Security is one of the most vital topics in Web services development today and will be for the foreseeable future. The lack of maturity of standards and tools in this area is the reason most often cited for large organizations delaying their commitment to Web services. The most important security standards are ready now, though, and the tools are coming online. Importantly, these Web Services Security standards are really not groundbreaking; they are, in turn, just extensions of very well-established information security standards.

In Chapter 2, “The Foundations of Web Services,” we introduced Web services as a new form of middleware for building and integrating distributed applications by sending XML messages between computing nodes. Making Web services secure means making those messages secure and keeping them secure wherever they go. This chapter builds on the preceding chapter by adding a solid foundation in the principles of distributed message-level security. Those principles depend on solid knowledge of shared key cryptography and public key cryptography. We begin by setting the information security context for Web services.

The Challenges of Information Security for Web Services
Securing distributed systems is hard. Securing exchanged information between those systems is harder. Securing Web services with their distributed, shared, and exposed information (XML messages) is much harder still. In the following sections, we address each of these challenges in order.

Security of Distributed Systems Is Hard
In distributed systems, you are not securing just one system, but many. In addition, you are securing the interconnections between the distributed systems. The goal of
Chapter 3 The Foundations of Distributed Message-Level Security

distributed systems is to make a collection of independent computers appear to their users as a single integrated system. When you can successfully integrate separate and distinct networked computers together, the integrated system can handle higher workloads, aggregate more functionality, and share data that previously was locked up and inaccessible. But the more access points there are, the more places an attacker has to attack. In other words, a distributed system has all the security issues of one system multiplied many times over.

These distributed system security issues include access control, identity management, authentication, password management, authorization, encryption of confidential information on each node, integrity of information passing between nodes, and more. This is the reason the CORBA security specification is 430 pages long.

Because Web services use the Web and enable and even encourage integration between systems across organizational boundaries using public networks, they are, in fact, much more distributed than CORBA or any previous form of middleware could contemplate. Web services, when used in remote procedure call mode, involve some of the same issues as CORBA object invocation security.

A service-oriented architecture—such as what can be achieved when Web services become pervasive—projects a vision of geographically distributed services, many of which are publicly available shared services incorporated into numerous applications. Knowing who is using these services, what these users are authorized to do, and how to protect all information at these nodes is a difficult security task.

Security of Exchanged Information (Messages) Is Harder

When security administrators think about securing their systems today, they typically think about securing their organization’s “four walls.” They think about perimeter security such as firewalls, intrusion detection, honeypots, and DMZs. None of these elements deal with the security of messages in transit. Security administrators focus most on access control security believing that, if they know for sure who has access to their network, they implicitly have secured their critical assets: their information. In fact, perimeter security schemes are really a proxy for information security, because all along, what people are really trying to protect is the information created, processed, or stored on the machines inside their perimeters.

Distributed applications send back and forth messages that are the command, control, and coordination that make the distributed applications function. This means you have the problems of computer and network security you always had, plus the problems of information security for critical command and control messages flowing through the distributed system. Transport-based security like SSL provides one type of protection for messages in transit—just server-to-server non-persistent confidentiality—but offers little or no control at the application level, which is the place where the messages have meaning.

Messages must remain intact from sender to receiver regardless of how many hops occur in between. Shared services create a many-hops scenario in which messages go
through many service endpoints on the way to their destination. The receiver must know for sure who the sender was to establish trust, which is critical in all aspects of business. The sender must be able to control who is allowed to see her messages because vital (and confidential) information is contained in them. In the case of transmitted legal documents, being able to prove what transpired and repudiate any denials is critical. Messages need to remain persistently secure not just while in transit, but also when residing in permanent storage because the message might contain highly confidential information such as a credit card number, Social Security number, or business deal terms.

### Security of Web Services Is Hardest

Web services add a new dimension to the challenges already faced with distributed systems. Critical security questions must be reliably answered. Who is the end user? How do you maintain security when routing messages through multiple servers? How do you maintain security when using shared services?

Web services are fundamentally about how to share the burden of—and derive the benefits from—computing across a distributed network of computers connected with Web infrastructure and standards. These messages—being XML—are text-based, readable, and self-describing. Security of Web services is about knowing whether the Web services are talking to the correct endpoints; the communications between all the disparate elements are kept confidential; the messages transported maintain their integrity at all times; and the entities for which these Web services are being invoked are known, trusted, and authorized to use the services and can be clearly identified to all other authorized services.

The good news is that Web Services Security builds on existing security standards; very little that is new is being proposed. In general, Web Services Security provides an XML-based abstraction layer for established security technologies that delivers confidentiality, integrity, non-repudiation, authentication, and authorization.

### Why All the FUD about Web Services Security?

Some Web Services Security vendors, the analysts that cover these companies, and the media are spreading fear, uncertainty, and doubt (FUD) about how an organization must fully implement Web Services Security before deploying the first Web service. Because early Web service deployments tend to be simple, straightforward, internal, \textit{behind the firewall}, point-to-point integrations, sophisticated Web Services Security technologies are not always necessary. In fact, the same security used by these companies for securing their Web sites—namely SSL—is usually sufficient.

For the vendors who are selling either a product or service that implements Web Services Security in some form, however, it is in their best interest to have the early Web services adopters firmly believe they cannot be successful without these vendors’ products or services. This thinking is a huge stretch, to put it mildly.

When designing an information security strategy for a Web services deployment, you must think like an information security professional. Think in terms of confidentiality, integrity, and the other building block security principles. If your messages need legal non-repudiation, perhaps you need to employ more sophisticated technologies. But if all you need to do is keep messages confidential between point A and point B, maybe what you already know how to do so well (use SSL) is perfectly sufficient.
To build on your understanding of these information security building blocks, you need to understand how the discipline of cryptography is applied to distributed messages. The following sections cover the shared key technologies and then cover public key technologies.

**Shared Key Technologies**

Shared key technologies, including shared key encryption and Kerberos, will be your critical tool for guaranteeing information security principle number one: confidentiality of Web services messages. The following sections discuss the cryptography behind the algorithms, the mechanisms for managing keys, and the relationship between shared key technologies and public key technologies.

**Shared Key Encryption**

Other synonyms for shared key encryption are frequently used. This technology is sometimes referred to as *symmetric encryption* or *secret key encryption* because the same key is used to both encrypt and decrypt the message, and this key must be kept secret from all non-participating parties to keep the encrypted message secret. For clarity and simplicity, in this book we consistently use the term *shared key encryption* because sender and receiver share the same key, which is distinct from public key encryption where one key is made open and public to the entire world. Shared key encryption is the key technology to achieve the security principle of message confidentiality. Message confidentiality means that no one other than the intended recipient will be able to read a message because it is encrypted strongly enough that no one without the correct key can decrypt it by any other means. We reproduce the shared key encryption diagram from Chapter 1, “Basic Concepts of Web Services Security,” here in Figure 3.1.

![Figure 3.1 Shared key encryption.](image)

The key to message confidentiality is *in the key*, so to speak. Because both sender and receiver must utilize the exact same key (it is *shared*), this solitary key must somehow be separately and confidentially exchanged so that both parties have it—and no one else...
Shared Key Technologies

does—prior to message exchange. Under other circumstances, you might think you could send the key via U.S. Mail or read it over the phone, but neither of these approaches would be consistent with the goals of computer-to-computer Web services communication. For Web services, this exchange must be made over the same channel that the encrypted messages will flow. Cryptographers and security practitioners have determined that the best way to accomplish this is to encrypt the shared key using public key encryption before exchange. Public key encryption is described in the next section.

