d>
```

Although we haven't explicitly defined what canonical form *is* yet, let's assume that both D and D' are in canonical form for the sake of this example. Consider the question, "Are D and D' logically equivalent?" The answer is both yes and no. The reason for the duality here lies in the qualifier tacked on to the end of Definition 5-2. If we were to consider the hash values of D and D', then clearly the octets are different and each document would produce a different hash value. A change like this would break an XML Signature; however, at this point in our discussion, we are evaluating the theoretical goal of Canonical XML, and if we choose to let "Four" be equivalent to "4" within our application, then we *could* say that these two documents are indeed logically equivalent.

To push this example further, consider the following pair of XML Documents:

Document D:

```
<d>throw my basketball into the hoop</d>
```

Document D':

```
<d>throw my boss into the pool</d>
```

Suppose that in our application, we decided that "boss" is logically equivalent to "basketball" and "hoop" is logically equivalent to "pool." This would be an outlandish thing to do, but this example shows that we cannot possibly account for application-customized semantics in a single canonicalization algorithm. If we made D and D' logically equivalent within our application, it would be impossible to incorporate this sort of arbitrary association within a generalized algorithm like Canonical XML. This realization leads us to our last definition regarding canonicalization. This statement is a negative statement and explicitly specifies a nongoal of Canonical XML:

Definition 5-3

Canonical XML Nongoal   Two XML Documents, D and D', are equivalent if and only if their canonical forms are identical.

As we have seen from our previous examples, Definition 5-3 is completely unachievable and is not the ultimate goal of Canonical XML.

Now that we have some broad notions in place, we can begin to look at the specifics of how Canonical XML works and some of the important things to notice. First off, two flavors of Canonical XML are available. One version of the algorithm preserves comments in the original input data, which is called *Canonical XML with comments*. The version of Canonical XML that omits (removes) comments from the final canonicalized form is called *Canonical XML without comments*. The XML Signature Recommendation refers to these two canonicalization algorithms as separate transformation algorithms; the distinction between the two is really quite trivial. Both define the same algorithm, but one version explicitly omits comments from the input node-set with a simple boolean test.

Secondly, Canonical XML uses the concept of a node-set for its main processing mechanism. A node-set is more correctly referred to as an XPath node-set and is the same abstraction discussed in the XPath primer in Chapter 3. A node-set is defined in the data model for XPath and is simply a logical representation of an XML document. The types of possible nodes include *element, attribute, namespace, text, comment, processing instruction*, and *root*.

The ultimate product of Canonical XML is a physical representation; that is, it produces binary octets as its final output data type. It takes as its input either an octet stream or a node-set. If the input is an octet stream, it must be first be converted to a node-set. The reason why is because all of the processing done by Canonical XML is based around the abstraction of a node-set.

Canonical XML can be thought of as operating on two node-sets, an *input node-set* and an *output node-set*. If a node is in the output node-set, it will be rendered as text in the final physical representation. The transformation of the input node-set to the output node-set defines the nature of the canonicalization. For example, data comes into an instance of the Canonical XML transformation. If it is binary, the octets are converted to an input node-set. If it is already a node-set, then this node-set becomes the input node-set.

From here, we select all of the nodes in the input node-set and go to work. At this point, if we are performing canonical XML without comments, we will omit the comments in this selection process. Then we apply the rules of Canonical XML, which are detailed in the following list (taken from the Canonical XML 1.0 Recommendation: **http://www.w3.org/TR/ 2001/REC-xml-c14n-20010315**. The rules listed here summarize two

steps of the canonicalization algorithm: (1) the transformations that happen to convert the input node-set to the output node-set and (2) the steps that occur when Canonical XML converts the output node-set to binary octets. For those readers interested in more detailed implementation information for Canonical XML, they should refer to the current Canonical XML Recommendation.

- The document is encoded in UTF-8.
- Line breaks are normalized to #xA on input before any node-set operations occur.
- Attribute values are normalized, as if by a validating processor.
- Character and parsed entity references are replaced.
- CDATA sections are replaced with their character content.
- The XML declaration and document type declaration are removed.
- Empty elements are converted to start-end tag pairs.
- Whitespace *outside* of the document element and within the start and end tags is normalized.
- All whitespace in character content is retained (excluding characters removed during linefeed normalization).
- Attribute value delimiters are set to quotation marks (double quotes).
- Special characters in attribute values and character content are replaced by character references.
- Superfluous namespace declarations are removed from each element.
- Default attributes are added to each element.
- Lexicographic order is imposed on the namespace declarations and attributes of each element.

A list of changes like this does little to convey any real understanding. As mentioned before, canonicalization is a complex operation with many aspects. Another way of understanding Canonical XML 1.0 is to fixate around the common permissible changes that occur during XML processing. The XML Signature Recommendation defines four basic types of change categories that are expected. These categories include *XML Processor changes, XML Parser changes, UTF conversion changes,* and *Namespace changes.* Let us examine these four categories.

XML Processor Changes The simple act of reading an XML document incurs some basic normalization. The easiest example of basic normalization is concerned with line endings. Different operating systems use different combinations of the carriage-return and line-feed characters to denote ends of lines. XML attempts to put an end to this madness by specifying that when an XML document is processed, before it is *parsed,* all line-endings must be normalized to the single line-feed character. That is, any combination of line-feed and carriage return or just a lone carriage-return are all normalized to a single line feed character.

In addition to line feed normalization, the XML 1.0 Recommendation also specifies what is called *attribute-value normalization.* This is normalization that also happens before an XML document is parsed. This type of normalization includes replacing character references, entities references, and normalizing whitespace within attribute values.

Finally, additional normalization that is done when XML is processed includes providing values for attributes with default values (if they are missing) and replacing character and entity references that appear outside of attribute values. All the details of this type of normalization are spelled out in great detail in the XML 1.0 Recommendation and the XML Signature Recommendation. Remember, we are focusing on these changes because they are material changes to the physical representation of an XML document that *preserve* the semantics of the document. For example, the computed hash value of an XML document with a carriage-return and line feed for a line ending is completely different than the hash value for a line-feed alone, although the meaning or intent of the character sequences are exactly the same. They both denote an end of line.

XML Parser Changes Two well-known methods are used for creating a data-structure representation of an XML document. One is to use the document Object Model (DOM), which views an XML document as a tree of nodes. The other is to use Simple API for XML (SAX), which views an XML document as a string of events. The simple use of these parsers also incurs material physical changes that do not affect the meaning of the data.

UTF Conversion Changes The XML Recommendation specifies that both UTF-8 and UTF-16 (full Unicode) must be supported as a viable character encoding. Because of this, it is possible that an XML Processor may convert from one encoding to the other and introduce subtle syntactic changes. For example, if an XML Processor reads an XML document as

UTF-8 and then subsequently converts it to UTF-16 and adds characters outside of the UTF-8 character set, these characters must be converted to numbered character entities if yet another XML Processor uses UTF-8. This amounts to a material syntactic change that preserves the meaning of the data (a well-defined relationship exists between a numbered character entity and its corresponding character). Such a change would break a signature because the two documents would produce different hash values.

Namespace Changes When XML namespaces are used, assigning a namespace to a particular element or a group of elements can be done in a number of ways. This is another case where the syntax is different, but the meaning is the same. For example, consider the following two XML documents:

Document D:

```
<?xml version="1.0"?>
<superns:herolist
xmlns:superns="http://www.superheroes.org/superns/">
  <superns:superheroes>
    <superns:hero>Superman</superns:hero>
    <superns:hero>Batman</superns:hero>
    <superns:hero>Wonderwoman</superns:her>
  </superns:superheroes>
</superns:herolist>
```

Document D':

```
<?xml version="1.0"?>
<herolist xmlns="http://www.superheroes.org/superns/">
  <superheroes>
    <hero>Superman</hero>
    <hero>Batman</hero>
    <hero>Wonderwoman</hero>
  </superheroes>
</herolist>
```

It is clear that the syntax of these two documents is completely different, but according to the allowable rules for Namespaces in XML, these two documents *mean* the same thing. Document D relies on an explicit namespace declaration to provide the association between the elements in the document, and their associated namespace and document D' relies on the *default namespace declaration* to accomplish the same thing. This example, however, isn't something that is handled by Canonical XML. When the explicit namespace prefix is used, the name of the element is effectively changed. This isn't something that can be forseen by Canonical

XML. Other problems occur with XML Namespaces especially when signatures are moved from one XML document to another. The correct canonicalization of XML Namespaces in a portable document often requires an alternative canonicalization algorithm called exclusive canonicalization, which is discussed in the Appendix for this book.

The number of cases that must be considered and dealt with in terms of permissible changes to XML is vast. It is because of this that great care must be taken with the use of the canonicalization algorithm. An XML Signature application must be certain that it is not using a rogue canonicalization algorithm. For example, if an attacker has the means to replace the canonicalization algorithm used during XML Signature processing, the rogue algorithm could be used to transform the input into arbitrary signatures that always fail or pass validation. Canonicalization is a major security concern and therefore this algorithm must be completely trusted at all times because an XML Signature application is essentially relying on it to produce the original data that was signed before XML Processing occurred.

Because the canonicalization algorithm is complex and normalizes many cases, it is also possible for an attacker to replace the canonicalization algorithm with an algorithm that works *almost* the same way as it is supposed to. It may conveniently omit a certain normalization step that is engineered to affect the result of a transformation that occurs further along, which could subsequently alter the nature of what was signed.

Other potential problems with canonicalization are that it is slow in terms of its performance and that no easy way exists for checking to see if it has been performed ahead of time. Because of this, canonicalization must be performed *every time* a signature is generated or verified. The algorithm itself also has little room for optimizations. Every input node in the document must be considered for canonicalization and subsequently passed on to the output node-set or explicitly ignored.

Base64 Decoding

The second transform algorithm that we will discuss is base64 decoding. This is a well-known algorithm and is given full treatment in the primer in Chapter 2. Base64 decoding is used in XML Signatures to decode and sign encoded binary files or to decode and sign data referenced in an <Object> element or other external XML resource that contains base64-encoded data. This transform always produces an octet-stream as output and can accept either an XPath node-set as input or an octet stream as input. This may seem a bit confusing at first. Newcomers to XML

Signatures view base64 decoding as an algorithm that operates on text and produces binary as its output. It isn't defined on a node-set data type.

The reason why support for this is necessary is because of the same-document URI references that were discussed in Table 5-2. For example, consider Listing 5-7.

A few things should be noticed about this example. First is the base64-encoded data that resides in the `<Object>` element at the bottom of the signature. This element contains arbitrary base64-encoded data and must be decoded in order to verify the digest value. The same document reference used here (URI=#object) identifies a node-set.

It is precisely here where we have the data type problem. Base64 decoding normally relies on an octet stream as input. We now have a situation where the node-set must be converted to an octet stream. The mechanism for how this is done is specified in the XML Signature Recommendation and involves stripping the tags to get at the actual data. Notice that the way the `<Object>` element is processed is dependent

---

Listing 5-7 Enveloping a XML Signature with an embedded `<Object>` element

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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="#object">
      <Transforms>
        <Transform
         Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"/>
      <DigestValue>DiHakF8GUVLTCii+GPBhB3gEJMk=</DigestValue>
    </Reference>
  </SignedInfo>
<SignatureValue>
  Byg6L8qTg5a7vqDd9ViPgkoGW7mBpUlIDx9/aYAd2NdiU4ev
  0S+9Df5YtCv9I/G1TlIr5pYxuHpl1gXDFzlNkQ==
</SignatureValue>
<Object Id="object">
  PE1hcnRpYWxBcnRRzPg0KICA8QWlraWRvIElkPSJGaXJzdEV
  sZW1lbnQiPg0KICAgIDxHcmFkZW5jz4NCiAgICAgIDxHcm
  FkaW5nMT5TYW4gRGFuPC9HcmFkaW5nMT4NCiAgICAgIDxHc
  mFkaW5nMj5OaSBBSEEYW48L0dyYWRpbmcyPg0KICAgICAgPEdy
  Pg0KPC9NYXJ0aWFsQXJ0cz4NCg==
</Object>
</Signature>
```

upon the transforms used. If base64 decoding is not specified as a transform, the entire contents of the `<Object>` element (including start and end tags) would have been digested, giving us the digest value of the base64 representation instead of the underlying data.

It is also pertinent to note that the `<Object>` element needs to be canonicalized before processing. In the previous situation, the canonicalization transform is explicitly left out because it would have ruined the example. The reason why is because the canonicalization algorithm would have produced octets as its output, successfully canonicalizing the node-set for the `<Object>` element. We would now have a situation where octets are being supplied to the base64 decoding algorithm, giving us nothing interesting to talk about for the example.

Careful readers may have noticed that only base64 *decoding* is mentioned as a transform, not base64 *encoding*. This may seem odd at first, but Base64 encoding is a pervasive operation frequently used to transmit binary data in a text format. Because of this notion, it seems like a perfect addition to XML Signatures. To add to the confusion, the XML Signature Recommendation lists base64 under the heading "Encoding" in the XML Signature Recommendation. This can be confusing because it is referred to as an *encoding algorithm*, but it is only used for *decoding* in XML Signatures. Base64 encoding is not used as a transform because the `<Object>` element is the standard way of including additional data (whether encoded or not), and this data isn't modified as part of the transformation waterfall. Further, user-defined transforms are possible, so an application that really wanted to implement this transform could do so.

XPath Filtering

The XPath transform is an interesting transform because it lets us filter the document to a very fine level of granularity. XPath is used in a straightforward manner in XML Signatures. The expression itself is simply embedded inside an `<XPath>` element that is placed inside a `<Transform>` element. When it comes time for this particular transform to be applied within the transformation waterfall, the XPath transform operates on a node-set as input. More specifically, all nodes that comprise the document are selected, and then the XPath expression is evaluated *for each node in the document*.

The result of the XPath evaluation is always a Boolean value, true or false. The XPath transform produces an output node-set; a value of true means that the node just evaluated will be placed in the output node-set and a value of false implies that the node will be discarded. Because this

evaluation happens for each node in the document, XPath filtering can be quite slow. An example of the syntax of an XPath transform as it is used in an XML Signature is shown in Listing 5-8.

Notice that the transform is called XPath *filtering,* which is an important point. The XPath transform enables us to select pieces and portions of a document. Consider the following example XML document, as shown in Listing 5-9.

Suppose we are concerned with only selecting the subset of Listing 5-9 containing the <Good_Cheese> element and all of its descendents, but nothing more. A proper expression for selecting <Good_Cheese> and its children would be the expression ancestor-or-self:Good_Cheese. This expression is evaluated for each node in the XML document. In this case, the expression asks the question: " Am I (the current node) a <Good_Cheese> element, or do I have <Good_Cheese> as my ancestor?" Once this expression is evaluated, if the answer to the question is yes, then the node is placed in the output node-set. The output node-set result of this expression is shown here:

```
<Good_Cheese>
    <Cheddar_Type> Extra Sharp Cheddar </Cheddar_Type>
</Good_Cheese>
```

Listing 5-8

An example showing the syntax of the <XPath> element

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116
  <XPath>
  ancestor-or-self:Good_Cheese
  </XPath>
</Transform>
```

Listing 5-9

An example XML Document for describing XPath filtering

```
<Cheese_Types>
  <Favorite_Cheese>
    <Good_Cheese>
      <Cheddar_Type> Extra Sharp Cheddar </Cheddar_Type>
    </Good_Cheese>
  </Favorite_Cheese>
  <Bad_Cheese>
    <Soft_Type> Gouda </Soft_Type>
  </Bad_Cheese>
</Cheese_Types>
```

It is also important to notice that the XPath expression used here would collect *all* occurrences of <Good_Cheese>, not just the first occurrence. That is, if Listing 5-9 contained four or five distinct <Good_Cheese> elements, all of them would be selected for the final node-set. This is because all such elements would evaluate to true for the given XPath expression. To expand this expression such that it selects occurrences of <Good_Cheese> that have <Favorite_Cheese> as a parent, one might instead use ancestor-or-self:Good_Cheese [parent:Favorite_Cheese].

Enveloped Signature Transform

The final transform that we are going to discuss is the enveloped signature transform. An enveloped signature is a <Signature> element that is child to the data being signed. This type of signature configuration is useful for modeling real-world documents that have signature blocks in various places. Enveloped signatures are also extremely useful because the signature and the original document are tightly coupled, effectively removing a need to hunt down and match up original documents during verification.

This transform is a required transform for implementations of XML Signatures. To place some motivation behind this transform, consider the core generation process for an *enveloped* XML Signature. During core generation, <Reference> elements must be created and subsequently hashed to obtain a digest value. For example, the use of a same document reference to include the current document will probably be used (see Listing 5-10).

Listing 5-10   A snapshot of an enveloped signature being created

```
<document1>
  <msg> This document needs to be signed, it is very important </msg>
  <Signature>
    <SignedInfo>
      <Reference URI="">
        <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"/>
        <DigestValue>ktEp0766IBHkHN8mxV5U9o1FIYs=</DigestValue>
      </Reference>
    </SignedInfo>
  <SignatureValue>  . . .  </SignatureValue>
  </Signature>
</document1>
```

Suppose this is a snapshot of core generation. We haven't done signature generation yet, but only calculated the digest value of our `<Reference>` element. In this case, it is the digest value of the node-set that represents the entire `<document1>` XML document, including all of its children. Because a node-set is provided as input to the digest operation, it must be canonicalized and converted to an octet stream before it is digested. Once the digest happens, we store the digest value in the `<DigestValue>` element and continue on our way, proceeding to add the signature value.

However, an inherent problem exists in our process thus far. If we complete signature generation and proceed to add the contents of `<SignatureValue>`, we will have already broken our signature because the digest content has already been changed. That is, the content of the `<DigestValue>` element shown previously is the result of digesting the document *before* the `<SignatureValue>` has been added. Once we add the actual `<SignatureValue>`, we have changed the original document that we are signing. When the signature is verified at a later time, the digest values won't match and core validation will fail. Further, because the signature value must always be generated *after* the `<Reference>` elements have been digested, no means exists for generating the `<SignatureValue>` beforehand.

The solution to this messy situation provided by the XML Signature Recommendation is to use what is called an *enveloped signature transform*. This transform removes the *entire* `<Signature>` element from the digest calculation. An XPath transform can be used to accomplish the enveloped signature transform, but the XML Signature Recommendation doesn't explicitly specify that XPath *must* be used. It is simply one way of performing the transformation. For example, the XPath expression for removing the `<Signature>` element from a node-set is shown in Listing 5-11. This is the same expression that is given in the XML Signature Recommendation.

Listing 5-11

XPath expression for accomplishing the enveloped signature transform

```
<XPath>
  not(ancestor-or-self:dsig:Signature)
</XPath>
```

This expression removes the `<Signature>` element from the input node-set by explicitly adding only those nodes that are *not* a `<Signature>` element or *don't* have `<Signature>` as an ancestor.

Transform Security: Seeing What Is Signed

The XML Signature Recommendation is very flexible in the number of transforms that are allowed. For example, a handful of transforms are specified in the XML Signature Recommendation and custom transform algorithms are permitted for application-specific purposes. Also, no explicit restriction has been made on the inherent power of the transforms used.

Transformations like XPath and XSLT are very powerful and robust; because of this, they carry with them an inherent danger. They have the potential to make drastic changes to the input document. As an example, the XML Signature Recommendation does not preclude the use of a transform that deletes all elements in an input document, changes arbitrary octets, or otherwise corrupts the data to be signed.

To provide some motivation for this topic, let's consider a fictional courtroom example where an XML Signature is presented as evidence of a contract. Suppose that Clint (a fictional person) is signing an electronic contract with the use of an XML Signature. The contract is shown to Clint over the Web in HMTL. Clint reads the document, understands the terms and conditions of the contract, and provides his private key to the XML Signing engine and creates the signature. Let's suppose for the sake of simplicity that the contract is for a job from the local middle school to provide after-hours janitorial services.

A week passes and Clint finds himself with a subpoena from the school asking him why he hasn't handed over his automobile, house, and contents of his bank account. Puzzled by the action taken by the school, Clint goes to court to defend himself. On the witness stand, the plaintiff presents the contract signed with Clint's private key. The contract says: "I, Clint, will hand over my automobile, house, and contents of my bank account." Obviously, the contract has been changed before it was signed, yet the signature still verifies. How could this have happened?

This concocted scenario is possible if rogue transforms are used. For example, the contract that Clint saw when he made the decision to sign was not the same contract that was actually digested and included as a

part of the `<Signature>` element. The signer didn't see what was signed because the transforms were not made explicit to the signer. Transforms are powerful enough to easily deceive users because they have the capability to change the document in arbitrary and unbounded ways. The following scenario brings about the first of three maxims for understanding the security issues surrounding transforms:

Security Transform Maxim I

A signer should only sign what is seen.

The term *seen* used previously is a broad term that implies that the signer needs to understand what he or she is signing in a general sense, including all transforms that will be applied to the data. This is sometimes difficult to achieve because some transforms are harmless, such as a transform that alters the presentation of a document. For example, the HTML document that Clint signed could have been transformed into plain, unformatted text. Clint would look at the document and might say that this *isn't* the same document that was initially signed. On some level, he would be correct because the document has gone through a transformation, albeit one that did not alter semantics. A natural logical conclusion from the previous maxim and the example scenario is the second maxim:

Security Transform Maxim II

Only what is signed is secure.

This maxim follows from the way core generation works. The transforms and source URI collectively describe the data to be signed, not the URI alone. This means that when an XML Signature is applied to a URI, you are not securing the source URI, but instead the result of the signature transforms applied to the octet stream or node-set from the source

URI. That is, if a transform discards or changes information, the original document is not actually secured, only the transformed document. This maxim implies the last maxim, which places some responsibility on the application:

Security Transform Maxim III

An application should see what is signed.

This maxim implies that an application should only make trust decisions based on the transformed document, not the original document. Again, the word *see* is used here again in a broad sense. Because only documents that are signed are secure (Maxim II), it would be erroneous for an application to make a security decision based on an untransformed document. The untransformed document may be completely different from the transformed document (this is exemplified with Clint's problem with the middle school in the scenario described).

All three of these maxims provide evidence for each other in a syllogistic, circular way. The problem with these maxims is they are quite strict in and of themselves. For example, some applications may try to make inferences about the relationship between the transforms used and the original document. Extra inferences can help with some aspects of workflow efficiency, while some inferences, if understood correctly, may enable an XML Signature application to operate on the original, untransformed document. This concept is called *document closure* and was invented by John Boyer. A simplified version of document closure says that if an application understands the transforms that are being applied to an original document, then it may also understand and continue to "see" the transformed document in terms of the relationship between the transforms and the original document. That is, when transforms are well understood, the result of a transformation is a member of a finite set of well-understood transformed documents. An XML Signature application that signs something using document closure as a conceptual model would not be signing the result of the transformed document, but instead the relationship between the source document and the transforms applied.

Document closure can be thought of as a conceptual compromise between the strict maxims presented here and a loose interpretation of what it means to transform an original document even given arbitrary transforms. The point is largely theoretical and gives us another way of thinking about the security of transforms. Figure 5-5 shows a pictorial representation of the semantics of document closure.

Again, document closure only has meaning if the application understands the transforms used. For example, consider an arbitrary, fictional, transform T that produces different outputs based on dynamic state in the operating system. That is, it is hard to predict the output of the transform because it is always changing, and the internals of how the transform works are complex or hidden. It would be difficult to argue that an application understands this transform well enough to make inferences based on the transformed content. It would also be hard to argue for document closure because the set of possible relationships between the original document and the transformed output is difficult to fully define.

Contrast this with a transform T_2 that adds a large amount of reoccurring "boilerplate" text to a document. This type of transform is quite simple and the set of possible relationships is better understood. It might be

Figure 5-5

A pictorial representation of transformation semantics

Weak Transform Semantics   Strong Transform Semantics

S_1 S_2 S_3
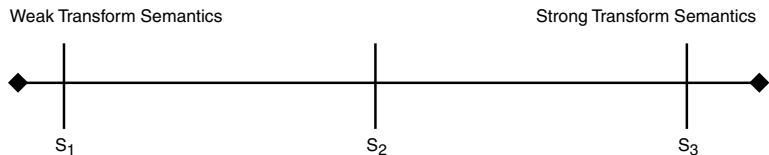
Transform Semantics

S_1: We are signing the original document given any arbitrary transforms.
S_2: We are signing a well-defined relationship between an original document and the transforms used (Document Closure).
S_3: We are only signing the result of transformations that are applied. We are signing no part of the original document.

possible to make trust inferences about the original document without having to go through the trouble of applying the well-defined T_2 transformation each time processing needs to happen on the document. Document closure can be useful for optimizing workflow in the sense that certain applications, when using well-defined, well-understood transforms, can cut down on the amount of processing that must happen.

To bring this theoretical discussion back down to the real world, the bottom line (for Clint's case) is that he should have *applied* the transforms *before* signing and looked at the document. This would have enabled Clint to see what he was signing, instead of what was presented to him initially. In the same vein, applications need to understand and try to effectively "see" (or understand) the implications of signing or verifying data that has undergone (or that will undergo) transformations.

Chapter Summary

This chapter was concerned with examining how XML Signatures are processed. We began the chapter by first adding detail to the `<Reference>` element and expanded on the use of the URI attribute as an identifier for the source data to be transformed. We looked at the various ways of identifying the data source for an XML Signature and noted that two basic, broad data types are used when processing XML Signatures: octets and node-sets. A node-set is an abstract data model for processing an XML document as XML and is defined in the XML Path Language Recommendation (XPath). Further, we discussed the *core generation* process for creating a XML Signature. Core generation is divided into two steps: reference generation and signature generation. Reference generation creates the data objects to be digested and signature generation executes the actual signature algorithm.

An XML Signature is verified using *core validation*. Core validation is also divided into two sub-steps: reference validation and signature validation. Some applications may deem an XML Signature valid even if some portions of *core validation* fail. This feature is not part of the XML Signature Recommendation, but it is instead an application-specific notion that aids in the practical use of XML Signatures when the number of reference elements grows.

Finally, we ended the chapter with a discussion of signature transforms. We covered canonicalization, XPath, enveloped signatures, and base64 decoding transforms. We examined canonicalization in some depth and learned how XPath can be used as a filtering mechanism. The last topic of discussion included a brief look at the semantics of signature transforms, and how one should interpret what it means for a transform to be applied to an original document. We also looked at the maxims provided by the XML Signature Recommendation as well as John Boyer's document closure concept.

XML Signature Examples

An XML Signature is rich in its features and syntax. The sheer number of elements and the number of valid permutations of elements and transformations tend to make the XML Signature confusing, especially when applied to practical scenarios. In order to facilitate better understanding of the intended use of XML Signatures, we will move from a more conceptual discussion to practical use cases of XML Signatures. The source for many of the questions or problems posed here comes from an early XML Signature Scenarios FAQ W3C Note as well as practical experience. The specific W3C Note is given in the references section. This chapter represents a cornucopia of differing questions and scenarios that should assist the reader in applying XML Signature concepts to real-world applications.

XML Signature Examples and Frequently Asked Questions

Each example begins with an intended goal, question, or simply a clarification concerning a particular aspect of XML Signatures. Some of the scenarios are quite simple and general, while others are quite specific to a particular problem that has risen during the practical use of XML signing.

Scenario 1

Consider an arbitrary XML document *D*. Next, consider the problem of creating a new XML document *D'* that is tightly coupled with an XML Signature *S*, where *S* signs the document element of *D'*. That is, we would like *S* to appear in the same document as *D'*. What are the two document models that would allow for this?

Proposed Solution

The XML Signature Recommendation provides for *enveloped* and *enveloping* signatures. Recall from Chapter 4 that an *enveloped* signature is defined to have the `<Signature>` element as child to the element being signed. Similarly, an *enveloping* signature has the `<Signature>` element as parent to the element being signed. Both of these document models tightly couple the `<Signature>` element with the data being signed, irrespective of the relationship of the `<Signature>` element to the document element or vice versa. Listing 6-1 and 6-2 shows two high-level examples of this coupling.

Notice that in Listing 6-2, the document that we are signing appears inside an `<Object>` tag.

Scenario 2

Consider the problem of signing an arbitrary octet stream. What are two ways in which an XML Signature can reference and sign arbitrary octets?

Listing 6-1

Enveloped XML Signature

```
<DocumentD>
  <Signature>
  </Signature>
</DocumentD>
```

Listing 6-2

Enveloping XML Signature

```
<Signature>
  <Object>
    <DocumentD>
    </DocumentD>
  </Object>
</Signature>
```

Proposed Solution

There are two ways of including an arbitrary octet stream in an XML Signature. The simplest (and arguably less useful) way to do this is to create a *detached* signature and reference the octets via a Uniform Resource Identifier (URI). This is shown in Listing 6-3. It is important to note that although it is possible to reference a local arbitrary file in the URI attribute of a `<Reference>` element, doing so has little utility. If the intended recipient wants to verify a signature that uses a local binary file, the recipient must have the same original document in the same location.

It can be argued, however, that a detached signature over a local file via a URI reference has value in certain restricted cases. Consider the case where the recipient ignores the URI attribute of the `<Reference>` element and knows *a priori* where to obtain the necessary octets for digesting. In this case, it matters little where the sender obtained the octets to sign, as long as the recipient obtains the same octets, the signature can still be effective.

The more complex (and arguably more useful) way to include arbitrary octets is to use an `<Object>` element. In Chapter 4, we discussed the `<Object>` element and mentioned that this element is the chief mechanism for including arbitrary data objects in an XML Signature. When the `<Object>` element is used, the data inside will ultimately be child to the parent document element and consequently, the `<Signature>` will be enveloping.

Listing 6-3

A detached XML Signature over a local binary file

```
<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="file:///C:\foo.bin">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>qR73c21p0dfJB4OuZEaC46WE1qg=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
    az14TVzmAUzq53cdQ6PF5Z/wNO9akwEhW7lVTVuy5hLh+TJLZpKhx8dtnnZoFx+cW
    AumfQbvhdCP8jmuLrwnH2nwtbPLqQg5Lf3fK4a4Qrk2XOUAkwjxZEoAELbGVfKQaU
    nOo84oPUvXDhiY5oJCIgJhtw4kCGmlHdVKeFb3Js=
  </SignatureValue>
  </Signature>
```

To actually make this scenario work, the arbitrary octets *must* be encoded in a printable format for inclusion in an <Object> element. This <Object> element is then treated as a normal XML document subset and signed via a bare fragment identifier with <Reference> element. The potentially confusing part of this process is that the *encoding* that must happen is orthogonal to the transformations that are permissible with the XML Signature Recommendation. That is, the octet stream must be encoded *before* it is placed in the <Object> element. Because transforms happen as a means to produce the data to be digested, there is no way to encode the octet stream as a part of the transforms. The transforms refer to the <Object> element, which in the case of an enveloping signature, must already have printable data inside of it. Put another way, if one were to use a printable encoding mechanism (such as Base-64 encoding) as a part of the transformation waterfall, one would have to include octets in the <Object> element, which is impossible by the rules of XML (all element context must be in a text format). The printable encoding that will be used in this case is going to be Base-64 encoding. This process is shown in Figure 6-1. The input is an octet stream that is encoded in a printable format and then placed inside an <Object> element. From here it is treated as normal XML and is processed as such. The XML Signature doesn't know the difference between arbitrary printable data and printable data that encodes some underlying binary data object.

Scenario 3

Consider the problem of signing a multiple references with *n* signing keys. Describe XML Signature syntax to accomplish this.

Proposed Solution

A simple approach is to create *n* separate <Signature> elements. Each <Signature> element has a duplicate <SignedInfo> element containing the referenced resources. That is, each <Signature> element uses the same <SignedInfo> child element. To reduce the complexity of this syntax, a <Manifest> element can be utilized to consolidate the shared resources. One detail that is often overlooked with this approach is the location of the <Signature> elements in relation to the <Manifest>

First, consider a snapshot of *reference generation* for an arbitrary XML Signature S.

```
<Signature>
  <SignedInfo>

    <Reference URI=?>
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue> ? </DigestValue>
    </Reference>
  </SignedInfo>

</Signature>
```

Second, encode an arbitrary binary octet stream *B* to produce a printable version of B_p.
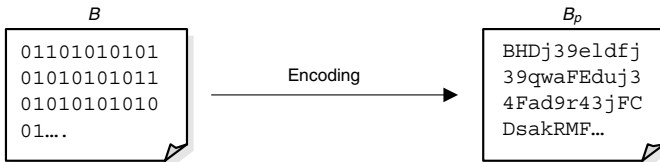
Finally, we include *Bp* in an `<Object>` element and give it an arbitrary Id. Now we can finish *reference generation* and complete the signature.

```
<Signature>
  <SignedInfo>

    <Reference URI="#arbitraryId">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>hR7gd2jpgd4JGDFuZEgfd6WE1qg=</DigestValue>
    </Reference>

  </SignedInfo>

  <Object Id="arbitraryId">

   BHDj39eldfj39qwaFEduj34Fad9r43jFCDsakRMF…

  </Object>
</Signature>
```

element. For example, reconsider Listing 4-29. All of the `<Signature>` elements are placed in the XML document, and the appropriate bare fragment identifier is used to reference the `<Manifest>` element (such as, `#ThreeReferences`). XML Signatures are not limited in this respect—each `<Signature>` element that references a `<Manifest>` may exist outside of the current XML document. Furthermore, the `<Manifest>` element may be fully divorced from the XML Signature if so desired. This malleable feature of XML Signatures provides for a great deal of flexibility within the syntax itself. Figure 6-2 shows a `<Manifest>` element in a separate XML document that is referenced by several `<Signature>` elements. Compare this to Listing 4-29. The difference here is the coupling of the `<Manifest>` element. In Listing 4-29, the `<Manifest>` is in the same document (and is part of an enveloping signature), while in Figure 6-2, the `<Signature>` elements shown are purely detached signatures.

In Figure 6-2, there are some important things to note. Consider the structure and location of the `<Manifest>` element. It is shown nested below two arbitrary ancestor elements—p1 and e1. The idea here is to show that a `<Manifest>` element can exist quite happily outside a -`<Signature>` block. There is no explicit requirement to tightly couple a

Figure 6-2

Multiple references with *n* signers via a `<Manifest>`


```
<Signature Id="FirstSigner">
  <SignedInfo>
    <Reference URI="http://remote.server.com/XMLDoc.xml">
      <Transforms>
        <Transform Algorithm=
        "http://www.w3.org/TR/1999/REC-xpath-19991116">
          <XPath>
            ancestor-or-self::Manifest
          </XPath>
        </Transform>
      </Transforms>
    </Reference>
…
```

XMLDoc.xml

```
<p1>
  <e1>
    <Manifest>
      <Reference> … </Reference>
      <Reference> … </Reference>
      <Reference> … </Reference>
    </Manifest>
  </e1>
</p1>
```

```
<Signature Id="SecondSigner">
  <SignedInfo>
    <Reference URI="http://remote.server.com/XMLDoc.xml">
      <Transforms>
        <Transform Algorithm=
        "http://www.w3.org/TR/1999/REC-xpath-19991116">
          <XPath>
            ancestor-or-self::Manifest
          </XPath>
        </Transform>
      </Transforms>
    </Reference>
…
```
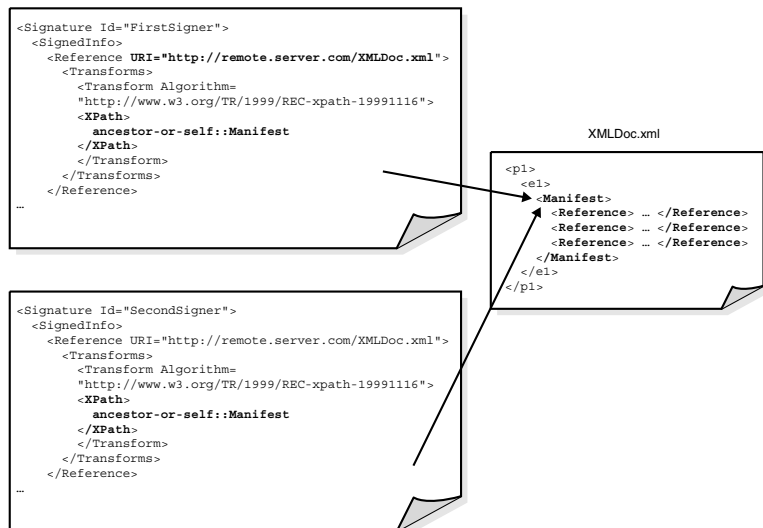
<Manifest> element within an existing XML Signature. It may be the case that an arbitrary XML document contains many <Manifest> elements scattered throughout its structure.

Furthermore, notice *how* the <Manifest> element has been referenced. In Listing 4-29, the <Manifest> was pointed at via a bare fragment identifier. In Figure 6-2, it is done instead with a very simple XPath expression that looks for an element (and all children of) called *Manifest*. Although useful as an example, this XPath expression will likely be too simple for most real-world scenarios. It doesn't incorporate very much context and omits things like namespace information. For example, if multiple <Manifest> elements were present in XMLDoc.xml, the entire set of them would be selected—in most cases, we would want just a single <Manifest> element.

Scenario 4

Consider the problem of signing an arbitrary element *E* within an XML document where *E* is not the document element. What are some possible ways to make this type of signature?

Proposed Solution

Targeting an individual element to participate in an XML Signature is one of the most powerful features of XML Signatures. This specific feature is often used to *exclude* other portions of the source document. If only a portion of an XML document is signed and the granularity is at the element level, what is happening is that we are really providing a mechanism for enabling changes to the rest of the XML document that are irrelevant to validity of the signature.

There are some obvious ways to pinpoint a specific element, and most of these mechanisms have been previously discussed. Listing 6-4 shows two ways of attempting to sign a particular element (called <myElement>) in an XML document via a *detached* signature.

In Listing 6-4, notice the use of the XPath expression once again. This particular expression is used pervasively in the examples here because it conveniently includes those nodes that are themselves myElement nodes or those that have myElement as an ancestor node. Also in Listing 6-4 is the use of a *fragment identifier*. This type of reference can be used to select

Listing 6-4

Two ways of
referencing an
element via a
detached
signature

```
<SignedInfo>
  <Reference URI="http://www.fake-site.com/file.xml#myElement">
    . . .
  </Reference>

  <Reference URI="http://www.fake-site.com/file.xml">
    <Transforms>
     <Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-
     19991116">
       <XPath>
         ancestor-or-self:myElement
       </XPath>
     </Transform>
    </Transforms>
    . . .
  </Reference>
  . . .
</SignedInfo>
```

a particular element within an XML document, but the XML Signature
Recommendation cautions against it.

   To be more precise, the fragment identifier shown is actually shorthand
for a slightly more complex expression defined by a language referred to
as *XPointer*. XPointer is used in places where a traditional HTML frag-
ment identifier is employed. The intention of XPointer is to provide a rich
language for selecting portions of XML documents. In this respect,
XPointer provides some of the same functionality as XPath in the context
of XML Signatures. The fragment identifier shown in Listing 6-4 is called
a *bare name* XPointer and is a syntactical shorthand for the expression:
`xpointer(id(myElement))`. This expression simply says: Give me the
element whose Id value is myElement, which is exactly what we want in
the scenario shown in Listing 6-4. The reader should understand that we
are looking for an element whose Id value is MyElement. This means that
the actual element inside file.xml would look as follows:

```
<MyElement Id="MyElement"> ... </MyElement>
```

   It is merely coincidental that the Id attribute value and element name
match, this doesn't have to be the case. The Id can be artibrary, such as
"elem1". In this case, the URI Reference in Listing 6-4 would need to be
changed as follows:

```
URI="http://www.fake-site.com/file.xml#elem1"
```

The reason why the XML Signature Recommendation cautions against the use of a bare name XPointer reference is because bare name XPointer resolution is not in scope for the XML Signature Recommendation. Pervasive use of bare name XPointer syntax can cause interoperability problems if these identifiers are not de-referenced in a standard way. The XML Signature Recommendation instead advises the use of an XPath expression for element level selection.

Scenario 5

Consider a similar scenario to Scenario 4, but add the requirement of *multiple signers* to an arbitrary element *E* where *E* is not the document element. Further, add the requirement that the signatures must be tightly coupled with the source XML document. What are two ways of accomplishing this type of signature?

Proposed Solution 1

The first solution involves adding separate `<Signature>` elements as children to the original document. This is a curious case because although the `<Signature>` elements appear to be enveloped, they are in fact detached. The reason why is because the document element is not included in the signature. The `<Signature>` elements are side-by-side the data being signed—the signature is neither parent nor child to the data being signed. This is a case where a detached signature appears to behave like an enveloped or enveloping signature in its relation to the source document. Listing 6-5 shows an example of three signers signing an arbitrary element within a source XML document.

The first thing to notice about Listing 6-5, like some of the other examples, is that it is not a complete signature. Elements, such as the signature value, signature method, digest value, and others have been left out for clarity. The thing to focus on is the arbitrary element, which has been named `<Element1>`. This element is a child element of the document element `<SourceDocument>`. In order to sign `<Element1>` with multiple signers, we have added three `<Signature>` child elements to `<Source-Document>`. Each of these three child `<Signature>` elements references `<Element1>`. Two of the `<Signature>` elements use a bare name fragment identifier to identify `<Element1>` by its Id value and one of the

Listing 6-5

Example of
Multiple
Detached
Signatures over
an arbitrary
element

```
<SourceDocument>
  <Element1 Id="SignHere"> This is important information!
</Element1>
  <Signature Id="Signer1">
    <SignedInfo>
      <Reference URI="#SignHere"> ... </Reference>
    </SignedInfo>
   ...
  </Signature>
  <Signature Id="Signer2">
    <SignedInfo>
     <Reference URI="">
        <Transforms>
         <Transform
          Algorithm="http://www.w3.org/TR/1999/REC-xpath-
          19991116">
           <XPath>
            ancestor-or-self:Element1[parent:SourceDocument]
           </XPath>
         </Transform>
        </Transforms>
     ...
     </Reference>