Only the shared key is secret, not the encryption algorithm itself. It is very important that the encryption algorithms themselves are public and well scrutinized. In the early days of information security, encryption algorithms were kept strictly secret and proprietary; however, after they were put into use, users found that they had dangerous flaws. Now the field of cryptography understands that the open study of algorithms enhances confidence in the algorithm's security due to the extensive testing and analysis by all the world's cryptographers. A proof of this point is how long the modern shared key encryption algorithms Advanced Encryption Standard (AES) and Triple Data Encryption Standard (3DES) have been in use, withstanding extensive testing with no flaws found.

Without going into extensive mathematical derivations to prove our points (see Appendix A for some details), we need you to take on faith the three most important factors you need to know about shared key encryption:

- Shared key encryption is much faster than public key encryption.
- Shared key encryption can operate on arbitrarily large plaintext messages, whereas public key encryption cannot.
- Shared key encryption utilizes public key encryption to manage distribution of the shared key securely to the recipient.

Shared key encryption uses cryptographic algorithms known as block ciphers. This means that the algorithms work by taking the starting plaintext message and first breaking it into fixed-size blocks before encrypting each block. Two strong encryption algorithms used throughout the software industry are part of the XML Encryption standard: the long-standing 3DES and the relatively new, but highly reviewed and analyzed AES.

DES involves many computationally fast and simple substitutions, permutations, XORs, and shifting on a data block at a time to produce ciphertext from an input plaintext message. The design of DES is so clever that decryption uses exactly the same algorithm as encryption. The only difference is that the order in which the key parts are used in the algorithm is exactly reversed.

DES is 25 years old and still going strong. Part of the reason for this is that DES was designed (and adapted over the years) to work really well in hardware. Numerous companies now include DES chips in their devices. Virtual private networks (VPNs) frequently use 3DES—the stronger version of DES described next—directly in the hardware to create secure IPSec tunnels. DES uses a 64-bit key: 56 effective bits, and 8 bits for parity. DES operates on an 8-byte fixed-block size. In the past decade, weaknesses found in DES have been addressed by 3DES.
Chapter 3 The Foundations of Distributed Message-Level Security

Note
It is rumored that the NSA has machines costing under $50K that can brute-force crackDES with its 64-bit key in 15 minutes.

3DES saved DES from being abrogated to the cryptography scrap heap by responding to the fact that $2^{64}$ (72 quintillion) possible keys made plain DES susceptible to a brute-force attack, so the DES algorithm is run three times with a much longer key. 3DES uses a 192-bit key, of which the effective key length is 168 bits. The idea is to use DES three times: sequencing Encrypt-Decrypt-Encrypt with a different third of the 168-bit key used at each stage, as shown in Figure 3.2. This approach makes it relatively slow but very secure; on hardware, relatively slow is still plenty fast. Note that the middle step—Decrypt—does not actually decrypt the ciphertext created by the first step because a completely distinct key is used; it actually scrambles the message further.

![Figure 3.2](http://example.com/figure3.png)

Figure 3.2 How 3DES uses DES encryption three times in succession to be much more secure than DES alone.

AES is the newest shared key encryption standard. AES came out of a government-sponsored contest won by the cryptographers Joan Daemen and Vincent Rijmen (the algorithm was called Rijndael, pronounced “rain doll”). It also works well in hardware and can have keys up to 256 bits long, taking it out of the realm of ever being susceptible to a brute-force attack.

A critical concept for shared key encryption is padding. The blocks that are input to the cipher must be of a fixed size, so if the plaintext input is not already of the correct size, it must be padded. However, you cannot just add random data; to do so, you would need to communicate to the receiver the correct size of the data, which would be a dan-
Shared Key Technologies

Gerous clue to an attacker. You also cannot just add zeros because the message itself might have leading or trailing zeros. Consequently, padding must include sentinel information that identifies the padding and must not give an attacker any critical information that might compromise the message.

Kerberos

No discussion of shared key technologies would be complete without mention of Kerberos. Kerberos was originally part of MIT’s project Athena. The goal set out for Kerberos was real-time authentication in an insecure (the client is not trusted) distributed environment. This technology predates the Web. Plus, it appears to be getting a new lease on life with Microsoft’s adoption of it in the base Windows operating system security model. Kerberos is also one of the recognized security tokens—pieces of information used for authentication or authorization added to a SOAP header—in the emerging WS-Security standard, which is presented in Chapter 7, “Building Security into SOAP.”

Kerberos is a trusted third-party authentication protocol. Users and services obtain an authentication token called a ticket that they use for identity and shared keys that they use for secure communications. A shared key known only to the server that encrypts tickets. The Kerberos model calls for a centralized authentication server. Kerberos performs centralized key management and administration. The server maintains all users’ shared keys and generates session keys whenever two users want to communicate securely. It authenticates the identity of a user who requests access to secure network services. A client must have a valid password and ticket (which only the server can validate) to use any services.

The basic Kerberos authentication flow is as follows:

1. The client sends a request to the authentication server requesting “credentials” for a given application server. These credentials can be applied directly to an application server or to a ticket-granting server.

2. The authentication server responds with credentials encrypted with the client’s key; it includes a ticket for the server and a session key.

3. If the ticket is for the ticket server, the client then requests a ticket for the application server from the ticket server.

4. The ticket server responds with the ticket.

5. The client transmits its ticket with the client identity and a session key copy encrypted using the server’s key to the application server.

6. The session key is now shared between the client and application server, which authenticates the client and can be used to authenticate the server. It can also be used to encrypt further communication between the two parties.

Kerberos plays an important, but not leading, role in Web services security. Systems running Microsoft Windows frequently use the built-in Kerberos model, which is appropriate only for closed systems, but they still may want to connect to external organizations.
via Web services. Those external systems need to have their authentication scheme mapped into the Kerberos model for the two organizations to interact.

The WS-Security standard recognizes Kerberos tickets as one of the valid types of security tokens. A very important attribute and goal of WS-Security is to be able to bridge between different security models and trust domains—for example, one system that runs Kerberos connected to another that uses X.509 public key infrastructure.

Limitations of Shared Key Technologies

As we said earlier, shared key technologies fail to solve the problem of scalable key distribution. Shared key algorithms are fast and can handle infinitely large messages; however, both ends of the communication need access to the same key, and you need to get it to your recipient securely. Shared key cryptography also fails to solve the issue of repudiation. At times, you will need to be able to prove that a certain identity created and attests to sending a message (or document) and no one else could have. The sender must not be able to deny having sent this exact document at this moment in time. Shared key cryptography provides no help here.

Public Key Technologies

Public key technologies, including public key encryption and digital signature, will be your tools for delivering integrity, non-repudiation, and authentication to XML messages and more generally to Web services. The following sections begin with an explanation of the concepts behind public key encryption. We expand this description so that you can apply this knowledge to build the basis for digital signatures and then apply these concepts more specifically to digital signatures in XML. A discussion of public key technologies is not complete without a description of public key infrastructure and the issues of establishing trust, which are covered at the end of this section.

Public Key Encryption

Public key encryption is also referred to as asymmetric encryption because there is not just one key used in both directions, as with the symmetric form of encryption used in shared key algorithms. In public key encryption, there is a matched set of two keys; whichever one is used to encrypt requires the other be used to decrypt. In this book, we use the term public key encryption to help establish context and contrast it with shared key encryption.
The keys in public key encryption are non-matching, but they are mathematically related. One key (it does not matter which) is used for encryption; that key is useless for decryption. Only the matching key can be used for decryption. This concept provides the critical facility you need for secure key exchange to establish and transport a shared key.

The diagram from Chapter 1 that shows how basic public key encryption works is reproduced here in Figure 3.3.

![Figure 3.3](image)

Figure 3.3 Public key encryption uses one key to encrypt the incoming plaintext, and only the other key can be used to successfully decrypt that ciphertext back into its original plaintext.

Although it is true that Kerberos provides a mechanism for distributing shared keys, Kerberos applies only to a closed environment where all principals requiring keys share direct access to trusted Key Distribution Centers (KDCs) and all principals share a key with that KDC. Thus, Kerberos does not provide Web services with a general mechanism for shared key distribution; public key systems without the restriction of working only in closed environments take that role. Public key systems work with paired keys, one of which (the private key) is kept strictly private and the other (the public key) is freely distributed; in particular, the public key is made broadly accessible to the other party in secure communications.

Parties requiring bidirectional, confidential communication must each generate a pair of keys (four keys in total). One of the keys, the private key, will never leave the possession of its respective creator. Each party to the communication passes his public key to the other party. The associated public key encryption algorithms are pure mathematical magic because whatever is encrypted with one half of the key pair can only be decrypted with its mate. Figure 3.4 shows how the key pairs are each used for one direction of the confidential message exchange.
Chapter 3 The Foundations of Distributed Message-Level Security

Figure 3.4 Two key pairs are involved in bidirectional, confidential message exchange. One pair is used for each direction. The sender always uses his private key for encryption, and the recipient always uses the sender's public key for decryption.

For Alice to send a confidential message to Bob, Alice must obtain Bob's public key. That's easy because anyone can have Bob's public key at no risk to Bob; it is just for encrypting data. Alice takes Bob's public key and provides it to the standard encryption algorithm to encrypt her message to Bob. Because of the nature of the public-private key pair and the fact that Alice and Bob agree on a public, standard encryption algorithm (like RSA), Bob can use his private key to decrypt Alice's message. Most importantly, only Bob—because no one will ever get her hands on Bob's private key—can decrypt Alice's message. Alice just sent Bob a confidential message. Any outsiders intercepting it will see useless scrambled data because they don't have Bob's private key.

We will describe digital signatures in a moment, but first we want you to notice something interesting about doing things just the reverse of Alice's confidential message. If Alice encrypts a message with her private key, which only Alice can possess, and if Alice
Public Key Technologies

makes sure Bob has her public key, Bob can see that Alice and only Alice could have encrypted that message. In fact, because Alice’s public key is, in theory, accessible to the entire world, anyone can tell that Alice and only Alice encrypted that message. The identity of the sender was established. That is a principle of digital signatures.

The simple fact that in public key cryptography, whatever is encrypted with one half of the key pair can only be decrypted with its mate, combined with the strict rule that private keys remain private and only public keys can be distributed, leads to a very interesting and powerful matrix of how public key encryption interrelates to confidentiality and identity. This matrix is shown in Table 3.1.

Table 3.1 The Matrix of Uses for Key Pairs in Public Key Cryptography Showing What Security Principle Applies and What Part of Web Services Security It Affects

<table>
<thead>
<tr>
<th>Public Key</th>
<th>Private Key</th>
<th>What This Means</th>
<th>WS Security Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt (with recipient’s)</td>
<td>Decrypt (with recipient’s)</td>
<td>Confidentiality (no one but intended recipient can read)</td>
<td>XML Encryption shared key exchange</td>
</tr>
<tr>
<td>Decrypt (with sender’s)</td>
<td>Encrypt (with sender’s)</td>
<td>Signature (identity) (it could only have come from sender)</td>
<td>XML Signature</td>
</tr>
</tbody>
</table>

Public key encryption is based on the mathematics of factoring large numbers into their prime factors. This problem is thought to be computationally intractable if the numbers are large enough. But a limitation of public key encryption is that it can be applied only to small messages. To achieve the goal of distributing shared keys, this is no problem; those keys are not larger than the message size limitation of public key algorithms. To achieve the goal of digital signatures on arbitrarily large messages, you can apply a neat trick to remain within this size limitation, as discussed next.

**Limitations of Public Key Encryption**

Even when implemented in hardware, shared key algorithms are many orders of magnitude faster than public key algorithms. For instance, in hardware, RSA is about 1,000 times slower than DES.

The first performance hit comes from key generation. You must find two multi-hundred-bit prime numbers that are near the same length. Then these two primes must be tested for primality. Primality testing is a very expensive operation requiring a series of steps that each has a certain probability of determining that the values are relatively prime. These steps must be run several times to make the probability high enough as to have an acceptably infinitesimal risk of being wrong.

The second reason that public key encryption is so much slower than shared key algorithms is that RSA encryption/decryption is based on the mathematics of modular exponentiation. This means you are taking each input value, raising it to a power (requiring a large number of multiplications), and then performing the modulo operation (the
remainder after doing integer division). On the other hand, shared key ciphers are based on much faster logical operations on bit arrays. Public key algorithms are called asymmetric for a reason. Because the private key has a much larger exponent than the public key, private key operations take substantially longer than do public key operations. To establish identity and support non-repudiation (for example, in digital signatures) where the public key is used for encryption, decryption takes substantially longer than encryption. In integrity applications (also in digital signatures) where the private key is used for encryption, it is the other way around. This imbalance would be a problem when applied to large messages but is not an issue when applied only to small messages such as the 200-bit key being exchanged to enable shared key encryption or the 20-byte message digests used in digital signatures, which will be discussed shortly.

The third reason to be concerned about the computational complexity of public key encryption is the padding issue. The input to RSA encryption operations is interpreted as a number, so special padding is required to make the input totally consistent. The total length of the data must be a multiple of the modulus size, and the data must be numerically less than the modulus. A 1,024-bit RSA key has a 128-byte modulus. Therefore, data must be encrypted in blocks of 128 bytes. Each input number must be padded with zeros until its numerical value is less than that of the modulus. This padding places a critical restriction on the size of data that RSA can encrypt. This is why RSA is never used to encrypt the entire plaintext message but only encrypts the shared key being exchanged between communicating parties. Then, after the shared key is established safely between the parties, RSA (public key) encryption is no longer used, and instead AES (shared key) encryption is used on the plaintext message itself.

### Digital Signature Basics

Digital signature is the tool you use to achieve the information security principle of integrity. Digital signature involves utilizing a one-way mathematical function called hashing followed by public key encryption. The basic idea is to use a hash function to create a message digest of fixed and short length and then to encrypt this short message digest. A message digest is a short representation (usually 20 bytes) of the full message. You need to do this because, as you have just seen, public key encryption is slow and is limited in the size of message it can encrypt. A hash is a one-way mathematical function that creates a unique fixed-size message digest from an arbitrary-size text message. One-way means that you can never take the hash value and re-create the original message. Hash functions are designed to be very fast and are good at never creating the same result value from two different messages (they avoid collisions). Uniqueness is critical to make sure an attacker can never just replace one message with another and have the message digest come out the same anyway; that would essentially ruin the goal of providing message integrity through digital signatures.

You know that public key encryption works only on small-size messages. You also know that if Alice encrypts a small message with her private key and sends the message to Bob, Bob can use Alice’s public key to prove that the message could only have come from
Alice (as long as you are sure she protected her private key). This identification of the sender is one half of what digital signature is all about. The other half relates to obtaining the goal of verifying the integrity of the message. By *integrity*, we mean that you can tell whether the message has changed by even one bit since it was sent. The key to integrity in the digital signature design is the use of a hash function to create a message digest.