    </SignedInfo>
  ...
  </Signature>
  <Signature Id="Signer3">
    <SignedInfo>
     <Reference URI="#SignHere">  . . .  </Reference>
    </SignedInfo>
  ...
  </Signature>
</SourceDocument>
```

<Signature> elements uses an XPath expression to target the desired
element. The point here is to show that the <Signature> elements may
reference <Element1> in any way they want.

The XPath expression shown is similar to the same ancestor-or-
self expression used in previous examples. The difference here is that
this expression is a bit more robust. The additional [parent:Source-
Document] syntax indicates that a node should be signed if it has
<SourceDocument> as a parent. This additional constraint would be use-
ful in situations where <SourceDocument> was placed into another XML
document as a child that also had an element named <Element1>. That
is, ancestor-or-self:Element1[parent:SourceDocument] means
a node should be included in the node-set to be signed if the node is an
<Element1> node with <SourceDocument> as the parent, or has
<Element1> as its ancestor, with <SourceDocument> as the parent.

Proposed Solution 2

Another way of adding multiple signers to an arbitrary element *E* within an XML document involves making *enveloping* signatures over *E*. This method of creating an XML Signature contrasts the previous solution in that it is a bit more complex. Each <Signature> element is tightly attached to the previous, creating a more cumbersome structure. This type of solution might be useful if it is anticipated that the <Signature> element may be moved at a later date. If it is removed, the original document goes with it. In the previous example, one would have to remove the original document and <Signature> blocks separately and maintain the child relationships between them. This solution makes for more portable signature elements. Listing 6-6 shows an example of how this might be accomplished.

Notice in Listing 6-6 the increased complexity of the structure. Only two signers are shown in this example to reduce confusion. The first signer is denoted by the <Signature> element with an Id attribute of Signer1. This <Signature> element references the arbitrary element, Element1 inside an <Object> element using a bare fragment identifier called <#thisElement>. The next signer, which is denoted by the <Signature> element with the Id attribute of Signer2 further envelopes the

---

Listing 6-6

Example of Multiple Enveloping Signatures over an arbitrary element

```
<SourceDocument>
  <Signature Id="Signer2">
    <SignedInfo>
      <Reference URI="#thisElement">
        . . .
      </Reference>
    </SignedInfo>
    <Object Id="FirstSigner">
      <Signature Id="Signer1">
        <SignedInfo>
          <Reference URI="#thisElement">
            . . .
          </Reference>
        </SignedInfo>
        <Object Id="thisElement">
          <Element1 Id="SignHere">
            This is important information!
          </Element1>
        </Object>
      </Signature>
    </Object>
    . . .
  </Signature>
</SourceDocument>
```

first signature, which is also placed inside an `<Object>` element. Notice how the second signer uses the *same* bare fragment identifier to sign `Element1`. Furthermore, because of the way the `<Object>` element is defined (see Chapter 4), the beginning and ending `<Object>` tags are also signed. That is, in the previous solution (see Listing 6-5), when we signed `<Element1>`, we signed the following data in all cases:

```
<Element1 Id="SignHere"> This is important information! </Element1>
```

In the solution shown in Listing 6-6 the data that is actually signed is as follows:

```
<Object Id="thisElement">
  <Element1 Id="SignHere"> This is important information!
  </Element1>
</Object>
```

The reason why this happens is because of the way that the XML Signature Recommendation defines `<Object>`. When an element is signed as a part of an enveloping signature, the encasing `<Object>` tags are also signed. In this case, then, it may be desirable to add an XPath transform that omits the `<Object>` tags in the node-set being signed. This type of transform would make Listing 6-5 and Listing 6-6 equivalent in terms of the data actually being signed.

Scenario 6

Consider the problem of signing an entire XML document. What is the obvious problem associated with signing a full-featured XML document?

Proposed Solution

At first, this scenario may seem trivial, but a fully featured XML document may not be well formed for the purposes of insertion into a `<Signature>` element. For example, consider the sample XML document shown in Listing 6-7 that uses a variation of an XML application called *DocBook*, which is used for authoring books or articles:

The interesting problem here is that although this document contains well-formed XML, the entirety of the document is not well-formed XML. In particular, the XML declaration and the document type declaration do not have ending tags, yet they are required for most XML parsing appli-

Listing 6-7

An Example
DocBook XML
Instance

```
<?xml version='1.0'?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD Docbook XML V4.1.2//EN"
                    "http://www.oasis-
open.org/docbook/xml/4.1.2/docbookx.dtd">
<article>
  <title>Chocolate: The definitive guide</title>
  <sect1>
    <title>Why dark chocolate is bad</title>
    <para>
      Everyone knows that dark chocolate is vastly inferior to milk
      chocolate.
    </para>
  </sect1>
</article>
```

cations in order to process and possibly validate the document properly.
This means that it would be impossible to simply insert the entire contents of Listing 6-7 into an enveloping signature.

There are at least three possibilities for dealing with a fully featured
XML document similar to the one shown in Listing 6-7. The document can
be treated as arbitrary octets and referenced via a detached signature, or
the document can be encoded in a printable format and inserted into an
enveloping signature. Both of these possibilities mean that the XML document is treated in exactly the same way as described in Scenario 2—
arbitrary octets. One solution for treating the XML document as XML is
to sign the document element of Listing 6-7 with an enveloped signature.
This is shown in Listing 6-8 and is a useful way of inserting an XML Signature into an existing XML document. It may be disappointing or
counter-intuitive that a fully-featured XML document must be treated as
binary in the context of XML Signatures when an enveloped signature is
not used, but the nature and syntax of the XML declaration and document
type declaration leave little other choice.

The <Reference> element shown in Listing 6-8 refers to the document
element, <article>. The association is made with a bare fragment identifier using the Id attribute of the <article> element. If <article> had
no Id attribute, the element could still be signed with an XPath expression. Another nuance is the presence of the enveloped-signature transform. This transform is required for all enveloped signatures and is used
to exclude the <Signature> block from the computation of the digest
value. Another detail that is overlooked in this example is the possibility
of conflicting namespaces. Our DocBook instance doesn't use any of the
same element names as XML Signatures, but care must be taken to
include the appropriate namespace declaration to prevent conflicts

Listing 6-8

Inserting an enveloped signature into a fully featured XML document

```xml
<?xml version='1.0'?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD Docbook XML V4.1.2//EN"
                    "http://www.oasis-
open.org/docbook/xml/4.1.2/docbookx.dtd">
<article Id="chocolate">
  <title>Chocolate: The definitive guide</title>
  <sect1>
    <title>Why dark chocolate is bad</title>
    <para>
      Everyone knows that dark chocolate is vastly inferior to milk
      chocolate.
    </para>
  </sect1>
  <Signature Id="Signer1">
    <SignedInfo>
      <Reference URI="#chocolate">
        <Transforms>
          <Transform
           Algorithm=
           "http://www.w3.org/2000/09/xmldsig#enveloped-signature"
/>
        </Transforms>
        . . .
      </Reference>
      . . .
    </SignedInfo>
    . . .
  </Signature>
</article>
```

between elements of the same name. An easy way to do this is to add a default namespace declaration to the <Signature> element as follows:

```xml
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#" Id="Signer1">
```

The correct XML Signature namespace should be included anytime there is a possibility of merging an XML Signature with an existing XML document—in fact, it is good practice to include it all of the time to prevent name conflicts in general. We have been omitting it from most of the examples thus far in an effort to avoid confusing details.

Scenario 7

What types of considerations arise when using XPath as a selection mechanism for pinpointing elements to participate in an XML Signature? That is, common XPath expressions such as /element seem to give erroneous results for the XPath transform.

Proposed Solution

When XPath is used as a transformation for an XML Signature, it is used as a filtering mechanism and not as a selection mechanism. This nuance causes headaches for those who have worked with XPath as a selection mechanism. Before an XPath transform occurs, the entire node-set of the current document is collected. For each node in this node-set, the desired XPath expression is evaluated against the current node and the result is converted into a Boolean value. If the value is true, the node is included in the node-set that will eventually be canonicalized and then signed. If the value is false, the node is excluded from the output node-set. For simple XPath processing, this type of Boolean conversion complicates simpler selection expressions. Consider Listing 6-9, which shows an arbitrary well-formed XML document.

Suppose that it is our job to write an XPath expression to select `<element3>` and all of its children. A vacuous XPath expression that might be considered for this task would simply be: `/element1/element2/element3`. This particular expression exemplifies the nature of XPath as a simple selection language based on the concept of a path similar to its use in a URI. Unfortunately, this particular XPath expression will produce a completely erroneous result if it is used with an XML Signature. In fact, instead of selecting only the `<element3>` node, it will usually inadvertently select all nodes of the input document. The reason why this occurs is because of the way the XPath expression is interpreted based on the XML Signature Recommendation. The given XPath expression is evaluated against each node, and the result is converted to a Boolean value. Furthermore, the XPath Recommendation defines that any non-empty node-set be evaluated as true when it is converted into a Boolean value. The expression `/element1/element2/element3` will always be a non-empty node-set and will cause every node in the input node-set to be evaluated as true and transferred to the output node-set. This is why the expression `ancestor-or-self` is used pervasively as an

---

Listing 6-9

An arbitrary well-formed XML document

```
<element1>
  <element2>
    <element3> This is my target ! </element3>
  </element2>
</element1>
```

element selection mechanism. This particular expression produces the correct Boolean value for selecting an element and its children. Care must be taken when designing XPath expressions to ensure that they evaluate to a proper Boolean value. This is especially important with XML Signatures, because it is often too difficult to discern what the XPath expression produces because the result is canonicalized and then signed—the signer must have complete confidence that the XPath expression chosen is indeed correct. Because the XPath transform and canonicalization method together define the data that is actually signed, a rogue or poorly understood XPath expression can completely alter the actual bytes that participate in the signature. This security consideration is related to the security maxims discussed in Chapter 5, specifically the third security maxim that states that an application should see what is signed.

Scenario 8

Consider an arbitrary `<Signature>` element that contains a `<Signed-Info>` element with several `<Reference>` elements. Each of the `<Reference>` elements refers to data external to the `<Signature>` element. That is, the `<Signature>` element is the document element of the given XML document. It is expected that this particular signature will be verified by a large number of recipients over an extended period of time. Moreover, it is anticipated that the semantics of the signature will change over time; some of the items referred to will be removed or changed. It is desirable to maintain the validity of the signature throughout material changes to external data. How can this requirement be met using XML Signatures?

Proposed Solution

The details of Scenario 8 represent a very application-specific requirement that can be met with the use of a `<Manifest>` element. This element has been discussed before, and by now this solution is most likely review for the reader. The use of a `<Manifest>` to house `<Reference>` elements is a convenient way of beating the definition of core validation. Each `<Reference>` element that anticipates change should be placed in a `<Manifest>` element. From here, it is up to the application to validate the `<Reference>` element as it sees fit—the validation of `<Reference>`

elements that reside inside a `<Manifest>` element is not part of the core validation process. For completeness, another example use of the `<Manifest>` element is shown in Listing 6-10.

Listing 6-10 illustrates the use of the `<Manifest>` element once again. The files that are referenced, `file1.bin` and `file2.bin`, can change at will without affecting the value of core-validation as defined by the XML Signature processing model.

Scenario 9

Canonicalization is an expensive operation and seems to be used in various places within the processing model for XML Signatures. How can we know if canonicalization is being used unnecessarily? Is it possible to omit canonicalization to save on performance?

Proposed Solution

There are three places where canonicalization is specified for the XML Signature processing model. It is a *required* operation during core validation and core generation and must be used to normalize the `<SignedInfo>` element before signing and verification. Unless a highly customized XML Signature implementation is being used, it is unlikely

---

Listing 6-10

Another example use of `<Manifest>`

```
<Signature>
  <SignedInfo>
    <Reference URI="#refList">
      . . .
    </Reference>
    . . .
  </SignedInfo>
  <Object>
    <Manifest Id="refList">
      <Reference URI="http://www.some-site.com/file1.bin">
        . . .
      </Reference>
      <Reference URI="http://www.some-site.com/file2.bin">
        . . .
      </Reference>
    </Manifest>
  </Object>
  . . .
</Signature>
```

that canonicalization can be omitted at this stage. The second place where canonicalization is required is directly before the data from a `<Reference>` element is digested. If the `<Reference>` element is being treated as binary, then canonicalization is meaningless because the operation is only defined on XML data—arbitrary octets cannot be canonicalized. When a `<Reference>` element specifies a node-set (as in a same-document reference), this node-set is canonicalized *implicitly* when it is converted to an octet stream. Canonicalization is defined to produce a physical representation based on a node-set, and it is this node-set that is converted into binary form to be used in the digest algorithm. This implicit canonicalization is a defined part of the processing model and cannot be omitted. The third place where canonicalization can be used is as an explicit transform from within the transformation waterfall. At first glance, this may seem like a redundant operation (see Listing 6-11).

If canonicalization is used implicitly before the digest operation, isn't adding the canonicalization method as an exclusive transform a redundant operation? In short, the answer here is that the addition of a canonicalization algorithm in the transform waterfall shouldn't be redundant in proper implementations of XML Signatures. The reason why is because the canonicalization algorithm produces an octet stream as its output. This means that the implicit canonicalization will be skipped because the output will already be the octet stream that is necessary for the digest function. A redundant use of canonicalization would be two transforms back-to-back within the transformation waterfall. This scenario is unlikely to occur for any practical situation and will probably be a non-issue.

Listing 6-11

Using canonicalization as an explicit transform

```
<Signature>
  <SignedInfo>
    <Reference URI="">
      <Transforms>
        <Tranform
         Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
         20010315" />
      </Transforms>
    </Reference>
    . . .
  </SignedInfo>
  . . .
</Signature>
```

Scenario 10

Consider a situation where it is desirable to tightly couple arbitrary binary data with an XML Signature. A solution for this would be to use an enveloping XML signature and encode the binary data in some printable format (see Scenario 2). The most popular pintable-encoding scheme used is Base-64 encoding—defined in RFC 1421, 1521, and 2045. Base-64 encoding makes use of the carriage return and line feed character to denote a line break. What sort of practical implications does this have for XML and Canonical XML, which normalize carriage return characters?

Proposed Solution

It is natural to experience problems with reference validation on a `<Reference>` element that specifies some arbitrary Base-64 encoded data. To give a concrete example, consider Listing 6-12, which shows some arbitrary Base-64 encoded data produced with a commercial cryptography toolkit.

Next, suppose we take this data and interpret it as binary. We can get more information about exactly how this particular piece of data was encoded by looking at the actual underlying octets. The resulting hexadecimal values that correspond to the ASCII printable characters are shown in Listing 6-13.

The carriage return and line feed (CRLF) characters that break up the Base-64 encoded data into separate lines are shown in bold in Listing 6-13. Furthermore, the XML Recommendation specifies that all occurrences of CRLF should be converted into a single LF character (see Section 2.11 in the XML 1.0 Recommendation). The problem, then, occurs when this particular piece of data is added to an XML Signature after the XML has been processed; it is expected that the data has already had its line endings converted. This situation is likely to occur any time an XML Signature is added to an existing DOM tree (for more information on

Listing 6-12

Arbitrary Base-64 encoded data

```
hH6wIolYX86HCvgTNFMFn/vFAqcYry2vjnVlwZsHwCdLXH3EtTfNYVx3YTgvUYAIOfey
G7/8HqkkEBXvcAU5a09ynnfuqhtI/XagQ2NCjcCIobypJCD8jS/QksHLmxcHdqmN8/9
n9T+o8SvvHkarj7oAHC3IYjChCbyELYR8EyvRf5HePYY7tgXxM3OxNVVy3u/LTLaeJt2
BzQudP3uBtQ==
```

Listing 6-13

Hexadecimal
values for
Listing 6-12

```
68 48 36 77 49 6F 6C 59 58 38 36 48 43 76 67 54
4E 46 4D 46 6E 2F 76 46 41 71 63 59 72 79 32 76
6A 6E 56 6C 77 5A 73 48 77 43 64 4C 58 48 33 45
74 54 66 4E 59 56 78 33 59 54 67 76 55 59 41 49
4F 66 65 79 47 37 2F 38 48 71 6B 6B 0D 0A 45 42
58 76 63 41 55 35 61 30 39 79 6E 6E 66 75 71 68
74 49 2F 58 61 67 51 32 4E 43 6A 6A 63 63 43 49
79 70 4A 43 44 38 6A 53 2F 51 6B 73 48 4C 6D 78
63 48 64 71 6D 4E 38 2F 39 6E 39 54 2B 6F 38 53
76 76 48 6B 61 72 6A 37 6F 41 0D 0A 48 43 33 49
59 6A 43 68 43 62 79 45 4C 59 52 38 45 79 76 52
66 35 48 65 50 59 59 37 74 67 58 78 4D 33 4F 78
4E 56 56 79 33 75 2F 4C 54 4C 61 65 4A 74 32 42
7A 51 75 64 50 33 75 42 74 51 3D 3D
```

DOM, see the primer in chapter 3). That is, the carriage return character is not considered a valid line-ending character in the context of XML, and when the XML is canonicalized, occurrences of x0D are converted to a character reference (&#xD). This has the undesirable side effect of altering the meaning of the Base-64 encoded data and subsequently the digest value, making reference validation nearly impossible for applications that aren't aware of this normalization. To make this point more salient, Listing 6-14 shows the actual data that is digested after canonicalization occurs. Remember, the data is canonicalized when it is converted from a node-set to an octet stream. If Base-64 encoded data is added to an <Object> element, this type of reference will be treated as a node-set and will be canonicalized in all cases.

Notice the additional characters references that have been added to the data shown in Listing 6-14. These additional references are included by the definition of Canonical XML, which states that any occurrences of carriage return are converted to &#xD. From here, the data is digested, and the digest value is placed inside a <DigestValue> element. This situation, however, may seem irrelevant at first glance. A common objection says that because the input data should be canonicalized before it is verified, these changes are irrelevant. The reason why this objection is not valid in this case is because in the case of an enveloping signature, the XML document containing the signature will most likely be processed by a conformant XML processor before verification. By definition of XML line feed normalization, any carriage return characters will be normalized and converted to line feed characters, and the recipient will be unable to successfully verify the signature even though the actual data inside the <Reference> remains unchanged. To further clarify the situation, sup-

Listing 6-14

Arbitrary Base-64
encoded data
after
canonicalization

```
hH6wIolYX86HCvgTNFMFn/vFAqcYry2vjnVlwZsHwCdLXH3EtTfNYVx3YTgvUYAIOfeyG7/8
Hqkk&#xDEBXvcAU5a09ynnfuqhtI/XagQ2NCjcCIobypJCD8jS/QksHLmxcHdqmN8/9
n9T+o8SvvHkarj7oA&#xDHC3IYjChCbyELYR8EyvRf5HePYY7tgXxM3OxNVVy3u/LTLaeJt2
BzQudP3uBtQ==
```

pose that the data shown in Listing 6-14 produces an arbitrary digest value, such as the one shown as part of a `<Reference>` element that references the data shown in Listing 6-13:

```
<Reference URI="encoded-data">
  <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1" />
  <DigestValue>cvjp9GJOX69990Kqew9ioBGGEio=</DigestValue>
</Reference>
```

The source URI is simply the Id of an `<Object>` element that houses the encoded data. The full `<Object>` element is shown as follows:

```
<Object Id="encoded-data">
hH6wIolYX86HCvgTNFMFn/vFAqcYry2vjnVlwZsHwCdLXH3EtTfNYVx3YTgvUYAIOfeyG7
/8HqkkEBXvcAU5a09ynnfuqhtI/XagQ2NCjcCIobypJCD8jS/QksHLmxcHdqmN8/9
n9T+o8SvvHkarj7oAHC3IYjChCbyELYR8EyvRf5HePYY7tgXxM3OxNVVy3u/LTLaeJt2Bz
QudP3uBtQ==
</Object>
```

It is very important to re-iterate that the digest value shown is not actually the digest value of `<Object>` element with the Id value of encoded-data, but it is instead the digest value of the `<Object>` tags and the data shown in Listing 6-14. The reason why is, again, because of Canonical XML, which converts carriage return characters to character references. Furthermore, suppose that this signature was written to a file and transmitted to a recipient. When the recipient reads the XML document containing the XML Signature, the carriage return characters that are not visible in the transmitted data are stripped as a part of standard XML normalization. This means that when the digest value of the `<Object>` element is calculated as a part of signature validation, the digest value will be different because the &#xD character references do not participate in the digest value. These character references are stripped out even before the data can be canonicalized for verification.

This particular situation represents a clash between the XML Recommendation, RFC 2045, and the XML Signature Recommendation. The

proper solution for this situation is to ensure that any Base-64 encoded data that is added to an XML Signature *after* line-ending normalization has occurred (such as to a DOM tree), should be free of any carriage return values. This is a tricky situation in particular because any Base-64 encoded data that is removed from MIME messages must be further massaged before it is added directly to an XML Signature.

Scenario 11

What are the implications of creating a `<Reference>` element whose URI attribute has a value that uses `https` as its protocol scheme?

Proposed Solution

An XML Signature can rely on any resource accessible via a URI for use in a digital signature. If a URI reference uses `https` as its protocol scheme, the referent is a web resource that resides on a server that supports SSL or TLS. The existence of `https` as a protocol scheme implies *nothing* about how the `<Reference>` element that contains this URI is handled. That is, de-referencing a Web resource is *out of scope* for the XML Signature processing model. This point may be especially confusing for some readers because processing a `<Reference>` element that uses an `https` URI seems to mix security technologies. The XML Signature processing model simply demands data residing at a particular URI; whether this data is protected by another security protocol or is sent in the clear is of no concern to the XML Signature processing model.

Scenario 12

Consider a remote URI reference used an XML Signature. Suppose it is anticipated that over time, the URI will change, but the actual data signed will remain constant. For example, let the original URI reference be `http://www.server1.com/document.xml`. The anticipated change moves document.xml to `http://www.server2.com/document.xml`. How would one create an XML Signature that would enable the validity of the signature to survive a change in the source location?

Proposed Solution

The way to achieve this is to utilize a `<Manifest>` element and add semantics to the core validation process with the use of a transform that omits the URI attribute. In a strict sense, the processing model for XML Signatures does not enable for such a change in the source location, and signature validity will be delegated to the application. The use of the `<Manifest>` element, however, enables the application to reserve some reference validation semantics for itself. This scenario is very similar to Scenario 8 discussed previously. The only addition is a signature transform that pinpoints the desired URI attribute and leaves it out of the final data that is signed. This type of transformation can be dangerous, and the signer must implicitly trust this change in server location. The digital signature no longer protects the location, and the application must properly validate the location *beforehand*. This is part of the requirement that the application "sees what it signs."

A new concept that is introduced in this section is the *XSLT transform*. This type of transform is an *optional* transform of the XML Signature Recommendation. XSLT is the Extensible Stylesheet Language for Transformations. This technology is an integral part of transforming XML in general. It relies on XPath as well as the language defined by Extensible Stylesheet Language (XSL). We have chosen this type of transformation for this particular scenario because we need a robust way of providing transformation semantics that may be confusing or impossible to do with an XPath transformation alone. To bolster this scenario with some concrete examples consider Listing 6-15; this figure shows an example `<Manifest>` element containing several `<Reference>` elements that will participate in an XML Signature.

Our goal here is to target the first `<Reference>` element with the Id attribute SubjectToChange. Suppose we know in advance that the location of the data object will change over time. This means that we would like to *exclude* the string URI=http://www.server1.com/document .xml from the actual digest value. By excluding this URI attribute from the computation, we are allowing for the location of this document to change from one server to another without breaking the signature. Realize, however, that these extra processing rules are going to be application-specific. What we mean here is that the application will have to decide its own rules for processing the internals of the `<Manifest>` element; The `<Reference>` elements inside `<Manifest>` are *not* processed as a part of core generation. Furthermore, if the application feels that it doesn't need

Listing 6-15

A `<Manifest>` element with three `<Reference>` elements

```
<Manifest Id="Manifest1">
  <Reference Id="SubjectToChange"
URI="http://www.server1.com/document.xml">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>vvVZy3yga7CyFcPzPt0C7cA3as8=</DigestValue>

  </Reference>
  <Reference URI="#ref2">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>mBeIaZFGfdsDI48HGcvzQA3qio=</DigestValue>
  </Reference>
  <Reference URI="http://www.server5.com/document5.xml">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>qBeBaZFGqweDI48KZcvztv3qyo=</DigestValue>
  </Reference>
</Manifest>
```

the URI attribute at all, nothing precludes the application from obtaining the proper document from a cache or from an alternate location. Our final goal is to create an XSLT transform that produces the same XML document shown in Figure 6-10, but with the URI attribute omitted.

When the `<Manifest>` element shown in Listing 6-16 is finally validated, the XML verification engine doesn't know or even care about a URI attribute that once appeared in an attribute list inside a `<Reference>` element somewhere in a `<Manifest>` element. Once the transformation is performed, the verification happens, and the URI attribute is forgotten —in a sense, it is completely out of scope for the XML Signature verification. The URI attribute is removed long before the XML digest value is checked as a part of core validation. To make this picture a bit more explicit, consider Figure 6-3, which shows a pictorial representation of the relationship between the application, the two XML documents and the core validation process.

Figure 6-3 is quite rich, and there are a lot of things described in the picture. To begin with, the figure is split into two parts. Figure 6-3(a) shows the meta-steps of core generation; Figure 6-3(b) shows the same for core validation. The complete syntax for the `<Signature>` and `<Reference>` elements is omitted for clarity, and not all of the core generation steps are shown. There are two main points of concern about Fiure 6-3(a). The first point is the presence of the large bubble that is prepended to the core generation process. This bubble denotes application-specific

Listing 6-16

Omitting a URI attribute

```
<Manifest Id="Manifest1">
  <Reference Id="SubjectToChange">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>vvVZy3yga7CyFcPzPt0C7cA3as8=</DigestValue>
  </Reference>
  <Reference URI="#ref2">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>mBeIaZFGfdsDI48HGcvzQA3qio=</DigestValue>
  </Reference>
  <Reference URI="http://www.server5.com/document5.xml">
    <DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
    <DigestValue>qBeBaZFGqweDI48KZcvztv3qyo=</DigestValue>
  </Reference>
</Manifest>
```

Figure 6-3

Signature Creation and Validation with an XSLT Transform to omit a URI
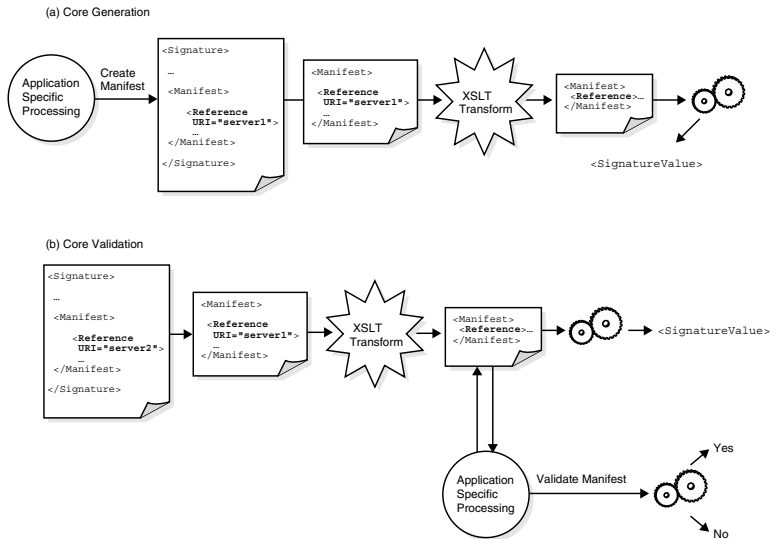
processing that is happening outside of the normal core generation process. In particular, the scenario with the `<Manifest>` element requires that the `<Reference>` elements inside this element be de-referenced and subsequently digested. If you look back at the definition of core generation in Chapter 5, you will notice that reference generation is defined around the contents of a `<SignedInfo>` element. It is *these* `<Reference>` elements that are explicitly de-referenced and digested. That is, the application that houses the signing engine cannot necessarily rely on the semantics of core generation for XML Signatures to de-reference and digest its `<Reference>` elements in any other place *but* the `<Signed-Info>` element. Most XML signing packages provide for this functionality as a convenience, but it is not a required part of the processing model.

The second point is the XSLT transform. Notice that this transform omits the URI attribute in the `<Reference>` (after the transform, `server1` has been omitted) element. The actual data that is signed is comprised of the `<Reference>` element *except* for its URI attribute. This has the obvious consequence of allowing anyone to change the URI attribute without altering the signature. Finally, the `<Manifest>` element is digested, and then the signature value is created. This last step omits several intermediate steps (if only for lack of room on the page). Remember, the `<Manifest>` that is digested is just one of many data objects (it too is referred to from a `<Reference>` element), so to be complete, Figure 6-3(a) should probably also show the `<SignedInfo>` element being signed to create the final `<SignatureValue>`.

Figure 6-3(b) shows the same process for core validation. It is extremely important to reiterate that the application is responsible for establishing the proper validity semantics when a `<Manifest>` element is used. The `<Reference>` elements that live inside a `<Manifest>` element are *not* de-referenced and validated as a part of core validation (to double check this, reconsider the core validation process shown in Chapter 5). To rephrase, each `<SignatureValue>` calculated in Figure 6-3 is identical and is impervious to changes in the location of the file (URI) *and* the contents of the file. The URI and the file can change or be completely removed, and core validation would still pass in this instance. This fact provides justification for the bubble that is attached to the `<Manifest>` element in Figure 6-3(b). In this case, the application is responsible for dereferencing the `<Reference>` inside the `<Manifest>` element and checking the digest itself.

Some readers may be wondering about the technical details of the XSLT transform and how it fits into the logistics of signature creation.

The XSLT transform is a transform just like any other; the only main difference between this transform and others is that it is an *optional* transform for the XML Signature Recommendation. This means that you might not find it in all implementations of XML Signatures. Listing 6-17 shows a sample XSLT transformation that contains logic to omit a URI attribute within a <Manifest> element provided that it has a certain Id attribute. In this case, if the <Reference> element has an Id attribute of SubjectToChange, then the URI attribute of that particular <Reference> element will be left out of the result set for the transform. We will not discuss the specifics of how XSLT works in this book. It is recommended that the reader visit the references section at the end of the book for more information on XSLT.

The XSLT code in Listing 6-17 consists of two meta-steps. The first step has the effect of an identity transform and collects all of the nodes in the

Listing 6-17

A sample XSLT transform for excluding an attribute

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output encoding="UTF-8" indent="no" method="xml" />

<!-- Collect all nodes in the document -->
<xsl:template match="node()|@*">
  <xsl:copy>
    <xsl:apply-templates select="node()|@*"/>
  </xsl:copy>
</xsl:template>

<!-- Find the element with the URI we want to nuke -->
<xsl:template match="Reference[@Id='SubjectToChange']">

  <xsl:element name="Reference">
    <xsl:attribute name="Id">
      <xsl:value-of select="@Id" />
    </xsl:attribute>

    <!-- Recreate the DigestMethod element -->
    <xsl:element name="DigestMethod">
      <xsl:attribute name="Algorithm">
        <xsl:value-of select="DigestMethod/@Algorithm" />
      </xsl:attribute>
    </xsl:element>

    <!-- Recreate the DigestValue element -->
    <xsl:element name="DigestValue">
      <xsl:value-of select="DigestValue" />
    </xsl:element>
  </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

input document. The second step catches all `<Reference>` elements with an `Id` attribute of `SubjectToChange` and recreates these elements without their URI attribute. This can be done because the schema definition for `<Reference>` is well defined with regard to the ordering of its child elements `<DigestMethod>` and `<DigestValue>`.

The final topic of discussion for this scenario is how to associate an XSLT transform with a particular `<Reference>` element. We have seen the syntax for an actual XSLT transform; now it is time to see how this XSLT code is associated with the proper `<Reference>` element. We must be especially careful and remember that we are talking about the `<Reference>` element in two distinct ways: There is the set of `<Reference>` elements inside the `<Manifest>` (where we would like to omit the URI attribute), and then there is the set of `<Reference>` elements inside the `<SignedInfo>`. It is one of these `<Reference>` elements that will ultimately refer to the `<Manifest>` element. Figure 6-4 makes this point explicit in a pictorial view.

We have seen figures similar to Figure 6-4 in previous discussions. This picture is repeated again for clarity. Creating a `<Transforms>` child element with the proper `<Transform>` subchild and necessary URI identifier is the designated way of adding an XSLT transform to a `<Reference>` element. In most cases, the XSLT style sheet code is added directly to the `<Transform>` child element, but it can be inline within the XML document being signed. Listing 6-18 shows how the XSLT code (given initially in Listing 6-17) is added to become part of a `<Reference>` element that references the necessary list of resources inside the `<Manifest>` element.

Notice in Listing 6-18, we have simply taken the XSLT code and placed it inside a `<Transforms>` element. Notice the URI identifier (shown in bold) for XSLT; this is the URI identifier specified by the XML Signature Recommendation for use with the XSLT transform. The only thing that prevents our XSLT transform from becoming completely functional is its lack of namespace-qualified elements. When using an XSLT transform, all of the names of the elements that are referred to should be namespace-qualified; this prevents the transformation from getting confused between possible name clashes when the transformation is underway. In this case, we would add the XML Signature Namespace, `xmlns="http://www.w3.org/2000/09/xmldsig#"`, which was explicitly left out of the example in the interest of clarity.

```
<Signature>

 <SignedInfo>

   <Reference URI="#Manifest1">


   </Reference>

 </SignedInfo>

 <Object>
  <Manifest URI="Manifest1">

   <Reference
    Id="SubjectToChange"
    URI=
    "http://www.server1.com/document.xml">


   </Reference>

  </Manifest>

 </Object>

</Signature>
```

XSLT

Figure 6-4

Associating the
XSLT Transform
with a
`<Reference>`
element
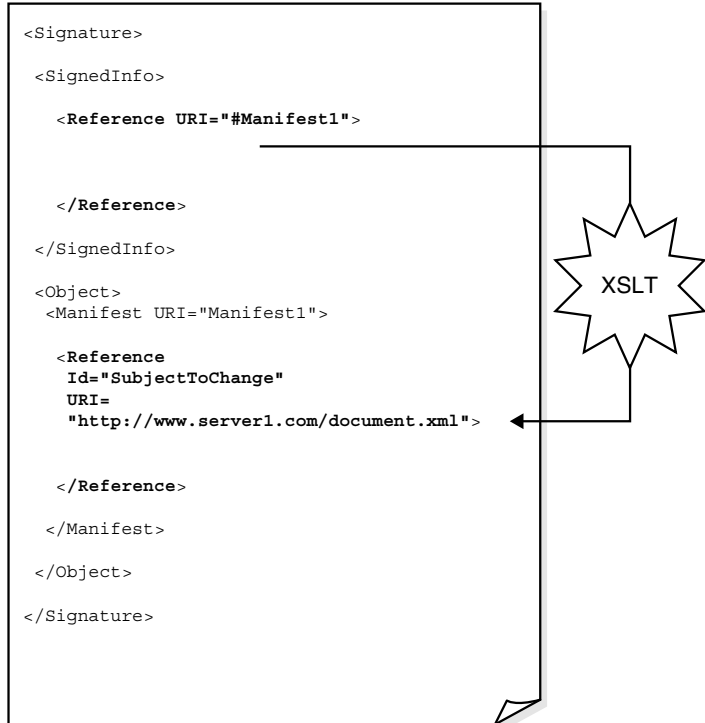
Scenario 13

How is it possible for an XML document *D* containing an XML Signature
S to exhibit all three types of XML signatures (enveloped, enveloping, and
detached) simultaneously?

Proposed Solution

As the complexity of a signed XML document increases, the classification
of an XML Signature into three distinct types begins to decay. All three
types of signatures can be combined and exhibited in a single document.

```
<Reference URI="#Manifest1">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-
    19991116">
    <xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output encoding="UTF-8" indent="no" method="xml" />

    <!-- Collect all nodes in the document -->
    <xsl:template match="node()|@*">
      <xsl:copy>
        <xsl:apply-templates select="node()|@*"/>
      </xsl:copy>
    </xsl:template>

    <!-- Find the element with the URI we want to nuke -->
    <xsl:template match="Reference[@Id='SubjectToChange']">

      <xsl:element name="Reference">
        <xsl:attribute name="Id">
          <xsl:value-of select="@Id" />
        </xsl:attribute>

        <!-- Recreate the DigestMethod element -->
        <xsl:element name="DigestMethod">
          <xsl:attribute name="Algorithm">
            <xsl:value-of select="DigestMethod/@Algorithm" />
          </xsl:attribute>
        </xsl:element>

        <!-- Recreate the DigestValue element -->
        <xsl:element name="DigestValue">
          <xsl:value-of select="DigestValue" />
        </xsl:element>
      </xsl:element>
    </xsl:template>
    </xsl:stylesheet>
    </Transform>
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"
  />
  <DigestValue>b4wq5Qxos7IqNVkiPy7/ffI+dCd=</DigestValue>
</Reference>
```
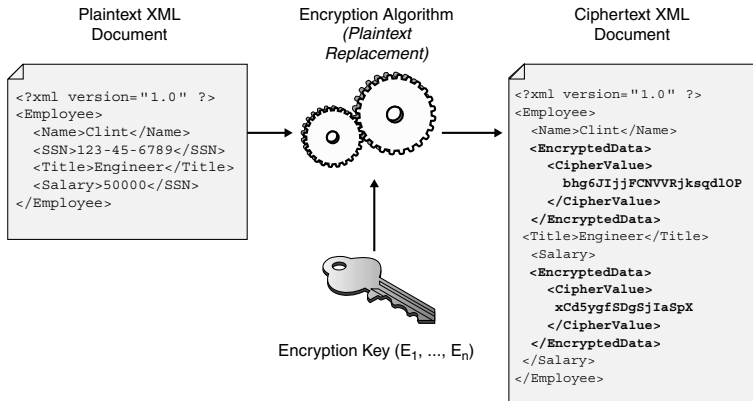
This leads to a signed XML document that cannot be neatly grouped into
one of the three categories. Listing 6-19 shows an example XML Signa-
ture whose `<SignedInfo>` element contains three disparate resources,
all of which represent a different signature type.

Notice in Listing 6-19 that the `<Signature>` element (with an Id
value of ThreeTypes) is enveloped by the `<Contract1>` element. This
particular association gives the XML Signature the enveloped property.

Listing 6-19

An XML
Signature that is
detached,
enveloped, and
enveloping

```
<Contract1>
  <ImportantContent Id="ImportantElement">
    This is important content!
  </ImportantContent>
  <Signature Id="ThreeTypes">
    <SignedInfo>
      <Reference URI="http://www.remote-server.com/file.doc">
        . . .
      </Reference>
      <Reference URI="#contract2">
        . . .
      </Reference>
      <Reference URI="#ImportantElement">
        . . .
      </Reference>
    </SignedInfo>
    <SignatureValue>  . . .  </SignatureValue>
    <Object Id="contract2">
      <Contract2> This is also very important content! </Contract2>
    </Object>
  </Signature>
</Contract1>
```

Furthermore, because the `<Signature>` element also does some of its
own enveloping (it signs the `<Object>` element that houses `<Con-
tract2>`), the signature is enveloping. Finally, the XML Signature also
references a remote file (`http://www.remote-server.com/file.
doc`). This gives the signature the detached property.

Scenario 14

An XML document containing an XML Signature *S* has white space
added to the `<SignedInfo>` element. The addition of white space seems
to change the `<SignatureValue>` and result in a broken signature,
despite the fact that Canonical XML is being employed. How can this be
explained?

Proposed Solution

Canonical XML only normalizes white space in certain areas of the XML
document being canonicalized. That is, placing white space haphazardly
inside a signed XML document cannot be done without breaking the sig-
nature because Canonical XML treats some white space as semantically

meaningful. For example, consider Listing 6-20. Notice that there is white space between elements in the XML document—this type of white space is actually *preserved* by Canonical XML, despite the fact that one might think otherwise.

Canonical XML treats white space *outside* of the document element or *inside* start and end tag pairs as semantically meaningless. Notice that we have said *inside* start and end tags, not between. Consider Listing 6-21(a), which shows an input XML document before canonicalization takes place. Listing 6-21(b) shows the same document after applying Canonical XML.

Listing 6-20

Canonical XML preserves white space between elements.

```
<Cheese_Types>

  <Cheese> Swiss </Cheese>

  <Cheese> Monterey Jack </Cheese>

  <Cheese> Cheddar </Cheese>

</Cheese_Types>
```

Listing 6-21(a)

Superfluous white space is inside tags and outside the document element.

```
<Cheese_Types      >

  <Cheese> Swiss </Cheese       >

  <Cheese> Monterey Jack </Cheese>

  <Cheese> Cheddar </Cheese>

</Cheese_Types      >




<!-- These are my favorite Types of Cheese! -->
```

Listing 6-21(b)

Superfluous white space is normalized after applying Canonical XML 1.0.