### Hashing the Message to Create a Message Digest

A hash function creates a message digest that can be used as a proxy for the original message. You want this function to be very fast because, as you will see, you need to run this function on both the sending and verifying ends of a communication. Most importantly, you must be certain that it is impossible for two messages to create the same output hash value to achieve integrity; otherwise, you lose your goal of integrity. If it were possible for two messages to create the same message digest, it would mean that someone could substitute a new message for the original and fool the recipient into thinking the new fraudulent message is the correct one. For integrity's sake, it must also be impossible to reverse the function and take the small output value and regenerate the original message (that is, it must truly be a one-way function).

Functions that avoid duplicate values are exceedingly rare. The birthday paradox demonstrates this point. With any random 23 people in a room, the chance that two have the same birthday is 50%. Think of these people as representing 23 different plaintext messages. If a hash function applied to just 23 messages (or 2,300 or 23,000 messages) had a 50% chance of two of them resulting in the same message digest, you would not accept this type of hash function to deliver integrity.

Several one-way hash functions that have excellent collision avoidance properties have been designed and deployed, including MD4, MD5, and SHA1. Weaknesses have been found in the first two, and currently most security systems, and all the standards for Web services security, use SHA1. (Interestingly, SHA1 is a replacement for SHA which corrected a flaw in the original SHA algorithm.)

---

**Note**

SHA1, which stands for Secure Hash Algorithm 1, refers to it being the first (implying there may be the need for variants in the future). NIST and the NSA designed the algorithm for use with the Digital Signature Algorithm (DSA).

### SHA1

The SHA1 standard specifies a Secure Hash Algorithm, which is necessary to ensure the security of the Digital Signature Algorithm. When a message of any length < $2^{64}$ bits is input, the SHA produces a 160-bit message digest output. The message digest is then input to the DSA, which computes the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process because the message digest is usually much smaller than the message. The verifier of the signature should obtain the same message digest when the received version of the message is used as input to SHA.

---

1. [http://www.rsasecurity.com/rsalabs/faq/3-6-5.html](http://www.rsasecurity.com/rsalabs/faq/3-6-5.html)
Chapter 3 The Foundations of Distributed Message-Level Security

The SHA is called secure because it is designed to be computationally infeasible to recover a message corresponding to a given message digest or to find two different messages that produce the same message digest. Any change to a message in transit will, with a very high probability, result in a different message digest, and the signature will fail to verify. The SHA is based on principles similar to those used by Professor Ronald L. Rivest of MIT when designing the MD4 message digest algorithm, and is closely modeled after that algorithm.

SHA1 meets all the requirements for a secure one-way hash algorithm. In particular, it is not reversible. Guessing a 20-byte hash (160 bits) has a 1 in $2^{160}$ chance of coming up with the original message.

Note
By way of scaling how unlikely guessing a 1 in $2^{160}$ chance is:

- $2^{61}$ sec is the total lifetime of the universe.
- $2^{170}$ is the total number of atoms in the earth.

On a computer running 1 billion hashes per second (still beyond computing capacity today), brute-force guessing of a 20-byte hash would take $10^{22}$ billion years to accomplish, which is more than the lifetime of the universe.

If you hash the entire plaintext message and then protect the resulting message digest from being modified in any way, and if the sender and receiver use the exact same input message and hash algorithm, the recipient can check and verify the integrity of the message without going to the huge expense of trying to encrypt the entire message. So now let's discuss how to protect the message digest.

Public Key Encryption of the Message Digest
Protecting the message digest simply involves encrypting it with the private key of the sender and sending the original message along with the encrypted message digest to the recipient. Public key encryption of the message digest provides non-repudiation because only the subject identity with the private key that did the encryption could have initiated the message. Protecting the message digest by encrypting it such that no middleman attacker could have modified it provides message integrity.

Digital Signature Signing Process
You are now ready to put all this information together and create a digital signature. You will combine the hashing function (SHA1) used to convert the plaintext message into a fixed-size message digest with public key encryption (RSA or DSA) of that message digest to create a digital signature you can send to the selected recipient. The basic steps for creating a digital signature are as follows:

1. Hash the entire plaintext message, creating a 20-byte message digest.
2. Encrypt this message digest using the sender’s private key.
3. Send the original message and the encrypted message digest along with the sender’s public key to any desired recipients.

These steps are shown in Figure 3.5, where the hash algorithm being used is SHA1.

**Digital Signature Verification Process**

Signature verification is the process the message recipient goes through to determine the identity of the sender and to determine that the message arrived intact and unaltered. In other words, signature verification is necessary to achieve the security principles of message integrity and non-repudiation. The steps in signature verification are as follows:

1. The recipient receives the original plaintext message and the encrypted message digest from the sender.
2. Separately or at the same time, the recipient receives the sender’s public key.
3. The original plaintext document is run through the same SHA1 hash algorithm originally performed by the signer. This algorithm is identical on all platforms, so the recipient has confidence that the exact same result will occur if the document has not been altered in any way.
4. The recipient uses the sender’s public key to decrypt the message digest. If the decryption is successful and the recipient trusts the sender’s public key to be valid, and the recipient also trusts that the sender has protected her private key, the recipient knows that it was the sender who sent this message.
5. The final step of the verification process is a bit-for-bit comparison of the message digest computed locally from the original document with the one just decrypted. If they match exactly, the signature is valid.

The verification process just outlined is shown in Figure 3.6.

![Digital Signature Verification Diagram]

**Figure 3.6** Digital signature verification process.

You now know for sure that the unique private key that matches this public key is the one that encrypted the message digest, which tells you the identity of the signer if you are certain she has protected her private key—all of which gives you non-repudiation. You also know that the message was sent unaltered, so you have integrity. What you still need, and will be discussed in a few moments, is assurance that you know the identity of the owner of the public key you just used.

RSA is the most commonly accepted digital signature algorithm used in Web services security, although officially DSA is also allowed.

**Integrity Without Non-Repudiation**

When non-repudiation is not a goal, a very different approach to verifying message integrity called Message Authentication Code (MAC) is used. This approach is like creating a cryptographic checksum of a message. The MAC’s class of algorithms provides pure message integrity protection based on a secret shared key. To limit the size message MAC
will operate on, Web Services Security combines pure MAC with a hash, so the acronym becomes HMAC. Figure 3.7 shows how an HMAC functions.

Think of an HMAC as a key-dependent one-way hash function. Only someone with the identical key can verify the hash. You know that hashing is a very fast operation, so these types of functions are useful for guaranteeing message authenticity when secrecy and non-repudiation are not important but speed is. These algorithms are different from a straight hash because the hash value is encrypted and protected with a key. The algorithm is symmetric: The sender and recipient possess a shared key. You will see HMAC again in Chapter 4, “Safeguarding the Identity and Integrity of XML Messages,” as part of the XML Signature discussion.

A Digital Signature Expressed in XML
Web services messages are XML-based. Message-level security in Web services requires that you apply digital signatures in an XML setting. Therefore, you need a digital signature expressed in XML. That is the simplest description of what the XML Signature standard is. XML Signature was designed to have a great deal of flexibility. An XML Signature can be placed inside an XML document, or it can refer to external elements that need to be signed. For example, an XML Signature could be applied just to a credit card number inside the XML document, or it could be applied to a complete personal medical history. It is also important to be able to sign external elements like online resources such as Web pages to prevent defacement. XML Signature supports that option as well.