```
<Cheese_Types>

  <Cheese> Swiss </Cheese>

  <Cheese> Monterey Jack </Cheese>

  <Cheese> Cheddar </Cheese>

</Cheese_Types>

<!-- These are my favorite Types of Cheese! -->
```

Chapter Summary

This chapter looks at a number of different problems and scenarios where XML Signatures are used. Some scenarios try to solve a particular problem, while others show general solutions for constructing XML Signatures of different types. There is focus in this chapter on how signatures look when different scenarios are applied that require multiple signers, multiple keys, or application-specific constraints. There is also some discussion on Base-64 encoding and how the format for Base-64 printable encoding conflicts with normalization done as a part of standard XML processing and the application of Canonical XML. The main goal of this chapter is to provide a playing field where the reader can test out the knowledge learned in Chapters 4 and 5.

This page intentionally left blank.

CHAPTER 7

Introduction to XML Encryption

Unlike XML Signatures, XML Encryption has a somewhat less exciting introduction. However, this shouldn't be the case. XML Encryption builds on some of the elements and ideas present in the XML Signature syntax and allows for some interesting combinations of XML documents that exhibit both signed and encrypted properties. This normative dependence of one technology on another is a recurring theme in XML Security (as well as a useful general idea for any set of related standards). XML Encryption and the XML Key Management Specification (XKMS) (discussed in Chapter 9) share elements, algorithms, and general concepts that were first introduced in the XML Signature drafts.

Throughout this chapter, the reader will notice the mixing of elements from different namespaces; some confusing situations arise from the sharing of elements between XML Signatures and XML Encryption. To keep things clear, the reader should assume that all elements shown in this chapter belong in the XML Encryption namespace unless they are prepended with the `ds` (digital signature) namespace prefix, which means that they are from the XML Signature namespace. For instance, consider the following example. The `<EncryptionMethod>` element is in the XML Encryption namespace, but the `<DigestMethod>` element belongs to the XML Signature Recommendation:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
 <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
</EncryptionMethod>
```

The abbreviation `ds` comes from a namespace declaration that maps the `ds` prefix to the XML Signature namespace. The previous example isn't complete and doesn't represent a very useful construct for XML Encryption. This is intentional; the goal is to simply show an example of how we can differentiate between various namespaces in a somewhat standard way. Namespace designation is a very important aspect of XML documents that rely on different standards or technologies, because this is the only way to disambiguate between element names.

XML Encryption Basics and Syntax

Some people might argue that that XML Encryption is a bit simpler than XML Signatures. This argument might come from the fact that the physical size of the standard is shorter than the XML Signature Recommendation. Some people might also argue that the normative dependence of XML Encryption on XML Signatures simply means that some of the material is repeated. Still, others might claim that XML Encryption has simpler semantics; no messy transforms get in the way of the actual plaintext being encrypted and there are far fewer optional elements that complicate things. Whether or not these statements are true makes little difference; we will proceed in the same rigorous manner as we did in Chapter 4 and focus on how XML Encryption works from a conceptual point of view.

When we approached XML Signatures, we looked at some fundamental definitions that helped us understand the nature of an XML Signature. In order to keep the discussion consistent, we will do the same with XML Encryption. The only problem is that XML Encryption doesn't have a proper noun to define. For example, in Chapter 4 we were able to define an XML Signature. That is, we were able to identify a single entity as "an XML Signature" with certain properties. There is no such thing as an XML Encryption. This sentence isn't even grammatical.

Instead, XML Encryption defines a process for encrypting data and representing the result using the syntax of XML. In some ways, the XML

Encryption draft is more of a packaging technology rather than a completely new idea. In order to see how XML Encryption packages data, it is useful to examine some use cases for this technology.

XML Encryption Use Cases

It is useful to divide the operation of XML Encryption into two broad use cases: encrypting arbitrary octets and encrypting XML data. This division may seem like a line in the sand because one can argue that XML data can also be considered arbitrary data. The reason why this division is necessary is because the XML Encryption draft recommends slightly different semantics for the encryption of XML data versus just plain octets. In particular, in some cases it is convenient to perform a replacement operation on the source XML document where the encrypted XML replaces the source plaintext in the original XML document. This use case and others will be examined in the coming sections and will be called *plaintext replacement*.

The first broad use case that we will examine is encrypting arbitrary octets. This is perhaps the simplest case and makes it easy to see the nature of XML Encryption. Let us consider a standard web resource in the form of a Uniform Resource Identifier (URI). Suppose that we want to dereference the URI and apply an encryption algorithm to the data to create an encrypted XML representation. The high-level view is shown in Figure 7-1.

Figure 7-1

A bird's eye view of XML Encryption on arbitrary data

Encryption Algorithm

```
<?xml version="1.0" ?>
```
Encrypted
XML document

Source
URI

Encryption Key (E_k)
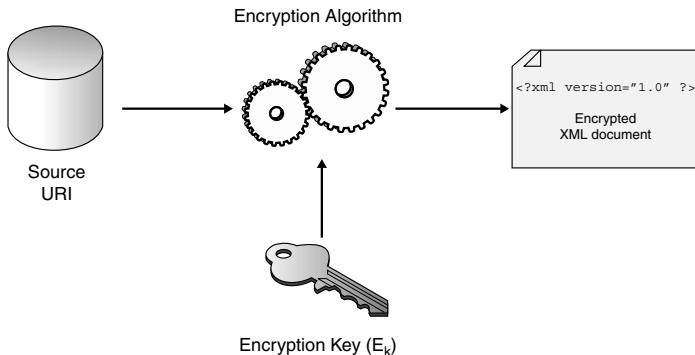
The high-level view shown in Figure 7-1 is a simplification of the entire encryption process, but all of the necessary pieces are present. The *source URI* can be a remote web resource that is retrievable by any sort of protocol scheme, or it can be a local file. The *encryption algorithm* is a block cipher or stream cipher that operates with an *encryption key* (E_k), and the output is an encrypted form of the source URI as represented in XML. The overall picture looks almost like normal non-XML encryption. The two changes include the input data, which is specified as a web resource instead of just arbitrary octets, and the final encrypted form, which is represented in XML instead of raw octets. The fact that we have modeled the input data as a URI is quite significant and is part of a paradigm shift toward disparate distributed resources that persist across potentially vast networks. Shifting the perception of the plaintext from a single stationary physical document to a URI changes the scope of encryption and expands its role. This contrasts with the classic view of encryption, which views the plaintext as simple binary. At some level, however, this high-level view breaks down because the actual encryption algorithm treats the input data as octets.

The result of the encryption operation is some sort of XML document. We haven't yet described what this is yet, but we'll discuss it soon. The requirements of the output document include some of the same properties as an XML Signature. It should be human-readable and should provide a facility for the discovery of the decryption key. Furthermore, it should also provide enough context and additional information about the algorithm used and any additional required semantics. This XML representation is the `<EncryptedData>` element, which is discussed in the following section.

The <*EncryptedData*> Element

The `<EncryptedData>` element is one of the fundamental elements in the XML Encryption syntax. Some people might say that this element is the most fundamental; this point can be argued because the `<EncryptedData>` element actually derives from a more general `<EncryptedType>` element that exists only in the XML Encryption schema definitions. The reader should ignore this distinction for now, as it will be revisited with more detail at a later time.

When arbitrary data is encrypted (such as in Figure 7-1), the result is an `<EncryptedData>` element. The `<EncryptedData>` element has the structure shown in Listing 7-1.

Listing 7-1

The
<Encrypted-
Data> element

```
<EncryptedData Id? Type?>
  <EncryptionMethod/>?
    <ds:KeyInfo>
      <EncryptedKey>?
      <AgreementMethod>?
      <ds:KeyName>?
      <ds:RetrievalMethod>?
      <ds:*>?
    </ds:KeyInfo>?
    <CipherData>
      <CipherValue>?
      <CipherReference URI?>?
    </CipherData>
    <EncryptionProperties/>?
</EncryptedData>
```

At first glance, the <EncryptedData> element looks very confusing. In order to get around the initial confusion, we will take an intellectual knife (once again) and create the vacuous version of <EncryptedData>. The vacuous <EncryptedData> element is intended to represent the simplest case of the encryption syntax and make it comparable to raw encryption, as presented in Chapter 2. The vacuous <EncryptedData> element is shown in Listing 7-2.

The vacuous <EncryptedData> element has only one direct child element called <CipherData>. The <CipherData> element is responsible for housing the actual encrypted value in some form or another. In this example, the encrypted data is inside the <CipherValue> element; in Listing 7-2, we show only the opening tag <CipherValue> and omit any content. To fill this element and make it a real example, an instance of the vacuous <EncryptedData> element is presented in Listing 7-3.

Listing 7-3 shows the addition of some arbitrary encrypted data inside the <CipherValue> child element. The reader should notice the nature of the data inside the <CipherValue> tag—it is printable ASCII as encoded via RFC 2045. In other words, it is Base-64-encoded data. Naturally, Base-64-encoded data is used to provide a printable representation of the otherwise binary output from the encryption operation.

The most important thing to notice about Listing 7-3 is that the basic concept is simple and intuitive. We have a single element called <EncryptedData> that has cipher data inside of it. This cipher data is given as a direct value, and the value is shown. There is nothing to hide and very little confusion at this point. In this broad use case, we are

Listing 7-2

The vacuous
`<Encrypted-Data>` element

```
<EncryptedData>
  <CipherData>
    <CipherValue>
  </CipherData>
</EncryptedData>
```

Listing 7-3

An instance of the
vacuous
`<Encrypted-Data>` element

```
<EncryptedData>
  <CipherData>
    <CipherValue>
      EwJVUzEPMA0GA1UECBMGQXRoZW5zMRUwEwYDVQ
      QKEwxQaGlsb3NvcGhlcnMxETAPBgNVBAMTCFNv
      Y3JhdGVzMSIwIAYJKoZIhvcNAQkBFhNzb2NyYX
      Rlc0BhdGhlbnMuY29tMB4XDTAxMDIxNjIzMjgz
    </CipherValue>
  </CipherData>
</EncryptedData>
```

treating the source data as an octet stream. It could be an entire XML document or a binary format graphic file; the nature of the binary stream is irrelevant. When we treat the source plaintext as XML, the semantics of XML Encryption take a slight turn.

Plaintext Replacement

Suppose we have an XML document that contains sensitive data. This sensitive data can be one or more elements and their children (document subsets), or element content (just the text). XML Encryption allows for element-level and element-content encryption. Because XML is structured using markup, this markup takes on a role of establishing a boundary between the data that needs to be protected and the data that can be sent in the clear. As the number of sensitive elements in an XML document grows, the number of potential privacy configurations increases. The additional possible application semantics that are provided based on element-level encryption is rather exciting and contrasts with raw encryption, which is usually a binary operation (either the data is encrypted or sent in the clear). It is difficult to create a piece of plaintext that can be selectively encrypted without severely fragmenting the plaintext or using a heavyweight encoding scheme such as ASN.1. The picture for plaintext replacement is much different than the picture presented in Figure 7-1. Most diagrams in security or cryptography books show the same ho-hum Alice and Bob picture; Figure 7-2 is quite different and showcases one of the powerful features of XML Encryption.

Plaintext XML Document

```
<?xml version="1.0" ?>
<Employee>
  <Name>Clint</Name>
  <SSN>123-45-6789</SSN>
  <Title>Engineer</Title>
  <Salary>50000</SSN>
</Employee>
```

Encryption Algorithm
(Plaintext Replacement)

Ciphertext XML Document

```
<?xml version="1.0" ?>
<Employee>
  <Name>Clint</Name>
  <EncryptedData>
    <CipherValue>
      bhg6JIjjFCNVVRjksqdlOP
    </CipherValue>
  </EncryptedData>
 <Title>Engineer</Title>
  <Salary>
  <EncryptedData>
    <CipherValue>
     xCd5ygfSDgSjIaSpX
    </CipherValue>
  </EncryptedData>
 </Salary>
</Employee>
```

Encryption Key (E_1, ..., E_n)

In Figure 7-2, our input plaintext represents an employee record written using a fictional XML-based markup language. The employee record has four elements: <Name>, <SSN>, <Title>, and <Salary>. For the sake of the example, let us assume that the entire <SSN> element and the contents of the <Salary> element are sensitive and need to be encrypted. With XML Encryption, this can be done using plaintext replacement. In Figure 7-2, the relevant data (the entire <SSN> element and the contents of the <Salary> element) are encrypted and replaced with an <EncryptedData> element. The actual encrypted data appears as the Base-64-encoded text content of <CipherValue>.

Figure 7-2 leaves out many details of the <EncryptedData> element. The figure shows a simple case to illustrate the main idea: Many <EncryptedData> elements are possible in a single XML document, and each of these <EncryptedData> elements represents the encrypted form of an element, a set of elements, or element content.

The other subtle point that should be mentioned is that the encryption keys used can be numerous for the given plaintext XML document. For example, the first <EncryptedData> element in the ciphertext XML document may be encrypted with encryption key E_1, the second such <EncryptedData> element may be encrypted with encryption key E_2, and so on. This idea is a bit different than normal raw encryption, where it is usually assumed that the plaintext will be encrypted in its entirety with a single key. The XML Encryption process enables element-level encryption using multiple symmetric or asymmetric keys.

The two previously discussed XML Encryption use cases are foundational to the rest of the syntax and processing rules. In order to avoid as much confusion as possible, we will leave the plaintext replacement use case for now and return to the discussion of encrypting arbitrary octets. From here, we will look at the rest of the features of `<EncryptedData>` as well as `<EncryptedKey>` and the schema-defined `<EncryptedType>` parent. Once we have exhausted the possibilities for the arbitrary octets use case, we will return to plaintext replacement and apply the details.

The `<EncryptedData>` Element: Details

Listing 7-1 shows the possible elements used in the `<EncryptedData>` structure. There are a total of four possible child elements as well as two attributes: `Id` and `Type`. The `Id` element is a straightforward arbitrary identifier that can be used for application-specific processing or for something as simple as distinguishing between multiple `<EncryptedData>` elements in a single XML document. This `Id` attribute is similar to the one found in the `<ds:Signature>` element, as discussed in Chapter 4. The `Type` attribute is a bit more interesting and is used to identify the type of the plaintext. We are using the term *type* in a broad sense here; the actual attribute value can be a *media type* (in the case of arbitrary octet encryption) or one of two prespecified values: `Element` or `Content`.

If the `Type` attribute is specified as media type, this means we chose an Internet Address Naming Authority (IANA) media type from the tree hosted at **www.isi.edu**. For example, if our plaintext is an HTML document, we should use `http://www.isi.edu/in-notes/iana/assignments/media-types/text/html` as the attribute value. The other prespecified values, `Element` and `Content`, are used when the plaintext is an XML document subset. That is, if we perform plaintext replacement, we must identify what we are encrypting: an element or element content. For example, if we were to use XML Encryption to encrypt a single element or a set of elements, we would use `http://www.w3.org/2001/04/xmlenc#Element` as the attribute value. Similarly, if we were performing plaintext replacement on element content, we would use `http://www.w3.org/20001/04/xmlenc#Content` as the attribute value.

The `Type` attribute, although optional, is vitally important in order to correctly process an XML document that contains `<EncryptedData>` elements. For example, in the case of plaintext replacement, the decrypting

application must know the original type of the plaintext (either `Content` or `Element`) in order to correctly reconstruct the original plaintext XML document. This differs from the similar `Type` attribute (in the `<ds: Reference>` element) used in the XML Signature syntax (discussed in Chapter 4). In the XML Signature processing rules, the correct processing of the `Type` element is not required and might not be necessary in all cases. The XML Encryption processing rules will be discussed further in coming sections. The information about `Type` usage for `<Encrypted- Data>` is summarized in Table 7-1.

The `Id` and `Type` attributes are the only two possible attributes defined for the `<EncryptedData>` element. The possible child elements for `<EncryptedData>` follow next. There are only four possible child elements for `<EncryptedData>`, including `<ds:KeyInfo>`, which we have seen in previous chapters. `<ds:KeyInfo>` is borrowed from the XML Signature Recommendation.

The <*EncryptionMethod*> Element

The `<EncryptionMethod>` element is the first possible child element of `<EncryptedData>` and is responsible for identifying the encryption algorithm along with possible auxiliary parameters such as key size padding scheme (for asymmetric ciphers), or encryption mode. The encryption algorithm is denoted with a URI identifier in the same vein as similar elements defined in the XML Signature Recommendation (such as `<ds:DigestMethod>` or `<ds:SignatureMethod>`). The `<Encryption- Method>` element, however, differs markedly from `<ds:DigestMethod>` or `<ds:SignatureMethod>` in that it has possible child elements. `<ds:DigestMethod>` and `<ds:SignatureMethod>` are always empty and usually comprise only a single line.

The XML Encryption draft makes the distinction between implicit algorithm parameters and explicit algorithm parameters. The implicit

	Plaintext Type	**Type Attribute Value**
Table 7-1	An XML element	`http://www.w3.org/2001/04/` `xmlenc#Element`
Content Types for XML Encryption	XML element content (includes a set of elements)	`http://www.w3.org/20001/04/` `xmlenc#Content`
	IANA media type	`http://www.isi.edu/in-notes/iana/` `assignments/media-types/*/*`

parameters of an encryption algorithm include the plaintext data, the encryption key, and the initialization vector (IV). The IV is used in some modes of symmetric cipher operation and provides a kickstart for a block cipher. For more information on IVs, see the primer in Chapter 2. Implicit parameters for cipher operation are not shown in any of the `<EncryptionMethod>` child elements or attributes. It is assumed that the application knows where to get things such as the plaintext and key (these are obvious implementation issues). The IV is actually melded into the ciphertext (in the case of a decryption operation). Finding the IV is functionally equivalent to finding the ciphertext.

The implicit algorithm parameters contrast the explicit parameters, which are either part of the URI identifier or a child element of `<EncryptionMethod>`. Explicit parameters denote things like key size, encryption mode, padding scheme, or any optional algorithm-dependent parameters. The `<EncryptionMethod>` element enables the addition of arbitrary namespace qualified child elements for any parameters required of custom encryption algorithms. The set of algorithms specified with the XML Encryption draft is not rigid; it is possible to add new algorithms as they become available. The XML Encryption syntax supports a number of encryption algorithms. An example of identifying Triple-DES in Cipher Block Chaining (CBC) mode as the encryption algorithm would look like the following:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
```

In a similar manner, the Advanced Encryption Standard (AES) cipher (also in CBC mode) is denoted as follows:

```
<EncryptionMethod
  Algorithm=http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
```

The reader should note that this URI identifier contains explicit parameters in the string itself. In the case of Triple-DES, the substring `cbc` is appended to the identifier for CBC mode. For the case of AES, the substring `128` and `cbc` are both appended, indicating that 128-bit AES is being specified in CBC mode.

Given that many explicit cipher parameters are present in the string, the reader may be wondering about some of the possible child elements of `<EncryptionMethod>`. The XML Encryption draft relies on three such elements: `<KeySize>`, `<ds:DigestMethod>`, and `<OAEPparams>`. The

`<KeySize>` element is used to denote the size of the encryption key, the `<DigestMethod>` element specifies the digest method to use during certain padding operations, and the `<OAEPparams>` object is used for transmitting parameters necessary for RSA operations that use the Optimal Asymmetric Encryption Padding (OAEP) scheme. For more information on OAEP, see the primer in Chapter 2. It is important to note that the `<ds:DigestMethod>` element used is the same one from the XML Signature Recommendation; this is a clear case of the normative dependence of one XML Security technology on another.

The `<KeySize>` child element may seem odd because we have just looked at two examples of ciphers that specify the key size in the string itself. The reader may be asking, Why specify the key size twice? The answer to this objection lies in the fact that some ciphers (such as stream ciphers) take a variable key size. That is, not all uses of a stream cipher implicitly denote a given key size, and a mechanism must exist to make this explicit so the decryption operation is possible. Further, some block ciphers can use variable key sizes (such as RC5). The `<KeySize>` element fulfills this role for ciphers that are not specified by the XML Encryption draft. The XML Encryption draft does, however, offer a constraint on the use of the `<KeySize>` child element. The value inside the `<KeySize>` element must match the key size implied in the URI identifier for the given cipher. If there is a mismatch, an error must be reported. For example, the following example would be invalid because the `<KeySize>` element does not match the implied key size of the chosen symmetric cipher. (Triple-DES uses a key size of 192 bits.)

```
<EncryptionMethod
  Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc">
  <KeySize>256</KeySize>
</EncryptionMethod>
```

The next two elements, `<ds:DigestMethod>` and `<OAEPParams>`, are specific to the OAEP padding scheme used for RSA operations. There is no restriction, however, from using `<ds:DigestMethod>` along with another padding scheme that might also require a hash function. The contents of the `<OAEPParams>` corresponds to the optional encoding parameter *P,* as specified in RFC2437 (PKCS#1). This parameter is an arbitrary Base-64-encoded octet string that is hashed during the encoding operation for OAEP.

The <*CipherData*> *and* <*EncryptionProperties*> *Elements*

The next element immediately after <EncryptionMethod> inside the <EncryptedData> is <ds:KeyInfo>. The <ds:KeyInfo> element is the same element shared from the XML Signature Recommendation with a few additions. It makes sense to defer our discussion of this element, skipping over to the last two elements of <EncryptedData>, which are <CipherData> and <EncryptionProperties>. We defer talking about <ds:KeyInfo> because this element opens the door to additional complexity for XML Encryption. The <ds:KeyInfo> element displays the recursive nature of XML Encryption and helps introduce the important <EncryptedKey> element. It's best to hold off on the excitement right now and finish up with the two simpler elements <CipherData> and <EncryptionProperties>.

The <CipherData> element is one of the few mandatory child elements of <EncryptedData>. <CipherData> either *envelopes* or *references* the encrypted data. If <CipherData> is acting as an envelope, the <CipherValue> child element will be present and contain the Base-64-encoded cipher data (this enveloping is shown in Listing 7-3). Conversely, if <CipherData> is acting as a reference, the <CipherReference> element will be present. The <CipherReference> element behaves a lot like the <ds:Reference> element from the XML Signature Recommendation. The only marked difference is that <CipherReference> references cipher data (encrypted), whereas <ds:Reference> references plaintext for signing. In all cases, the XML Encryption schema restricts <CipherData> to either <CipherReference> or <CipherValue>. Both elements cannot be present.

The <CipherReference> child element is useful in cases where the encrypted data becomes so large that it becomes infeasible to put it in a text representation for participation in an XML document. In this case, the location of the ciphertext is specified with a URI. This is shown in the following example.

```
<CipherData>
  <CipherReference URI="www.fake-site.com/encryptedfile.bin"/>
</CipherData>
```

In the previous example, the <CipherData> element points to an encrypted file somewhere on **www.fake-site.com**. One subtle complication to mention is the format of the encrypted data. In the case of <CipherValue>, the cipher data has been Base-64 encoded. In the alter-

native case of `<CipherReference>`, the raw (unencoded) octet stream is expected. This slight variation must be taken into consideration if ciphertext is transported from a `<CipherValue>` element to a file on a remote host.

Fortunately, the `<CipherReference>` element supports transforms that are similar to those used in the XML Signature Recommendation. For example, it is possible to specify a Base-64 decoding transform that would decode any Base-64-encoded ciphertext and yield the raw octet stream. This would be useful for transporting the contents of a `<CipherValue>` element directly into a file; this would enable the decryption of the content without further altering the cipher data. Assuming that `encryptedfile.b64` contains some Base-64-encoded cipher data, we might alter the previous example as follows:

```
<CipherData>
  <CipherReference URI="www.fake-site.com/encryptedfile.bin"/>
  <Transforms>
    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
  </Transforms>
</CipherData>
```

The careful reader might notice that there is no `ds` prefix on the `<Transforms>` child element, but there is a `ds` prefix on the actual `<Transform>` element used. In fact, the `<Transforms>` element shown in the previous example actually belongs to the XML Encryption draft and isn't part of the XML Signature Recommendation. This is because the `<Transforms>` element specified in XML Encryption has different semantics. Namely, the transforms in this example are a means to produce the decrypted octet stream and aren't defined for the encryption operation. This contrasts with the `<ds:Transforms>` element, which specifies that transformations should be applied during signature generation as well as during signature validation.

Nonce Value in `<EncryptedData>`

The `<EncryptedData>` element has a single optional attribute called *nonce*, which is used to alert the decrypting application that a nonce value has been mixed in with the encrypted data. The attribute specifies the length of the nonce value so that it may be stripped off once the decryption is complete.

A nonce value used in this context is a means to add more entropy to plaintext and thwart some forms of attack. In the case of XML documents, the plaintext data may have a great deal of known structure. That is, an

attacker may have access to a schema or document type definition (DTD) and might be able to perform a dictionary attack of some sort on the encrypted data. The nonce value prevents this from happening in this instance by increasing the size of the plaintext with some prepended randomness. The length of the nonce is specified as an integer attribute value and is stripped off during the decryption stage. The specifics of how the nonce is handled will be covered in the section "XML Encryption Processing Rules." The utility of a nonce value used with a block cipher in cipher block chaining mode can be argued. In particular, the initialization vector plays the same sole as the nonce value and doesn't add extra security properties for this specific case. It is likely that the nonce value will be removed from the XML Encryption drafts as it matures.

The `<EncryptionProperties>` Element

The final element in the `<EncryptedData>` structure is the `<EncryptionProperties>` element. This element is inspired from the `<ds:SignatureProperties>` element from the XML Signature Recommendation. It carries the same basic function: to include extra semantics and properties along with the encrypted data. The `<Encryption-Properties>` element contains one or more `<EncryptionProperty>` child elements, which can contain arbitrary elements from any namespace. Each `<EncryptionProperty>` element also has a provision for a `Target` attribute that enables the matching of a given `<Encryption-Property>` to an `<EncryptedData>` element. Furthermore, the XML Encryption draft also allocates a type value for use with the XML Signature Recommendation. That is, it may be useful to sign an `<Encryption-Properties>` element using an XML Signature. The value `http://www.w3.org/20001/04/xmlenc#EncryptionProperties` can be used in a `<ds:Reference>` element as the value of the `Type` attribute. This is one example where the standards mix together and share semantics easily. Listing 7-4 shows how the `<EncryptionProperties>` element can be signed using an XML Signature. Some readers may have skipped around a bit and haven't yet studied the previous chapters on XML Signatures. In order to understand what is happening in Listing 7-4, it is recommended that the reader understand how a basic XML Signature works. The discussion of XML Signatures covers Chapters 4, 5, and 6. Another point to note is that the signing operation used has the potential to affect overall performance because the signing operation is considerably slower than symmetric key encryption.

In Listing 7-4, we have an XML document called `<SecureDoc>` that contains a `<ds:Signature>` element as well as an `<EncryptedData>` element. Although this example is incomplete (many of the required elements from the XML Signature are omitted for brevity), it does include the namespace declarations, which are very important for disambiguating between the elements of XML Encryption and XML Signatures.

Listing 7-4 contains a `<ds:Reference>` element that signs the `<EncryptionProperties>` element using a fragment identifier with a detached signature. The signature is detached because it is neither a parent nor child to the data actually being signed (the `<Encryption-Properties>` element). In addition, the `Type` attribute on the `<ds:Reference>` element is also used. This enables a processing application to make better decisions about reference processing (in this case, it is an `<EncryptionProperties>` element); however, this `Type` attribute is not mandatory and may be omitted. Moreover, the `Target` attribute is also used to match the `<EncryptionProperties>` element to the

Listing 7-4

Signing an `<Encryption-Properties>` element with an XML Signature

```
<SecureDoc>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    ...
  <Reference
    URI="#EncProps"
    Type="http://www.w3.org/2001/04/xmlenc#EncryptionProperties">
    <DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
  </Reference>
  <SignatureValue>
    SJTDijgDflhdMlNjrWWKdkgj1hDhvhsdfx5BwpjfAuJ6T3ZysdfggA==
  </SignatureValue>
  </Signature>
  <EncryptedData Id="EncData1" xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>
        ErBgcqhkjOOAQBMIIBHgKBgQDaJjfDTrawMHf8MiUt
        Y54b37hSmYNnR3KpGT10uU1Dqppcju06uN0iGbqf94
        7DjkBC25hKnqykK31xBw0E
      </CipherValue>
    </CipherData>

    <EncryptionProperties Id="EncProps">
      <EncryptionProperty Target="EncData1">
        <TokenId>123456789</TokenId>
      </EncryptionProperty>
    </EncryptionProperties>
  </EncryptedData>
  ...
</SecureDoc>
```

correct `<EncryptedData>`. Although the use of the `Target` attribute in Listing 7-4 may seem trivial (no confusion is possible because there is only a single `<EncryptedData>` element), a case may arise where multiple `<EncryptedData>` elements reside in the same XML document context. The careful reader might notice that the use of `<Encryption-Properties>` in Listing 7-4 contrasts with the way `<ds:Signature-Properties>` is used. In the case of an XML Signature, the `<ds:SignatureProperties>` element resides inside a `<ds:Object>` container element. This difference in structure is fully expected because `<EncryptionProperties>` is specified as part of the `<Encrypted-Data>` element and must be part of the `<EncryptedData>` element.

Listing 7-4 is an interesting use case because it allows for the authentication of the `<EncryptionProperties>` element. In this example, the `<EncryptionProperties>` element contains a single `<Encryption-Property>` child that denotes a fictional serial number of a hardware token (one that ostensibly contains an encryption or decryption key). Without an XML Signature, an attacker would be able to modify the contents of the `<EncryptionProperties>` element with little recourse. It is important to realize that the `<EncryptedData>` element doesn't offer any sort of authentication other than that provided by the ciphertext. Altering the ciphertext will cause the plaintext to be incorrect upon decryption, but it is not certain that the change was malicious; for example, a data transfer error could have occurred.

Listing 7-4 shows one way of signing the `<EncryptionProperties>` element. In addition to the detached signature shown, another useful case would be an enveloped signature. For example, the `<ds:Signature>` element can reside inside the `<EncryptionProperties>` as an actual `<EncryptionProperty>`. This type of signature is shown in Listing 7-5.

Some people might argue that the use of `<EncryptionProperties>` in Listing 7-5 isn't semantically correct. A `<ds:Signature>` is not a property of the encrypted data, but instead it adds a property to the `<Encryption-Properties>` element, namely the signed property. It is possible to stick the `<ds:Signature>` element here because the schema definition for `<EncryptionProperty>` allows for arbitrary child elements.

This use case may seem a bit odd, but it may be useful in some circumstances because it provides a clean way of packaging the `<ds:Signature>` block within the `<EncryptionProperties>` element. The evidence for this can be seen in Listing 7-5, which has a cleaner structure than Listing 7-4. The important point to note is that we are not signing the actual contents of the `<EncryptedData>` element, we are

Listing 7-5

An enveloped signature for `<Encryption-Properties>`

```xml
<SecureDoc>
  <EncryptedData Id="EncData1" xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>
        ErBgcqhkjOOAQBMIIBHgKBgQDaJjfDTrawMHf8MiUt
        Y54b37hSmYNnR3KpGT10uU1Dqppcju06uN0iGbqf94
        7DjkBC25hKnqykK31xBw0E
      </CipherValue>
    </CipherData>
    <EncryptionProperties Id="EncProps">
      <EncryptionProperty Target="EncData1">
        <TokenId>123456789</TokenId>
      </EncryptionProperty>
      <EncryptionProperty Target="EncData1>
        <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
           . . .
          <Reference URI="#EncProps"
            Type="http://www.w3.org/2001/04/xmlenc#EncryptionProperties">
            <DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
          </Reference>
          <SignatureValue>
            RJTDhjgsalhdMlNjrWWKdkgjfhDh7hsddj5BwpjfAuJ6T3ZysdfbvA=
          </SignatureValue>
        </Signature>
      </EncryptionProperty>
    </EncryptionProperties>
  </EncryptedData>
  ...
</SecureDoc>
```

only signing the `<EncryptionProperties>` element. Signing the `<EncryptedData>` element is also very important because XML Encryption doesn't have a mechanism for structural integrity checking (unlike an XML Signature). For example, it is possible for an attacker to arbitrarily change (delete or remove) components of an `<EncryptedData>` element. An example of such an attack is altering the encryption algorithm specified in the `<EncryptionMethod>` element. Doing something like this might cause an otherwise properly encrypted message to fail the decryption step (the algorithm would now be incorrect and the recipient would receive garbage instead of the real plaintext). The reader might argue that this is okay in this case because the attacker hasn't actually received any secret information. Although this is entirely true, such antics leave the recipient unclear of the status of the sender as well as the integrity of all data that might be received. The other issue is the weight of the performance toll. The signature operation is much more expensive than symmetric key

encryption and the signature doesn't authenticate the plaintext; it only authenticates the structure of the `<EncryptedData>` element.

The `<ds:KeyInfo>` Element

The `<ds:KeyInfo>` element is the second element in the `<Encrypted-Data>` parent and is probably one of the most important elements in the XML Encryption syntax. The `<ds:KeyInfo>` element opens the door to complexity for XML Encryption by introducing (among other elements) the `<EncryptedKey>` element. The `<EncryptedKey>` element is another fundamental type of the XML Encryption draft, and once the reader understands how `<ds:KeyInfo>` and `<EncryptedKey>` work, he or she will have nearly mastered the XML Encryption syntax.

Thus far, we have assumed that the retrieval of the decryption key is designated by the application. Although this case is entirely possible, it is more likely that hints or direct pointers to decryption keys will be more usable and common for real-world scenarios. Listing 7-6 repeats the structure of `<ds:KeyInfo>` as it is described in the XML Encryption draft. This listing is provided as a convenience to the reader (this is the same structure that is embedded inside Listing 7-1).

Simply put, the `<ds:KeyInfo>` element describes where to obtain the key to decrypt the contents of the `<CipherData>` element. The basic purpose of `<ds:KeyInfo>` for XML Encryption is analogous to its use for an XML Signature. There is, however, one very important distinction. For a signed object, it is possible (and recommended) to package the verification key inside the `<Signature>` element. This is done because it is assumed that the recipient will first assert trust over the holder of the verification key and then proceed to perform the signature verification. This contrasts with an encrypted object, which must omit the actual decryption key. An encrypted object that is packaged with a decryption key is rather useless for data privacy unless the key itself is somehow protected!

Listing 7-6

`<ds:KeyInfo>` as defined in the XML Encryption draft

```
<ds:KeyInfo>
   <EncryptedKey>?
   <AgreementMethod>?
   <ds:KeyName>?
   <ds:RetrievalMethod>?
   <ds:*>?
</ds:KeyInfo>?
```

The previous discussion highlights the differences between how `<ds:KeyInfo>` is used in XML Encryption versus how an XML Signature is used. For the case of XML Encryption, there are two broad categories of items that belong inside `<ds:KeyInfo>`: pointer information and encrypted keys. When we use the term *pointer information*, we are talking about ways of identifying an end recipient, either by name, digital certificate, or some other means. When we use the term *encrypted keys*, we are talking about the `<EncryptedKey>` element that is defined in the XML Encryption draft. As the name suggests, the `<EncryptedKey>` element is an element designed to package encrypted keys. We have been very good so far about starting with simple examples, and this case will be no different. First we will look at some examples of possible pointer information that might be stored inside `<ds:KeyInfo>`, and then from there we will proceed into a discussion of the `<EncryptedKey>` element.

Pointer Information for <ds:KeyInfo>

The `<KeyName>` element is the most basic type of pointer information. In vacuous cases, it may hold the name of the recipient, but not necessarily. The `<KeyName>` element represents a hint of sorts and is meant to identify the ostensible holder of the decryption key (whether it is symmetric or asymmetric). Other types of pointer information include `<ds:RetrievalMethod>` and the mysterious `<ds:*>` element. The `<ds:RetrievalMethod>` element is the same element used in the XML Signature syntax, but it has been extended to handle the `<EncryptedKey>` element. That is, one can designate the location of an `<EncryptedKey>` element remotely by setting the `Type` attribute (an attribute of `<ds:RetrievalMethod>`) to `http://www.w3.org/2001/04/xmlenc#EncryptedKey`. Even though the `<ds:RetrievalMethod>` is a proper pointer, we will wait and show an example of its use when we discuss the `<EncryptedKey>` element in full detail in the next section. Aside from `<KeyName>`, the XML Encryption draft also specifies the `<ds:*>` element. This notation simply means that any child element defined previously in the XML Signature Recommendation can be used in an instance of `<ds:KeyInfo>`. The reader may be wondering about this statement and may question the use of XML Signature `<ds:KeyInfo>` child elements for an `<EncryptedData>` element. The most useful additional element defined by the `<ds:KeyInfo>` element is the `<ds:X509Data>` element. The list of possible child elements for `<ds:X509Data>` as specified in the XML Signature Recommendation is repeated in Table 7-2.

Element Name	Description
`<X509IssuerSerial>`	X.500 issuer name and serial number
`<X509SKI>`	X.509 subject key identifier
`<X509SubjectName>`	X.500 subject name
`<X509Certificate>`	X.509 certificate
`<X509CRL>`	X.509 CRL

The careful reader will notice that each of the child elements of `<X509Data>` serves to identify a public verification key. As a side effect, however, all of these child elements (except for `<X509CRL>`) also uniquely identify an end-entity. This fact is significant because although a simple string value (as is used in `<KeyName>`) may be used, it can be an ambiguous way of identifying an individual.

In some cases, the element directly identifies an end-entity and in other cases, the individual is indirectly identified. For example, in the case of `<X509Certificate>` and `<X509SubjectName>`, we are identifying the recipient in a very strong manner. The distinguished name that is used in an X.509 certificate is intended to be unique by design. In this case, we can either get the entire certificate (which contains the subject name) or the subject name itself (the `<X509SubjectName>` element). The distinguished name or X.509 certificate itself cannot help us because it doesn't directly contain the decryption key. Instead, it might be useful to look up the proper decryption key based on this nonambiguous information.

This contrasts with other child elements such as `<X509Issuer-Serial>` and `<X509SKI>`. For this case, the certificate must be first retrieved based on the issuer name and serial number or the subject key identifier. These elements only implicitly identify a given certificate; some sort of database lookup or retrieval must occur to find the name of the end-entity. Figure 7-3 shows a pictorial view of some of the different types of objects that `<ds:KeyInfo>` can point to for an encrypted data element.

In Figure 7-3(a), the vanilla case of a simple string name is shown. Remember, we are attempting to identify a particular decryption key, not a verification key (as is the case with an XML Signature). In Figure 7-3(b), the `<ds:KeyInfo>` element points to an X.509 certificate (which maps to someone who has the private key). Finally, in Figure 7-3(c), the `<ds:Key-Info>` element points to a X.509 `SubjectKeyIdentifier`. In this case, some additional mapping must be done by the application to first locate

Figure 7-3

Some choices for the `<KeyInfo>` element

(a)

```
<EncryptedData>
…

   <KeyInfo>

   </KeyInfo>
…
</EncryptedData>
```

→ **John Q. Recipient**

(b)

```
<EncryptedData>
…

   <KeyInfo>

   </KeyInfo>
…
</EncryptedData>
```

→ **X.509 Certificate**

(c)

```
<EncryptedData>
…

   <KeyInfo>

   </KeyInfo>
…
</EncryptedData>
```

→
```
50F1 861B C105
D8B7 15F4 BF25
99CC 20C2 1942
2C1D
```
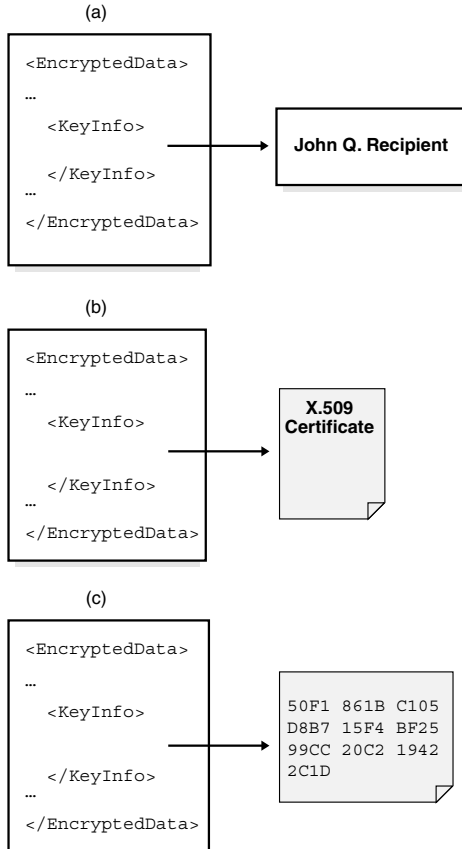
the identity of the certificate to which the SubjectKeyIdentifier points. The idea with the SubjectPublicKeyIdentifier is to identify a given public key, which will eventually map to an X.509 certificate. This type of identifier is a less-direct way of specifying an X.509 certificate.

Encrypted Keys and the `<EncryptedType>` Element

A likely scenario for the `<ds:KeyInfo>` element is the inclusion of an *encrypted decryption key*. This seems a bit odd a first because we are adding another recursive step. That is, we have an `<EncryptedData>`

element that points to or contains sensitive cipher data to be decrypted. Further, the actual decryption key is specified as being encrypted (which means it must also be decrypted). At this point, one might argue that the distinction between a decryption key and the actual encrypted data begins to blur. After all, an encryption key (for the case of symmetric encryption) is simply a string of bits. This means that one could think of the encrypted key as simply another `<EncryptedData>` element. Why does it really matter? The content of the ciphertext is simply binary data! Any string of bits that is the appropriate size can serve as a usable symmetric key for most symmetric encryption algorithms. Because of this fact, the `<EncryptedKey>` element matches the structure of `<EncryptedData>` exactly (with the exception of a few additional attributes and an additional element). The reader should think of these two elements as being functionally equivalent in what they represent (some encrypted data), but one marks this encrypted data as a key and the other designates the encrypted data as arbitrary.

The realization of this trait of symmetric encryption keys yields the `<EncryptedType>` element, which is an abstract type defined in the XML Encryption schema and provides the basic behavior for both `<Encrypted-Data>` and `<EncryptedKey>`. A picture of how `<EncryptedData>`, `<EncryptedKey>`, and `<EncryptedType>` are related is shown in Figure 7-4.

The object-oriented concept of inheritance is a loose analog to the way Figure 7-4 is arranged. The reader will never see an `<EncryptedType>` element in the wild because it is an abstract parent type that dictates its behavior to those elements that borrow from it. Figure 7-4 shows the fictional shorthand notation for `<EncryptedType>`. This is fictional notation because an actual instance of `<EncryptedType>` will never be created—only elements that derive from it such as `<EncryptedData>` and `<EncryptedKey>` will be created. The reader should notice that `<EncryptedData>` has the exact same structure as `<EncryptedType>`. This contrasts with `<EncryptedKey>`, which adds an attribute value called `Recipient` as well as two elements: `<CarriedKeyName>` and `<ReferenceList>`. These additional data items are shown in bold in Figure 7-4.

Another property of `<EncryptedType>` is the possibility for *recursive nesting*. This has been alluded to before and occurs when a given `<EncryptedData>` element contains an `<EncryptedKey>` element (this is inside `<ds:KeyInfo>`). At this point, we have essentially nested `<EncryptedType>` inside of itself a single time. The chain doesn't have to

Figure 7-4

The abstract
<Encrypted-
Type> element