The structure of an XML Signature is shown in Listing 3.1:

Listing 3.1  The Structure of an XML Signature

```xml
<Signature>
  <SignedInfo>
    (CanonicalizationMethod)
    (SignatureMethod)
  </SignedInfo>
</Signature>
```
Chapter 3 The Foundations of Distributed Message-Level Security

We expand on this structure and explore the full richness of XML Signature in Chapter 4.

**Public Key Infrastructure**

As you saw earlier, digital signature verification required that the recipient obtain the sender’s public key. We were not specific about how that happened. This step is incredibly important to establishing trust between sender and recipient. The recipient needs to rely on the trust implied in the public key that it is really from the sender and that the sender has maintained custody and sanctity of his private key. This is the domain of Public Key Infrastructure (PKI). PKI encompasses certificates, certificate authorities, and trust.

In all our discussions of public key encryption and its application to digital signatures, we oversimplified to the extreme when we said the public key is just sent to a recipient. In fact, the key alone is not enough. You need more than just the public key itself if the public key is from someone you don’t know well. You need identity information associated with the public key. You also need a way to know whether someone you trust has verified this identity so that you can trust this entire transaction. Trust is what PKI is all about.

**Digital Certificates Are Containers for Public Keys**

A digital certificate is a data structure that contains identity information along with an individual’s public key and is signed by a certificate authority (CA). The official designation of standard digital certificates with which we will be dealing is X.509. By signing the certificate, the CA is vouching for the identity of the individual described in the certificate. This is so that relying parties (message recipients) can trust the public key contained in the certificate.

Bob, the relying party, must be certain that this is really Alice’s key. He ensures that it is by checking the identity of the CA that signed this certificate (how he trusts the CA in the first place we will get to in a moment) and by verifying both the identity and integrity of the certificate through the CA’s attached signature. A validity date included in X.509 certificates helps ensure against compromised (or out-of-date and invalid) keys.

The X.509 digital certificate trust model is a very general one. Each subject identity has a distinct name. The subject must be certified by the CA using some well-defined certification process it must describe and publish in a Certification Practice Statement (CPS). The CA assigns a unique distinguished name to each user and issues a signed certificate containing the name and the user’s public key.

```xml
<Signature>
  <SignedInfo>
    < digestMethod>
    </digestMethod>
    <digestValue>
    </digestValue>
    <reference>
      (Reference URI=?)>
      <transforms>
      </transforms>
      <keyInfo>
      </keyInfo>
      <object>*
    </reference>
  </SignedInfo>
  <signatureValue>
  </signatureValue>
</Signature>
```
Version 3 of the X.509 standard specifies a certificate structure including certificate extensions. The v3 certificate extensions enable extra functionality. For example, they allow incorporating authorization information in the certificate, providing the possibility of defining special authorization certificates. However, there is no guarantee of uniformity in the use of extensions, which can lead to interoperability problems.

The most important fields in the X.509 structure include:

- **Version**—Which version of the X.509 standard this certificate conforms to.
- **Serial number**—A unique identifier for this certificate. This number can be used for revoking certificates, as described later in this chapter.
- **Signature algorithm**—The algorithm used to produce the digital signature this certificate bears.
- **Issuer**—The name of the organization that issued this certificate.
- **Valid from** and **valid to**—The validity times of the certificate (that is, when the certificate begins being valid and when it will expire).
- **Subject**—The name of the principal whose public key is contained in this certificate.
- **Public key**—The public key of that principal.
- **Various other fields**—Some of which are referred to as extended attribute fields.

The issuing certificate authority always signs the certificate. Figure 3.8 shows how a digital certificate is represented by the Windows XP operating system.
Certificate Authorities Issue (and Sign) Digital Certificates

The CA signs the certificate with a standard digital signature using the private key of the CA. Like any digital signature, it allows anyone with the CA's matching public key to verify that this certificate was indeed signed by the CA and is not fraudulent. The signature is an encrypted hash (called the thumbprint) of the contents of the certificate, so standard signature verification guarantees integrity of the certificate data as well. That, in turn, allows you to believe the information contained in the certificate. Of course, what you are really after is trust in the validity of the Subject's (that is, the sender's/signer's) public key contained in the certificate.

So far, so good—if you trust the CA who signed this certificate.

The entire world may rely on such a signature, so you can be sure the CA goes to extraordinary lengths to protect its private key, including armed guards, copper-clad enclosures, and special hardware protecting the private key.

Note

Typical trusted CAs really do have armed guards protecting their buildings and areas called man traps that protect entry to the rooms in which the keys are contained. Two people must enter at once or not at all. The room contains a copper-clad enclosure that prevents stray electromagnetic radiation from emanating where it could be picked up by an intruder. Inside this copper-clad enclosure called a tempest room are the hardware devices that contain the keys. Changes in temperature or any attempts to move these devices cause them to self-destruct—yes, really self-destruct. To re-create a destroyed key requires five or more separate individuals taking their own special hardware card and combining them all under the observation of a trained auditor. The point is that these private keys are well protected.

The public key of the CA is typically very widely distributed. In fact, the public keys for SSL certificates—X.509 certificates issued to organizations for their Web sites—are found in all Web servers and all Web browsers to make sure relying parties can always verify certificates signed by those CAs.

The key to trusting the signed certificate is what process the CA used to verify the identity of the subject prior to the issuance of the certificate. It might be based on individuals being employees. It might require they produce a driver's license. Or it might be that they must correctly answer a set of shared secret questions drawn automatically from databases that know about all individuals, such as the telephone company, driver's license bureau, or credit bureau. In extreme cases in which no doubt is tolerable (national security, for example), a blood or DNA sample might have to be produced.

You can think of the CA as a digital notary. An individual's identity is based on the assurance (honesty) of the notary. A certificate policy specifies the levels of assurance the CA has to provide, and the CPS specifies the mechanisms and procedures to be used to achieve a level of assurance. Development of the CPS is the most time-consuming and essential component of establishing a CA. The planning and development of the certificate policies and procedures require the definition of requirements, such as key escrow, and processes, such as certificate revocation, which are covered later in this chapter.
A CA may be the guy down the hall, the HR department of your company, a local external company, a public CA, or the government.

**Levels of Identity**

One level of identity verification is never suitable for all situations. There is a need to declare different levels of identity “strength” so that some standardization can occur to benefit relying parties. A relying party is someone who is presented with a digital identity and must decide whether the process used to establish this identity is acceptable for his application. These strength levels are chosen based on

- Degree of confidence that the individual is who he says he is
- Risk of being wrong (release of information, completion of transaction, and so on)
- Severity of consequences of being wrong

Two different organizations are driving the push for standard levels of identity: the AICPA and the Federal Bridge Authority. Both deal with federated identity, but unlike the commercial federated identity projects Passport and Liberty Alliance, this application of federated identity is purely business-to-business or agency-to-agency in the government. One level of identity does not suffice in these situations because numerous different applications are involved with large variances in their assessment of the three criteria identified here.

The levels being proposed by the AICPA and Federal Bridge are similar in description, as shown in Table 3.2.

**Table 3.2 Federal Bridge and Proposed AICPA Criteria for Identity Levels Used in CA Processes When Establishing Digital Identities**

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Appropriate Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rudimentary—Whatever the individual claims, such as an email address</td>
<td>For anonymous transactions</td>
</tr>
<tr>
<td>2</td>
<td>Basic—May be based on employment records or consumer credit card and address verification</td>
<td>For pseudonymous transactions in which specific identity is not critical, but follow-up or delivery information is necessary</td>
</tr>
<tr>
<td>3</td>
<td>Medium—May use the consumer credit file and other databases that provide reliable shared secrets for identity verification</td>
<td>For identified transactions that require a person to be specifically identified</td>
</tr>
<tr>
<td>4</td>
<td>High—Person must be physically present</td>
<td>For verified transactions, the person must be identified, integrity of data and transaction event guaranteed, and evidence created to prove these were the parties to the transaction</td>
</tr>
</tbody>
</table>
Chapter 3 The Foundations of Distributed Message-Level Security

CAs Must Be Trusted or Vouched For by a Trusted CA
If the CA that signed the certificate is not known to or not trusted by the relying party, that CA must itself be vouched for by a more trusted CA that satisfies the relying party. This certificate chain can continue indefinitely until eventually the chain reaches a trusted CA or it reaches a root certificate. In a root certificate, the issuer of the certificate is also the subject of that certificate (that is, the certificate is self-signed); it’s a dead end. If the relying party does not trust the root CA, that party is out of luck and will have to reject the transaction being requested. If you are using a certificate issued by GeoTrust, you might have two certificates in a certificate chain, as shown in Figure 3.9.

![Certificate chain example](image)

**Figure 3.9** Screenshot showing a certification path. Certificate issued to www.geotrust.com is linked back to a certificate issued to (and by) Equifax Secure which is self-signed.

Two certificates are involved in this example. One certificate is issued directly to www.geotrust.com, who is the subject, and Equifax Secure is the issuer. The second certificate is the Equifax Secure root certificate, which is self-signed: The subject is Equifax, and the issuer is Equifax. Typically, you have a “trust store” or “trust list” in some database (it exists in Web servers, Web browsers, and in the operating system itself in many cases and for Web services will be placed in additional accessible places) containing the certificates of the certificate issuers that you are willing to trust. Through a process called certificate path validation, an attempt is made to create a “path” of valid, non-revoked certificates to one of the defined trusted certificate issuers in your trust list. This process can become quite complex; hence, one of the goals of the XML Key Management Specification (XKMS) discussed in Chapter 9, “Trust, Access Control, and Rights for Web Services,” is to offload this complexity to a “trust engine” Web service that will handle this validation for other Web services.
A reason for this complexity and for the long discussion about it here is that people believe that there will be hundreds and thousands of registration authorities eventually. Registration authorities are the organizations that establish and maintain the identities of individuals known to them. Company HR departments, universities, retailers, and hundreds of other types of organizations either already are or plan to become registration authorities. CAs are registration authorities themselves, or they can accept input from RAs and just be certificate issuers. Given this explosion in the number of RAs/CAs, it becomes quite clear why certification chains become complex and important in establishing trust.

If the relying party does not have the public key of the CA that signed the certificate presented, that party must acquire the key from someone else who vouches for that CA for the sub-CA below her. This means moving up the trust hierarchy to another CA that signed the certificate. This concept is shown in the diagram in Figure 3.10. Here, relying parties are end-entities (EE) that are presented with certificates from sub-CAs. Since the sub-CAs are not known or trusted by the EEs, they require the certificate of the CA that signed the sub-CAs certificates. These CAs further up the trust hierarchy vouch for the sub-CA. These root CAs are the trust anchors for the EE allowing them to trust the certificates from the sub-CAs.

Figure 3.10 Certificate authority trust hierarchy.

Cross-Certification: Federated Trust
In the next few years, you will be seeing a lot more of cross-certification in contrast to the rooted chains of trust. In this trust model, another CA has performed the identification procedure on an individual who is otherwise a total stranger to your CA and all organizations yours serves. But the two CAs have agreed that their processes are in lock-step and agree to cross-certify each other’s certificates. In other words, you will accept the other CA’s certificates on faith. Figure 3.11 shows how the cross-certification model links together the sub-CAs and end entities (EEs) of two top-level CAs that have agreed to cross-certify each other.
Several organizations have implemented this model. The first was Identrus, now merged with Digital Signature Trust (DST). Several dozen banks have agreed to cross-certify based on standards and a root certificate from Identrus. If a Bank A customer is identified and issued a certificate, Bank B, being part of the same Identrus-rooted network, would accept the Bank A customer as if she were a Bank B customer.

The Federal Bridge program is a newer example of this model that is currently being deployed. Different departments within the federal government are agreeing to accept certificates from other departments. This has led to a strong push for levels of trust in certificates. A level from 1 to 4 would be based on how thorough the identification process was, as described earlier in Table 3.2. The AICPA—the organization that sets standards for all the audit firms—has recently decided to adopt this model and make it part of the standards to which it audits certificate authorities. The good news for Web services is that, because these levels would be specified as a standard X.509 extension, the Web services themselves can check and enforce compliance with a predetermined trust level.

Root CAs Are Trusted by Everyone

There are not many root CAs because, for their certificates to be understood by your tools, the public key for their self-signed certificates must already be accessible to those tools. For the most common kinds of certificates—those used for SSL—this has occurred by embedding the root keys for the root CAs right in the browser. In fact, the browsers typically have dozens of root keys and another batch of intermediate CA keys as well. Just a portion of the pre-installed roots on Windows XP are shown in Figure 3.12.

With Web services, it is not the browser having the keys that matters. It is the nodes or termination points of the Web service that matter, where signature validation or decryption must occur. As a Web service is deployed, a crucial step in that deployment is to make sure the appropriate public keys of all CAs whose certificates may be seen by that Web service are installed at all endpoint servers.
Figure 3.12  Pre-installed roots in Windows XP.

Controlling Trust in the Root CAs
You might be surprised to learn how the CA business started. How did companies such as VeriSign ensure their root keys were placed in browsers? Initially, they just made deals with Netscape and Microsoft. Netscape considered these deals much like a partnering “pay-to-play” program, so if you paid the fee, your key went in. Microsoft did not charge for the honor. But there was still very little scrutiny or process in choosing whose key went in.

The situation changed in 2001 when Microsoft decided that it could no longer use the ad hoc process. By this time, more than 300,000 active SSL certificates were in use, and consumers were committing serious money based on trust in these certificates. Microsoft realized it had a house of cards if it did not do something.

As mentioned previously, an organization called the AICPA establishes best practices for the accounting/auditing industry. Its members are accountants who do financial and other sorts of audits. The leadership of the AICPA comes from the big multinational accounting firms such as Ernst & Young, KPMG, Price Waterhouse Coopers, and Deloitte & Touche. The AICPA wanted to create new business for its members and so established best practices around the processes CAs must use to issue and manage digital certificates. With no other standard to rely on, Microsoft decided this approach was better than none and made an announcement in 2002 that the AICPA’s WebTrust audit for CAs would be the new bar that CAs must get over to be allowed into the root store of the browser and of its application servers (including NET servers used in Web services).

Key Escrow for Recovering Lost Private Keys
Key escrow is a very important and controversial aspect of PKI. This technology is about storage and retrieval of private keys to recover data in the absence of the private key
owner. Key escrow goes against the very idea of a private key. The private key may ultimately be accessed by more than the owner of the key, which lessens the case for non-repudiation. Key escrow is often considered a necessary evil when critical information may be encrypted with this private key and loss of the key, death of its owner, or some sort of fraud means that information might be forever irretrievable. Requirements for key escrow/recovery systems may come from customer support or legal or policy requirements. International PKI implementations may require key escrow to comply with government and law enforcement restrictions.