```
<EncryptedType Id? Type?>
  <EncryptionMethod/>?
  <ds:KeyInfo>?
  <CipherData>
  <EncryptionProperties/>?
</EncryptedType>
```

```
<EncryptedData Id? Type?>
  <EncryptionMethod/>?
  <ds:KeyInfo>?
  <CipherData>
  <EncryptionProperties/>?
</EncryptedData>
```

```
<EncryptedKey Id? Type? Recipient?>
  <EncryptionMethod/>?
  <ds:KeyInfo>?
  <CipherData>
  <EncryptionProperties/>?
  <ReferenceList>?
  <CarriedKeyName>?
</EncryptedType>
```
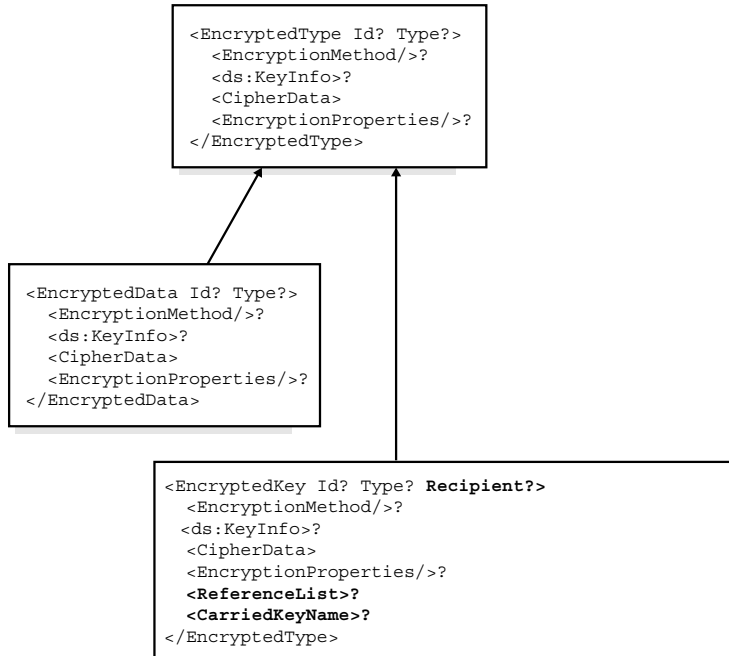
stop here and can continue. It is possible for a given <EncryptedKey> element to be encrypted by another <EncryptedKey> element (the most common case of this a digital envelope). There are two common cases where recursive nesting will appear. These cases are shown in Figure 7-5.

At first glance, Figure 7-5 may seem a bit abstract. The trouble is that many details have been omitted and some assumptions must be made. For example, consider Figure 7-5(a). In this case, we have an <Encrypted-Data> element whose <ds:KeyInfo> points to an <EncryptedKey>. This <EncryptedKey> element is the end of the chain on the recursion because it doesn't point to anything else. The unstated assumption for this example is that the <ds:KeyInfo> element contains some sort of tangible identifier for the decryption key (instead of a reference to yet another <EncryptedKey> element). This case represents a digital envelope where an asymmetric key represents the end of the chain and is responsible for encrypting a symmetric key, which then actually encrypts the original plaintext. The concept of a digital envelope is not new and is known as the

Figure 7-5

Recursive containment for the `<EncryptedKey>` element
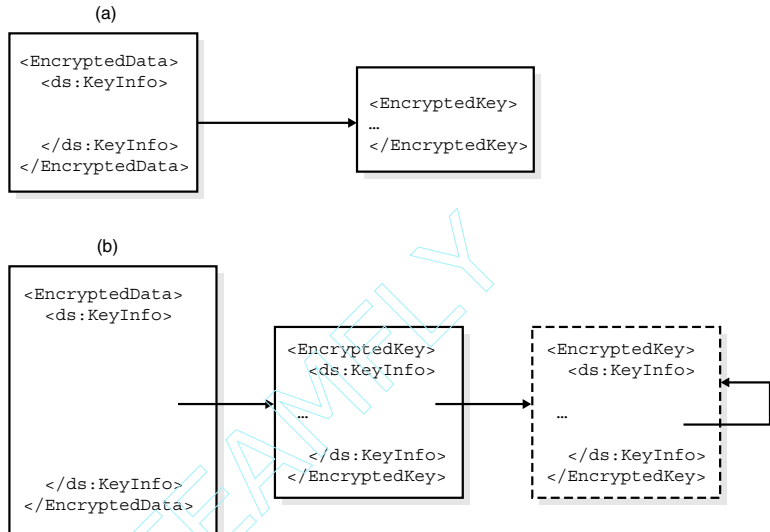
equally obscure term *key transport*. More information on digital envelopes and why they are useful for data privacy is provided in Chapter 2. One thing to note is that in most cases where a digital envelope is used, the asymmetric decryption key is also encrypted (usually with a password-based scheme). The password-based encryption scheme is convenient from a usability standpoint for human users.

Finally, Figure 7-5(b) shows the last case of *symmetric key wrap*, whereby one symmetric key is encrypted by another symmetric key (which is also encrypted and so on). This chain can be arbitrarily long, but it will usually be fairly short in the interest of usability. In general, the most important thing that the reader should take away from Figure 7-5 is the fact that `<EncryptedKey>` can be nested inside another `<EncryptedKey>` element.

At this point, it is best to proceed with some examples. It is challenging to see how everything fits together without something to look at. Listing 7-7 is meant to simply mirror the structure shown in Figure 7-5(a).

Listing 7-7 shows some cipher data that has been encrypted with AES using a 128-bit key. In this example, the media type is text/html, so the assumption is that we are encrypting an HTML document. The encryption key used is encrypted (this is represented by the `<EncryptedKey>`

Listing 7-7

The <Encrypted-Data> element with an <EncryptedKey> (digital envelope)

```
<EncryptedData
  xmlns="http://www.w3.org/2001/04/xmlenc#"
  Type="http://www.isi.edu/in-notes/iana/assignments/media-types/text/html">
  <EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <EncryptedKey>
      <EncryptionMethod
       Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:KeyName> John Q. Recipient </ds:KeyName>
      </ds:KeyInfo>
      <CipherData>
        <CipherValue>
          zc1msFhM1GKMYDgYQAAoGActA8YG
        </CipherValue>
      </CipherData>
    </EncryptedKey>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>
      NFNNJoqMcT2ZfCPrfvYvQ2jRzBFMB4GA1UdEQQXMBWBE2
    </CipherValue>
  </CipherData>
</EncryptedData>
```

structure, which is shown in bold). This symmetric key has been encrypted using the RSA algorithm with Public Key Cryptography Standards (PKCS) #1 padding. Moreover, the specific RSA key used to perform the encryption was John Q. Recipient's public key. The only person who should be able to decrypt the AES key and read the message is John Q. Recipient. The reader should understand that the enforcement of uniqueness for the <KeyName> element is implementation dependent and is not part of the XML Encryption draft. Practical scenarios suggest that a key identifier that is more unique than the string John Q. Recipient will be used.

The reader should notice that we haven't used any of the new features of <EncryptedKey> (such as the <ReferenceList> element) yet. We'll get to these elements in a later section. The most important thing to notice about Listing 7-7 is the nesting of <EncryptedKey> inside <EncryptedData>.

A case similar to Figure 7-5(a) is shown in Listing 7-8 and is a symmetric key wrap. This corresponds to Figure 7-5(b).

Listing 7-8 is structurally identical to Listing 7-7. There are only two significant differences. The first is the fact that we are encrypting a file

Listing 7-8

The
`<Encrypted-
Data>` element
with an
`<EncryptedKey>`
(symmetric key
wrap)

```
<EncryptedData
  xmlns="http://www.w3.org/2001/04/xmlenc#"
  Type="http://www.isi.edu/in-notes/iana/assignments/media-types/image/gif">
  <EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <EncryptedKey>
      <EncryptionMethod
       Algorithm="http://www.w3.org/2001/04/xmlenc#kw-tripledes"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:KeyName> John Q. Recipient </ds:KeyName>
      </ds:KeyInfo>
      <CipherData>
        <CipherValue>
          49dGMTPKg/JJjvqu+wWkYbaQ39G
        </CipherValue>
      </CipherData>
    </EncryptedKey>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>
      ZIhvcNAQkBFhNzb2NyYXRlc0BhdGhlbnMuY29tMB4XDTAxMD
    </CipherValue>
  </CipherData>
</EncryptedData>
```

that has a media type of `image/gif` instead of `image/html` (as is done in Listing 7-8). Secondly, we specify a very different encryption algorithm for the encryption key. That is, we are using the value `http://www.w3.org/2001/04/xmlenc#kw-tripledes` instead of specifying an asymmetric encryption algorithm like RSA. Thirdly, for variety we have chosen the Triple-DES encryption algorithm instead of AES. This is denoted by the `<EncryptionMethod>` element whose `Algorithm` attribute value is `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`. To summarize Listing 7-8, one might say that we have encrypted some data (a gif file) with a symmetric encryption key using the Triple-DES algorithm. This key has also been encrypted, using a key-wrapping algorithm that includes Triple-DES as the chief encryption mechanism. The concept of symmetric key wrapping can be a bit odd at times. The reader may be thinking, Why bother encrypting a symmetric key with another symmetric key? The act itself seems a bit redundant. There are, however, some good reasons for doing something like this. For example, if a lot of effort was put into generating a given symmetric key (as with a key exchange algorithm such as Diffie-Hellman), it might be useful to encrypt this with another symmetric key.

Another case is a situation where a set of keys is encrypted with one master key (this situation might arise in some sort of database context). To unlock a given field or record, the master key must be obtained; this master key is simply a symmetric encryption key that wraps other symmetric keys. This example assumes that each field or record is encrypted with a different symmetric key. Listing 7-8 only shows one level of recursive nesting; the reader should understand, however, that more levels are permitted by the XML Encryption draft, although their use will be quite seldom.

The <EncryptedKey> Element: Details

The previous examples (Listings 7-7 and 7-8) showcase some common uses of the <EncryptedKey> element. There are, however, some extra details that must be mentioned about <EncryptedKey>, which provide some additional semantics. The structure of the <EncryptedKey> element is repeated in Listing 7-9.

The goal of this section is to teach the reader about the additional `Recipient` attribute as well as the <CarriedKeyName> and <ReferenceList> elements.

<ReferenceList> Details

So far we have seen a general paradigm in terms of how keying information is associated with an XML structure (either an XML Signature or encrypted XML element). That is, the assumption has been that the given XML structure envelops or references (for encryption) the keying material. In addition, we have also assumed that a key is tightly wed with a given structure. That is, we only talk about a <ds:KeyInfo> as a component element and not as a first-class object.

The careful reader may have noticed that we can turn this association around and begin to think of the decryption key as a first-class object and

---

Listing 7-9

<EncryptedKey> element details

```
<EncryptedKey Id? Type? Recipient?>
  <EncryptionMethod/>?
  <ds:KeyInfo>?
  <CipherData>
  <EncryptionProperties/>?
  <ReferenceList>?
  <CarriedKeyName>?
</EncryptedType>
```

the data it encrypts is the item being referenced. The reason why this is possible is because the `<EncryptedKey>` element inherits from `<EncryptedType>`, which is the parent object of XML Encryption. This means that an `<EncryptedType>` can be the root of its own XML document and reference the data (or keys) that have been encrypted by it. This reverse referencing is done with the `<ReferenceList>` element. A `<ReferenceList>` points to data items that have been encrypted with a given `<EncryptedKey>`. Furthermore, the `<ReferenceList>` element might also employ transforms, if, for example, the actual data or key encrypted is part of some larger XML document (or stored in a compressed or encoded form). The `<ReferenceList>` element is simple; its structure is given in the following example:

```
<ReferenceList>
  <DataReference URI?>*
  <KeyReference URI?>*
</ReferenceList>
```

There are only two elements that can be inside a `<ReferenceList>` element: `<DataReference>`, which identifies some encrypted data, and `<KeyReference>`, which identifies some encrypted key. In both cases, the data item is specified with the use of a URI. The URI can be used in the same way as it is used in an XML Signature and can specify either an absolute URI or a bare name fragment identifier. Listing 7-10 shows an example of how `<EncryptedKey>` references the data that it has encrypted when both `<EncryptedKey>` and `<EncryptedData>` reside in the same XML document.

Listing 7-10 shows an XML document with a parent element called `<SecureDoc>`. Inside this document are two elements from the XML Encryption syntax. The reader should assume that other arbitrary elements also exist in `<SecureDoc>`, but they are not shown for brevity. The `<EncryptedKey>` element represents some sort of decryption key that has been encrypted with the RSA algorithm for key transport. This `<EncryptedKey>` contains a `<ReferenceList>` element with a URI attribute value of #EncryptedDataItem1. This bare name fragment identifier has the same semantics as a URI attribute that is used in a `<ds:Reference>` element. That is, it refers to the element whose Id attribute value is EncryptedDataItem1.

Listing 7-10 shows a one-way association between the `<Encrypted-Key>` element and the `<EncryptedData>` element. That is, if one was to stumble across only the `<EncryptedData>` element first, it would be difficult to determine the appropriate decryption key. However, if we begin

Listing 7-10

An example of using
`<Reference-List>`

```
<SecureDoc>
...
  <EncryptedKey>
    <EncryptionMethod
     Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1 5"/>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:KeyName> John Q. Recipient </ds:KeyName>
    </ds:KeyInfo>
    <CipherData>
      <CipherValue>
        mPCadVfOMx1NzhDaKMHNgFkR9upTW4kgBxyPWjFdW
        UhiE4uQpww+t681UIuZ9y5QVhRlEdfZ5H4Ytza2v8
        anvv6YwVwBhjHU3vSm49FgZp
      </CipherValue>
    </CipherData>
    <ReferenceList>
    <DataReference URI="#EncryptedDataItem1"/>
    </ReferenceList>
  </EncryptedKey>

   . . .

  <EncryptedData Id="EncryptedDataItem1">
    <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <CipherData>
      <CipherValue>
        7KZ5l1KMKp54PyQNAkE9iQofYhyOfiHZ29kkEFV
        J30CAwEAAaMSMBAwDgYDVR0PAQH/BAQDAgSQMA0
        GCSqGSIb3DQEBBQUAA4GBACSzF
      </CipherValue>
    </CipherData>
  </EncryptedData>

</SecureDoc>
```

with the `<EncryptedKey>` element, we can refer to the `<Reference-List>` element to determine where an encrypted data item resides. This will be either directly inside the `<EncryptedData>` element with a `<CipherValue>` child element (shown in Listing 7-10) or indirectly with a `<CipherReference>` element. The reader should think of a `<ReferenceList>` as an arrow (or set of arrows if multiple data items are included) to other `<EncryptedData>` or `<EncryptedKey>` elements. The conceptual picture of this is shown in Figure 7-6.

Figure 7-6 simply shows an `<EncryptedKey>` element that points to two pieces of encrypted data and one encrypted key, indicating that those items can be decrypted with the initial `<EncryptedKey>`. At this point, the reader may be wondering about the reverse association. That is, it might be useful to identify the particular `<EncryptedKey>` structure associated with a given `<EncryptedData>` element (instead of just the

Figure 7-6

The conceptual
view of
`<Reference-
List>`
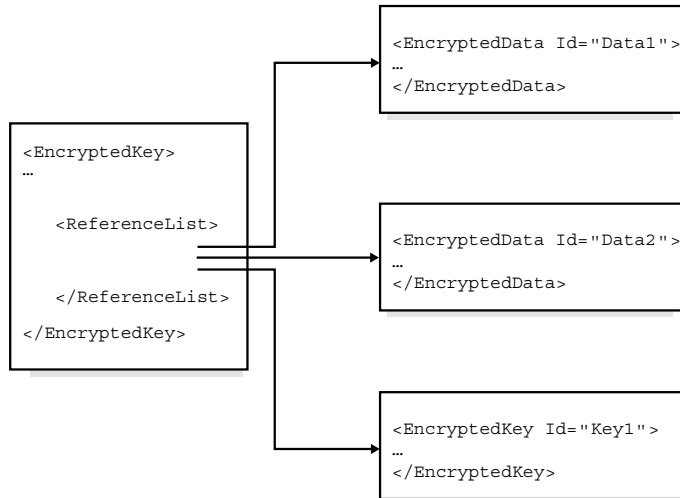
identity of the private key holder). This can be confusing at first because
we mentioned previously that discerning a decryption key is the role of
`<ds:KeyInfo>`. Although this is entirely correct, in the case where we
want direct access to the actual `<EncryptedKey>` element, the XML
Encryption draft has us rely on the `<CarriedKeyName>` element. We are
in essence drawing an arrow back from the various encrypted data items
(keys or otherwise) directly to the proper `<EncryptedKey>` structure.
Although the `<ReferenceList>` element shown in Figure 7-6 points to
two `<EncryptedData>` elements and one `<EncryptedKey>` element, the
schema definition for `<ReferenceList>` constrains its contents to choice
but not both.

The `<CarriedKeyName>` Element

The `<CarriedKeyName>` element is meant to identify an `<Encrypted-
Key>` structure and not necessarily the identity of a recipient. It is the
name carried by a given `<EncryptedKey>` structure. This means that it
is used as an identifier and is referenced from another `<Encrypted-
Data>` element.

The introduction of the `<CarriedKeyName>` element also brings about
another semantic for `<ds:KeyName>`. Thus far, we have assumed that
`<ds:KeyName>` is a mechanism to identify some holder of a verification

key. Although it is possible to use this element for such a task, it is also used to refer to the value given to <CarriedKeyName>, which isn't necessarily the name of a recipient. An example of this is shown in Listing 7-11, which is simply a repeat of Listing 7-10 but with <CarriedKeyName> and <ds:KeyName>.

Listing 7-11 adds a <CarriedKeyName> element with a value of MasterEncryptionKey to <EncryptedKey> as well as a <ds:Key-Name> element to <EncryptedData>. This means that an association can be made from the <EncryptedData> element to the <EncryptedKey> using the value MasterEncryptionKey.

Listing 7-11

The addition of <CarriedKey-Name> and <ds:KeyName> to Listing 7-10

```
<SecureDoc>
...
  <EncryptedKey Id="Key1">
  <EncryptionMethod
     Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1 5"/>
     <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
       <ds:KeyName> John Q. Recipient </ds:KeyName>
     </ds:KeyInfo>
     <CipherData>
       <CipherValue>
         mPCadVfOMx1NzhDaKMHNgFkR9upTW4kgBxyPWjFdW
         UhiE4uQpww+t68lUIuZ9y5QVhRlEdfZ5H4Ytza2v8
         anvv6YwVwBhjHU3vSm49FgZp
       </CipherValue>
     </CipherData>
     <ReferenceList>
       <DataReference URI="#EncryptedDataItem1"/>
     </ReferenceList>
     <CarriedKeyName> MasterEncryptionKey </CarriedKeyName>
  </EncryptedKey>

   . . .

  <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
   Id="EncryptedDataItem1">
    <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <ds:KeyInfo>
      <KeyName> MasterEncryptionKey </KeyName>
    </ds:KeyInfo>
    <CipherData>
      <CipherValue>
        7KZ5l1KMKp54PyQNAkE9iQofYhyOfiHZ29kkEFV
        J30CAwEAAaMSMBAwDgYDVR0PAQH/BAQDAgSQMA0
        GCSqGSIb3DQEBBQUAA4GBACSzF
      </CipherValue>
    </CipherData>
  </EncryptedData>
</SecureDoc>
```

Another thing to notice about the use of `<ds:KeyName>` and `<CarriedKeyName>` is that the association between `<EncryptedData>` and `<EncryptedKey>` isn't direct. That is, if a processing application were to begin with `<EncryptedData>` and then proceed to discern the key, it would be an application task to actually find the key given only the value of `<CarriedKeyName>`. To help with this problem, the `<ds:RetrievalMethod>` element can be used.

The <ds:RetrievalMethod> Element

The `<ds:RetrievalMethod>` element is borrowed from the XML Signature Recommendation. In the context of an XML Signature, the `<ds:RetrievalMethod>` element is designed to associate remote verification key information with a given `<ds:KeyInfo>` element. The role of `<ds:RetrievalMethod>` is similar for the case of XML Encryption. Instead of pointing to a signature verification key, the `<ds:RetrievalMethod>` points to a decryption key and, more specifically, an `<EncryptedKey>` element. The reader can think of `<ds:RetrievalMethod>` as a way of providing a functional link to a given `<EncryptedKey>` element. To illustrate this point, it is useful to repeat the `<EncryptedData>` structure as shown in Listing 7-11 with the added `<ds:RetrievalMethod>` element, which is shown in Listing 7-12.

There are two things to notice about `<ds:RetrievalMethod>` as it is shown in Listing 7-12. First of all, it uses a bare name fragment identifier

Listing 7-12

Providing a functional link to `<EncryptedKey>`

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
 Id="EncryptedDataItem1">
 <EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
 <ds:KeyInfo>
   <ds:KeyName> MasterEncryptionKey </KeyName>
   <ds:RetrievalMethod URI="#Key1"
    Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
 </ds:KeyInfo>
 <CipherData>
   <CipherValue>
     7KZ5l1KMKp54PyQNAkE9iQofYhyOfiHZ29kkEFV
     J30CAwEAAaMSMBAwDgYDVR0PAQH/BAQDAgSQMA0
     GCSqGSIb3DQEBBQUAA4GBACSzF
   </CipherValue>
 </CipherData>
</EncryptedData>
```

to link to an `<EncryptedKey>` element with an `Id` value of `Key1` (this is the same `<EncryptedKey>` element that is shown in Listing 7-11). Further, the `<ds:RetrievalMethod>` element qualifies what it is linking to with the use of the `Type` attribute. The value used, `http://www.w3 .org/2001/04/xmlenc#EncryptedKey`, is fixed for `<ds:Retrieval- Method>` as used in the XML Encryption draft. Multiple instances of `<ds:RetrievalMethod>` can appear within a given `<ds:KeyInfo>` element, but if they do, they must all point to the same key. The reader can think of three conceptual links between the `<EncryptedData>` element and the `<EncryptedKey>` element. First, there is a link from the `<EncryptedKey>` element by virtue of the `<ReferenceList>` element. Second, there is a link from the `<EncryptedData>` element to the `<EncryptedKey>` with the use of `<ds:KeyName>` and `<CarriedKey- Name>`. Finally, there is a functional link from `<EncryptedData>` to `<EncryptedKey>` with the use of the `<ds:RetrievalMethod>` and its bare name fragment identifier.

The Recipient Attribute The final addition to `<EncryptedKey>` is the `Recipient` attribute. This attribute is used to offer an application-dependent hint for the recipient of the `<EncryptedKey>` in which it appears. Some people may argue that we already have a vehicle for denoting the recipient of an encrypted key; for example, the `<ds:KeyName>` attribute does this quite easily. The reason why the `Recipient` attribute might be required in some situations comes from the fact that the `<ds:KeyName>` (if used) may contain the name or identifier of a key, rather than a recipient. This case is more common when a single key is encrypted for multiple recipients. As an example, consider Listing 7-13, which is based on Listing 7-11.

Listing 7-13 includes the `Recipient` attribute with the value of `John Q. Recipient` as well as a second `<EncryptedKey>` element with the

Listing 7-13

Using the Recipient attribute in `<EncryptedKey>`

```
<SecureDoc>
...
  <EncryptedKey Id="Key1"
                Recipient="John Q. Recipient">
  <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1 5"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:KeyName> MasterEncryptionKey.bin </ds:KeyName>
      </ds:KeyInfo>
      <CipherData>
```

(continues)

```xml
        <CipherValue>
          mPCadVfOMx1NzhDaKMHNgFkR9upTW4kgBxyPWjFdW
          UhiE4uQpww+t68lUIuZ9y5QVhRlEdfZ5H4Ytza2v8
          anvv6YwVwBhjHU3vSm49FgZp
        </CipherValue>
      </CipherData>
      <ReferenceList>
        <DataReference URI="#EncryptedDataItem1"/>
      </ReferenceList>
      <CarriedKeyName> MasterEncryptionKey </CarriedKeyName>
  </EncryptedKey>

  <EncryptedKey Id="Key2"
                Recipient="Sue B. Recipient">
  <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:KeyName> MasterEncryptionKey.bin </ds:KeyName>
      </ds:KeyInfo>
      <CipherData>
        <CipherValue>
          FRlY2hub2xvZ2llcywgTHRkKupTW4kgBxyPWjFdW
          AkGA1UEBhMCSUUxDzANBgNVJhRlEdfZ5H4Ytza2v8
          KFwoOqgn5aKIkICGMlv6SgAH
        </CipherValue>
      </CipherData>
      <ReferenceList>
        <DataReference URI="#EncryptedDataItem1"/>
      </ReferenceList>
      <CarriedKeyName> MasterEncryptionKey </CarriedKeyName>
  </EncryptedKey>
   . . .

  <EncryptedData Id="EncryptedDataItem1"
   xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <ds:KeyInfo>
      <ds:KeyName> MasterEncryptionKey </KeyName>
      <ds:RetrievalMethod URI="#Key1"
      Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
      <ds:RetrievalMethod URI="#Key2"
      Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
    </ds:KeyInfo>
    <CipherData>
      <CipherValue>
        7KZ5l1KMKp54PyQNAkE9iQofYhyOfiHZ29kkEFV
        J30CAwEAAaMSMBAwDgYDVR0PAQH/BAQDAgSQMA0
        GCSqGSIb3DQEBBQUAA4GBACSzF
      </CipherValue>
    </CipherData>
  </EncryptedData>
   . . .
</SecureDoc>
```

value of `Sue B. Recipient`. The `<ds:KeyName>` no longer identifies the identity of the recipient, but identifies the name of the actual key instead. In Listing 7-13, we have some encrypted data that has been encrypted for multiple recipients using their RSA public keys. Each recipient has a different encrypted form of `MasterEncryptionKey`, but the reader should notice that `<CarriedKeyName>` has the same element value for both encryption keys. This would not be possible if a reference was made to the `Id` attribute of `<EncryptedKey>` because `Id` attributes must be unique in a given XML document context. Finally, the reader should notice that we have added another `<ds:RetrievalMethod>` element. This second element points to the second encrypted key with an `Id` value of `Key2`. This signifies that the `<EncryptedData>` element in Listing 7-13 has been encrypted with the same key, but is for different recipients. That is, one could decrypt the `<EncryptedData>` with either `Key1` or `Key2`.

The <*AgreementMethod*> Element

At this point, we have exhausted most of the details of the `<Encrypted-Key>` element. The last element to be covered is the `<AgreementMethod>` element, which is the second element in `<ds:KeyInfo>`. The `<AgreementMethod>` element is used to support the agreement to a shared secret value for the purposes of data encryption.

The XML Encryption draft only supports asynchronous key agreement. This means that the sender and recipient participating in the key agreement will generate their keys at different times. This contrasts with the key agreement that might be used in a communications protocol, where it is generally expected that a key will be negotiated by both parties simultaneously and then used for encryption.

In XML Encryption, a sender must first retrieve the recipient's public value and compute a shared secret. This shared secret is then used to encrypt data as part of an `<EncryptedData>` element. It is expected at this point that the sender will include his or her public value so that the recipient can compute the shared secret and decrypt the message. This process can be a bit confusing and is normally divided into two phases or parts. The first phase or part is shown in Figure 7-7.

In Figure 7-7, a recipient begins by generating some sort of key agreement material. Once this material is finally generated, it is stored in a central repository. This can be a certificate database or some sort of key

Figure 7-7

Asynchronous
key agreement
(recipient)

Recipient

Key Agreement
Material Generation

Central Repository
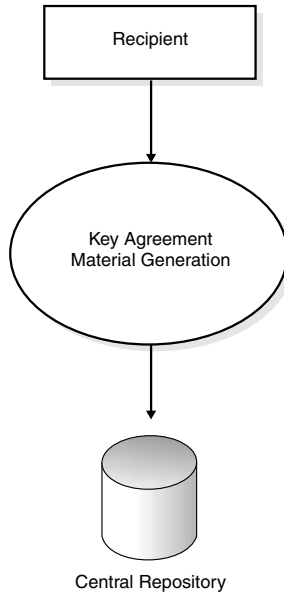
directory. Once the recipient generates the proper key agreement material, the sender can do the appropriate lookup and create an encrypted message. This phase is shown in Figure 7-8.

In Figure 7-8, the sender wants to encrypt data for the recipient using a shared secret. To do this, the central repository must be queried to obtain the recipient key agreement material (public value). Once obtained, the sender performs the shared secret generation and proceeds to encrypt data and send it to the recipient. The only detail that must be considered is the presence of the sender's public values. The recipient must have the sender's key agreement material (public value) in order to generate the shared secret and decrypt the information. The sender's key agreement material is communicated with the `<OriginatorKeyInfo>` element, which is shown in bold in Figure 7-8. The `<AgreementMethod>` element is considered a proper child of `<ds:KeyInfo>` for XML Encryption because it eventually produces a key value. In addition to the `<OriginatorKeyInfo>` element, the `<AgreementMethod>` element also communicates an optional nonce value and digest method that is

Figure 7-8

Asynchronous
key agreement
(sender)


```
<EncryptedData>
  …
  <ds:KeyInfo>
    <AgreementMethod>
      <OriginatorKeyInfo>
      …
      </OriginatreKeyInfo>
    </AgreementMethod>
  </ds:KeyInfo>
</EncryptedData>
```
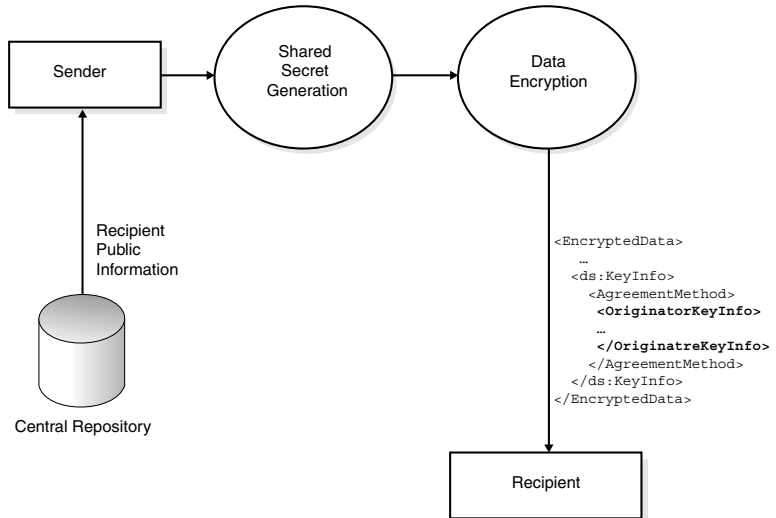
used for the actual key derivation. For example, the XML Encryption draft supports a key derivation function that produces any number of key bytes based on the shared secret value, nonce, key size, and string URI identifier.

Plaintext Replacement

In every example so far, we have talked about encrypting arbitrary data. This can be an entire XML document, an HTML file, or some binary format file or resource. Now it is time to transition to the case of plaintext replacement. We began this chapter with an overview of plaintext replacement, which is illustrated in Figure 7-2 (the reader should flip back now and refresh themselves). It is useful to note that almost nothing changes in terms of the syntax for the case of plaintext replacement. The only difference is that the `<EncryptedData>` element replaces the original XML plaintext (element or element content), effectively creating a new XML document. One syntax change that is necessary is the designation of the type of XML structure being replaced. This is going to be either

an element or element content (attribute encryption is not currently possible). The identifiers that will be used in the `Type` attribute for the `<EncryptedData>` element are given in Table 7-1. The behavior of the plaintext replacement case is very similar to the examples given previously; the only thing that changes is the addition of `<EncryptedData>` to form the new ciphertext. Consider Listing 7-14, which shows a bank transfer authored in a fictional markup language that uses the syntax of XML.

In Listing 7-14, there are disparate elements and content. Not all of the data items enumerated in Listing 7-14 can be considered especially sensitive. For example, the client name and the transaction ID are probably not very sensitive. Contrast this with the account numbers or transfer amount. This type of information might be considered sensitive. Furthermore, it may not be practical to encrypt the entire structure. For example, some application-specific processing or routing may need to take place that is based on the `<TransferTime>` or the `<TransactionID>` element. In order to selectively encrypt on the sensitive information, we can rely on plaintext replacement. The updated, encrypted structure is shown in Listing 7-15.

In Listing 7-15, we have actually encrypted a set of elements, namely the content of the `<SensitiveInformation>` element. This means that `<SourceAccountNumber>`, `<DestinationAccountNumber>`, and `<TransferAmount>` have all been encrypted using Triple-DES in CBC mode. Furthermore, the key has been given a `<ds:KeyName>` called `TransactionKey`. It is assumed that the recipient who is receiving the message knows how to obtain the proper decryption key based on this identifier. Another equally likely scenario is embedding an `<Encrypted-`

Listing 7-14

A fictional markup language for a banking transaction

```
<Transfer>
  <TransactionId> 548356 </TransactionId>
  <ClientName> Dale Reed </ClientName>
  <ClientSSN> 123-45-6789 </ClientSSN>
  <TransferTime> 08/13/01 11:03:36.23 </TransferTime>
  <SensitiveInformation>
    <SourceAccountNumber> 123456789 </SourceAccountNumber>
    <DestinationAccountNumber> 987654321 </DestinationAccountNumber>
    <TransferAmount> 500.00 </TransferAmount>
  </SensitiveInformation>
  <CurrencyType> USD </CurrencyType>
</Transfer>
```

```
<Transfer>
  <TransactionId> 548356 </TransactionId>
  <ClientName> Dale Reed </ClientName>
  <ClientSSN> 123-45-6789 </ClientSSN>
  <TransferTime> 08/13/01 11:03:36.23 </TransferTime>
  <SensitiveInformation>
    <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
     Type="http://www.w3.org/2001/04/xmlenc#Content"/>
    <EncryptionMethod
     Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
    <ds:KeyInfo>
      <ds:KeyName> TransactionKey </KeyName>
    </ds:KeyInfo>
    <CipherData>
      <CipherValue>
        A1UEChMcQmFsdGltb3JlIFRlY2hub
        2xvZ2llcywgwTHRkLjEWMBQGA1UEAx
        MNTWVybGluIEh1Z2hlczCCAbYwggE
        rBgcqhkjOOAQBMIIBHgKBgQDaJjfDT
        rawMHf8MiUt
      </CipherValue>
    </CipherData>
    </EncryptedData>
  </SensitiveInformation>
<CurrencyType> USD </CurrencyType>
</Transfer>
```

Key> structure inside the <Transfer> parent element if we desire to transport the decryption key along with the message.

The other thing to notice in Listing 7-15 is the designation of type information for <EncryptedData>. That is, we have given the Type attribute a value of http://www.w3.org/2001/04/xmlenc#Content. This may seem odd because we are encrypting elements and one might think that the http://www.w3.org/2001/04/xmlenc#Element identifier is more appropriate. We use the Content identifier because we are really encrypting the contents of the <SensitiveData> element. The Element designation is used when we are encrypting a single element.

XML Encryption Processing Rules

At this point, we have covered the majority of the useful syntax for XML Encryption and now it is time to transition into a discussion of the XML Encryption processing rules. For XML Signatures, the processing rules comprise a great deal of discussion and make up the bulk of Chapter 5.

For XML Encryption, the processing rules are a bit simpler and more straightforward. The XML Encryption draft specifies three entities for describing the processing rules: the application, the encryptor, and the decryptor.

The Application

The term *application* refers to any sort of entity that relies on a given XML Encryption implementation. This distinction is necessary because some actions are left out of scope for XML Encryption. Certain things such as plaintext validation (after decryption has been performed) and serialization (the conversion of XML to octets) are left to the application.

The Encryptor

The term *encryptor* has a very specific meaning in terms of XML Encryption. In short, it is a processing entity that performs the actual encryption operation. The encryptor is defined around the operations it performs to produce some sort of `<EncryptedType>` element, whether this is a key or data. The encryptor is responsible for generating and properly combining all of the elements of the XML Encryption syntax including keys, encoded data, and all attributes. It is also responsible for the storage of keys and intermediate structures during the actual processing steps. Its job is to put together and build up a package that contains the encrypted data.

The Decryptor

The term *decryptor* refers to the entity that performs the exact opposite function of the encryptor. It is responsible for tearing down and decrypting any packaged `<EncryptedType>` elements that exist in a given XML document context. It is responsible for decoding and decrypting any encrypted data and provides the result back to the application. The decryptor is not responsible for validating the result of the decryption operation or ensuring that any resultant XML is well formed or valid.

The Encryptor: Process

The entire process for the actions of the encryptor can be captured in a single flowchart. This process is shown in Figure 7-9. The reader should look this over and try to follow it. The process is extremely straightforward and is shown with only a few simplifications. This process comes from the text description in the XML Encryption draft and should be considered repurposed and non-normative. The full normative process should be obtained from the latest XML Encryption draft. The flowchart notation shown is simple—ovals represent actions and diamonds represent choices.

The encryptor process begins with accepting three inputs: the encryption key, the algorithm, and the input data. The encryption process completes when we perform plaintext replacement or return an `<EncryptedType>`. These states are shown in bold. The information about the algorithm that will be used implicitly includes things like key

Figure 7-9

The encryptor process flowchart
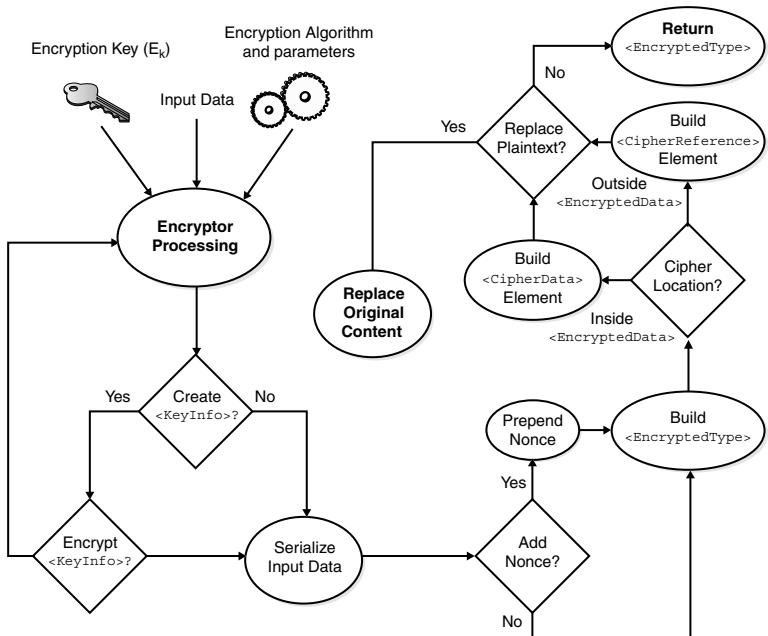
size and encryption mode, if applicable. At this point, we are not making any assumptions about the input data (whether it is treated as XML or as octets) because it will eventually be serialized into octets for actual encryption. Once these three inputs are processed, the next step is to decide whether or not we are including a `<ds:KeyInfo>` element in the `<EncryptedType>`. Notice that we are on the path to the creation of the `<EncryptedType>`. The final result can be either an `<EncryptedData>` element or an `<EncryptedKey>` element and depends on which path is taken farther down in the flowchart.

If we decide that a `<ds:KeyInfo>` should be created, we will then ask if we need to add an `<EncryptedKey>` element to the `<ds:KeyInfo>` element. If so, we follow the path back to the beginning of the process and proceed to create the `<EncryptedKey>` element. This step is the only recursive step in the entire process; the careful reader should notice that there is no upper bound on the number of times a given `<EncryptedKey>` element can be encrypted. It is possible to have many levels of encrypted keys (although most practical scenarios will afford just a few). Once this recursive step is finished and we have created the proper `<ds:KeyInfo>` element, we then proceed to serialize the input data. When we use the word *serialize*, we are referring to converting the input data into octets. It is possible that an application might use some other encoding or data content type, but in the end the encryption algorithm must operate on the raw binary form.

Once we have properly serialized the input data into octets, it is time to decide on a nonce value. A nonce value is usually a good idea if the type of data you are encrypting has some sort of a priori known structure. This means that if you have a structure markup language that only allows a few values by virtue of its schema definition or DTD, it is a good idea to add the nonce value at this time to prevent an attacker from deducing too much information. The nonce adds entropy to the plaintext and helps protect against certain cryptographic attacks. The reader should also note that a nonce value is not used in the case of an `<EncryptedKey>` element; it is only used in an `<EncryptedData>` element.

To build the `<EncryptedType>`, we need to know if we are going to be using a `<CipherReference>` element or a `<CipherValue>` element. We are in effect asking if the cipher data is enveloped by or referenced by the `<EncryptedType>` element. If we are using the `<CipherReference>` element, the element must be created and the appropriate URI location for the cipher data must be provided, as well as any applicable transforms. If we are using the `<CipherValue>` element, the encryptor is

responsible for Base-64-encoding the cipher data and including the encrypted data directly.

Finally, once we have built our `<EncryptedType>` element, we need to make a determination about plaintext replacement. This is easy if it is an `<EncryptedKey>` element because encrypted keys can't be used for plaintext replacement. However, if the final `<EncryptedType>` is an `<EncryptedData>` element, the encryptor needs to check the `Type` attribute and determine the type. This will be `Element`, `Content`, or a media type of some sort (this is specified in Table 7-1). If the `Type` attribute is not `Element` or `Content` (it is some media type), then the encryptor must return this `<EncryptedData>` element to the application (plaintext replacement cannot be performed).

If the `<EncryptedType>` is supposed to replace some original XML plaintext, the application must provide the entry point and the encryptor should be able to perform this replacement. You should note that an `<EncryptedData>` element with a `Type` attribute of `Element` or `Content` isn't required to perform plaintext replacement; it may also return the `<EncryptedData>` element to the application. This is useful in circumstances where the XML data is being used as a flat file database. For example, there would be no need to replace the original plaintext, but there might be a need to send a piece of the original XML document in encrypted form to some other recipient.

The Decryptor: Process

The decryptor fulfills the reverse process of the role of the encryptor and is responsible for decrypting and pulling apart an incoming `<Encrypted-Type>` element. The flowchart for the decryption process is shown in Figure 7-10.

The decryptor process begins by assuming that it has all of the necessary inputs to kickstart the decryption. The necessary parameters include the algorithm, parameters, padding information, and encryption mode. The IV (initialization vector) for block ciphers is implicit in the `<Cipher-Data>` and is assumed to be prepended to the ciphertext. If any information is missing from the `<EncryptedType>` that is required for the decryption process, the application must somehow supply it.

Once the algorithm and parameters are known, the first step is to check and see if a `<ds:KeyInfo>` element is present in the `<Encrypted-Type>`. If the `<ds:KeyInfo>` element is not present, the application must

supply the necessary decryption key. If the `<ds:KeyInfo>` element is present and is not encrypted (for example, a `<ds:KeyName>` is used), the application must successfully map this name or identifier to some type of nonencrypted key for use in the decryption process. However, if the `<ds:KeyInfo>` contains an `<EncryptedKey>` type, then the recursive step comes into play and we must repeat the process from the beginning until we somehow arrive at an unencrypted key. The astute reader should realize that this recursive process must eventually stop because some key must exist as a prime mover in order to be able to decrypt the `<EncryptedData>` element.

Once the decryption key has been obtained in some usable form (unencrypted), the decryptor must discern the location of the `<CipherData>`. That is, the actual encrypted data may be pointed to by a URI reference and transforms (in the case of a `<CipherReference>` element), or it may be enveloped directly inside the `<EncryptedData>` element. Either way, the end result of this step is the same—the cipher data must be decoded

and eventually decrypted. If a nonce is present, it also must be stripped at this time to ensure that the proper plaintext is eventually generated.

Once the nonce is stripped, the decryptor must determine if the value decrypted is a key value. If so, it must be saved at this time for use in a repeated instance of the decryptor operation. If the value decrypted is not a key value (for example, it is an `<EncryptedData>` element), the `Type` information must be returned to the application. This means that the decryptor is responsible for returning one of the values in Table 7-1. Finally, if plaintext replacement is to be performed, the decryptor is responsible for replacing the `<EncryptedData>` element with the proper plaintext. It is important to note that the decryptor is not required to actually validate that the end result of the decryption was proper XML; this task is delegated to the application. At this point, we have either performed a replacement operation on the input ciphertext or have returned the decrypted plaintext to the application. The work of the decryptor is finished for this specific `<EncryptedType>` element. If the XML document contains other `<EncryptedType>` elements, the decryption process must be repeated for each of these.

XML Encryption: Other Issues

At this point, the reader should have a good understanding of how XML Encryption works in terms of its basic syntax and processes. At the time of this writing, the XML Encryption draft is still young and more complex examples and implementations have yet to be worked out. This means that this entire chapter held slightly less detail than the material on XML Signatures (which are much more mature at this time). Specific information that is left out includes the specification of the actual symmetric key wrap functions as well as the shared secret derivation function provided by the XML Encryption draft. For those readers interested in more details about XML Encryption, the References section at the end of this book includes a link to the official World Wide Web Consortium (W3C) working draft.

This final section discusses some aspects of XML Encryption that just don't seem to fit well in any other broad category. The remaining topics of discussion include more information on key transport, decryption transform for XML Signatures, and a quick word on security considerations for XML Encryption.

More Information on Key Transport

The asymmetric key transport algorithms that are defined by the XML Encryption draft include the RSA algorithm with PKCS#1 padding and the RSA algorithm with OAEP padding. Both of these algorithms are useful for encrypting key data, but we have not mentioned these algorithms for encrypting arbitrary plaintext. That is, our previous discussion doesn't include examples or a discussion of how one might encrypt some arbitrary plaintext using the RSA algorithm. Instead, we specify how the RSA algorithm can be used to encrypt a symmetric key. This is the nature of asymmetric key transport and is also known by the term *digital enveloping* (this concept is touched on in the primer in Chapter 2).

The question remains, however, of whether it is possible (or useful) to use the key-transport algorithm defined in XML Encryption for the protection of arbitrary data. In fact, the XML Encryption draft doesn't prohibit the use of the key transport algorithm for arbitrary data, but there are two reasons why this may not be an especially useful operation. The first reason is because the nature of the padding scheme used in the RSA operation puts an artificial limit on the length of the data that can be encrypted. For example, if RSA with PKCS#1 padding is used, the upper bound on the number of bytes that can be encrypted is k-11, where k is the size of the modulus in bytes. So for a 512-bit RSA key, the largest possible piece of data that can be encrypted with the RSA PKCS#1 key transport algorithm is 53 bytes ($64 - 11$). Although the amount of data increases with the key size, this operation is still limiting (for a 2,048-bit key, the largest piece of data that can be encrypted is 245 bytes). The second reason for avoiding direct RSA operations on plaintext is because the operation is horrendously slow compared to analogous symmetric key encryption operations. In short, the key transport is designed to only encrypt symmetric keys and although it can be used for arbitrary plaintext, it should probably be avoided unless a special fringe case arises.

The Decryption Transform for XML Signatures

This topic actually comprises a short separate W3C working draft. The purpose of the decryption transform is to help manage the complexity that arises when XML documents begin to mix XML Signatures and XML Encryption (which is a common use case). The decryption transform is still in its early stages (at the time of this writing), so a conceptual overview of how it works will be shown, but the actual normative algorithm will be left up to the appropriate W3C document. Once the reader

understands the purpose behind the decryption transform, the rest is just details. The decryption transform is a transform that is intended to be used within an XML Signature, but it is meaningless without knowledge of XML Encryption. This is why it is included in this chapter rather than earlier chapters.

Decryption Transform Motivation

An easy way to understand why the decryption transform is necessary is to think about the plaintext replacement case for decryption. That is, when we have an incoming XML document that contains both a `<ds:Signature>` element and at least one `<EncryptedData>` element, it is possible in some cases that decrypting the `<EncryptedData>` element will break an XML Signature. This happens if the `<Encrypted-Data>` element is signed with an XML Signature because when we perform the plaintext replacement, we are in effect altering the original document. When signature verification occurs, the hash values won't match because plaintext replacement has been performed on the `<EncryptedData>` element. We can build up to an example of what we mean by this by first considering Listing 7-16, which shows an arbitrary XML document that we might want to sign and encrypt.

Suppose we begin by performing plaintext replacement and encrypt `<SensitiveInfo1>`. This is shown in Listing 7-17 in bold.

So far things are still simple. We have merely performed some content encryption on the value Send 5 million to the North base!. We have used the Triple-DES algorithm in CBC mode and have assumed that the receiver implicitly knows the decryption key. Let's now take this a step further and create an enveloping XML Signature over the `<SecureDoc>` element. The final output from this operation appears in Listing 7-18.

Things have gotten a bit more complex, but all that has really happened is that we have placed the `<SecureDoc>` element inside a `<ds:Object>` element and signed its structure. At this point, there is

Listing 7-16	
| An arbitrary XML document to encrypt and sign | ```
<SecureDoc>
  <SensitiveInfo1>
   Send 5 million to the North base!
  </SensitiveInfo1>
  <SensitiveInfo2>
    Attack at dawn!
  </SensitiveInfo2>
</SecureDoc>
``` |

```
<SecureDoc>
  <SensitiveInfo1>
    <EncryptedData Id="ED1"
                    Type="http://www.w3.org/2001/04/xmlenc#Content"
                    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <CipherData>
        <CipherValue>
          qF1z+e5Jwvy49vVmZpMkb/3aMdr4ESGmTbc7FcQ
        </CipherValue>
      </CipherData>
    </EncryptedData>
  </SensitiveInfo1>
  <SensitiveInfo2>
    Attack at dawn!
  </SensitiveInfo2>
</SecureDoc>
```

already an ordering problem. That is, if the receiver decides to decrypt the data before verifying the signature, plaintext replacement will occur and the digest value of the #envelopedData reference will be incorrect. One can argue that this is not an issue at this point because it is fairly clear that the data inside `<ds:Object>` has been signed by inspecting the source reference. However, consider what happens when we modify the contents of `<ds:Object>` further by encrypting (but not signing) the `<SensitiveInfo2>` element. Only the extracted `<ds:Object>` element is shown in Listing 7-19.

At this point, the document is more complicated than ever before. The `<ds:Object>` element now contains two `<EncryptedData>` elements, and a recipient has no way to know the order in which the signing or encryption operations were performed. That is, if a recipient were to try to validate the XML signature, it would break because the signature is only over one of the `<EncryptedData>` elements. A transform that designates which `<EncryptedData>` elements should be excepted from the decryption operation is needed. For example, if we can specify to the verifier that the `<EncryptedData>` element with an Id value of ED1 should not be decrypted (but all other instances of `<EncryptedData>` should be), then we would have a way to properly verify the XML Signature over `<SecureDoc>`. This is the nature of the decryption transform for XML Signatures—it is a list of `<EncryptedData>` elements that should be skipped over in order to preserve the proper order for signature verification to be successful. Those readers who want more details about the

Listing 7-18

Signing the
`<SecureDoc>`
element

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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="#envelopedData">
    <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>3NMzvYIWFHiF3LStTgxtQkS9NpI=</DigestValue>
  </Reference>
  </SignedInfo>
  <SignatureValue>
    F2dXbU267Zaw/bsDfpM4GkqbDIl7JcdU6mR+yYtvEFAK87v6j5vyf8X8TF0HWqMK
    BPlvQthSFBEcKurEkHxcvQ==
  </SignatureValue>
  <ds:Object Id="envelopedData" xmlns=""
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <SecureDoc>
     <SensitiveInfo1>
      <EncryptedData Id="ED1"
                     Type="http://www.w3.org/2001/04/xmlenc#Content"
                     xmlns="http://www.w3.org/2001/04/xmlenc#">
      <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04 /xmlenc#tripledescbc"/>
       <CipherData>
         <CipherValue>
           qFlz+e5Jwvy49vVmZpMkb/3aMdr4ESGmTbc7FcQ
         </CipherValue>
       </CipherData>
      </EncryptedData>
     </SensitiveInfo1>
     <SensitiveInfo2>
       Attack at dawn!
     </SensitiveInfo2>
   </SecureDoc>
  </ds:Object>
</Signature>
```

decryption transform should visit the References section at the end of this
book for links to web resources and the current W3C Note regarding this
subject.

Security Considerations

There are a few security considerations for the XML Encryption syntax as
well as for XML documents that exhibit both signed and encrypted prop-
erties. The first issue is that of data integrity, or authenticity, of a given
`<EncryptedType>` element. The XML Encryption draft does not provide

```
<ds:Object Id="envelopedData" xmlns=""
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <SecureDoc>
     <SensitiveInfo1>
      <EncryptedData Id="ED1"
                     Type="http://www.w3.org/2001/04/xmlenc#Content"
                     xmlns="http://www.w3.org/2001/04/xmlenc#">
      <EncryptionMethod
         Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
       <CipherData>
         <CipherValue>
           qF1z+e5Jwvy49vVmZpMkb/3aMdr4ESGmTbc7FcQ
         </CipherValue>
       </CipherData>
      </EncryptedData>
     </SensitiveInfo1>
     <SensitiveInfo2>
      <EncryptedData Id="ED2"
                     Type="http://www.w3.org/2001/04 /xmlenc#Content"
                     xmlns="http://www.w3.org/2001/04 /xmlenc#">
      <EncryptionMethod
        Algorithm="http://www.w3.org/2001/04 /xmlenc#tripledes-cbc"/>
       <CipherData>
         <CipherValue>
           tvEFAK87v6j5vyf8X8TF0HWqMKBPlvQthSFBEcKu
         </CipherValue>
       </CipherData>
      </EncryptedData>
     </SensitiveInfo2>
   </SecureDoc>
</ds:Object>
```

the integrity of its structure. For example, an attacker listening in on encrypted messages can easily change (or delete) pieces of the `<EncryptedType>` structure that may be present, such as the `<ds:Key-Info>` element or the `<EncryptionMethod>` element. This will obviously cause havoc with the recipient, who will most likely be unable to continue processing the given `<EncryptedType>` message. Once the reader realizes this, the importance of applying a digital signature to any `<Encrypted-Type>` elements becomes especially clear. This issue should not be overlooked—there is no structural integrity built into XML Encryption.

The second issue has been noted by Hal Finney (as noted in the current XML Encryption draft) and is in regards to XML data that is signed and then encrypted. There is a slight security issue here because in the case of an XML Signature, the `<DigestValue>` element is still in the clear. This means that there is extra information about the plaintext along with the ciphertext. This information is a digest of the plaintext, which means that

it may be possible to try arbitrary plaintext guesses (which might be possible or likely if the XML structure is known ahead of time) and see if the correct digest value can be computed. Once this has been computed, the encryption key can be determined with ciphertext, plaintext, and knowledge of the encryption algorithm.

Chapter Summary

This chapter provides an overview of XML Encryption with an end goal of conveying a conceptual understanding of XML Encryption syntax and processes. The chapter begins with a distinction between encrypting arbitrary octets and encrypting XML. The former case produces an `<EncryptedData>` element as its result, whereas the latter case may result in plaintext replacement, where the original plaintext will have pieces of its structure replaced with one or more `<EncryptedData>` elements. The chapter then transitions into a piece-by-piece discussion of the possible child elements of `<EncryptedData>`, including the `<ds:KeyInfo>` element, which opens the door to complexity for XML Encryption. The `<ds:KeyInfo>` element is responsible for communicating information about the decryption key and can house the `<EncryptedKey>` element, which is a type similar to `<EncryptedData>`. Both `<EncryptedData>` and `<EncryptedKey>` are derived types from the parent `<EncryptedType>` element, which provides default behavior for these elements.

After a discussion of syntax, the main processing rules for XML Encryption are shown in a flowchart format. Three entities are defined by the XML Encryption draft: the application, the encryptor, and the decryptor. The application refers to any entity that uses a given XML Encryption implementation. The encryptor is the entity that is responsible for the creation of a given `<EncryptedType>` element. The decryptor is responsible for decrypting and breaking apart a given `<EncryptedType>` element. The chapter concludes with a quick discussion of the decryption transform for XML Signatures as well as a brief look at some security considerations for XML Encryption.

This page intentionally left blank.

XML Signature Implementation: RSA BSAFE© Cert-J

This chapter explores the design and usage of XML Signatures in RSA's *Cert-J* product. Cert-J is a PKI toolkit for Java that contains an implementation of the XML Signature Recommendation. Cert-J is layered on top of RSA's cryptography toolkit for Java, called *Crypto-J*. The XML Signature implementation relies on classes from Cert-J, Crypto-J, and Apache's Xerces and Xalan packages. Cert-J uses the Document Object Model (DOM) for representing and processing XML structures within memory. For more information on DOM, see the primer in Chapter 3.

The class hierarchy is presented using class diagrams. This approach gives the reader an overall view of the design approach taken by Cert-J as well as important relationships between classes in the toolkit. Code examples are given that illustrate the most important and useful function calls and classes. The anxious reader looking to get something up and running quickly should look at the Signature Processing section. This section has quick and dirty examples of creating and verifying an XML Signature. Finally, the chapter concludes by showing some more advanced code examples that aim to solve certain specialized problems.

RSA BSAFE Cert-J: Class
Diagrams and Code Examples

One way to gain an understanding of a toolkit product is to look at how the class relationships are structured. Knowledge of the inheritance hierarchy can provide insight into the subclass relationships and help the programmer understand when subtype polymorphism is useful. Class diagrams also illustrate how Cert-J sees the world of XML Signatures by showing abstractions for the corresponding XML syntax.

This section represents a walk-through of the public classes that are of any use to the developer. Class diagrams are given where appropriate; the intent is not to flesh out the entire design, but instead to show specific classes and their relationships. The class diagrams are presented using syntax similar to the Unified Modeling Language (UML). Readers unfamiliar with class diagrams or the UML should have little trouble following along; the syntax is decidedly simple, and the real goal is to convey relationships and conceptual understanding.

Syntax and Processing Revisited

The XML Signature Recommendation divides the specification of XML Signatures into syntax and processing rules. A similar dichotomy is present within the class hierarchy for XML Signatures as they are implemented in Cert-J. There is a clear division between classes designed to handle the syntax of an XML Signature versus classes that deal directly with the *parsing* or processing of an XML Signature as it relates to the DOM. Table 8-1 shows this loose division.

Table 8-1	XML Signature related classes	Processing related classes
XML Signature Classes in Cert-J	XMLSignature	ParserHandler
	Reference	VerificationInfo
	Transformer	
	KeyInfo	
	Manifest	

In general, all of the classes on the left side of the table are strongly related with some corresponding element or concept that is central to XML Signatures while the classes on the right side of the table represent classes that are necessary for the practical implementation of XML Signatures. Table 8-1 is not a complete list of classes, but simply the most central ones in each category. Cert-J makes heavy use of subclassing and most of the classes shown in Table 8-1 have multiple child classes that extend the behavior of the parent class. Discussion will begin with syntax-oriented classes followed by the classes that pertain to the processing and actual creation of XML Signatures.

XMLSignature

The most important and fundamental class for XML Signatures as they are implemented in Cert-J is the *XMLSignature* class. This class encapsulates the entire `<Signature>` element. Its contents eventually come to represent a fully featured `<Signature>` element as shown in Listing 4-3 (see Chapter 4). Most of the work done when creating an XML Signature with Cert-J begins with creating an instance of `XMLSignature`. This instance is modified with accessor functions and finishes when the `sign()` method is called.

There are a number of constructor options for the `XMLSignature` class. The choice of constructor is strongly related to the *type* of signature that one wants to create. For example, if the goal is a *detached* signature, then the toolkit needs no additional data up front, and the empty constructor can be used. The data to be signed will be added later as a `Reference` object. If the goal is an *enveloped* or *enveloping* signature, the toolkit needs additional data to envelope or become enveloped by. In most cases, this means additional XML data will be added as an argument to the constructor. Finally, regardless of the constructor used, Cert-J provides methods for adding additional data to an instance of `XMLSignature` through the use of an accessor function called *setDocument()*. This means that even if the empty constructor is chosen, XML data can be added for use in an enveloped or enveloping signature at any time before the signature is created. Listing 8-1 shows different ways in which an XMLSignature instance can be created. There are three separate examples inside Listing 8-1.

In Listing 8-1, the code marked in bold shows the use of the XMLSignature constructor and the related `setDocument()` function. Examples 2 and 3 accomplish the same thing, but example 3 does it in

Listing 8-1

Using the
`XMLSignature`
constructor

```
import com.rsa.certj.xml.*;
import org.w3c.dom.*;
import java.io.FileInputStream;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.InputSource;

class CodeListing81 {
  public static void main (String args[]) throws Exception {
  /* Example #1:
     Empty XMLSignature constructor.
     Most often used for a detached signature.*/
     XMLSignature xmlSig1 = new XMLSignature();
  /* Example #2:
     Instantiates an XMLSignature object with file.xml as the input
     data
     for an enveloped or enveloping signature.*/
     XMLSignature xmlSig2 = new XMLSignature("file.xml");
  /* Example #3:
     Creates a org.w3.dom.Document from a file and adds it to an
     empty
     XMLSignature object. */
     InputSource inputSource =
     new InputSource(new FileInputStream("file.xml"));
     // Creates a new DOM Parser for reading in file.xml
     DOMParser domParser = new DOMParser();
     domParser.parse(inputSource);
     Document xmlDoc = domParser.getDocument();
     // Use Cert-J to associate the org.w3.Document with an
     XMLSignature
     XMLSignature xmlSig3 = new XMLSignature();
     xmlSig3.setDocument(xmlDoc);
  }
}
```

a more roundabout way. Example 3 might be used in situations where
the program already has access to an `org.w3.dom.Document` type.

Once an instance of `XMLSignature` is made, the *type* of signature must
be explicitly communicated to the object. For example, if we create an
instance of `XMLSignature` with a filename (as is done in example 2 inside
Listing 8-1), how will the library know which type of signature to create?
If we create an `XMLSignature` instance as done in example 2, it is possi-
ble that an enveloped or enveloping signature can be created.

To solve this particular problem, the library uses three static
integer variables to denote the signature type. The available types
are *DETACHED_SIGNATURE*, *ENVELOPING_SIGNATURE*, and
ENVELOPED_SIGNATURE. The chosen type is set with the use of the
`setSignatureType()` method. An additional *SIGNATURE_MASK* inte-

ger is used to determine which type of signature has been selected. This code is shown in Listing 8-2 and is prevalent throughout the sample code that is included with Cert-J.

Once the signature type is set, the details of the <Signature> element need to be fleshed out. Considering the previous discussion, the toolkit only knows about the type of signature to create and any possible input data used to create the signature. Details, such as the list of <Reference> elements, the signature algorithm, and canonicalization method, have yet to be specified. Furthermore, we still need to make provisions for additional optional elements such as <KeyInfo> or <Manifest>. There are two important accessor functions that help us fill in some of this information: setSignatureMethod() and setCanonicalizationMethod(). As their names imply, these functions are used for specifying the URI identifiers for the corresponding signature method and canonicalization method. These functions operate on string objects, and the URI identifiers specified in the XML Signature

Listing 8-2

Setting the signature type on an instance of XMLSignature

```java
import com.rsa.certj.xml.*;

class CodeListing82 {
  public static void main (String args[]) throws Exception {
    // First create an instance of XMLSignature
    XMLSignature xmlSig = new XMLSignature();
    // Set the signature type to be used
    xmlSig.setSignatureType(XMLSignature.ENVELOPING_SIGNATURE);
    int sigType = xmlSig.getSignatureType();
    int maskResult = (XMLSignature.SIGNATURE_MASK & sigType);

    /* This following code snippet shows how one might want
       to check and see what type of signature we have. */
    if (maskResult == XMLSignature.DETACHED_SIGNATURE) {
      System.out.println("Signature object is a Detached
      Signature");
    } else if (maskResult == XMLSignature.ENVELOPING_SIGNATURE) {
        System.out.println("Signature object is an Enveloping
        Signature");
    } else if (maskResult == XMLSignature.ENVELOPED_SIGNATURE) {
        System.out.println("Signature object is an Enveloped
        Signature");
    } else {
      System.out.println("Unknown Signature type");
    }
  }
}
```

Recommendation should be used as input. An example of how these are used is shown as follows:

```
// Set the signature method to RSA with SHA-1
xmlSig.setSignatureMethod
("http://www.w3.org/2000/09/xmldsig#rsa-sha1");
// Set the canonicalization method to Canonical XML without
comments
xmlSig.setCanonicalizationMethod
("http://www.w3.org/TR/2001/REC-xml-c14n-20010315");
```

The signature method chosen is RSA with SHA-1, and the canonical-ization method chosen is Canonical XML 1.0 without comments. Notice that the string URI identifiers are taken directly from the XML Signature Recommendation.

At this point, we have filled in most of the details required to actually sign some data. The remaining structures that need to be specified are the list of <Reference> elements and accompanying transforms as well any additional optional elements such as <KeyInfo>, <Object>, or <Manifest>. Figure 8-1 shows a class diagram that represents class *association*. That is, the figure represents how the XMLSignature class is associated with additional classes that will enable us to fill in the final details of the <Signature> element.

Figure 8-1 is telling us that an XMLSignature instance is composed of (this is the *diamond notation*) one or more instances of Reference and zero or one instance of KeyInfo. A valid XML Signature can be created without a <KeyInfo> element (<KeyInfo> is optional), but at least one

Figure 8-1

XMLSignature
class
relationships
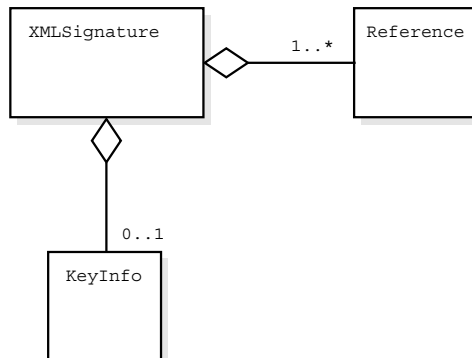
<Reference> element is required. Once a Reference object is created, it can be added to an XMLSignature instance with the accessor function setReferences(). This function takes an array of Reference objects; it is possible to add many References to an XMLSignature at a single time. The next section discusses the Reference class in more depth.

Reference and Transformer

The Reference class is perhaps the second most important class next to the XMLSignature class. Although it is possible to give data to an XMLSignature object with the use of the correct constructor, the toolkit doesn't know which portion of this data to sign without the use of a Reference object. If no data is given to the XMLSignature object, a Reference object fully defines the data source.

The simplest possible <Reference> element in an XML Signature may omit everything but the <DigestValue> and <DigestMethod> child elements (see Chapter 5). Furthermore, a practical implementation cannot create a digest value without some sort of digest method or algorithm to use. This being the case, the simplest possible useful Reference object that can be created must specify a digest method. From here, additional properties, such as the URI, Type, and <Transforms>, can be added. Cert-J offers a simple constructor and accessor functions for creating a Reference object. A small code listing is shown in the following example:

```
// First use the default Constructor
Reference ref = new Reference();
// Set the Digest Method (URI Identifier)
ref.setDigestMethod("http://www.w3.org/2000/09/xmldsig#sha1");
// Set the URI (Detached Signature)
ref.setURI("http://www.server.com/foo.xml");
```

Adding transforms to a Reference object involves just another accessor function call. To understand how transforms are implemented in Cert-J, consider Figure 8-2 that shows the class hierarchy and subtype relationships between a Reference object and the Transformer parent class.

The Transformer class is an abstract parent class that has four children that implement its interfaces. The four children represent the four required and recommended transformations as specified in the XML Signature Recommendation: Base-64 decoding, Canonical XML 1.0, enveloped signature, and XPath processing. The corresponding classes are Base64Transformer, C14NTransformer, EnvelopedTransformer,

Figure 8-2

`Reference` class relationships and `Transformer` subtypes
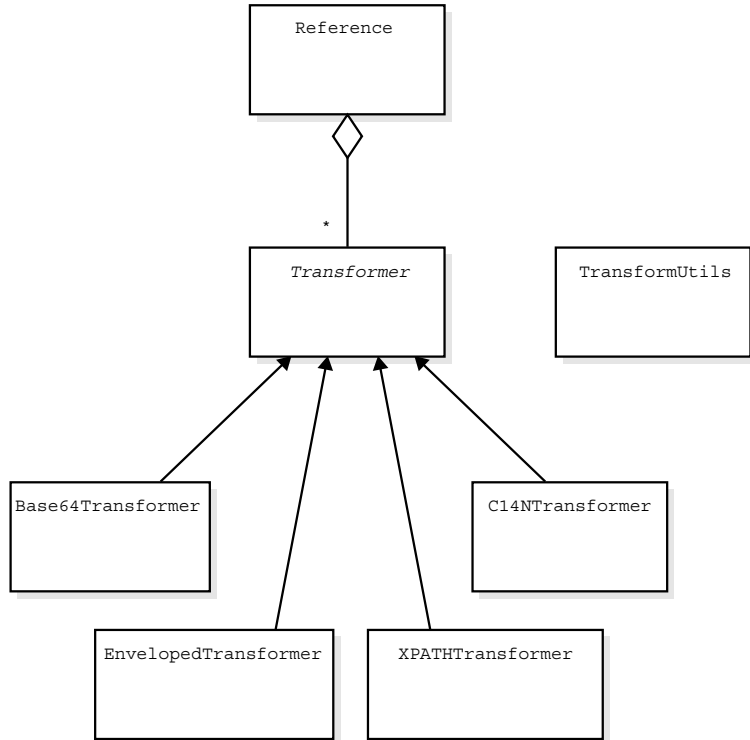
and `XPATHTransformer`. The `TransformUtils` class shown in Figure 8-2 is used for adding user-defined transforms to the list of acceptable transformations. This topic is revisited in a later section where we will investigate how to add our own custom transformation that does ZIP decompression.

There are two ways to instantiate `Transformer` instances in Cert-J, a standard constructor and a factory method. For some applications, the *factory pattern* might be more appropriate, but the end result is the same. The factory method is convenient if you need to instantiate a `Transformer` in a general way; this is shown in Listing 8-3.

In Listing 8-3, the `Transformer` base class is used for all types of Transformer objects. The input URI identifier determines the type of

Listing 8-3

Factory method for instantiating a `Transformer` type

```
import com.rsa.certj.xml.*;
class CodeListing83 {
  public static void main (String args[]) throws Exception {

  String[] algs = {
    "http://www.w3.org/TR/2001/REC-xml-c14n-20010315",
    "http://www.w3.org/2000/09/xmldsig#base64",
    "http://www.w3.org/2000/09/xmldsig#enveloped-signature",
    "http://www.w3.org/TR/1999/REC-xpath-19991116",
    "http://www.w3.org/TR/2000/CR-xml-c14n-20001026" };

    /* Factory method for a Canonical XML Transformer (Without
    Comments) */
    C14NTransformer c14n =
    (C14NTransformer)Transformer.getInstance(algs[0]);
    System.out.println("Transformer: " +
    c14n.getTransformAlgorithm());

    /* Factory method for a Base64 Transformer */
    Base64Transformer b64 =
    (Base64Transformer)Transformer.getInstance(algs[1]);
    System.out.println("Transformer: " +
    base64.getTransformAlgorithm());

    /* Factory method for an EnvelopedSignature Transformer  */
    EnvelopedTransformer env =
    (EnvelopedTransformer)Transformer.getInstance(algs[2]);
    System.out.println("Transformer: " +
    envSig.getTransformAlgorithm());

    /* Factory method for instantiating an XPath Transformer */
    XPATHTransformer xpath =
    (XPATHTransformer)Transformer.getInstance(algs[3]);
    System.out.println("Transformer: " +
    xpath.getTransformAlgorithm());

    /* Factory method for Canonical XML under an older namespace */
    C14NTransformer c14nOld =
    (C14NTransformer)Transformer.getInstance(algs[4]);
    System.out.println("Transformer: " +
    c14nOld.getTransformAlgorithm());
  }
}
```

Transformer generated—this is especially useful in cases where older namespaces must be supported. For example, the last example in Listing 8-3 uses an old namespace for Canonical XML as specified in RFC 3075. Support for generating signatures with older namespaces is useful for maintaining interoperability between different XML Signature implementations, which may have implemented an older revision of the specification.

Using XPATHTransformer

All of the `Transformer` objects shown in Listing 8-3 are instantiated using only the URI identifier; no other initialization information for these objects is required. This is acceptable for all of the Transformer objects except for the `XPATHTransformer`. The reason why is because the user needs a way to give the toolkit an XPath expression to use as the basis of the transformation. There are two ways to provide an XPath expression to the toolkit, a simple accessor function that accepts a string input, or a function that accepts a `org.w3c.dom.Node` type. Both of these functions operate on an instance of the `XPATHTransformer` object. The first method is quite easy and is shown in a code snippet as follows:

```
// First make a new XPATHTransformer. Use the normal constructor
here.
XPATHTransformer xpath = new XPATHTransformer();
// Here is a simple String XPath Expression
String xpathExpression = "ancestor-or-self:Element1";
// Use an accessor function to set the String
xpath.setXPathExpression(xpathExpression);
```

The second method of initializing the `XPATHTransformer` with a valid expression involves using the Xerces org.w3c.dom package to manually create the actual `<XPath>` element and text content. The XML Signature Recommendation specifies that the expression to be evaluated appears as the content of an element named `<XPath>`. Cert-J gives the user the option of adding this element manually as an `org.w3c.dom.Element` type. The primary reason for this application programming interface (API) in the toolkit is to provide for additional flexibility. In the first method where only the accessor function is used, Cert-J will internally create the `<XPath>` element appropriate text content. The Xerces and Cert-J code required for the creation of this element is shown in Listing 8-4.

In Listing 8-4, most of the work done is with the Xerces API to create the proper `<XPath>` element and expression content. There are only two calls made to Cert-J in this instance, one call to create an instance of the `XPATHTransformer` and one call to add the `org.w3c.Node` type to this Transformer. Both Cert-J specific calls are shown in bold in Listing 8-4. The creation of the `org.w3c.Element` type (which is a subtype of `org.w3c.Node`) also includes initializing the namespace of the `<XPath>` element that contains the expression.

```
// Import the Crypto-J Classes
import com.rsa.certj.xml.*;
// Import Xerces packages
import org.apache.xerces.dom.DocumentImpl;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
class CodeListing84 {
  public static void main (String args[]) throws Exception {
  // First create a new XPATHTransformer instance.
  XPATHTransformer xpath = (XPATHTransformer)
  Transformer.getInstance("http://www.w3.org/TR/1999/REC-xpath-
  19991116");
  // Arbitrary XPath Expression
  String xpathExpression = "ancestor-or-self:Element1";
  // Create a <XPath> element that wraps the XPath expression using
  the DOM
  DocumentImpl documentImpl = new DocumentImpl();
  Element xpathNode = documentImpl.createElement("XPath");
  Text textNode = documentImpl.createTextNode(xpathExpression);
  xpathNode.appendChild(textNode);
  // Finally, add the org.w3c.Node to the XPATHTransformer
  xpath.setXPathExpression(xpathNode);
 }
}
```

Multiple Transforms

Multiple transforms are usually associated with a given `<Reference>`
element. This processing is accomplished with a method called *set-
Transform()* that operates on an array of `Transformer` object types.
An example code snippet is shown in the following example:

```
Transformer b64 = << previously initialized Base64 transformer >>
Transformer xpath = << previously initialized XPath transformer >>
Transformer[] transforms = {b64,xpath};
Reference ref = new Reference();
ref.setTransform(transforms)
```

In the previous code example, the Reference object in question now has two transforms associated with it, Base64Transformer and XPATH-Transformer. Conceptually, the <Reference> element should appear as follows when the signature is finally created:

```
<Reference URI="file:///C:\file.b64">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/2000/09 /xmldsig#base64"/>
    <Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-
    19991116">
      <XPath>
        ancestor-or-self:Element1
      </XPath>
    </Transform>
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09 /xmldsig#sha1"/>
  <DigestValue>T2UhxvKRnqtupNLDuIkYanwZBYg=</DigestValue>
</Reference>
```

In this example, the source file has been Base-64 encoded and needs to be decoded (here we assume the decoded file is an XML document). Once we obtain the raw XML document, it is converted to a *node-set* and sent to the XPATHTransformer, which filters the document with the given expression (ancestor-or-self:Element1). This particular expression selects an element named <Element1> and all of its children for the actual reference data used in the signature.

KeyInfo

The <KeyInfo> element defined by the XML Signature Recommendation is shared by two other XML Security related technologies: XML Encryption and XML Key Management (XKMS). Discussion of the <KeyInfo> element here will be constrained to its practical use in Cert-J with XML Signatures.

The <KeyInfo> element is represented in Cert-J with the KeyInfo class. According to the XML Signature Recommendation, there can be at most one <KeyInfo> element per <Signature> element. The same constraint holds true with the KeyInfo class and its relationship to the XMLSignature class. Figure 8-3 shows a class diagram that describes the relationships KeyInfo has to other classes in Cert-J.

Figure 8-3 shows the KeyInfo class and its three subclasses, Key-Value, RetrievalMethod, and X509Data. The KeyInfo class is an
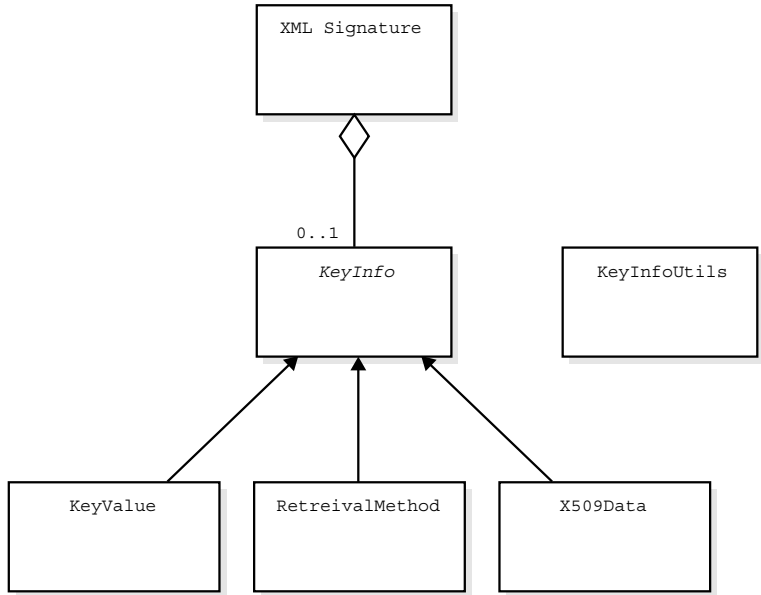
abstract class and is designed in much the same way as the `Transformer` class. The XML Signature Recommendation specifies seven elements that can appear as children of `<KeyInfo>`. Cert-J only implements three of these: `<KeyValue<`, `<RetrievalMethod>`, and `<X509Data>`. New Key-Info child classes can be added through the use of the KeyInfoUtils class —it is possible for the user to add the missing types or create application-specific KeyInfo types.

For most practical applications that intend to interface with a traditional PKI, the contents of the `<KeyInfo>` element can be best understood as a container for an X.509 certificate or public verification key. The `<KeyInfo>` child elements that are designed to store these two types of cryptographic objects include `<KeyValue>` and `<X509Data>`. These elements are represented in Cert-J with the KeyValue and X509Data subclasses, respectively.

KeyValue with Raw Key Data

Perhaps the easiest way to think of the `KeyInfo` class and its subclasses is to think of a simple public verification key. When a digital signature is created, it must be verified with some sort of public verification key (See the primer in Chapter 2). The `KeyValue` subclass is designed to represent the raw, mathematical key data that is divorced from any sort of encoding scheme. In simple terms, it is just a key value. Creating a `KeyValue` type is simple; the *factory pattern* is used here again much like the `Transformer` class. Listing 8-5 shows how to create a `KeyValue` type and set the verification key with raw public key data.

There are a number of important things to note in Listing 8-5. The first thing to realize is that you will almost never see a raw verification key used in practice. Most public verification keys are in a BER encoded binary format. The example is shown for illustrative purposes and highlights how Cert-J and Crypto-J fit together. The first line of code shows the factory constructor used to make a `KeyValue` type. From here on, work is done to create a `JSAFE_PublicKey` instance. The `JSAFE_PublicKey` class is part of Crypto-J and is the concrete class that holds any sort of key data, whether it is raw or encoded. Once a `JSAFE_PublicKey` is instantiated, the raw key data (in this case, a 512-bit RSA key) is used to complete the initialization. The `JSAFE_PublicKey` is further used to complete the initialization of the `KeyValue` type. The last point of possible confusion is the `setKeyInfos()` method. The name of this method implies that it creates more than one `<KeyInfo>` element in the final signature, but in reality, it is simply creating a single `<KeyInfo>` element, and the contents of the `KeyInfo[]` array contains the child elements that will appear inside the `<KeyInfo>` element when the signature is finally created.

KeyValue with an Encoded Key

Setting the raw public key data manually is a very impractical way of handling verification key information. Most public keys arrive in a single binary encoded blob or live inside an X.509 certificate. Fortunately, the basic process for creating a `KeyValue` is the same. The `KeyValue` type has its value set with the `setKey()` function and it always takes a `JSAFE_PublicKey` as input. This means that the problem of handling the encoded key type is deferred to the `JSAFE_PublicKey` class. We have already seen how to create a `JSAFE_PublicKey` with raw key data; this

Listing 8-5

Using raw key data to create a `KeyValue` type

```
import com.rsa.certj.xml.*;
import com.rsa.jsafe.*;
class CodeListing85 {
  public static void main (String args[]) throws Exception {
    // First create a KeyValue type using the factory constructor
    KeyValue keyValue = (KeyValue)KeyInfo.getInstance("KeyValue");
    // Create an empty key object. This is a call to Crypto-J
    JSAFE_PublicKey verificationKey =
    JSAFE_PublicKey.getInstance("RSA","Java");
    // Here comes the raw key data
    byte[] modulus = {
      (byte)0xC5,(byte)0xAC,(byte)0x92,(byte)0xE8,(byte)0x9E,(byte)0x96,
      (byte)0xBC,(byte)0xC8,(byte)0x3A,(byte)0x36,(byte)0x2D,(byte)0x22,
      (byte)0xE1,(byte)0xD7,(byte)0x99,(byte)0x52,(byte)0xAC,(byte)0x71,
      (byte)0x73,(byte)0xE1,(byte)0x80,(byte)0x60,(byte)0xA5,(byte)0xE5,
      (byte)0xDA,(byte)0x62,(byte)0xAE,(byte)0xF7,(byte)0x32,(byte)0x00,
      (byte)0x12,(byte)0x39,(byte)0x9E,(byte)0x05,(byte)0x46,(byte)0x83,
      (byte)0xD1,(byte)0x03,(byte)0x68,(byte)0xDD,(byte)0x41,(byte)0xE3,
      (byte)0x74,(byte)0x86,(byte)0x69,(byte)0x81,(byte)0x7D,(byte)0xDB,
      (byte)0xE0,(byte)0x70,(byte)0xB7,(byte)0x39,(byte)0xDE,(byte)0x05,
      (byte)0xA9,(byte)0xDC,(byte)0xD4,(byte)0x5F,(byte)0xDE,(byte)0xF2,
      (byte)0x72,(byte)0x18,(byte)0x9C,(byte)0xF9
    };
    byte[] exponent = { (byte)0x11 };
    byte[][] rawKeyData = {
      modulus,
      exponent
    };
    // Now we have a key object created with raw key data
    verificationKey.setKeyData(rawKeyData);
    // This is where Cert-J and Crypto-J meet
    keyValue.setKey(verificationKey);
    KeyInfo[] keyInfos = { keyValue };
    // We need an instance of XML Signature.
    XMLSignature xmlSig = new XMLSignature();
    // This creates a <KeyInfo> element with a single <KeyValue> child
    xmlSig.setKeyInfos(keyInfos);
  }
}
```

means that the problem of handling encoded public keys is reduced to understanding how `JSAFE_PublicKey` deals with this key type. The standard format defined by RFC2459 is the `SubjectPublicKeyInfo` type and is commonly referred to as a BER encoded X.509 public key. For more information on this key format, see the primer in Chapter 2. The X.509 public key is BER encoded and differs from the raw key data in that it appears as a single blob and can be thought of as a single byte array. The raw key data is two entities: a modulus and an exponent; in the X.509 public key type, the modulus and the exponent are included in the BER encoding. Furthermore, the `SubjectPublicKeyInfo` type is also an

integral part of any X.509 certificate. An ideal situation would be to have some code that deals with both situations. That is, we need a way to read a binary encoded public key from a disk and create a JSAFE_PublicKey instance as well as a way to extract a raw public key from an X.509 certificate. Once we have the JSAFE_PublicKey instance, it becomes useful to the programmer as input to the setKey() function for the initialization of the KeyValue type. Listing 8-6 shows how to read a BER encoded X.509 public key (SubjectPublicKeyInfo) from a file and creates a JSAFE_PublicKey instance.