The concept of a private key is crucial to the effectiveness of PKI because so many downstream PKI concepts depend heavily on the assumption that the private key is never compromised. And yet, humans are fallible and do indeed lose keys. If a critical private key is lost, there truly is no way to decrypt any data its matching public key encrypted. That data is lost forever—even to the CIA, NSA, or FBI. The risk of losing that data are too high for many reasons, which is what drives the need for a key recovery scheme. There are other reasons for a server-based key escrow scheme as well.

Key escrow may be an important consideration in Web service deployments. Critical information flowing through the Web services infrastructure may have been encrypted, and the governing organization, although it wants to maintain the confidentiality of this data, cannot afford to lose access to it forever. If the private keys used to encrypt it are lost, that is what would happen.

When an enterprise entrusts employees or agents with confidential data, this data is at risk of being forever lost if an employee loses her private key. Therefore, the employer may want a way to recover the private key in that eventuality. If the employee uses the key to encrypt email messages, and she violates company policy or if law enforcement subpoenas the email, key recovery is needed to decrypt the email. Finally, if an employee leaves the company, he may not cooperate with the return of the private key and that key may need to be recovered.

Certificate Revocation for Dealing with Public Keys Gone Bad

Key escrow is an optional feature of PKI, but certificate revocation is not. Revocation is an essential part of the certificate process to establish and maintain trust. Authentication of clients and servers requires a way to verify each certificate within the chain, as well as a way to determine whether a certificate is valid or revoked. A certificate could be revoked if a key is compromised or lost or as a result of modification of privileges, misuse, or employment termination. It is essential, especially for Web services, that near real-time revocation of certificates be achieved.

Currently, two technologies are used for revocation: certificate revocation lists (CRLs) and online certificate status protocol (OCSP).

CRL Certificate Revocation Checking

The CRL is an up-to-date list of all certificates revoked; a CA must keep this list accessible to relying parties. It goes without saying that the CA must make it easy for registration authorities to revoke any given certificate (but prove that they have the right to do
With CRLs, relying parties have the burden of checking this list each time a certificate is presented. Best practices call for a certificate deployment point (CDP) URL to be embedded in the certificate. A CDP is a pointer to the location of the CRL on the Internet, accessible programmatically by any relying party’s applications.

CRLs are usually updated once per day because the process of generating them is non-trivial and time-consuming. When an organization is dealing with a compromised key or a rogue employee, once-per-day updates can mean a huge loss during the day compromise occurred, especially when interactions are automated with Web services. Currently, almost no one checks revocation lists. Although CRLs are created obediently by the sponsoring CAs and numerous tools can and do process them, there are so many unsolved problems with them that, in our view, CRLs on the Internet are a technological failure.

**OCSP Certificate Revocation Checking**

OCSP was an attempt to create a much finer-grained protocol for essentially real-time revocation checking. But as in CRLs, the trust information provided requires it be signed by the originating CA, which is an expensive operation to perform in real time. The best case on an unloaded system of moderate speed is 26ms response time for a single OCSP request in our tests. In our view, this makes standard OCSP so limited in scope that it will continue to be only a bit player in revocation solutions.

In our opinion, there is much promise in emerging techniques that are scalable and respond in microseconds and still conform to the OCSP standard without requiring time-consuming digital signatures on each request. One such approach is Silvio Micali’s technique based on chains of hashed secret codes. Web services will require such a high-speed revocation system. For details, see the section on Silvio Micali’s High-Speed Validation/Revocation in Appendix A.

**Trust Services**

What this discussion about CAs, trust chains, cross-certification, key escrow, and certificate revocation leads to is an overriding need for Web services that offer trust services to other Web services. The developer of a Web service does not want to be burdened with all the issues we have just outlined. Furthermore, companies will consider how these issues are handled as core corporate security policies and will want consistent, reliable, and centralized administration of these policies.

Having trust services available as Web services means developers can avoid writing key management and signature processes in every application. All the critical trust services will be encapsulated into a reusable service. Trust services must deal with the complexities of PKI, keys, signatures, encryption, and the like. Trust services are Web services that provide these and other security services for any application that needs them. Because they are just Web services themselves, they are accessed through SOAP messages. With this approach, special PKI client code does not need to be deployed any longer.
Key Management Services

The first trust service needed is one that manages the registration, distribution, and lifecycle of public keys. Like any Web service, it needs to be a service that has a WSDL and is accessed via SOAP.

The W3C XML Key Management Specification (XKMS) specifies protocols for distributing and registering public keys suitable for use in conjunction with XML Signature and XML Encryption. See Chapter 9 for details on XKMS.

Digital Signature Services

The OASIS Digital Signature Services (DSS) is developing a set of services to help manage signature creation and validation. This will completely hide the PKI complexities of signature processing.

Single Sign-On Services

Single sign-on (SSO) services help facilitate authentication for Web services. Using SAML as the base standard (the subject of Chapter 6, “Portable Identity, Authentication, and Authorization”), a user can log in to the SSO service, provide one or two factor authentication challenges (user ID, password, and perhaps smart card or biometric), obtain a credential (or security token), and use that credential in SOAP headers on all subsequent Web services to provide necessary authentication information required by the Web services. Because Web services are not interactive (they are computer-to-computer), this approach is usually required when using Web services needing authentication. Single sign-on services are being developed by the Liberty Alliance and by Microsoft Passport.

Access Control Services

Sometimes referred to as Entitlement Services, Access Control Services (ACS) provide centralized access control policies. They support one of several evolving OASIS access control standards such as XACML and XrML. These standards are covered in Chapter 9.

Security in a Box

Because of its compute-intensive nature, application-level XML security is being built into hardware for acceleration. Such security will be required in high-volume applications. The box will be the termination point for encryption, signature, and transport security. These hardware accelerators will provide dramatic acceleration of time-consuming, compute-intensive cryptography tasks for Web services.

Billing and Metering Services

The last of the security-related Web services is billing and metering services. These services keep track of a user’s service utilization and accrue billing information. They are vital to the business models that Web services will enable and encourage. You can’t charge for the use of Web services if you cannot authenticate legitimate users and bill them for their usage.
“PKI’s not dead. It’s just resting!”
That controversial heading comes from a 2002 article by Peter Gutmann. Many had proclaimed PKI as dead because it was so hard to deploy and so few companies had deployed it successfully. It is true that PKI tried to be all things to all people and therefore sacrificed some utility.

One huge reason PKI has not “taken off” is that, because revocation is so difficult, it has never been implemented on broad networks such as the Internet. It is hard to know where to find revocation information. Revocation information cannot be issued often enough to be useful. Without effective revocation, PKI can never really be useful because trust in the privacy of private keys and the integrity and identity of public keys are absolutely fundamental to the whole notion of PKI.

Web services in general and XKMS in particular will bring back a new lease on life to PKI—but with some changes. CAs will have to provide a managed PKI service available as a high-performance, high-availability Web service. Locally meaningful identifier names will be used as opposed to the X.500-derived DN structure used in X.509 certificates.

Revocation is built into the XKMS protocol because you are establishing a direct relationship to the third-party trust provider (the CA) that issued the credential in the first place. Therefore, revocation is direct and explicit in the XKMS protocol.

Most importantly, Web services in general and the closed environments in which Web services will typically be used create application-specific PKIs, which always worked the best at solving the general problem with PKI.

(Based on an article by Peter Gutmann in IEEE Security, August 2002.)