Listing 8-6 assumes that there is a file on disk that contains the BER encoded public key called *publicKey.ber*. This file is opened, and its contents are read into a byte array called *publicKeyBER*. From here, this byte array is used to construct the JSAFE_PublicKey instance. Once this call is made, the rest of the code should match Listing 8-5. The KeyValue type simply accepts the JSAFE_PublicKey; it has no knowledge of where this key came from. The two snippets shown in bold in Listing 8-6 represent

Listing 8-6

Creating a
JSAFE_
PublicKey from
an X.509
SubjectPublic-
KeyInfo

```
import com.rsa.certj.xml.*;
import java.io.*;
import com.rsa.jsafe.*;
class CodeListing86 {
  public static void main (String args[]) throws Exception {

    // Read the file containing the SubjectPublicKeyInfo into a
    byte array
    File publicKeyFile = new File("publicKey.ber");
    long fileLength = publicKeyFile.length();
    FileInputStream publicKeyStream = new
    FileInputStream(publicKeyFile);
    byte[] publicKeyBER = new byte[(int)fileLength];
    publicKeyStream.read(publicKeyBER);
    publicKeyStream.close();
    // Create a new JSAFE_PublicKey object
    JSAFE_PublicKey verificationKey =
    JSAFE_PublicKey.getInstance(publicKeyBER,0,"Java");
    // Create a new KeyValue type
    KeyValue keyValue = (KeyValue)KeyInfo.getInstance("KeyValue");
    keyValue.setKey(verificationKey);
    KeyInfo[] keyInfos = {keyValue};
    // Add it to an XMLSignature instance
    XMLSignature xmlSig = new XMLSignature();
    xmlSig.setKeyInfos(keyInfos);
  }
}
```

the salient points of this example, the creation of a `JSAFE_PublicKey` instance from the raw encoded public key and the interface between the `JSAFE_PublicKey` object and the `KeyValue` type.

KeyValue and an X.509 Certificate

Although an encoded public key is certainly more pervasive than raw public key data, an X.509 certificate is far more likely to be used as a source of verification information. Certificates represent the chief authentication method in a traditional PKI. Perhaps we would like to include a public key from a certificate inside a `<KeyValue>` element; that is, our verification key is *inside* an X.509 certificate, and we want to use it for our verification material. The reader may begin thinking about the `<X509Data>` element that has been previously mentioned. We are not yet discussing this element, which implicitly includes more trust-based information. There is a subtle but important distinction here. We are merely trying to use a verification key inside a certificate to verify a signature. We are not considering a trust assertion about the certificate yet. In this example, the X.509 certificate isn't being viewed as a trustable entity, only as a convenient container for a public key (which is often the case).

Let us proceed by assuming that we have an X.509 certificate and want to create a `KeyValue` type that includes the public key from this certificate. All X.509 certificates contain a `SubjectPublicKeyInfo` object (this is the same as the binary blob used in Listing 8-6). Our goal is to reduce the problem of extracting the `SubjectPublicKeyInfo` object (which contains a `SubjectPublicKey`) from the certificate. Once we have this binary blob, we can proceed down the same path as Listing 8-6. Listing 8-7 shows how this can be done using Cert-J.

The careful reader may have noticed that we did not actually extract the encoded key from the certificate; instead, we used a shortcut accessor function called *getSubjectPublicKey()* that returns a `JSAFE_PublicKey` type. This code assumes that we are working with the `JSAFE_PublicKey` type; there is also a function called *getSubjectPublicKeyBER()* that will return the raw encoded key in the standard format defined by RFC 2459. Once we have the `JSAFE_PublicKey` type, we can create a `KeyValue` type and use this as a `<KeyInfo>` child element.

At this point, we have seen a modest amount of code, and it is likely that a small summary of what we are aiming at is useful. Listings 8-5, 8-6, and 8-7, have been centered around the `<KeyValue>` child element of

```
import com.rsa.certj.cert.X509Certificate;
import com.rsa.jsafe.*;
import java.io.*;

class CodeListing87 {
  public static void main (String args[]) throws Exception {

    // Read the file containing the X.509 certificate into a byte
    array
    File certFile = new File("x509.cer");
    long fileLength = certFile.length();
    FileInputStream certStream = new FileInputStream(certFile);
    byte[] certBER = new byte[(int)fileLength];
    certStream.read(certBER);
    certStream.close();
    // Create an X509Certificate instance from a byte array
    X509Certificate x509Cert = new X509Certificate(certBER,0,0);
    // Get the public key as a JSAFE_PublicKey object (shortcut)
    JSAFE_PublicKey publicKey =
x509Cert.getSubjectPublicKey("Java");
  }
}
```

<KeyInfo>. The goal with these code listings is to show practical ways of creating a KeyValue type from common (or possible) sources of verification key material. We have not yet begun discussion on how (or if) we trust this key material. Figure 8-4 shows a pictorial view of the three previous code listings and discussion.

The X509Data Type

The X509Data type is the most complex KeyInfo child that is also tightly coupled with a standard PKI. The X509Data type needs to be able to include various X.509 fields and objects. Each of the possible X509Data child elements (with the exception of X509CRL) serves to uniquely identify an X.509 certificate. Once the certificate itself has been identified, the key inside can be used to verify the signature. It is extremely important to note that any signature with a valid key will verify properly; this means that unless *trust* is asserted over the key or certificate somehow, the signature is meaningless. Because X509Data resolves (in most cases) to a X.509 certificate, the obvious thing to do once the certificate has been retrieved is to perform path validation and make a trust assertion over the certificate. Path validation may be simple and may only include chain-

Figure 8-4

Possible sources
for a `KeyValue`
type

```
X.509 Certificate


  Encoded Key
  (SubjectPublicKeyInfo)
```

```
Encoded Key
(SubjectPublicKeyInfo)
```

```
<KeyInfo>
  <KeyValue>
  ...
  </KeyValue>
</KeyInfo>
```

```
Raw Key Data
Modulus+Exponent
```
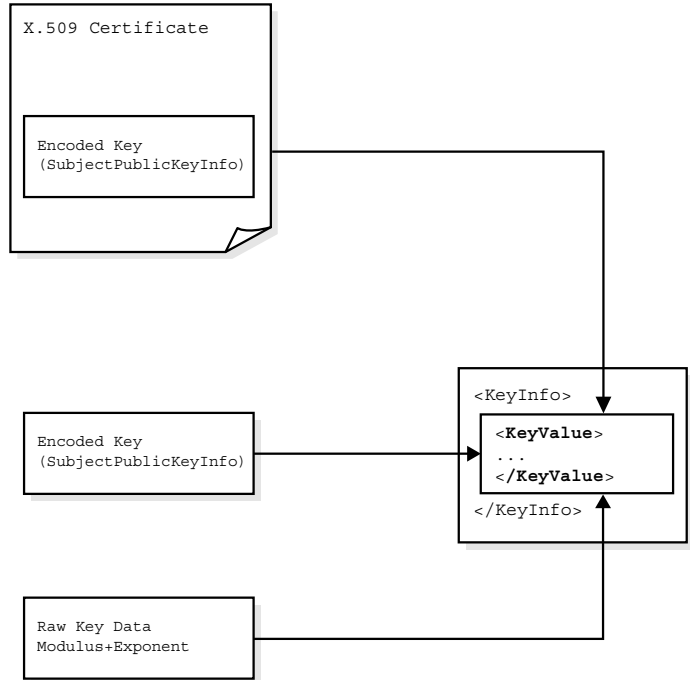
ing to trusted root certificate, or it may become more involved and perform status checks to ensure the certificate has not been revoked and can still be trusted. For more information on path validation, see the primer in Chapter 2. Sample code that shows how XML Signature verification is connected to path validation in Cert-J is shown in a later section. There are four child elements of X509Data that uniquely identify a certificate; these include `<X509IssuerSerial>`, `<X509SubjectName>`, `<X509SKI>`, and `<X509Certificate>`. The last element is a rather vacuous case because it *is* a certificate. Listing 8-8 shows how to create an X509Data type that contains all four of these child elements.

Listing 8-8

Creating an
<X509Data>
element with the
X509Data type

```
import com.rsa.certj.xml.*;
import com.rsa.certj.cert.*;
import com.rsa.certj.cert.extensions.*;
import java.io.*;
class CodeListing88 {
  public static void main (String args[]) throws Exception {
    // Read the file containing the X.509 certificate into a byte
    array
    File certFile = new File("testcert.cer");
    long fileLength = certFile.length();
    FileInputStream certStream = new FileInputStream(certFile);
    byte[] certBER = new byte[(int)fileLength];
    certStream.read(certBER);
    certStream.close();
    // Create an X509Certificate instance from a byte array
    X509Certificate x509Cert = new X509Certificate(certBER,0,0);
    // Create a new X509Data type using the factory constructor
    X509Data x509Data = (X509Data)KeyInfo.getInstance("X509Data");
    // Set the X509IssuerSerial
    x509Data.setX509IssuerSerial(x509Cert.getIssuerName(),
    x509Cert.getSerialNumber());
    // Set the X509SKI (Subject Key Identifier)
    X509V3Extensions extensions = x509Cert.getExtensions();
    // Find the Subject Key Identifier
    SubjectKeyID subjectKeyId =
(SubjectKeyID)extensions.getExtensionByType(X509V3Extension.SUBJECT
_KEY_ID);
    // Add the Subject Key Identifier to the <X509Data> element
    x509Data.setX509SKI(subjectKeyId);
    // Set the Subject Name
    x509Data.setX509SubjectName(x509Cert.getSubjectName());
    // Set the entire certificate
    X509Certificate certs[] = { x509Cert };
    x509Data.setCertificates(certs);
    // Now add the the X509Data type to an XMLSignature instance
    XMLSignature xmlSig = new XMLSignature();
    KeyInfo[] keyInfos = {x509Data};
    xmlSig.setKeyInfos(keyInfos);
  }
}
```

Listing 8-8 begins by assuming that we have an arbitrary certificate
(testcert.cer) that contains a SubjectKeyIdentifier extension. All
standard X.509 certificates must have an issuer name and a serial num-
ber and most will have a subject name field. The code begins by creating
an X509Certificate object and from here we extract the necessary
fields (issuer name, serial number, subject key identifier, and subject
name). Finally, the entire certificate itself is added. All of the fields are
then added to the X509Data type and included in the <KeyInfo> parent
with a call to setKeyInfos(). Once the signature is created, the <Key-
Info> element will appear as follows:

```
<KeyInfo>
    <X509Data>
      <X509IssuerSerial>
       <X509IssuerName>CN=Super Entity,ST=California,C=US</X509IssuerName>
       <X509SerialNumber>5293E647FD80DA8A86A02B4C2226A0F6</X509SerialNumber>
      </X509IssuerSerial>
      <X509SubjectName>CN=Super
      Entity,ST=California,C=US</X509SubjectName>
      <X509SKI>01020202020202020202020202020302020202</X509SKI>
      <X509Certificate>
        MIICKDCCAZGgAwIBAgIQUpPmR/2A2oqGoCtMIi
        ag9jANBgkqhkiG9w0BAQUFADA5MQswCQYDVQQG
        EwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEVMB
        MGA1UEAxMMU3VwZXIgRW50aXR5MB4XDTAxMDkw
        NDA2NTkwOFoXDTAyMDkwNDA2NTkwOFowOTELM
        AkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3mb3
        JuaWExFTATBgNVBAMTDFN1cGVyIEVudGl0eTC
        BnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
        ma4+Q/PmdPK2y9yLgAKltZe/8CKrtEBZ3nn5m
        eu9w+JCVRghgpB0VebZ9Ok55CXTa1UsGjc0Fq
        8wb7iZoaYdTOuciKZg7E+b3Ez0Z+yShPlUy8k
        vPyTaNOjVeN0Z5HgwOTorS08pWdYb+649LcWy
        Pj5HJN/hl0xEMXtEQuzcJzMCAwEAAaMxMC8wD
        gYDVR0PAQH/BAQDAgSQMB0GA1UdDgQWBBQBAg
        ICAgICAgICAgICAgIDAgICAjANBgkqhkiG9w0
        BAQUFAAOBgQASWO6Tgz9EZu1OcGBWlWOPFj6d
        ez3st8UCUsU8DhPcMMItH5MGHfIgMR6dPlwdz
        SegTFFt77KsfqtNUMJNYom5cbjCqF2I1eHGzh
        2X36HOQwIjB/DAJGzXaTZWynCNZnqJrprU2tM
        f3/yKlMuzXflSIzhl5zWOJSoz5Ff4bMbY2g==
      </X509Certificate>
    </X509Data>
</KeyInfo>
```

All possible X509Data child types are shown in the previous example, but not all are actually required. Because any single X509Data child element uniquely identifies a certificate, a single such child element is sufficient. The choice of child elements inside the <KeyInfo> parent is application-specific and is strongly related to the existing PKI infrastructure used.

RetrievalMethod

The final KeyInfo child type implemented by Cert-J is the Retrieval-Method type. This type represents the <RetrievalMethod> element as specified in the XML Signature Recommendation. This child element is used to obtain verification information from an external source. For example, if the certificate containing the verification key resides on a remote server, a RetrievalMethod type would be useful for identifying the location of this certificate. In addition to conveying location information, the

RetrievalMethod element shares some of the properties of the `<Reference>` element. Specifically, it is possible to add a `<Transforms>` child element to a `<RetrievalMethod>` such that transformations (such as decoding) can occur if the verification information is stored in a different format. RetrievalMethod also implements a typing scheme that identifies (using a URI identifier) the specific KeyInfo child that is pointed to. The URI identifier appears as the value of the type attribute of the `<RetrievalMethod>` element. The possible identifiers defined by the XML Signature Recommendation are listed in Table 8-2.

The careful reader may have noticed that the `<KeyValue>` type is absent from Table 8-2. Instead, two other types are present: DSAKeyValue and RSAKeyValue. These child elements are direct children of `<KeyValue>` and give the application more specific information about what type of key is being processed. Instead of a just a key, we know if it is a DSA key or an RSA key. One useful application of RetrievalMethod is to use it to reference a raw X.509 certificate that contains the verification key. Listing 8-9 shows how to use the factory method to create a RetrievalMethod type.

Once the signature has been generated, the resulting `<KeyInfo>` element appears similar to the snippet of XML shown in the following example. The URI given is fictional; in practice, the URI location should contain the X.509 certificate in a raw binary format (such as non-encoded). For more information on certificate formats, see the primer in Chapter 2.

Table 8-2	RetrievalMethod Type	URI Identifier
Retrieval- Method Identifiers	DSAKeyValue	`http://www.w3.org/2000/09/xmldsig#DSAKeyValue`
	RSAKeyValue	`http://www.w3.org/2000/09/xmldsig#RSAKeyValue`
	X509Data	`http://www.w3.org/2000/09/xmldsig#X509Data`
	PGPData	`http://www.w3.org/2000/09/xmldsig#PGPData`
	SPKIData	`http://www.w3.org/2000/09/xmldsig#SPKIData`
	MgmtData	`http://www.w3.org/2000/09/xmldsig#MgmtData`
	Raw X.509 Certificate	`http://www.w3.org/2000/09/xmldsig# rawX509Certificate`

Listing 8-9

Using
Retrieval-
Method to point
to a raw X.509
certificate

```
import com.rsa.certj.xml.*;
class CodeListing89 {
  public static void main (String args[]) throws Exception {
  // Create an instance of RetrievalMethod
  RetrievalMethod rMethod =
  (RetrievalMethod)KeyInfo.getInstance("RetrievalMethod");
  // Set the type of <KeyInfo> we are referencing

rMethod.setType("http://www.w3.org/2000/09/xmldsig#rawX509Certificate");
  // Set the URI to where our raw X.509 certificate lives
  rMethod.setURI("http://www.rsasecurity.com/x509.cer");
  }
}
```

```
<KeyInfo>
    <RetrievalMethod
    Type="http://www.w3.org/2000/09 /xmldsig#rawX509Certificate"
    URI="http://www.rsasecurity.com/x509.cer"/>
</KeyInfo>
```

Other useful scenarios for RetrievalMethod include referencing a chain of certificates that may be long or cumbersome to transport. In this case, RetrievalMethod is initialized with the <X509Data> type, and the <X509Data> element contains multiple children that uniquely identify a certificate chain. This could be a set of <X509Certificate> elements or any other <X509Data> child types that uniquely identify a certificate: <X509IssuerSerial>, <X509SubjectName>, or <X509SKI>.

Custom <KeyInfo> Types

One of the powerful features of the XML Signature Recommendation is the ability to add customized <KeyInfo> child elements for application-specific purposes. Cert-J enables the user to implement custom KeyInfo types with the KeyInfoUtils class. This class boasts a single function called *addKeyInfoClass()* that adds a user-defined KeyInfo type to the internal hash table that stores all of the supported KeyInfo types. As a simple example, we will implement the <KeyName> child element as defined in the XML Signature Recommendation. The idea here is to extend the KeyInfo abstract class and implement all of the functions necessary for the useful operation of <KeyName>. This case is almost trivial because the <KeyName> child element is nothing more than a string identifier for a key. There are no special constraints on the string; the idea

here is to have a way to communicate a flippant but useful string identifier that our recipient knows how to map to a real key. The mapping of the string value to a key is application specific and does not necessarily rely on any existing standard.

In addition to extending `KeyInfo` to enable us to represent a `<KeyName>` element, we also have to implement functions that enable the verifier (if the verifier is also using Cert-J) to automatically parse the `KeyName` child element and obtain a real verification key. There are two problems that need to be solved; the first is informing Cert-J how to *represent* a `<KeyName>` element, and the second is informing Cert-J how to *obtain* the actual verification key associated with the `<KeyName>` string value. Both problems are solved with a single child subclass of `KeyInfo`. Figure 8-5 shows a pictorial diagram of what we are trying to do.

Perhaps the most innocuous step in Figure 8-5 is Step 2. This doesn't say much; what is meant here is that we need to make our `KeyName` type work within the confines of Cert-J. The `KeyName` type holds a string value that will eventually map to a public key. This means at the very least that our implementation will need a way to set a string value and get a string value. The idea is to hook our `KeyName` type into Cert-J such that we can use it just like the other `KeyInfo` types we have discussed (`KeyValue`, `X509Data`, and `RetrievalMethod`). An example code snippet of what we might want out of `KeyName` is shown in the following example:

```
// Create a KeyName KeyInfo type
KeyName keyName = (KeyName)KeyInfo.getInstance("KeyName");
// Set the name of the KeyName
keyName.setKeyName("Dales Key");
Furthermore, the actual signature, when generated, should have a
<KeyInfo> element as follows:
<KeyInfo>
  <KeyName>Dales Key</KeyName>
</KeyInfo>
```

It is expected that the verifying application will know what the string Dales Key means and will be able to trust that this string value maps to a viable verification key. In addition to a function that will store and retrieve a string value, we need to properly implement the rest of the `KeyInfo` abstract functions. We can tell which functions we need to implement by getting a list of the abstract functions with the **javap** command (I have removed the throws clauses and fully qualified class names for readability). This list is shown in the following example:

```
public abstract int getKeyInfoType();
public abstract String getKeyInfoName();
```

```
public abstract boolean hasKey()
public abstract boolean hasCertificate()
public abstract Certificate[] getCertificates(CertJ)
public abstract JSAFE_PublicKey getKey()
public abstract void setKey(JSAFE_PublicKey)
public abstract void setCertificates(Certificate[])
protected abstract Element generateKeyInfo(Document)
protected abstract void parseKeyInfo(Element)
```

All of the functions are simple accessor functions except for the last two shown in bold. The `generateKeyInfo()` function is used to create the actual `<KeyName>` element. This shouldn't be too difficult—`<KeyName>` is just an `org.w3c.dom.Element` with a single text child element. The `Document` input argument is used to ensure that the `org.w3c.dom.Element` is created under the proper parent `org.w3c.dom.Document`.

Figure 8-5

The process for adding a new `KeyInfo` type

Step 1: Create a `KeyName` type that extends `KeyInfo`.
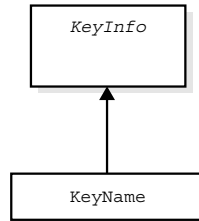
Step 2: Implement/Add functions to make `KeyName` useful.

Step 3: Add the new `KeyName` class to the list of available `KeyInfo` types using `KeyInfoUtils`.
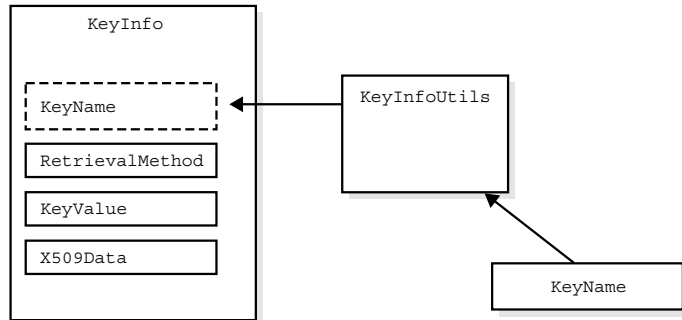
The `parseKeyInfo()` function is used to map the string to a verification key and will be called during signature verification. Its job is to take a `<KeyName>` element and decide how to map it to a real key. This is where our application-specific code will live. Conceptually, our `<KeyName>` element doesn't *have* a key because it is just a string, but the design of the toolkit is such that every `KeyInfo` type must somehow resolve to a key. This is one of the defining properties of the `<KeyInfo>` element in general. This is why the `KeyInfo` types have accessor functions for a `JSAFE_PublicKey` and/or an array of certificate objects. Eventually, a key or certificate containing a key must be added to the object so the signature can be verified. The fully implemented `KeyName.java` class is shown in Listing 8-10.

Listing 8-10

The fully implemented `KeyName` Type

```
/* KeyName.java
   This class implements the KeyName child element of KeyInfo
   as described in the XML Signature Recommendation
*/
import com.rsa.certj.cert.*;
import com.rsa.jsafe.*;
import com.rsa.certj.*;
import com.rsa.certj.xml.*;
import org.w3c.dom.*;
import java.io.*;
public class KeyName extends KeyInfo
  implements Cloneable, Serializable {
  // This is the key that <KeyName> resolves to
  private JSAFE_PublicKey publicKey = null;
  // This is the actual KeyName
  String keyName = null;
  /* This function Is used to retrieve the certificates */
  public Certificate[] getCertificates(CertJ certJ) {
    /* No certificates in this KeyInfo object! */
    return null;
  }
  /* We can retrieve the resolved key */
  public JSAFE_PublicKey getKey() {
    return this.publicKey;
  }
  /* Here is the name of our KeyInfo type */
  public String getKeyInfoName() {
     return "KeyName";
  }
  /* This is the type of KeyInfo class we have */
  public int getKeyInfoType() {
    return KeyInfo.KEYNAME_KEYINFO;
  }
  /* Check and see if this KeyInfo type has certificates */
  public boolean hasCertificate() {
    // No certificate here!
```

(continues)

Listing 8-10 (cont.)

The fully implemented KeyName Type

```
      return false;
    }
    /* Check and see if this KeyInfo type has a key */
    public boolean hasKey() {
      return true;
    }
    /* Initialize this KeyInfo type with certificates */
    public void setCertificates(Certificate[] certificates)
      throws XMLException {
          throw new XMLException ("KeyName should not contain
certificates");
    }
    /* Initialize this KeyInfo type with a public key */
    public void setKey(JSAFE_PublicKey publicKey)
      throws XMLException {
       this.publicKey = publicKey;
    }
    /* This is the accessor function for initializing the name */
    public void setKeyName(String keyName) {
      this.keyName = keyName;
    }
    /* This is our accessor function for retrieving the name */
    public String getKeyName() {
      return keyName;
    }
    /* This function parses the KeyName and calls MapKeyName() */
    protected void parseKeyInfo (Element keyInfoNode) throws
    XMLException {
      // Check to make sure the Element is not empty
      if (keyInfoNode == null)
       throw new XMLException ("KeyName is empty");
      String keyNameValue =
      keyInfoNode.getFirstChild().getNodeValue();
      try {
        this.publicKey = ExternalApp.MapKeyName(keyNameValue);
      } catch (Exception xp) {
        throw new XMLException("Could not map the key");
      }
    }
    /* This function generates the actual <KeyName> element */
    protected Element generateKeyInfo (Document document)
      throws XMLException {
      if (this.keyName == null)
        throw new XMLException
      ("Error in generating KeyName element: KeyName not set");
      String keyNameString = "KeyName";
      Element keyName = document.createElement (keyNameString);
      keyName.appendChild (document.createTextNode (this.keyName));
      return keyName;
    }
}
```

All of the functions in `KeyName` are trivial except for the last two shown in bold. The very last function, `generateKeyInfo()`, does the job of creating the `<KeyName>` element. The function first checks to see if a string key name has been set (`this.KeyName`). Then, using the Xerces API, the first thing done is the creation of a new `org.w3c.dom.Element` type called `KeyName`. Conceptually, this is the `<KeyName>` element. To add content to this, we create a text node and set the key name (`this.KeyName`) as the value. This element is created under the passed in `org.w3c.dom.Document` that refers to the parent document that Cert-J is using to create the signature. Once the element is created, it is returned, and Cert-J does the rest of the work by placing it correctly in the DOM tree in which the `<Signature>` element is being created.

The penultimate function, `parseKeyInfo()`, has the job of parsing the `<KeyName>` element in an incoming signature (such as a signature to be verified) and mapping the value to a `JSAFE_PublicKey`. The function initiates by checking to see if the node that we are parsing is null; if it is non-null, we go to the first child of the `org.w3c.dom.Node` (an element is also a node) and get the node value. This value is the string value inside `<KeyName>`. After storing the node value in a string object, a call is made to a static function called *MapKeyName()*. This user-defined function is responsible for translating the string value into a `JSAFE_PublicKey`. The `JSAFE_PublicKey` that is returned is used to initialize the publicKey member inside `KeyName`. This member variable will be queried during signature validation and used as a verification key source. Figure 8-6(a) shows a pictorial representation of the generation process; Figure 8-6(b) shows the validation process.

All of the pieces necessary for supporting `KeyName` have been shown except the details of the `MapKeyName()` function. This is the application-specific code that does the mapping to actual keys. The keys can be in a database or in some sort of indexed list, or they can come across a network or a web service. A vacuous case is shown for example purposes in Listing 8-11.

The `MapKeyName()` function shown in Listing 8-11 simply matches the string Dales Key to a filename on the local disk. If no match is found, an exception is thrown. A custom application would probably do a search on a key directory or key database.

Figure 8-6(a)

The creation of
<KeyName>

Step1: The KeyName type is added to the list of possible KeyInfo types.
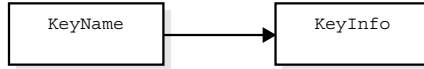
Step 2: A new KeyName type is created.
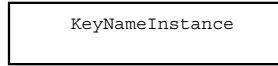
Step 3: The key name is set with setKeyName().
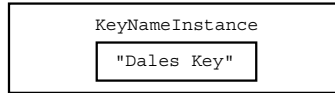
Step 4: During signature generation, the <KeyName> element is created with generateKeyInfo().
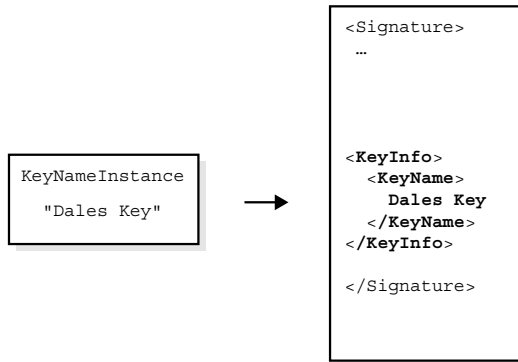
Manifest

The Manifest class is used to represent the <Manifest> element that stores a list of <Reference> elements. Discussion of this class brings with it more discussion about the Reference class. The Manifest class

Figure 8-6(b)

The processing of
`<KeyName>`

Step1: The KeyName type is added to the list of possible KeyInfo types.
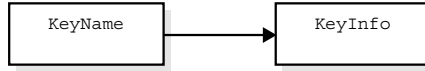
Step 2: A new KeyName type is created.
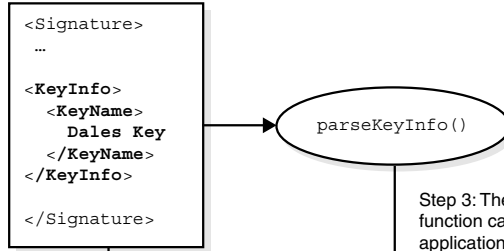
Step 3: The parseKeyInfo function calls an external application that maps the string to a key.
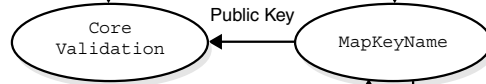
Step 4: The Public Key is used to complete Core Validation.
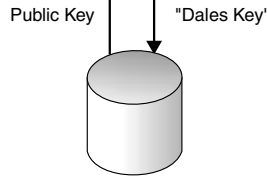
can be confusing at first because it is related to at least two different
`<Reference>` elements. Any given `<Manifest>` element within the
scope of Cert-J will have an associated `<Reference>` element inside
`<SignedInfo>`. This is separate and distinct from the list of `<Refer-
ence>` elements inside the `<SignedInfo>`. Listing 8-12 is shown in
Chapters 4 through 6, but it is shown again here for clarity.

In the interest of clarity, we will assign vocabulary words to the differ-
ent `<Reference>` elements associated with `<Manifest>`. Let the *source*

Listing 8-11

Mapping a name
to an actual key

```
import com.rsa.jsafe.*;
import java.io.*;
class ExternalApp {
  /* This function turns a string key name into an actual key. This
is
     an application specific function */
  public static JSAFE_PublicKey MapKeyName(String keyNameValue)
    throws Exception {
    JSAFE_PublicKey pubKey = null;
    if (keyNameValue.equals("Dales Key")) {
      File keyfile = new File("dale.key");
      long length = keyfile.length();
      byte[] keyData = new byte[(int)length];
      FileInputStream fis = new FileInputStream(keyfile);
      fis.read(keyData);
      fis.close();
      pubKey = JSAFE_PublicKey.getInstance(keyData,0,"Java");
      return (pubKey);
    } else {
      throw new Exception
       ("Could not map key, unknown KeyName" + keyNameValue);
    }
  }
}
```

Listing 8-12

The `<Manifest>`
element and its
relationship to
`<Reference>`
elements

```
<Signature>

  <SignedInfo>
    <Reference URI="#manifest" Id="Source">
    </Reference>
  </SignedInfo>


  <Object>
    <Manifest Id="manifest">
      <Reference Id="Target1" URI=?>
      </Reference>

      <Reference Id="Target2" URI=?>
      </Reference>
    </Manifest>
  </Object>

</Signature>
```

reference be the single `<Reference>` element that refers to a list of *target references* inside a `<Manifest>` element. In Figure 8-7, the *source* reference is the `<Reference>` element whose Id value is `Source`, and the target references are those that have the Id values of `Target1` and `Target2`. With these new terms in hand, we can easily slice apart the `Manifest` class and see how it is implemented within Cert-J. Figure 8-8 shows how the `Manifest` class relates to the `Reference` class.

Figure 8-8 can be confusing; it is telling us that an instance of `Reference` can be composed of zero or more instances of `Manifest`, and an instance of `Manifest` can be composed of one or more instances of `Reference`. `Manifest` is a container for references, but also is used in the creation of a `Reference` element itself. In short, if a `Manifest` is to be used in the creation of an XML Signature, it needs to be added to a source reference. Listing 8-13 shows how to create a set of target references and associate these with a source reference.

There are some additional typing features used by the `Reference` class that become evident when other classes, such as `Manifest`, are employed. Specifically, it is possible to designate the target URI as an `<Object>`, `<SignatureProperties>`, or `<Manifest>`. This is useful for additional application processing semantics. This is seen in Listing 8-13 with the use of the `Reference.MANIFEST_TYPE` identifier. The signature

Figure 8-7

The `<Manifest>` element and its relationship to `<Reference>` elements

```
<Signature>

  <SignedInfo>
    <Reference URI="#manifest" Id="Source">
    </Reference>
  </SignedInfo>


  <Object>
    <Manifest Id="manifest">
      <Reference Id="Target1" URI=?>
      </Reference>

  <Reference Id="Target2" URI=?>
      </Reference>
    </Manifest>
  </Object>

</Signature>
```

Figure 8-8

Manifest class
relationships
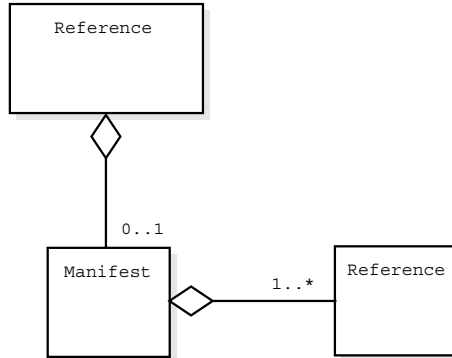
Listing 8-13

Creating an
instance of
Manifest

```
// Import the Crypto-J Classes
import com.rsa.certj.xml.*;
class CodeListing812 {
  public static void main (String args[]) throws Exception {
  /* Create three target references with a URI,
  digest function and empty list of Transformers */
  String hashAlg = "http://www.w3.org/2000/09/xmldsig#sha1";

  Reference target1 =
  new Reference("http://www.rsasecurity.com",hashAlg,null);
  Reference target2 =
  new Reference("http://www.securant.com",hashAlg,null);
  Reference target3 =
  new Reference("http://www.xcert.com",hashAlg,null);
  Reference[] targetReferences = {target1, target2, target3};
  // Create a Manifest with 3 target References and an Id
  Manifest manifest = new Manifest(targetReferences,"manifest1");
  // Now create a source Reference and designate the type
  Reference sourceReference = new
  Reference("manifest1",hashAlg,null,Reference.MANIFEST_TYPE,
  manifest);
  // Now add the source Reference to an XMLSignature instance
  XMLSignature xmlSig = new XMLSignature();
  Reference[] refs = {sourceReference};
  xmlSig.setReferences(refs);
  // Signing code not shown ...
  }
}
```

generated from these source and target reference configurations and `Manifest` is shown in the following example. The source and target references are shown in bold:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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
     Type="http://www.w3.org/2000/09 /xmldsig#Manifest"
     URI="#manifest1">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>1JsfksFOWRWthQt0VQS995SJfOU=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
ROKKGwMgMtXYlUzEZ9ZGv6tsx1Kt9ISjR7M5oPmtjlcJ
cY9amiNuHsQsdL2hiUDtrd6z5QbaxWQwjf1ssZLeng==
  </SignatureValue>
  <Object>
    <Manifest Id="manifest1">
    <Reference URI="http://www.rsasecurity.com">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>tptm5huZKMkSd/Y5FBletU/LtL8=</DigestValue>
    </Reference>
    <Reference URI="http://www.securant.com">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>tptm5huZKMkSd/Y5FBletU/LtL8=</DigestValue>
    </Reference>
    <Reference URI="http://www.xcert.com">
      <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>tptm5huZKMkSd/Y5FBletU/LtL8=</DigestValue>
    </Reference>
    </Manifest>
  </Object>
</Signature>
```

Another point that should be mentioned is the `<Manifest>` element is shown inside an `<Object>` element. The `<Object>` element is created automatically by Cert-J; the user doesn't have to worry about placing this element in the signature tree manually when creating the signature. This case differs from the more general case where an `<Object>` element is used to transport arbitrary data or application-specific signature information such as a `<SignatureProperties>` element or Base-64 encoded data. Both of these cases will be visited in detail in a later

section. The remaining functions that deal with the **Manifest** class are tightly integrated into the core validation processing done by Cert-J. We will save these additional details for the section on creating and verifying signatures.

The <Object> Element

Unlike the `<Manifest>` element, there is no associated Java class in Cert-J for creating an `<Object>` element. The creation of an `<Object>` element for an `XMLSignature` instance is deferred to the Xerces API. The reason for this is quite simple; because of the extensible nature of `<Object>` and its contents, it is hard to write a consistent interface for what might be placed inside this element. Instead, Cert-J uses the `setXMLObjects()` to enable the user to add any number of `org.w3c .dom.Element` objects as children of the main `<Signature>` element. The elements added by this function are placed in a vector and are added as children during the signature generation process. All `Element` instances added with `setXMLObjects()` must have the same parent document.

A useful example for the `<Object>` element is the `<Signature-Properties>` element. The `<SignatureProperties>` element appears within a given `<Object>` as a means to convey signed assertions about a specific XML Signature. Suppose we wanted to add (and sign) the fictitious `<SignatureProperties>` element shown in Chapter 4. The complete set of assertions and `<Object>` parent are shown in the following example:

```
<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>
```

All of the contents of the `<SignatureProperty>` element shown previously are fictional, and the tag names are invented. The way to add this to an XML Signature is to create the DOM representation of this `<Object>` element and then use `setXMLObjects()` to add the structure to an instance of `XMLSignature`. The `<SignatureProperty>` tag shown in bold has the Id value of the parent `<Signature>` as well as its own Id (`FictionalSignatureAssertions`) that will be used as a fragment identifier from within a `<Reference>` element.

Listing 8-13 shows how to create a `<SignatureProperties>` element and the first `<Assertion>` child element. Remember, the `<Assertion>` element is invented and is used for explanatory purposes. If this sort of custom markup is used, it is the application's responsibility to know how to understand and make decisions based on its meaning.

Listing 8-14 probably looks like a mess; the bulk of the confusing code is the use of Xerces to create the `<SignatureProperties>` element and child elements. Only one `<Assertion>` tag is shown in Listing 8-14; trying to fit them all in would only serve to confuse the reader with needless details. This code creates the `<SignatureProperties>` element as well as two `<Reference>` elements. The first `<Reference>` is for the actual electronic check (which in this case is assumed to reside on a remote server). The second `<Reference>` is the source reference for the `<SignatureProperties>` element. This means that our set of signature properties is signed along with the check, providing some protection against alterations. There are a few important things to notice about Listing 8-14. First, the signature type is set to `XMLSignature.DETACHED_SIGNATURE`. This may seem confusing at first, because the actual signature when generated would actually be *both* detached (reference to a remote file) and enveloping (reference to signature properties). This is an example where the classification of XML Signatures into distinct types begins to break down.

Amid the confusion of Listing 8-14, there are a few items shown in bold that are of special importance. The first item is the *setSignatureID() function*. This function is required here because it may be useful to match the Target attribute in the `<SignatureProperties>` element back to the signature that it refers to. The third item shown in bold is the creation of the `DocumentImpl` type. This provides a root `Document` type from which to create the `<SignatureProperties>` element and its children. The next call shown in bold is the *setXMLObjects() call* that takes an array of `org.w3c.dom.Element` types. This call actually initializes the `XMLSignature` with the list of child elements that will be added during the signing process. The following function is the important *setDocument() call*.

Listing 8-14

Adding a
`<Signature-Properties>`
element to
XMLSignature

```
import com.rsa.certj.xml.*;
import java.io.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.DocumentImpl;
class CodeListing813 {
  public static void main (String args[]) throws Exception {
    XMLSignature xmlSig = new XMLSignature();
    xmlSig.setSignatureType(XMLSignature.DETACHED_SIGNATURE);
    // Set the signature type to be used
    xmlSig.setSignatureID("SignedCheckToPaperBoy");
    // First we have to create an <Object> element.
    DocumentImpl documentImpl = new DocumentImpl();
    Element objectNode = documentImpl.createElement("Object");
    // Create the <SignatureProperties> element, Id and Target
    String sigId = "FictionalSignatureAssertions";
    String sigTarget = "SignedCheckToPaperBoy";
    Element sigPropsNode =
    documentImpl.createElement("SignatureProperties");
    sigPropsNode.setAttribute("Id",sigId);
    sigPropsNode.setAttribute("Target",sigTarget);
    // Create the <SignatureProperty> element
    Element sigPropNode =
    documentImpl.createElement("SignatureProperty");
    // Create the <Assertion> element. This is a fictional tag
    Element assertNode = documentImpl.createElement("Assertion");
    // Create the <GenerationTime> element. This is a fictional tag
    Element gtNode = documentImpl.createElement("GenerationTime");
    // Create the text contents of this tag
    Text textNode =
    documentImpl.createTextNode("Mon Jun 11 19:10:27 UTC 2001");
    // Append the text node to <GenerationTime>
    gtNode.appendChild(textNode);
    // Append <GenerationTime> to <Assertion>
    assertNode.appendChild(gtNode);
    // Append <Assertion> to <SignatureProperty>
    sigPropNode.appendChild(assertNode);
    // Append <SignatureProperty> to <SignatureProperties>
    sigPropsNode.appendChild(sigPropNode);
    // Set the XML Objects to be enveloped by <Signature>
    Element[] elems = {sigPropsNode};
    xmlSig.setXMLObjects(elems);
    // Tell Cert-J about the parent Document type
    xmlSig.setDocument(documentImpl);
    // Reference element for the check
    Reference checkRef = new Reference();
    checkRef.setURI("http://somelocation.com/check.txt");
    checkRef.setDigestMethod("http://www.w3.org/2000/09/
    xmldsig#sha1");
    // Reference element for the <SignatureProperties>
    Reference sigPropsRef = new Reference();
    sigPropsRef.setURI("#FictionalSignatureAssertions");
    sigPropsRef.setDigestMethod("http://www.w3.org/2000/09
    /xmldsig#sha1");
    Reference[] refs = {checkRef,sigPropsRef};
    // Actually add the Reference element.
    xmlSig.setReferences(refs);
  }
}
```

This function initializes the XMLSignature with the proper root `Document` type that will be used when generating the signature. Failure to make this call with the correct `Document` type will result in an exception thrown by Cert-J. The final calls shown in bold comprise the creation of the `Reference` that points to the `<SignatureProperties>` element. The `setURI()` call shows how a fragment identifier is used to point to the ID attribute of the `<SignatureProperties>` element. The value used here is `#FictionalSignatureAssertions`. The last thing to note here is that Listing 8-14 is not a complete signature. It is missing the designation of the canonicalization algorithm and signature method, as well as the code that actually does the signing operation. If this code were completed, the output signature (pictorial view) would look something like Figure 8-9.

The arrows in Figure 8-9 show what is signed. In this case, two entities are signed, an external piece of data (the electronic check) and the enveloped `<SignatureProperties>` element.

Signature Processing

At this point, we have covered all of the important classes for initializing and creating the syntax of an XML Signature. We have yet to see how to

Figure 8-9

A pictorial view of the signature generated with code from Listing 8-14
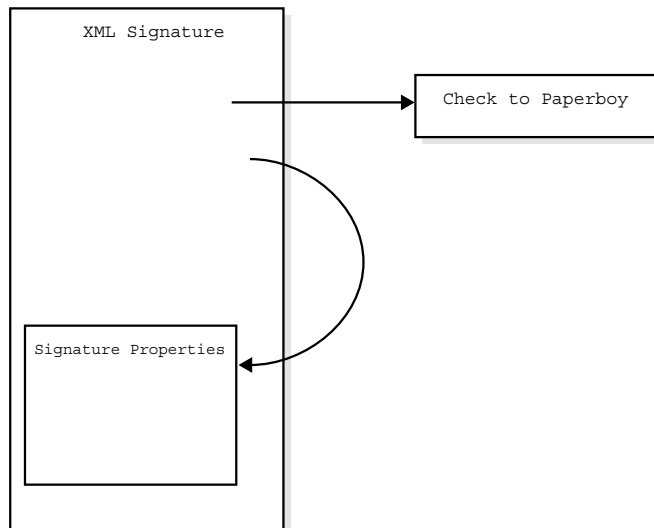
actually create an XML Signature. The previous classes are primarily used to build a structure in memory that is going to be signed. This next section covers the classes necessary for actually creating and writing signatures to a useful location such as a file or stream. For those readers who just want to sign something and just want to verify something, the next two examples are the first fully functional code examples that will actually create and verify a digital signature. First we will show the code, and then we will talk about how the processing and verification actually works (see Listing 8-15).

One of the most overlooked entities at this point has been the actual *signing key*. An XML Signature cannot be created without a cryptographic key of some sort, whether it is a symmetric key for use in an HMAC or an RSA or DSA key. For the previous example, code is shown that uses the Crypto-J toolkit to generate a new RSA key pair at run time. The previous code can be considered example code and is not really secure for a real application because the seeding method shown is extremely poor (the date is used). In reality, a good source of randomness should be used to seed the random number generator. More about random number generators and seeding can be found in Chapter 2. The first thing done in Listing 8-15 is to create the key pair. This is done with a static function called *createKeys()* shown in bold. This function does the job of initializing the two static objects that represent the private key and the public key (JSAFE_PrivateKey and JSAFE_PublicKey). Most of the gory details regarding the key generation are going to be skipped here; they are fully explained in the Chapter 2 primer.

Listing 8-15

Complete XML Signing example

```
import com.rsa.certj.xml.*;
import com.rsa.jsafe.*;
import java.util.*;
import java.io.*;
class CodeListing814 {
  private static JSAFE_PrivateKey privateKey = null;
  private static JSAFE_PublicKey publicKey = null;
  public static void main (String args[]) throws Exception {
    // First create the keys
    createKeys();
    // Create the XMLSignature instance
    XMLSignature xmlSig = new XMLSignature();
    xmlSig.setSignatureType(XMLSignature.DETACHED_SIGNATURE);
    xmlSig.setSignatureMethod("http://www.w3.org/2000/09/
    xmldsig#rsa-sha1");
    xmlSig.setCanonicalizationMethod
    ("http://www.w3.org/TR/2001/REC-xml-c14n-20010315");
```

(continues)

Listing 8-15 (cont.)

Complete XML Signing example

```
    // Create a reference to a URI
    Reference ref = new Reference();
    ref.setURI("http://www.rsasecurity.com");
    ref.setDigestMethod("http://www.w3.org/2000/09/xmldsig#sha1");
    Reference[] refs = {ref};
    xmlSig.setReferences(refs);

    // Add the public key so we can verify the signature
    KeyValue kv = (KeyValue)KeyInfo.getInstance("KeyValue");
    kv.setKey(publicKey);
    KeyInfo keyInfos[] = {kv};
    xmlSig.setKeyInfos(keyInfos);

    // Make sure the DTD doesn't show up for now
    ParserHandler.DTD_LOCATION = null;

    // Perform the signing operation and write the signature
    to the screen.
    xmlSig.sign(privateKey,null,"Java");
    ParserHandler.write(System.out,xmlSig);
}
 private static void createKeys() {
   JSAFE_KeyPair keyPair = null;
   JSAFE_SecureRandom random = null;
  try {
     System.out.println("Generating RSA key pair.");
     random = (JSAFE_SecureRandom)
     JSAFE_SecureRandom.getInstance ("SHA1Random", "Java");
     random.seed (new Date().toString().getBytes());
     keyPair = JSAFE_KeyPair.getInstance("RSA", "Java");
     int[] keyGenParams = { 512, 17 };
     System.out.println ("Modulus size:  " + keyGenParams[0]);
     System.out.println ("Public Exponent:  " + keyGenParams[1]);
     keyPair.generateInit( null, keyGenParams, random);
     System.out.println ("Generating the RSA keypair...");
     keyPair.generate();
     privateKey = keyPair.getPrivateKey();
     publicKey = keyPair.getPublicKey();
     keyPair.clearSensitiveData();
   } catch (Exception anyException) {
     System.out.println ("Exception caught while generating
keys");
     System.out.println ( anyException.toString());
   }
  }
}
```

Before the signing operation takes place, there is an odd flag being set on the ParserHandler class. The flag is DTD_LOCATION, and it is being set to null. When the signature is verified, some recipients may want to validate the structure of the XML Signature and check to see if it is *valid.* Here the term *valid* does not refer to the authenticity of the signed data,

but refers to *valid XML*. The choice of vocabulary words is unfortunate. For now, we will set it to null; this means that Cert-J will not include a *document-type declaration* in the output XML. For most applications, this will probably be acceptable, but some recipients require this declaration as well as a valid path to the actual document type definition (DTD). For more information on DTDs, consult the primer in Chapter 3.

Once we have a JSAFE_PrivateKey object in hand, we can use the sign() function (shown in bold in Listing 8-15) to actually create the signed XML structure in memory. The null second argument is for a random object that is only required during DSA signature generation. The third argument is a string value that represents the device setting. When the device setting is set to Java, we are telling Cert-J to attempt the signing operation using only software calls. Once the sign() function has been called, the ParserHandler class is required to actually write the signature to an OutputStream or file. In the simple case of Listing 8-15, we are simply writing to System.out. The second argument is the actual XMLSignature instance that we would like to send to an OutputStream. If we had multiple signatures, we would need to use the Parser-Handler.write() function multiple times, once for each signature to generate. This previous description is a bare-bones case of how signing works in Cert-J. Signature verification is quite similar and possibly even simpler because there is no need to generate a key pair. One thing to note about Listing 8-15 is that a <KeyInfo> element is added to the signature, making it easy to verify. If no <KeyInfo> element is present, the application must do the necessary work to obtain the verification key. Listing 8-16 shows how to verify an XML Signature stored in a file.

Listing 8-16 is quite subtle and fairly restrictive on what can be used as input, but it does the job of verifying an XML Signature stored in a file. Some assumptions are made; the first is that the input file contains a single XML Signature with a <KeyInfo> element containing a verification key. ParserHandler.read() is called and reads from an input file, returning an array of possible XMLSignature instances. This call returns as many <Signature> elements found in the input file. In our simple case, we assume that the <Signature> to verify is the first element in this array. A more practical example consists of looping through the array and verifying each signature found.

Next, the verification key is removed from the XMLSignature instance with getKeyInfos(). This call returns a list of all the KeyInfo types found inside the given XMLSignature instance. Although there can be only one <KeyInfo> element in an XMLSignature, this element may contain many KeyInfo types (such as a key, certificate, subject name, and so

```
import com.rsa.certj.xml.*;
import com.rsa.jsafe.*;
import java.io.*;
class CodeListing815 {
  public static void main (String args[]) throws Exception {
    // Assume the first argument contains a String filename
    XMLSignature[] sigs = ParserHandler.read(args[0]);
    // Assume the first signature is the one we want to verify
    XMLSignature sig1 = sigs[0];

    // Assume the signature has a <KeyInfo> element
    KeyInfo[] keyInfos = sig1.getKeyInfos();
    JSAFE_PublicKey publicKey = keyInfos[0].getKey();

    VerificationInfo info = sig1.verify(publicKey,"Java");
    // Find out what happened
    if (info.getStatus() == VerificationInfo.CORE_VERIFY_SUCCESS) {
      System.out.println("Core Validation Succeeded");
    } else {
      System.out.println("Core Validation Failed");
    }
  }
}
```

on). Once the list is obtained, the first element in the list is assumed to contain a verification key. This verification key is retrieved with getKey() and stored in a JSAFE_PublicKey object. The careful reader might ask the question: Why can't Cert-J automatically use the <Key-Info> element provided? Why must we manually extract the key and give something back to the toolkit that it presumably already has? The answer here lies in the fact that <KeyInfo> may contain many different types and not all of them may contain an actual key (some KeyInfo types serve to *identify* a key via X.509 entities). Additional application processing might have to be done (such as to a directory server) that will eventually provide the verification key.

Once we have the JSAFE_PublicKey object, we can use it directly with the XMLSignature.verify() method. This method is very similar to the sign() method shown previously, it takes a key and a device setting. In this case, we are going to perform verification with software and use the Java string.

The object returned after the verify() call is a VerificationInfo object. This object stores the result of what happened during signature verification. There are a number of static variables defined inside VerificationInfo that give the application more information about the granularity of signature validation. That is, Cert-J divides *core validation*

into two possible steps: *reference validation* and *signature validation*. The set of possible return values and their meaning are shown in Table 8-3.

According to the processing model defined by the XML Signature Recommendation, a signature fails verification if *core validation* fails. The previous constants enable an application to proceed under other circumstances, specifically when reference generation or signature generation fails. This might not always be a secure thing to do, but an application can make this decision when and if it needs to if it has enough contextual information.

Once a `VerificationInfo` object is generated, the call to `getStatus()` retrieves the integer constant, and a check is made to determine the status of the signature. This check is shown in the last bold line in Listing 8-16. Only the simplest case of core validation is considered here. The next section returns to the discussion of the `Manifest` class and showcases how verification works with this class to provide more information about the Reference objects kept within.

More on Manifest

We have seen in a previous section how to create an XML Signature that contains a `<Manifest>` element. We talked about a source reference that refers to the actual `<Manifest>` element and the list of target references inside the `<Manifest>`. Cert-J is designed such that the `Manifest` object

Table 8-3	**Identifier Name**	**Description**
Core Validation Semantics	`CORE_VERIFY_FAILURE`	Both signature validation and reference validation failed.
	`CORE_VERIFY_SUCCESS`	Signature validation and reference validation succeeded.
	`OTHER_FAILURE`	Some other failure occurred, such as unable to de-reference a URI or an unsupported algorithm.
	`REFERENCE_VALIDATION_FAILURE`	Signature validation succeeded, but reference validation failed.
	`SIGNATURE_VERIFICATION_FAILURE`	Reference validation succeeded, but signature validation failed.

must be accessed through a source reference. Any code that attempts to verify a `<Manifest>` should search through the `<Reference>` elements in the signature and find one that refers to a `<Manifest>` element. An easy way of doing this is to rely on the `Type` attribute of the `<Reference>` element. Listing 8-17 shows some example code that attempts to verify a `<Manifest>` in a given XML Signature.

The first thing done in Listing 8-17 is to read from a file and create a list of `XMLSignature` instances. Each instance in the list corresponds to a `<Signature>` element in the source file. In this sample, we are assuming that the input file only contains a single `<Signature>` element. The next thing done is similar to Listing 8-16. We extract the verification key (and make the assumption that one exists). From here we verify the signature using the `JSAFE_PublicKey` inside the provided `<KeyInfo>` element. This concludes the core validation process thus far.

Once the source reference is located, the next thing done is to count the number of `Reference` objects stored inside the `Manifest`. Once the number of `Reference` objects has been determined, a `StringBuffer` must be initialized for each `Reference` object. This `StringBuffer` object stores the descriptions of what happened once the `Manifest` is verified. After the `StringBuffer` objects have been initialized, the actual `verify()` method is called. Note that this method is called on the `Reference` object, the *source reference*, and not a `Manifest` object as one might think. The `Manifest` is accessed and verified through the source reference. The descriptions will not get printed unless the `<Manifest>` element fails completely, that is, at least one `<Reference>` element inside the `<Manifest>` element fails the digest value check. The output might look something like this if all of the `<Reference>` elements failed the digest check:

```
Verified the signature
Digest value not matched
Digest value not matched
Digest value not matched
```

Additional Classes

The classes covered in depth thus far include the syntax-based classes (`XMLSignature`, `Reference`, `Transformer`, `KeyInfo`, and `Manifest`) and the processing-based classes (`ParserHandler` and `VerificationInfo`). Although these classes represent the most important and visible

Listing 8-17

Verifying a
Manifest

```java
import com.rsa.certj.xml.*;
import com.rsa.jsafe.*;

class CodeListing816 {

  public static void main (String args[]) throws Exception {

    XMLSignature xmlSigs[] = ParserHandler.read(args[0]);
    // Verify the first signature
    XMLSignature sig = xmlSigs[0];
    // Get a key
    KeyInfo keyInfos[] = sig.getKeyInfos();
    JSAFE_PublicKey publicKey = keyInfos[0].getKey();
    // Verify the signature
    VerificationInfo vi = sig.verify(publicKey,"Java");
    if (vi.getStatus() == VerificationInfo.CORE_VERIFY_SUCCESS) {
      System.out.println("Verified the signature");
    } else {
      System.out.println("Signature verification failed");
    }
    // Now find the source reference and verify the Manifest....
    Reference refs[] = sig.getReferences();
    int sourceRefIndex = -1;
    for (int k=0; k<refs.length; k++) {
      Reference currentRef = refs[k];
      if
(currentRef.getReferenceType().equals(Reference.MANIFEST_TYPE)) {
        sourceRefIndex = k;
      }
    }
    if (sourceRefIndex == -1) {
      System.out.println("No soure Reference found");
      System.exit(1);
    }
    Reference sourceRef = refs[sourceRefIndex];
    // Figure out how many target References we have
    int numTargets =
    sourceRef.getManifest().getManifestContent().length;
    StringBuffer[] descriptions = new StringBuffer[numTargets];
    for (int j=0; j<descriptions.length; j++)
      descriptions[j] = new StringBuffer();
    // Try to verify the manifest
    boolean manifestResult =
    sourceRef.verifyManifest(sig,descriptions);
    if (manifestResult == true) {
      System.out.println("Verified the manifest");
    } else {
      // We couldn't verify all of the manifest
      for (int i=0; i<descriptions.length; i++)
      System.out.println(descriptions[i]);
    }
  }
}
```

classes in the toolkit, they do not represent all of the classes. There are a few additional helper classes that the user will probably never interact with, but they are listed for information purposes. Table 8-4 lists the remaining classes and a brief description of their function.

RSA BSAFE Cert-J: Specialized Code Samples

This section is devoted to showcasing three code samples that try to solve certain specialized problems. The first sample is called `XMLEnveloping-Binary` and shows how to create an XML Signature that signs some piece of arbitrary binary data and packages the data inside the `<Signature>` element (enveloping signature). The second code sample gives an example of a custom transformation that shows how the Java classes can be used to perform ZIP decompression. Finally, the last example is called `XPathTester` and is useful for viewing an input XML document after an XPath transformation and canonicalization takes place to help see what you sign.

Enveloping Arbitrary Binary Data

One application of XML Signatures is the ability to sign *non-XML data*. This can be done easily with a detached signature, but becomes more complicated with an enveloped or enveloping signature because the binary data must be encoded in a printable format. The other problem in this area is the conflict of the XML 1.0 Recommendation and existing MIME specifications with regards to line endings. The XMLEnvelopingBinary

Table 8-4	Class Name	Description
Additional Helper XMLSignature Classes	Canonicalizer	Stores the details of Canonical XML. The methods in this class should almost never be called directly.
	FunctionHere	Adds the here() function for XPath evaluation.
	SigNodeNameList	Stores the element names defined by the XML Signature Recommendation.
	XMLException	Exception class for exception messages related to XML Signing.

sample uses Crypto-J to omit the carriage return-line feed (CRLF) characters at the end of each line of Base-64 encoded text, thereby sidestepping the problem of line-ending problems when the resulting signature is normalized when read in by the XML Processor. For more information on this particular problem, see the relevant scenario in Chapter 6.

The entire `XMLEnvelopingBinary` sample is simply too large to present in its entirety, and presenting excerpts is also cumbersome because the actual file can be obtained at **www.rsasecurity.com/ go/xmlsecurity**. The best thing to do is a simple discussion of some of the salient points of the sample. First of all, because the sample aims to create an enveloping signature, the data to envelop must somehow be dereferenced (it is assumed to be a URI because this is the standard access mechanism for data in a distributed web environment) and encoded. Once this happens, it needs to be placed in an `<Object>` element. This is done using the Xerces API in much the same way as Listing 8-14. Once this is done, it is now considered to be XML data (start tags, content, and end tags) and can be signed with a *fragment* identifier and a normal `Reference` object. When encoded with Base-64 encoding, the line breaks are set to 0, which means that no CRLF characters will be added to the enveloped data effectively skirting the problem of inconsistent line endings during normalization. Sample output from `XMLEnvelopingBinary` is shown in the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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
      Type="http://www.w3.org/2000/09/xmldsig#Object"
      URI="#object">
      <DigestMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>9g95CeL/92y7rFMGMuLbZkjKljM=</DigestValue>
    </Reference>
  </SignedInfo>
<SignatureValue>qgDWBL9WM+3imJ8ID4EG5yhiGor6PVlFpxNSKx0Ol5EhjkyiyNH
vmczvoYcZ6gLBqNG+Ug7xCy7elrcLDCVuVA==</SignatureValue>
<KeyInfo>
  <KeyValue>
    <RSAKeyValue>
      <Modulus>
        r34zgMQpajMQWtFghKa+YglTcyxeJ7e13Gey
        M6nxIuRmkeXmxc84sp+vpLr1axBjQ5JdnrTkDip9
        ldPiafmPzw==
      </Modulus>
      <Exponent>EQ==</Exponent>
```

```
        </RSAKeyValue>
      </KeyValue>
  </KeyInfo>
  <ObjectId="object">
  eVBjL2Y1dzJRdnpXaTRaUktJTjhCUWs0VUw5PQ==
  </Object>
</Signature>
```

The most notable feature of the previous listing is the last `<Object>` element shown in bold. This element contains the actual enveloped binary data. It is created using Xerces and added to the `XMLSignature` instance before signing occurs. The ID value for this `<Object>` element is arbitrary; the one used in the sample is called *object*.

Custom Transformations

It is possible to add a user-defined transformation algorithm to Cert-J with the use of the `TransformUtils` class. This utility class is used to add a custom transform algorithm to the list of possible transforms stored inside `Transformer`. As an example, we will show here how to add a *decompression* transform that can decompress source files and use their uncompressed form as source data for a signature. The algorithm we will use is ZIP decompression, and the implementation will rely on the `java.util.zip` package that is part of Java.

To create a custom transformation algorithm, one must extend the `Transformer` class and implement the proper methods. There are two steps to implementing a custom transform algorithm. The first step is the creation of the class file that does the work. The second step comprises the calls needed when the transform algorithm is used in the creation of a signature. It is important to realize that because this is a custom transform algorithm, it will not be interoperable with all recipients. A recipient that wants to verify a signature using a custom transformation algorithm must not only have access to the algorithm itself, but must know of the string identifier used to denote this specific custom transform. Listing 8-18 shows the details of the `ZIPDecompressionTransformer` class.

Despite the length of Listing 8-18, there are really only four important functions: Two accessor functions for setting and getting the name of the Transformer and two functions that perform the transformation. The reader should consider the two `performTransformation()` functions first. Notice that the `perfomTransformation()` function that accepts a `NodeList` as input returns null. This is because ZIP decompression is not defined on a node-set. That is, given a set of abstract nodes, there is no

Listing 8-18

The `ZIP-Decompression-Transformer` class

```
import com.rsa.certj.xml.Transformer;
import com.rsa.certj.xml.XMLException;
import java.io.*;
import java.util.zip.*;
import java.util.Vector;
import org.w3c.dom.NodeList;
class ZIPDecompressionTransformer extends Transformer
  implements Cloneable, Serializable {
  private String transformAlgorithm = null;
  protected void setTransformAlgorithm (String transformAlgorithm) {
    this.transformAlgorithm = transformAlgorithm;
  }
  public String getTransformAlgorithm () {
    return this.transformAlgorithm;
  }
  public byte[] performTransformation(byte[] input, int inOffset, int
  inputLength) throws XMLException {
    return (zipDecompress(input));
  }
  public static byte[] zipDecompress(byte[] compressedData) throws
  XMLException {
  // Byte array for returning the uncompressed data.
  byte[] uncompData = null;
  // Buffer used for reading a byte at a time.
  byte[] buffer = new byte[1];
  // Vector for growing the decompressed data
  Vector decompressedSoFar = new Vector();
  // For converting the Vector to an array
  Object[] objectArray = null;
  int ret = 0;
  int bytesDecompressed = 0;
  // Create a ByteArrayInputStream for reading ZIP compressed data
  ByteArrayInputStream baIs = new
  ByteArrayInputStream(compressedData);
  // Create a new ZipInputStream
  ZipInputStream zIn = new ZipInputStream(baIs);
  try {
   ZipEntry ze = zIn.getNextEntry();
   if (ze == null) {
     throw new XMLException
     ("Error, no zip entry. Source is not a zip file");
   } else {
     // Crude way to figure out how many bytes we have decompressed.
     while ((ret = zIn.read(buffer,0,1)) != -1) {
       Byte thisByte = new Byte(buffer[0]);
       decompressedSoFar.add(thisByte);
       bytesDecompressed++;
     }
     zIn.close();
     uncompData = new byte[bytesDecompressed];
     objectArray = decompressedSoFar.toArray();
     for (int i=0; i<uncompData.length; i++)
       uncompData[i] = ((Byte)objectArray[i]).byteValue();;
     }
  } catch (Exception xp) {
     xp.printStackTrace();
     System.out.println
```

(continues)

Listing 8-18 (cont.)

The ZIP-
Decompression-
Transformer
class

```
    ("Caught an Exception while decompressing using ZIP");
  }
  return (uncompData);
}
public NodeList performTransformation(NodeList inputNodes) {
  NodeList nl = null;
  return (nl);
}
}
```

meaningful way to decompress these because the algorithm itself only operates on octets. It follows then that we only need to implement the performTransformation() function that accepts a byte[] array as input. The public performTransformation() algorithm calls down to an internal function called *zipDecompress()* that does the work of calling down to the java.util.zip package to do the actual decompression. Once the appropriate performTransformation() function has been called, the last thing that must be done is to ensure that there is a way to assign a string identifier to the transformation. Once these two tasks are completed, the class is generally ready to be used as a transform in an actual signature. Using the new transformation requires only a few calls shown in the following:

```
Reference ref = << previously initialized Reference object >>
// Make an instance of the ZIPDecompressionTransformer
ZIPDecompressionTransformer unzipper = new
ZIPDecompressionTransformer();
unzipper.setTransformAlgorithm("ZIPDecompression");
TransformUtils.addTransformer(unzipper);
ref.addTransform(unzipper);
```

There are really only two special calls that must be made. The first call assigns an identifier to the transformation algorithm. This is where a URI resides for other transformation algorithms already defined by the XML Signature Recommendation. In this example, the string ZIP-Decompression is used, and this is the identifier that will appear when the signature is generated (shown in the following):

```
<Reference URI="file:///C:\test.zip">
  <Transforms>
    <Transform Algorithm="ZIPDecompression"/>
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09
  /xmldsig#sha1"/>
  <DigestValue>X7sktJA/j4fBnfHg3Wb97i+Tt4M=</DigestValue>
</Reference>
```

The verifier must be able to recognize this specific string (ZIPDecompression) and must know that it refers to actual ZIP decompression. Interoperability problems that arise from failures to understand these string identifiers are a chief reason as to why custom transformation algorithms are likely to be scarce in practice.

XPath Tester

Transforms are very powerful as they are specified in the XML Signature Recommendation. Because it is the transformed data that is actually signed, signers must be extremely aware of what it is that is being signed. Often a string of transformations in the transform waterfall can obscure the final result sent into the digest function. The XPathTester sample enables for the visualization an XPath transformation (along with Canonicalization) to aid implementers and testers in seeing what they sign.

The basic idea behind the XPathTester is two inputs, an input XML document and an input string XPath expression. This sample works differently from other XPath visualization tools because it performs the XPath processing as specified in the XML Signature Recommendation. It converts the result of each expression to a Boolean value and uses this value to determine if the current node should appear in the output node set. This means that a basic expression to obtain an element, such as //ElementName, will not work. This type of expression will be converted to a Boolean for each node, and all nodes will be included; this expression does not do any sort of Boolean test.

The sample relies on protected functions in the ParserHandler class; because of this, it is necessary for the XPathTester class file to subclass ParserHandler. Furthermore, a node-set implementation is also required for the resultant node-set after the transformation is executed. The details of this sample can be found at the web site for this book at **www.rsasecurity.com/go/xmlsecurity**. A sample run is shown in the following. It is important to note that the input parameters should be placed in double quotes on some systems to prevent the shell from misinterpreting the input; XPath expressions can get fairly complex:

```
java XPathTester file:///sample.xml "ancestor-or-self:Karate"
Original Input Data
<?xml version="1.0" encoding="UTF-8"?>
<MartialArts>
  <Aikido Id="FirstElement">
```

```
       <Gradings>
         <Grading1>San Dan</Grading1>
         <Grading2>Ni Dan</Grading2>
         <Grading3>Sho Dan</Grading3>
       </Gradings>
     </Aikido>
     <Karate Id="SecondElement">
       <Gradings>
         <Grading1>Yellow Belt</Grading1>
         <Grading2>Brown Belt</Grading2>
         <Grading3>Black Belt</Grading3>
       </Gradings>
     </Karate>
   </MartialArts>
   *****************************************
   XPath Expression: ancestor-or-self:Karate
   *****************************************
   Transformed Data
   <Karate Id="SecondElement">
       <Gradings>
         <Grading1>Yellow Belt</Grading1>
         <Grading2>Brown Belt</Grading2>
         <Grading3>Black Belt</Grading3>
       </Gradings>
     </Karate>
```

In the previous example, we attempted to target the `<Karate>` element for signing. After testing our XPath expression with `XPathTester` (`ancestor-or-self:Karate`), we can see what the transformed output looks like; this is the actual data that will be signed. XPathTester only works on simple documents without namespace declarations.

Chapter Summary

This chapter represents an in-depth tour of a real XML Signature implementation in RSA's Cert-J product. The chapter begins with a walk-through of each of the classes that comprise the toolkit. Class diagrams are given, and relationships are explained. The discussion of the classes is divided into classes that support the syntax and processing of XML Signatures, much the same way as the XML Signature Recommendation itself is structured.

We saw how the `XMLSignature` class provides the main abstraction for an XML Signature, including accessor functions to set various algorithms such as the canonicalization method and the signature method. The `Reference` class is used to store a data stream to sign as well as transform algorithms. This class represents the `<Reference>` element as it is defined in the XML Signature Recommendation. An in-depth discussion of

the `KeyInfo` class was also given; this class is an abstraction for the `<KeyInfo>` element. We saw how this class can be used as a container for public keys and certificates as well as a container for a customized `<KeyInfo>` child element. Furthermore, the `ParserHandler` class is responsible for actually creating the final XML structure and writing the structure to a file or stream. Similarly, `ParserHandler` is also used to read from a file or stream and create a number of `XMLSignature` instances, one instance per `<Signature>` structure.

The chapter concludes by discussing three customized pieces of sample code that are used to solve certain specialized problems. The first piece of sample code discussed is the `XMLEnvelopingBinary` sample. This sample shows how to envelope arbitrary data as a Base-64 encoded `<Object>` element. The second piece of sample code is the `ZIPDecompression-Transformer`; this sample shows how to extend the `Transformer` base class and create a customized transformation that does ZIP decompression. The final piece of sample code discussed is the `XPathTransformer`. This sample is an aid in developing XPath expressions and helps users see what they sign by performing the transformations explicitly. The `XPath-Transformer` takes a well-formed XML document as input and a string XPath expression and produces the transformed XML document as specified via the XML Signature Recommendation.

This page intentionally left blank.

CHAPTER 9

XML Key Management Specification and the Proliferation of Web Services

XML Encryption builds upon the XML Signature Recommendation in an axiomatic way. It relies on shared elements and semantics, and uses XML to provide a simpler vision of applied security. While XML Signatures and XML Encryption can be considered powerful and elegant, it can be argued that they still need the help of a traditional PKI. For example, an application that verifies an XML Signature must properly validate the signers' identity before accepting a signed message. Similarly, an application that sends encrypted correspondence to a recipient must properly retrieve the encryption key. Moreover, traditional PKI issues such as key-pair generation and certificate revocation aren't considered in either XML Encryption or XML Signatures.

Traditional PKI issues are out of scope for XML Signatures or XML Encryption, but are in scope for another XML Security technology called XKMS, which stands for *XML Key Management Specification*. At this time, XKMS is a much younger body of work than the XML Signature Recommendation or the XML Encryption drafts. Because of this, a discussion of the details has been omitted and a more conceptual approach has been taken. The goal of this chapter is for the reader to understand *what* XKMS is, *why* it is considered important, and *how* it is used. The details are left to the XKMS draft, which is currently in progress. (See the references section at the end of this book.)

XKMS Basics

The topic of XKMS is large and covers many areas of applied security and key management. The first scenario that we will look at provides some intuition about how XKMS works and what it is used for.

Validation, Verification, and Trust

In Chapter 4, we drew a picture of XML Signature verification from a high-level point of view. This is shown again in Figure 9-1 and is the same picture shown in Figure 4-1.

Figure 9-1 shows a trust engine and denotes the difference between signature verification and signature validation. In Figure 4-1, cryptographic verification is performed using the public key contained in the accompanying X.509 certificate, but the issue of signer trust is delegated to a fictional trust engine. In Chapter 4, we didn't describe this engine further or say how it worked; it was assumed that some sort of simple result is returned (for example, trusted or not trusted).

Some readers may wonder why a trust engine is necessary; isn't cryptographic verification of the signature good enough? The reason this is *not* good enough is because any XML Signature can be doctored to pass cryptographic verification. That is, it is trivial to generate a key pair and sign a document to make a claim of identity. What is required here is an asser-

Figure 9-1

XML Signature
verification and
trust
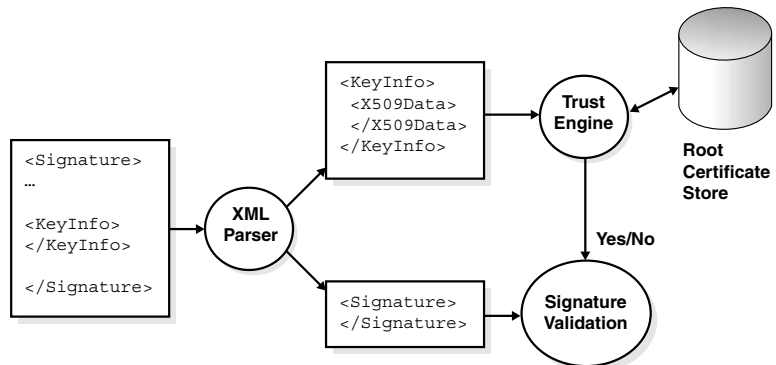
tion between an identity and a public key. This assertion is typically verified using path validation and is the same concept discussed in Chapter 2.

When we discussed path validation in Chapter 2, a number of things are mentioned. First and foremost, path validation is described with a state machine that is comprised of many steps and checks. Actions such as signature validation (for each certificate except the trusted CA), validity checking, name chaining, and certificate revocation must all be performed. Some checks, such as certificate revocation have the potential to be very expensive. It is possible that a certificate revocation list of hundreds or thousands of certificates must be checked for each certificate in the certification path.

In addition to the cryptographic load and revocation checking that must be performed, path validation also consists of policy mapping and certificate policy checking, both of which have the potential to be arbitrarily complicated.

Lastly, path validation is performed by the application receiving the certificate (this is called the relying party). This contrasts Figure 9-1, which shows a trust engine performing this validation step. The trust engine role is played by something called X-KISS, which stands for *XML Key Information Service Specification*. This particular acronym is one of the two parts of XKMS and defines a service for obtaining information about public keys, including trust assertions. The result of this design is the opportunity to offload the expensive and tedious path validation step to a service. As long as we access the service with a well-defined protocol, there is no need to have this heavy-duty path validation processing performed by the client.

This is one example of how XKMS promises to simplify the design of a client application that uses digital certificates. With X-KISS, the client application that wishes to verify a certificate doesn't have to create a certification path, verify numerous digital signatures, and perform revocation checking or policy mapping. All of this complexity is pushed away and only queries to a service are made.

XKMS Components

Two major components comprise XKMS. We have already briefly discussed one of them, X-KISS. The other component is called X-KRSS, which stands for *XML Key Registration Service Specification*. This component is responsible for registering public-key information to be used with X-KISS.

At this point, the reader can safely ignore X-KRSS because it is difficult to fully explain without a working knowledge of X-KISS. We will return to X-KRSS after X-KISS has been fully discussed. The bird's eye view of XKMS is shown in Figure 9-2.

Figure 9-2 is quite simple and reiterates previous discussion in the form of a picture. At this point, we will dig deeper into X-KISS and look at how X-KISS works with an XML Signature, as well as some sample X-KISS messages. Following this, a similar discussion will be presented with X-KRSS.

X-KISS: Tier 1

Suppose that we have a fictional application that needs to verify incoming XML documents that contain XML Signatures. That is, our program accepts files, streams, or URI values that eventually de-reference to well-formed XML documents containing `<ds:Signature>` elements. Given this, it follows that we would like to verify and validate these signatures. From Figure 9-1, we know that there is a big difference between cryptographic signature verification and signature validation. The former amounts to an integrity check that is only meaningful if we can assert trust over the purported signer. Asserting this trust usually comes from certificate path validation, which is described in some detail in Chapter 2.

Suppose, however, that the XML Signature that we are attempting to verify does not include a public key or a certificate, only a `<ds:KeyName>` element specifying a key identifier. Such an XML Signature is shown in Listing 9-1.

Figure 9-2

Bird's eye view of XKMS
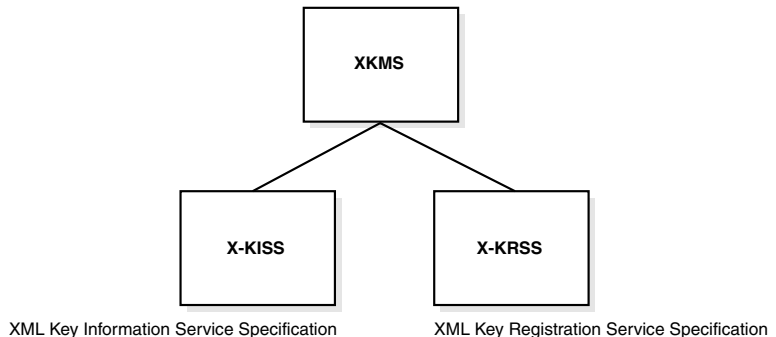
XML Key Information Service Specification   XML Key Registration Service Specification

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <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="file:///C:\foo.xml">
      <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>91cq1G+EfylJS6EPyy0kMWUOpVs=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>Y++WsqkpOHNwIr3zidvDQ92XzPGLhpt/t
  Dk9N9RE4xJ5QJo+9fKmUXVfzkTdcXqcl0OmRIN4IMSdF1sL8LxU+A==
  </SignatureValue>
  <KeyInfo>
    <KeyName> QR9432YZ5 </KeyName>
  </KeyInfo>
</Signature>
```

Notice the odd nature of the `<KeyName>` element in Listing 9-1. The element value is an odd-looking identifier that doesn't appear to have any relation to a user readable name. Suppose that this is a key identifier for a public key in a large organizational database. It could be a primary key in a database entry or some custom identification scheme. In any event, it needs to be mapped to a real public key value or the proper certificate that contains the verification key. What we are asking for here is a key location function. In particular, it might be nice to send the `<ds:KeyName>` element off to a service and have the service return a new `<ds:KeyInfo>` element containing a public key or certificate. This process is shown in Figure 9-3.

Figure 9-3 begins with an incoming XML Signature. The `<ds:Signature>` element is parsed for the `<ds:KeyInfo>` element that contains a `<ds:KeyName>` element including the odd key identifier. We are assuming that the signature processing application doesn't understand this identifier and must delegate the processing to a key location service. This key location service processes the key identifier and makes a database query that matches it to an X.509 certificate. This certificate is then formatted as a `<ds:KeyInfo>` element and passed back to the signature processing application. At this point the signature processing application has enough information to perform cryptographic validation of the signature. It now has a public key (contained in the certificate), whereas before it only had a single key identifier. The signature processing application may now choose to perform path validation on its own, or it may decide to delegate this action to a service as well. The key location service is the

Figure 9-3

Key location procedure
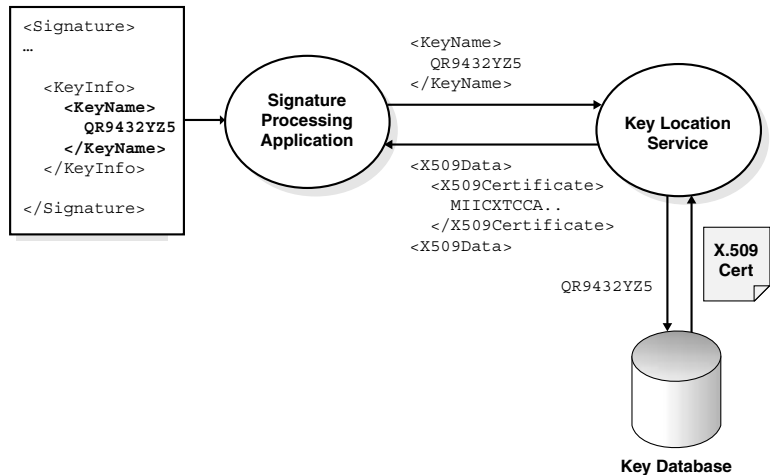
first tier of X-KISS, which is called the Locate service. In addition to pass-
ing off `<ds:KeyInfo>` elements, the signature processing application
may also pass off a `<ds:RetrievalMethod>` element if the signature
processing application doesn't have access to the necessary network or
server location. The syntax and meaning of the `<ds:RetrievalMethod>`
element is discussed in Chapter 4.

Another subtle use for the Locate service is to parse an X.509 certificate.
For example, consider a small signature processing application that does-
n't have the capability to parse an X.509 certificate, but requires the con-
tained public key to perform signature verification. The Locate service can
be queried with a `<ds:KeyInfo>` element that specifies a Base-64 encoded
X.509 Certificate. The X-KISS Tier 1 service can then parse the certificate
and return the actual public key value as a `<ds:KeyValue>` element con-
taining an `<ds:RSAKeyValue>` or `<ds:DSAKeyValue>` child element.

Syntax of the Locate Message

This is the first section where we actually look at an XKMS protocol mes-
sage for the Locate service, which is defined as a pair of messages. One
message, `<Locate>` is the request message, and the matching

<LocateResult> element is the response message. The Locate service can be thought of as a function that maps <ds:KeyInfo> elements to <ds:KeyInfo> elements where the input <ds:KeyInfo> element is usually inadequate for any sort of cryptographic processing. Listing 9-2 shows an example <Locate> message that asks a Locate service to map a string name to an X.509 certificate.

There are a few things to notice about Listing 9-2. First, notice that we are mixing namespaces. The <ds:KeyInfo> element comes from the XML Signature Recommendation (and is borrowed by XKMS). Further, XKMS has its own namespace, which has been given the identifier xkms. The proper use of namespaces is important when mixing XML Security technologies.

Listing 9-2 shows a few of the elements used in the <Locate> message. The essence of the <Locate> message is the <Query> element and the <Respond> element. The <Query> element asks the Locate service about the contained <ds:KeyInfo> element, and asks for specific information in return. The response requested is a Base-64 encoded X.509 certificate containing the public key bound to the name John Doe. The response message, <LocateResult> might look something like Listing 9-3.

In Listing 9-3, the <LocateResult> message contains a Base-64 encoded X.509 certificate containing the public key that corresponds to the name John Doe, as well as an indication of success or failure. It is extremely important to note that the Locate service does *not* assert validity over the binding of the name John Doe and the public key in the returned certificate. The Locate service simply maps <ds:KeyInfo> elements to <ds:KeyInfo> elements without any further assertions. In the case of the protocol exchange denoted by Listings 9-2 and 9-3, the client application is still responsible for ensuring that the binding between the

Listing 9-2

A sample <Locate> message

```
<?xml version="1.0" encoding="UTF-8"?>
<xkms:Locate xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">
<xkms:TransactionID>cb6f923a05d016575</xkms:TransactionID>
<xkms:Query>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsiga"
    <ds:KeyName>John Doe</ds:KeyName>
  </ds:KeyInfo>
</xkms:Query>
<xkms:Respond>
  <xkms:string>X509Cert</xkms:string>
</xkms:Respond>
</xkms:Locate>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xkms:LocateResult xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">
<xkms:TransactionID>gh5436f54923a05d65435</xkms:TransactionID>
<xkms:Result>Success</xkms:Result>
<xkms:Answer>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsiga">
    <ds:X509Data>
      <ds:X509Certificate>
        MIICcjCCAdugAwIBAgIQxo8RBl7oeoBUJR713
        41R/DANBgkqhkiG9w0BAQUFADBsMQswCQYDVQ
        ...
      </ds:X509Certificate>
    </ds:X509Data>
  </ds:KeyInfo>
</xkms:Answer>
```

name `John Doe` and the public key contained in the resultant certificate is valid.

The careful reader may question the usefulness of the Locate service under these circumstances. What has been gained? In this case, we started with simply a name, `John Doe`, and the Locate service provided a purported matching public key. It follows then that we gained the actual verification key whereas before we only had a name. In this case we can perform cryptographic verification of a signature whereas before it was impossible because we had no key. If we also want an indication of trust, we must turn to the next tier in the X-KISS protocol, which is called the Validate service.

X-KISS: Tier 2

The second tier is called the Validate service and is responsible for asserting trust over the binding of a name and a public key. The Validate service is a superset of the Locate service. This means that in addition to providing name-key assertions, it can also locate public key values. A more detailed example of the logistics of the validate service is shown in Figure 9-4.

In Figure 9-4 we have a situation similar to the one presented in Figure 9-3. The signature processing application receives an XML Signature with an X.509 certificate in the `<ds:KeyInfo>` element. In order for the signature application to assert validity over the signature, a proper certi-

Figure 9-4

Validate service


```
<Signature>
…

  <KeyInfo>
    <X509Data>
      <X509Certificate>
        MIICXTCCA..
      </X509Certificate>
    <X509Data>
  </KeyInfo>

</Signature>
```

```
<KeyInfo>
  <X509Data>
    <X509Certificate>
      MIICXTCCA..
    </X509Certificate>
  <X509Data>
</KeyInfo>
```

Signature Processing Application

Trust Service

<Result> Success </Result>

Certificate Path Validation Module

Certificate Repository
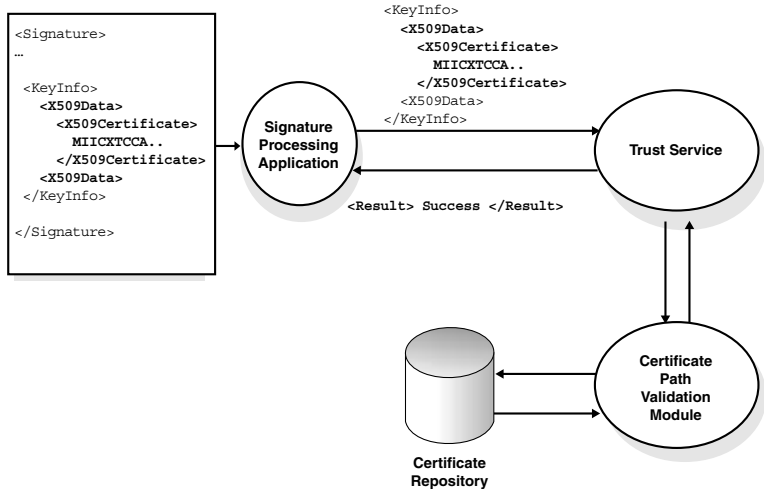
fication path must be built. Suppose that our signature processing application is quite simple and doesn't have any X.509 functionality or any other certificates. In short, our signature processing application can't build a certification path, nor can status checking (revocation) or policy mapping be performed. The solution is to ask a *trust service* using the protocol defined by X-KISS Tier 2.

The trust service shown in Figure 9-4 has access to the proper certificate store and relies on an additional module to perform the path validation. We can also assume that any CRLs (certificate revocation lists) can be retrieved from Certificate Repository 1. This figure shows one way in which processing complexity is moved from a client application to a service. The X-KISS Tier 2 service eventually completes with some sort of affirmative or negative result. In addition, the actual X-KISS protocol messages can communicate additional evidence to support or refute the final validation decision.

Some may argue that while path validation is complex, delegating the entire process to a service is not really necessary. Why can't the signature processing application simply perform these tasks? The answer to this argument lies in Figure 9-5, which shows a situation where the trust service must make further requests in order to build the proper certification path.

Figure 9-5 depicts a situation where the first trust service queried didn't have enough information to build a certification path. Instead of just returning an invalid result, the trust service instead asks another trust service about the purported certificate to be validated. This service can then ask another service and so on. This model is called service chaining and is one of the exciting capabilities of XKMS. The most important thing to notice about Figure 9-5 is that this additional complexity is hidden from the signature processing application, which just sees the end result of the query (basically a yes or no answer). This enables additional flexibility for the client application in that there is really no need to decide from multiple trust services. Any trust service can be queried with the expectation that it will do the dirty work and delegate further queries if necessary. A client application required to support the same type of chained functionality would be horrendously complex and would defeat the purpose of XKMS, which is simple client design.

Figure 9-5

Service chaining
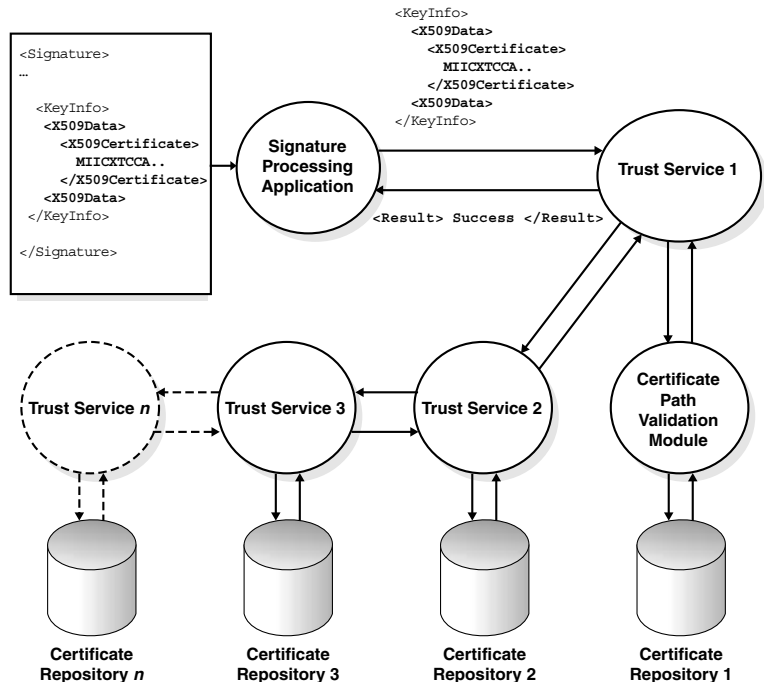
Syntax of the Validate Message

The Validate service is defined as a message pair in the same way as the Locate service. The request message is a `<Validate>` element and the response message is a `<ValidateResponse>` element. A sample `<Validate>` message is shown in Listing 9-4.

In Listing 9-4 we are passing a `<ds:X509Data>` element to the Validate service with the expectation of a status result and an indication of the key binding. When we use the term key binding we are referring to arbitrary data that the public key is bound to. In most cases the arbitrary data will be a name, but other useful data items such as a key identifier or key usage constraints are also possible.

In Listing 9-4 we are asking the Validate service to give us the name and public key from the queried certificate as well as make an assertion regarding the binding between the name in the certificate and the public key. The request for the assertion is provided by the `<Query>` element, which contains a prototype or model of the purported assertion. In Figure 9-4 the purported assertion is Valid as denoted by the `<xkms:Status>` element.

Listing 9-4

A sample `<Validate>` message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xkms:Validate xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">

<xkms:Query>
   <xkms:TransactionID>
     f9c6afa0-e6b3-11d5-81b0-a75f99b3a363
   </xkms:TransactionID>
   <xkms:Status>Valid</xkms:Status>
   <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <ds:X509Data>
     <ds:X509Certificate>
       MIICcjCCAdugAwIBAgIQxo8RBl7oeoBUJR713
       41R/DANBgkqhkiG9w0BAQUFADBsMQswCQYDVQ
          ...
     </ds:X509Certificate>
   </ds:X509Data>
   </ds:KeyInfo>
</xkms:Query>

<xkms:Respond>
  <xkms:string>KeyName</xkms:string>
  <xkms:string>KeyValue</xkms:string>
</xkms:Respond>
</xkms:Validate>
```

This particular situation shows how the Validate service is a super-set of the Locate service. The reason is because in addition to returning an assertion, we are also effectively locating the public key. This location functionality is useful in a situation where the client application has no X.509 processing capabilities. One possible response message would look something like Listing 9-5.

Listing 9-5 shows a sample `<xkms:ValidateResult>` element. The `<xkms:ValidateResult>` element contains a `<xkms:Result>` child element that conveys a result regarding the status of the request. This can be confusing because the `<xkms:KeyBinding>` element contains an element called `<xkms:Status>` that appears to be similar. The `<xkms:Result>` element refers to the success or failure of the actual transaction and not the semantics of *what* is asked about. The possible values of `<xmks:Result>` include Success, NoMatch, Incomplete, and Failure. The Success value implies that all of the request information was available; the NoMatch value implies that the transaction was executed, but there was nothing to return. The Incomplete value implies that some of the information requested was unavailable and the Failure value implies that the operation failed completely.

Listing 9-5

A sample `<Validate-Result>` message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xkms:ValidateResult
  xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">
<xkms:Result> Success </xkms:Result>
<xkms:Answer>
  <xkms:KeyBinding>
    <xkms:Status>Valid</xkms:Status>
    <ds:KeyInfo>
      <ds:KeyValue>
        <ds:RSAKeyValue>
          <ds:Modulus>
            tvvf9MtHPH+VVkG7nWENPK5N2Q+qm9PN
            +1luxVQg3QR+6WOXLZdAA21Hlm2qokqI
            MzuoF/0EY+k0vHZuwZmQOw==
          </ds:Modulus>
          <ds:Exponent>EQ==</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
      <ds:KeyName>John Doe</ds:KeyName>
    </ds:KeyInfo>
  <xkms:ValidityInterval>
    <xkms:NotBefore>2001-11-20T12:00:00</xkms:NotBefore>
    <xkms:NotAfter>2001-12-20T12:00:00</xmks:NotAfter>
  </xkms:ValidityInterval>
  </xkms:KeyBinding>
</xkms:Answer>
</xkms:ValidateResult>
```

The `<xkms:Result>` element contrasts the `<xkms:Status>` element; the `<xkms:Status>` element gives us information about the contained assertion and key binding. The `<xkms:Status>` element has three possible values: `Valid`, `Invalid`, and `Indeterminate`. In Listing 9-5 the status of the `<xkms:ValidateRequest>` message is `Valid`, indicating that that the binding of the name and public key is trusted. Following the `<Status>` element is a `<ds:KeyInfo>` element containing the actual public-key value and corresponding name. The `<xkms:Validate-Request>` message shown in Listing 9-5 does two things; it asserts trust as well as locates the public key corresponding to the certificate in the `<xkms:Validate>` request message.

X-KRSS

The X-KISS protocol is useful for locating public keys and determining how public keys are bound to names or other information. The only remaining problem lies in the registration of this information. If an X-KISS Tier 1 or Tier 2 service provides keys and binding information, there must be some way of informing the service that these keys or binding assertions exist in the first place. This is one of the tasks solved by the XML Key Registration Service protocol (X-KRSS). X-KRSS has three basic functions: Key Registration, Key Revocation, and Key Recovery.

Key Registration

Key Registration refers to the process generating a public/private key pair and registering the public key against some sort of key binding assertion. In most cases the public key is bound to a user name, although XKMS eventually promises to allow advanced assertions such as authorization information.

Key pair generation can be performed by the X-KRSS service or by the client. If the service performs key generation, the service must return the private key to the user in some sort of safe form. In the case of X-KRSS, this will be an encrypted RSA private key. If the client performs key-pair generation, the client must provide proof of possession of the private key. Proof of possession is used to prevent a certain form of identity theft. For example, suppose that Sally obtains the public key belonging to John (this

could come from an X.509 certificate or any public directory where public keys are accessible). Because a public key is public, there is nothing stopping Sally from attempting to register her identity against John's public key. If, however, Sally were required to provide proof that she owns John's private key, she would be unable to because only John knows his private key.

Proof of possession is often shown with an arbitrary signature. That is, the client making the registration request can show the X-KRSS service that they own the correct private key by signing an arbitrary piece of data (not chosen by the signer). The X-KRSS service will verify this signature using the public key in the registration request.

In addition to proof of possession, the registration service also needs to have some measure of client authentication for the client making the registration request. For example, Sally can generate a new key pair and register her public key against the name John Doe. In this case she has proof of possession because she owns a matching private key, but the public key binding is incorrect; her public key is not really bound to the name John Doe.

In order to prevent this sort of mischief, an out-of-band, shared secret is used by the X-KRSS registration service. This means that before a client registration request is made, the client must contact the service for a shared secret used to sign a future registration request. This authenticates the person making the request and prevents identity spoofing and false registration requests. Figure 9-6 shows the high-level overview of a client making a registration request as well as generating its own key pair. Once the registration request is sent, the service stores the name and key binding in some sort of repository (this could be any sort of PKI infrastructure). The service returns with an affirmative result informing the client that the registration request was successful.

In Figure 9-6 the client provides an authenticated request (via an HMAC) along with a name, public key, and some sort of proof of possession. The picture changes a bit when the service provides key pair generation. Service-provided key pair generation is useful for clients that have minimal processing capabilities. In general, RSA key pair generation is CPU intensive and it may be faster to have the service perform key pair generation. In addition, service-side key pair generation also allows for key recovery, because the service stores an additional copy of the private key that can be queried at a later date. Figure 9-7 shows a picture of service-side key pair generation.
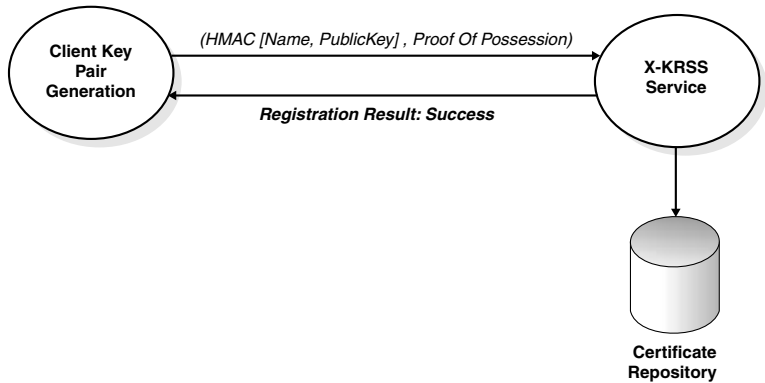
Figure 9-7 contrasts Figure 9-6 in that the client is only sending two pieces of information:

- A name that the public key will be bound to
- Proof of authenticity (with the use of an HMAC)

Upon receipt and successful processing of the request, the service returns an affirmative result: an RSA public key and an encrypted RSA private key. Again, the registration request is stored in some sort of certificate repository that corresponds to some sort of traditional PKI. Figures 9-6 and 9-7 both show how X-KRSS is capable of shielding the client from the underlying PKI by providing a service interface.

Key Registration Message Syntax

The key registration message is defined by the <Register> element. There are two important child elements for the <Register> element. These include the <Prototype> element and the <AuthInfo> element. The <Prototype> element is an example of the requested assertion. That is, it is a model for the assertion eventually registered with the X-KRSS service. The choice of wording here can be confusing. The term prototype refers to an example instance or basis from which to model the actual assertion. Another idea that fits here is purported assertion. The <Register> message contains a purported assertion that will eventually become real when finally registered.
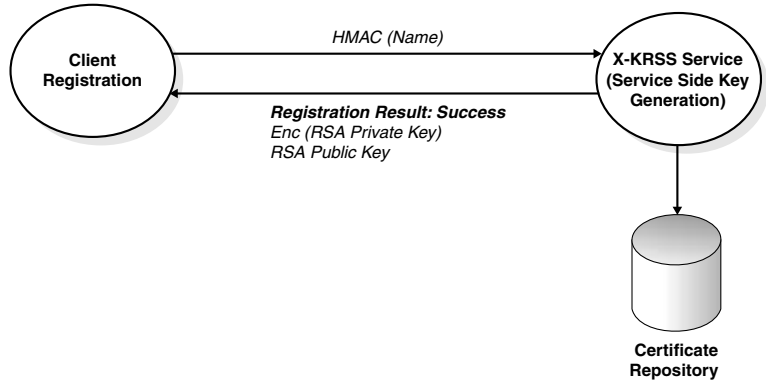
The <AuthInfo> element contains data items that relate to the authenticity of the requester. In the case of client-generated key pairs, this includes two signatures, one that determines proof of possession of the private key and one that authenticates the identity of the user. An example of a <Register> message is shown in Listing 9-6.

The <xkms:Register> message begins with the <xkms:Prototype> element, which contains a <ds:KeyInfo> element. The first child in the <xkms:Prototype> element is a purported assertion about the contained prototype. This is shown with the <xkms:Status> element containing the value Valid, which asserts that this particular key binding is purported to be valid and should be registered as such. Following the <xkms:Status> element is the <ds:KeyInfo> element that specifies the purported key binding between the <ds:KeyName> and the <ds:KeyValue> elements. An interesting element is the <xkms:Pass-Phrase> element, which is the last child element of the <xkms:Prototype> element. The <xkms:PassPhrase> element contains a user-chosen pass phrase (that is encrypted and encoded) that enables key recovery in the event that the client's key is compromised.

The next element is the <xkms:AuthInfo> element, which contains two signature values. The reader should notice that the signatures used are complete XML Signatures that specify the <xkms:Prototype> element as the source data via the Id attribute. The first signature in the <xkms:AuthUserInfo> element is the proof of possession information. The client must sign something to prove to the service that it owns the

Listing 9-6

An example
<Register>
message (client
generated key
pair)

```xml
<xkms:Register xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-
20">
<xkms:Prototype Id="bindingInfo">
  <xkms:Status>Valid</xkms:Status>
  <ds:KeyInfo>
    <ds:KeyValue>
      <ds:RSAKeyValue>
        <ds:Modulus>
          tvvf9MtHPH+VVkG7nWENPK5N2Q+qm9PN+1luxVQg
          3QR+6WOXLZdAA21Hlm2qokqIMzuoF/0EY+k0vHZu
          wZmQOw==
        </ds:Modulus>
        <ds:Exponent>EQ==</ds:Exponent>
      </ds:RSAKeyValue>
    </ds:KeyValue>
    <ds:KeyName>John Doe</ds:KeyName>
  </ds:KeyInfo>
  <xkms:PassPhrase>vtyhHnJzxBHJi</xkms:PassPhrase>
</xkms:Prototype>

<xkms:AuthInfo>
  <xkms:AuthUserInfo>
    <xkms:ProofOfPossession>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <ds:Reference URI="#bindingInfo">
          ...
        </ds:SignedInfo>
      </ds:Signature>
    </xkms:ProofOfPossession>
    <xkms:KeyBindingAuth>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
          <ds:Reference URI="#bindingInfo">
          ...
        </ds:SignedInfo>
      </ds:Signature>
    </xkms:KeyBindingAuth>
  </xkms:AuthUserInfo>
</xkms:AuthInfo>
<xkms:Respond>
  <string>KeyName</string>
  <string>KeyValue</string>
</xkms:Respond>
</xkms:Register>
```

proper private key. The data is signed with an RSA signature that requires a private key as input to the signature operation. This particular `<ds:Signature>` element contrasts the signature in the `<xkms:Key-BindingAuth>` element in that it uses an RSA operation where as the second signature in the `<xkms:Register>` element uses an HMAC to produce the signature value.

This point is especially important. The first signature specifies only proof of possession of the private key while the second signature is generated with a pre-shared secret value and provides authentication to the X-KRSS service. That is, it is used so the X-KRSS service can be confident that the `<xkms:Register>` request came from the purported client. The final part of the `<xkms:Register>` message is a list of data items that the service should return. The idea here is that the service should repeat back to the client the actual binding that was registered. In this case the data items returned include a public key and a name value.

The X-KRSS response message takes the supplied key binding prototype and produces the final key binding and presents it back to the client. The response is sent using the `<RegisterResult>` element. The syntax of a sample X-KRSS response message is shown in Listing 9-7.

In Listing 9-7, the `<xkms:RegisterResult>` element contains a `<xkms:KeyBinding>` element that represents the information actually bound. In this case it is a `<ds:KeyValue>` element and a `<ds:KeyName>` indicating that the name John Doe is bound to the returned RSA public key.

Listing 9-7

An example `<Register-Result>` message (client generated key pair)

```
<xkms:RegisterResult
 xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">
  <xkms:Result>Success</xkms:Result>
  <xkms:Answer>
  <xkms:KeyBinding>
     <xkms:Status> Valid </xkms:Status>
     <ds:KeyInfo>
     <ds:KeyValue>
       <ds:RSAKeyValue>
         <ds:Modulus>
           tvvf9MtHPH+VVkG7nWENPK5N2Q+qm9PN+1luxVQg
           3QR+6WOXLZdAA21Hlm2qokqIMzuoF/0EY+k0vHZu
           wZmQOw==
         </ds:Modulus>
        <ds:Exponent>EQ==</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
      <ds:KeyName>John Doe</ds:KeyName>
    </ds:KeyInfo>
    </xkms:KeyBinding>
  </xkms:Answer>
</xkms:RegisterResult>
```

The `<xkms:RegisterResult>` element also contains two types of status values. The `<xkms:Result>` element conveys information about the request itself (but not information about the semantics of the request) while the `<xkms:Status>` element contains the value Valid, which means that the binding in the returned assertion is valid. The `<xkms:Status>` element is the same element seen in the X-KISS Tier 2 response message and has similar meaning for the X-KRSS service.

Key Revocation

Some X-KRSS servers may enable clients to revoke key binding assertions that have been registered at an earlier time. If a key binding assertion is revoked, it is no longer considered a valid assertion and should no longer be trusted. The key revocation process is similar to the key registration process. The main idea is that the prototype for the assertion changes. This means that instead of asserting that a given key binding is Valid (as is done in an X-KRSS request message), the prototype changes to a purported value of Invalid. This change is done by altering the value inside the `<xkms:Status>` element from Valid to Invalid. The rest of the message remains similar; the only notable difference is that only one signature is required to provide evidence for proof of possession (the client must still own the private key if it is to be revoked). The additional presence of the HMAC signature is not required because the X-KRSS service has positive identification of the client from the initial registration request. One could argue that the act of key revocation is really just the registration of a negative assertion. An example key revocation message is shown in Listing 9-8.

Listing 9-8 differs from the registration request shown in Listing 9-6 in that it omits one of the signatures as well as the `<xkms:PassPhrase>` element, which is not needed for key recovery. The corresponding `<RequestResult>` message is identical to Listing 9-5 except for the fact that the `<xkms:Status>` element contains the value Invalid instead of Valid, which indicates that the key binding can no longer be trusted.

Security Considerations

The XKMS protocol as presented here still has a transport security problem. This has to do with the integrity, confidentiality, and authenticity of

Listing 9-8

A revocation request

```
<xkms:Register xmlns:xkms="http://www.xkms.org/schema/xkms-2001-01-20">
<xkms:Prototype Id="bindingInfo">
  <xkms:Status>Invalid</xkms:Status>
  <ds:KeyInfo>
    <ds:KeyValue>
      <ds:RSAKeyValue>
        <ds:Modulus>
          tvvf9MtHPH+VVkG7nWENPK5N2Q+qm9PN+1luxVQg
          3QR+6WOXLZdAA21Hlm2qokqIMzuoF/0EY+k0vHZu
          wZmQOw==
        </ds:Modulus>
      <ds:Exponent>EQ==</ds:Exponent>
      </ds:RSAKeyValue>
    </ds:KeyValue>
    <ds:KeyName>John Doe</ds:KeyName>
  </ds:KeyInfo>
</xkms:Prototype>
<xkms:AuthInfo>
  <xkms:AuthUserInfo>
    <xkms:ProofOfPossession>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:SignatureMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <ds:Reference URI="#bindingInfo">
            ...
        </ds:SignedInfo>
      </ds:Signature>
    </xkms:ProofOfPossession>
  </xkms:AuthUserInfo>
</xkms:AuthInfo>
<xkms:Respond>
  <string>KeyName</string>
  <string>KeyValue</string>
</xkms:Respond>
</xkms:Register>
```

the request and response messages for either X-KISS or X-KRSS. The best way to understand this issue is to think of how XKMS messages are actually transported at the application layer. XKMS messages and elements are not designed with an implicit transport mechanism. This means that one can't simply send XKMS request or response messages in the clear because there is no way to prevent against tampering. For example, if a client opens a socket connection to an X-KISS service and send a `<Locate>` or `<Validate>` message, any potential eavesdropper can alter the message and interrupt the request. Further, there is no authentication of the response from the server. The client has no way to trust the service.

There are two ways in which this transport security problem can be solved:

- Use the security services of the bound transport protocol.
- Use a transport protocol without security features enabled.

If the second option is chosen, one must use socket-level security or packet-level security to protect XKMS messages. Socket level security is achieved with something called TLS, which stands for "Transport Layer Security" and packet-levels security is typically achieved with a protocol called IPSEC, which loosely stands for Internet Protocol Security.

When we talk about the bound transport protocol, we are usually talking about another XML-based wrapper around XKMS messages that enables the encapsulation of the request and response paradigm. The SOAP protocol is currently used to provide this wrapper, although the XKMS draft also defines a content type for XKMS messages that permits XKMS over HTTP. In short, the security services can be delegated to the bound transport protocol, TLS, or the packet level. (For more information on TLS, see the references section at the end of this book.)

Although the transport security problem can be solved, care must be taken not to complicate the solution, especially when attempting to provide support for multiple trust services. For example, a client that wishes to use a single trust service must have some way of trusting the trust service. This is a fundamental problem of the delegated trust mechanism built into XKMS.

One can claim that an XKMS client is "PKI-free" and only uses a trust service to register and validate key binding assertions without any PKI knowledge, but a client must have some way of validating the authenticity of the service itself. XKMS clients are designed to be simple and because of this there is no second check of the information returned from an XKMS service; this implies that some minimal verification ability must be present on the client. One can argue that as the number of distinct trust services grows, the complexity of a client application that wishes to boast compatibility with all of them must also grow. If this growth continues unchecked, the client will become increasingly complex and defeat one of the purposes of XKMS—simple client design.

Chapter Summary

Chapter 9 is a conceptual discussion of XKMS, the XML Key Management Specification. The goals of this chapter include introducing XKMS and understanding how it fits in with XML Encryption and the XML Signature. XKMS is divided into two sub-technologies, X-KISS and X-KRSS. The first subtechnology, X-KISS, delegates the processing of key information while the second sub-technology, X-KRSS, delegates the registration of key information. The concept of delegation takes center stage for XKMS because the traditional PKI requires a rather heavyweight client. XKMS pushes complexity away from a client to a service and builds a simple interface around this service. This paradigm shift allows for a lightweight client to easily handle complicated authentication and authorization semantics.

Appendix

Additional Resources

This final section represents a cornucopia of topics that didn't make their way into the main body of the text, but are important to mention. The topics are varied and include information about another canonicalization algorithm called *exclusive canonicalization*, an algorithm list for XML encryption, as well as material related to RSA's BSAFE Cert-J product, discussed in Chapter 8. In addition, the reader will find a list of references to books and web sites organized by chapter.

Exclusive Canonicalization

The XML Signature Recommendation uses Canonical XML 1.0 as the default canonicalization algorithm for ensuring that semantically meaningless syntax alterations don't unnecessarily break an XML Signature (refer to Chapter 4 and Chapter 5). It was discovered, however, that Canonical XML can cause trouble when XML documents are reenveloped, causing signatures to break when they shouldn't.

This problem occurs because of the way Canonical XML deals with namespaces. In particular, the Canonical XML 1.0 algorithm "attracts" ancestor namespaces during the execution of the algorithm and makes it difficult for portable signatures to work properly. For example, consider two example XML documents, D_1 and D_2, as follows:

Document D$_1$
```
<?xml version="1.0" encoding="UTF-8"?>
<foodns:Food xmlns:foodns="http://food.com">
  <foodns:FrenchFries> Curly Fries </foodns:FrenchFries>
</foodns:Food>
```

355

Document D₂
```
<?xml version="1.0" encoding="UTF-8"?>
<Drinks>
  <Beer>
    <Good_Beer> Samuel Adams </Good_Beer>
    <Good_Beer> Guinness </Good_Beer>
    <Bad_Beer>  Budweiser </Bad_Beer>
    <Bad_Beer>  Fosters </Bad_Beer>
  </Beer>
</Drinks>
```

Suppose we create an *enveloped* XML Signature over document D_2 using an XPath expression to pinpoint only the `<Beer>` element (`ancestor-or-self::Beer`). We can call this signature S_1 as follows:

Enveloped Signature S₁
```
<?xml version="1.0" encoding="UTF-8"?>
<Drinks>
  <Beer>
    <Good_Beer> Samuel Adams </Good_Beer>
    <Good_Beer> Guinness </Good_Beer>
    <Bad_Beer>  Budweiser </Bad_Beer>
    <Bad_Beer>  Fosters </Bad_Beer>
  </Beer>
  <Signature>
    <SignedInfo>
      ...
      <Reference URI="">
        <Transforms>
          <Transform
           Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
            <XPath>
              ancestor-or-self::Beer
            </XPath>
          </Transform>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>DikaiaUpotHeKeIcJyTwm84DU=</DigestValue>
      </Reference>
    </SignedInfo>
    ...
  </Signature>
</Drinks>
```

Carrying the example further, let's envelope S_1 inside D_1. That is, we are "wrapping" document D_1 *around* S_1 (which contains document D_2 by virtue of the type of signature). This new aggregate document will be called D_3 and is shown as follows:

Document D₃
```
<?xml version="1.0" encoding="UTF-8"?>
<foodns:Food xmlns:foodns="http://food.com">
  <foodns:FrenchFries> Curly Fries </foodns:FrenchFries>
```

```
  <Drinks>
    <Beer>
      <Good_Beer> Samuel Adams </Good_Beer>
      <Good_Beer> Guinness </Good_Beer>
      <Bad_Beer>  Budweiser </Bad_Beer>
      <Bad_Beer>  Fosters </Bad_Beer>
    </Beer>
   <Signature>
     <SignedInfo>
      ...
       <Reference URI="">
         <Transforms>
           <Transform
             Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
             <XPath>
               ancestor-or-self::Beer
             </XPath>
           </Transform>
         </Transforms>
         <DigestMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
         <DigestValue>DikaiaUpotHeKeIcJyTwm84DU=</DigestValue>

       </Reference>
     </SignedInfo>
        ...
   </Signature>
  </Drinks>
</foodns:Food>
```

Finally, suppose we want to verify the signature inside document D_3, which is calculated over the original document D_2. During the *reference validation* process, the XPath expression will be used to obtain the proper node set. This node set is then digested and compared against the original node set in document D_2. The node set produced, however, is not what might be expected. If we try to extract the `<Beer>` element using an XPath expression over document D_3, the node set produced is as follows:

```
  <Beer xmlns:foodns='http://food.com">
    <Good_Beer> Samuel Adams </Good_Beer>
    <Good_Beer> Guinness </Good_Beer>
    <Bad_Beer>  Budweiser </Bad_Beer>
    <Bad_Beer>  Fosters </Bad_Beer>
  </Beer>
```

The reader should notice the apparently superfluous namespace declaration in the previous example (`xmlns:foodns="http://food.com"`). We should *expect* to see the exact same node set represented by document D_2. The reader should check document D_2 and notice the difference. This situation may seem errant, but it is not. Canonical XML attracts ancestor namespace information and in doing so adds the namespace value

`xmlns:foodns="http://food.com"`, which will break the signature upon verification because the original `<Beer>` element in D_3 did not know about this additional context. The problem comes from the way Canonical XML works; it is not a problem with the XML Signature format or standard, but an anomalous case that occurs because of the mechanics of canonicalization. The solution to this problem is an algorithm called *exclusive canonicalization*, which is an alternate canonicalization algorithm that repels ancestor context instead of attracting it. We won't cover the details of how this algorithm works here, but the reader is urged to visit the references section for more information.

XML Encryption: A List of Supported Algorithms

The following is a list of algorithms supported by the XML Encryption draft along with their URI identifiers. This list comes directly from the XML Encryption Working Draft.

Block Encryption

- Triple-DES (Required):
  `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`
- AES-128 (Required):
  `http://www.w3.org/2001/04/xmlenc#aes128-cbc`
- AES-256 (Required):
  `http://www.w3.org/2001/04/xmlenc#aes256-cbc`
- AES-192 (Optional):
  `http://www.w3.org/2001/04/xmlenc#aes192-cbc`

Key Transport

- RSA Encryption with PKCS#1 Padding (Required):
  `http://www.w3.org/2001/04/xmlenc#rsa-1_5`
- RSA Encryption with OAEP Padding (Required):
  `http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p`

Key Agreement

- Diffie-Hellman Key Agreement:
  `http://www.w3.org/2001/04/xmlenc#dh`

Symmetric Key Wrap

- Triple-DES Symmetric Key Wrap (Required):
  `http://www.w3.org/2001/04/xmlenc#kw-tripledes`
- AES-128 Symmetric Key Wrap (Required):
  `http://www.w3.org/2001/04/xmlenc#kw-aes128`
- AES-256 Symmetric Key Wrap (Required):
  `http://www.w3.org/2001/04/xmlenc#kw-aes256`
- AES-192 Symmetric Key Wrap (Required):
  `http://www.w3.org/2001/04/xmlenc#kw-aes192`

Message Digest

- SHA1 (Required): `http://www.w3.org/2000/09/xmldsig#sha1`
- SHA-256 (Recommended):
  `http://www.w3.org/2001/04/xmlenc#sha256`
- SHA-512 (Optional):
  `http://www.w3.org/2001/04/xmlenc#sha512`
- RIPEMD-160 (Optional):
  `http://www.w3.org/2001/04/xmlenc#ripemd160`

Message Authentication

- XML Digital Signature: `http://www.w3.org/TR/2001/`
  `PR-xmldsig-core-20010820/`

Canonicalization

- Canonical XML with Comments (Optional): `http://www.w3.org/`
  `TR/2001/REC-xml-c14n-20010315#WithComments`
- Canonical XML Without Comments (Optional):
  `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`
- Exclusive XML Canonicalization (Optional):
  `http://www.w3.org/TR/2001/WD-xml-exc-c14n-20011120`

Encoding

- Base 64 Encoding (Required):
  `http://www.w3.org/2000/09/xmldsig#base64`

References

This section lists all the source material used by the author for this book. Some of the references are on the Web and others are textbooks. The books listed here are especially good at conveying conceptual information, while the web links usually refer to XML or cryptography standards.

Chapter 2: Security Primer

The applied security-related references used for this chapter are listed here.

Cryptography Books

Burnett, Steve and Stephen Paine. *RSA Security's Official Guide to Cryptography.* Berkeley, California: Osborne/McGraw-Hill, 2001.

Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. New York: John Wiley and Sons, 1996.

Stinson, Douglas R. *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

PKCS Standards

RSA Laboratories, Public Key Cryptography Standards (PKCS). RSA Security Inc. 16 December 2001. www.rsasecurity.com/rsalabs/pkcs.

ASN.1

Larmouth, John. *ASN.1 Complete,* San Francisco: Morgan Kaufmann Publishers, 1999.

PKI

Housley, R., SPYRUS, W. Ford, VeriSign, W. Polk, NIST, and D. Solo, Citicorp. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile (RFC2459)*. January 1999.

Chapter 3: XML Primer

The XML resources used for this chapter are listed here.

XML Books

Hunter, David, et. al. *Beginning XML.* Birmingham UK: Wrox Press Ltd, 2000.

Ray, Erik T. *Learning XML*. Sebastopol, CA: O'Reilly and Associates, 2001.

XML and DTD

Bray, T., E. Maler, J. Paoli, and C. M. Sperberg-McQueen. "Extensible Markup Language (XML) 1.0 (Second Edition)." W3C Recommendation. October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006.`

Namespaces in XML

Bray, T., D. Hollander, and A. Layman. "Namespaces in XML." W3C Recommendation. January 1999. `http://www.w3.org/TR/1999/REC-xml-names-19990114/.`

XML Schema

Beech, D., M. Maloney, N. Mendelsohn, and H. Thompson. "XML Schema Part 1: Structures." W3C Recommendation. May 2001. `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.`

Biron, P. and A. Malhotra. "XML Schema Part 2: Datatypes W3C Recommendation." May 2001. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.`

DOM

Apparao, V., S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. "Document Object Model (DOM) Level 1 Specification." W3C Recommendation. October 1998. `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`

SAX
For information on SAX, see `www.saxproject.org.`

Xerces

Information about the Xerces XML parser can be found at `xml.apache.org.`

URI

Berners-Lee, T., R. Fielding, and L. Masinter. "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. Standards Track." August 1998. `www.ietf.org/rfc/rfc2396.txt`

Chapter 4 and Chapter 5: XML Digital Signatures Parts I and II

Note:

An interoperability matrix of known XML Signature implementations is hosted at `www.w3.org/Signature/2001/04/05-xmldsig-interop .html`.

Canonical XML 1.0

Boyer, J., March 2001. Canonical XML. Recommendation. `http://www .w3.org/TR/2001/REC-xml-c14n-20010315`. `www.ietf.org/rfc/ rfc3076.txt`.

Exclusive Canonicalization

Boyer, J., D. Eastlake, and J. Reagle. "Exclusive XML Canonicalization." W3C Working Draft. October 2001. `http://www.w3.org/TR/2001/ WD-xml-exc-c14n-20011120`.

XML Signature Proposed Recommendation

Eastlake, D., J. Reagle, and D. Solo. "XML-Signature Syntax and Processing." Proposed Recommendation. `http://www.w3.org/TR/2001/ PR-xmldsig-core-20010820/`.

Chapter 6: XML Signatures Frequently Asked Questions

XML Signatures Scenarios FAQ. WG Proposal. 18 February 2000. `http://www.w3.org/Signature/Drafts/PROP-xmldsig-faq- 20000218/Overview.html`.

XSLT

Tidwell, Doug. *XSLT*. O'Reilly and Associates, 2001.

Chapter 7: XML Encryption

Imamura, T., and H. Maruyama. "Decryption Transform for XML Signature." Working Draft. `http://www.w3.org/TR/2001/WD-xmlenc-decrypt-20010626`.

"XML Encryption Syntax and Processing." WG Working Draft 18. October 2001. `http://www.w3.org/TR/2001/WD-xmlenc-core-20011018/`.

Chapter 8: XML Signatures with RSA BSAFE Cert-J

For information on RSA Security BSAFE toolkits, please visit `www.rsasecurity.com`. The sample code and code listings for Chapter 8 can be downloaded at `www.rsasecurity.com/go/xmlsecurity/`.

Chapter 9: XKMS and the Proliferation of Web Services

XML Key Management Specification (XKMS). W3C Note 30. March 2001. `http://www.w3.org/TR/2001/NOTE-xkms-20010330/`.

Dierks, T., Certicom, and C. Allen. "The TLS Protocol Version 1.0 (RFC 2246)." January 1999.

Template Signing FAQs for RSA BSAFE Cert-J

What Is Template Signing?

The term *template signing* refers to building the template of a `<ds:Signature>` element inside an input document before signing occurs. The idea here is that the `<ds:Signature>` has the necessary cryptographic values filled in upon signature generation.

Why Is Template Signing Necessary?

Template signing is necessary in cases where the `<ds:Signature>` element needs to be placed in a specific location in the XML input document tree. In most cases, this will happen with a detached or enveloped XML Signature. In the case of a same-document-detached XML Signature, template signing is often the only way to create the desired structure.

How Does Template Signing Work in Cert-J?

If you want to create an XML document with a very specific structure and want the `<ds:Signature>` element to appear at a specific location in the tree, the following steps can be followed (for a DOM-based implementation):

1. Create a new DOM Parser object and load the input document that will eventually have the `<ds:Signature>` element added.

2. Search through the nodes of your input document until you find the position where you want the `<ds:Signature>` element to be added.

3. Once you have your target node, you must build an entire `<ds:Signature>` element using your DOM Parser, but leave the `<ds:SignatureValue>` and `<ds:DigestValue>` elements empty. That is, actually create a new subtree structure and append this structure to the desired target node. More than one `<ds:Reference>` element is permitted; be sure to add as many as are required.

4. Once your structure has been altered to include a `<ds:Signature>` element, the parent `Document` object will also be altered. At this point, you can create a new `XMLSignature` object and use the `setDocument()` function with the updated `Document` object and the boolean value "true," which tells Cert-J to use the existing template `<ds:Signature>` instead of creating a new one.

Is Template Signing Present in All Versions of Cert-J?

Template signing is available for Cert-J 2.01 and later. Cert-J 2.01 requires a patch for template signing to work. This patch adds a new API function, `XMLSignature.setDocument()`, which takes as its input a `Document` object and a boolean value (this should be set to "true" to indicate that a template signing operation is happening).

Is Any Sample Code Available?

Sample code has been created to help customers perform this important task. Please contact your RSA support representative for access to the patch and sample code.

Index