SSL Transport Layer Security
Secure Socket Layer (SSL), also called Transport Layer Security (TLS), was invented by Netscape to provide for secure e-commerce transactions between a Web browser and a Web server. SSL is arguably the most widely used implementation of PKI. It is important and relevant to a discussion of Web Services Security because it is so easy to use, it is already deployed in virtually every organization, and it is so effective for certain types of Web service deployments.

A Description of the SSL Protocol
SSL security is most commonly used for browser-to-server security in e-commerce transactions. Virtually all browsers support SSL. Likewise, virtually all Web servers—or more correctly, Web application containers, such as Microsoft Internet Information Server, BEA’s WebLogic Server, IBM’s WebSphere—do as well. Because most, if not all,
of the current Web services containers (for example, Microsoft’s .NET framework, BEA’s WebLogic Workshop, IBM’s WebSphere Studio Application Developer) are also Web application containers, you can use the existing built-in Transport Layer Security for Web services without modifying a thing.

SSL is effective at maintaining confidentiality of transactions. It implements shared key encryption between its endpoints after first transporting the shared key via public key cryptography. SSL will prove to be broadly useful for Web Services Security, especially in early implementations, because those early Web services require only simple point-to-point encryption and possibly the level of authentication SSL can provide. As Chapter 7 on WS-Security explains, SSL also will be useful as an added layer of transport security underneath the message-level security this book describes.

Four options are available when you are using SSL Transport Layer Security over HTTP (which you will see as HTTPS):

- **SSL/TLS (one-way)**—This is the same SSL that you use online when entering your credit card on a Web site. Using one-way SSL, you obtain two benefits:
  - Your client (browser) verifies the identity of the server.
  - You have an encrypted session between your client and the server.

**Note**

In one-way SSL, the idea that you are verifying the identity of the Web server’s owner is technically valid but currently extremely weak in browser implementations. When you go into an SSL session with a Web site, the lock symbol in your browser lights up. Double-clicking on the lock opens a dialog box showing the server’s certificate. You can view the identity of the company validated by the certificate authority by clicking on the Details tab and then the specific line item called subject. Look at the O= (for organization equals), and you will see the validated name of the company.

This information is buried ridiculously deep, and for all intents and purposes, it has no value to an individual viewing a Web page. Many legitimate sites transfer control to another company when they get into a secure session, and the user never knows this transfer happens. You would be surprised if the real name of the company running the secure pages you encounter on the Web were to show up.

- **Basic authentication (basic auth)**—With basic authentication, the client sends a username and password for authentication. These credentials are sent in the clear, so it is common practice to combine basic auth with one-way SSL.

- **Digest authentication**—Digest authentication addresses the issue of the password being in the clear by using hashing technology (for example, MD5 or SHA1). Basically, it involves the server passing a nonce (just a number or string chosen by the server) down to the client, which the client then combines with the password and hashes using the algorithm specified by the server. The server then receives the hash and runs the same hash algorithm on the password and nonce that it has.

A couple of problems with digest authentication prevent it from being used often.
Note

The main issue with digest authentication is that it is not supported in a standard way across Web servers and clients. The other issue is, for the server to participate, it must have access to a clear password, meaning the password must be stored in the clear. Many implementations store only a hashed password, making it impossible to participate in a digest protocol defined this way.

- **Client certificates (two-way or mutually authenticated SSL)**—This is one-way SSL, as discussed previously, with the addition that the client must also provide an X.509 certificate to authenticate itself with the server. The protocol involves challenge and response between the server and the client in which information is digitally signed to prove the possession of the private key. This, in turn, is to prove that the identity based on the public key contained in the certificate can be trusted. This option is powerful, but it adds a great deal of complexity, especially for Web applications with large numbers of consumer clients, because each client needs to be issued a certificate to gain access. In some Web services scenarios—such as business-to-business—this may not be quite so onerous because the number of server clients (a seemingly contradictory term but one that will be common in Web services where a server machine acts as a client because it is the service requestor) is typically small. However, one of the major complexities of using client certificates is that either
  - Your company needs to issue X.509 certificates to each client, meaning you need to get certificate management software and become a certificate authority.
  - You need to work with a certificate authority managed service. Also, unfortunately, configuring your Web server to accept client certificates is often not for the faint of heart, so you will need an experienced security systems administrator to be successful.

We will briefly explain how the SSL protocol works using an example in which the “client” is the Web service **requestor** and the “server” is the Web service **provider**. The steps outlined in Figure 3.13 are as follows:

1. The service requestor opens a connection to the service provider and sends a **ClientHello** message. This message lists the capabilities of the service requestor, including the version of SSL it is using and the cipher suites (cryptographic algorithms) it supports.
2. The service provider responds with a ServerHello message. The service provider returns the cipher suite it has chosen and a session ID that identifies this connection.

3. The service provider sends its certificate. This X.509 site certificate is signed by a certificate authority. The certificate contains the service provider’s public key. It is assumed that the service requestor has access to the signing CA’s public key perhaps because it is already installed in the trust store residing at the service requestor.

4. The service provider (optionally) sends the service requestor a request for its certificate. Client authentication will be necessary for almost all Web services, but if the Web service is a thin veneer directly to a human user, username/password authentication may be used instead.

5. The service requestor (optionally, if requested in step 4) sends its client certificate. It will have been signed by some trust authority to which the service provider will assign trust levels based on its policies in place. The service provider will also have to have direct access to the signing CA’s public key. The service provider may or may not choose to trust that this is really the entity it claims to be, which will determine whether it agrees to provide the service being requested.

6. The service requestor sends a ClientKeyExchange message. The service requestor has created a shared key and is sending it to the service provider with this message. The full session key is not created directly because different shared key ciphers use different key lengths. The service requestor encrypts this shared key using the service provider’s public key and sends it back to the service provider.

7. The service requestor (optionally, if requested in step 4) sends a CertificateVerify message. This is the authentication of the client step in client-authenticated, or two-way, SSL. The service requestor has to prove it knows the correct private key. The shared key from step 6 is signed using the service requestor’s private key (which only it has and which it guarantees it has kept secret) and sent to the service provider, which verifies this using the service requestor’s public key forwarded earlier embedded in the X.509 certificate.

8. Both service requestor and service provider send a ChangeCipherSpec message. It simply says both sides are ready to communicate in encrypted form using the shared secret session key.

9. Both service requestor and service provider send a Finished message. This is an MD5 and SHA hash of the entire conversation up to this point to confirm that the entire conversation was received by the other party intact and not tampered with en route. What is finished is just the handshake; the real communication of the confidential message is now about to begin.
When people talk about SSL Transport Layer Security, they sometimes use the term *secure pipe* as a metaphor. This means that after the SSL endpoints go through their protocol (either one-way or two-way SSL), a cryptographic pathway is created between the two endpoints (see Figure 3.14). Web services that are based on HTTP as the transport flow through this secure pipe, making all messages sent back and forth confidential. Remember, though, that the messages are encrypted just during transport. At the receiving endpoint, the messages are decrypted by the server. If the Web service uses multiple hops on its way to its real destination or if persistent confidentiality (encryption) is needed for the messages, SSL does not provide a solution. Hence, we still need to learn a lot more about message-level security applied to Web services messages.
Summary

This chapter layers an information security foundation on top of the essential nature of the Web services established in the preceding chapter. A solid understanding of both is necessary to build secure Web services. Securing Web services requires securing the messages used to communicate between distributed systems and the applications that run on them. Web services add significant complexity to the basic problem of securing distributed systems because Web services employ shared services, and messages may follow multi-hop topologies. Instead of a focus on perimeter or network security, what is needed instead is persistent message-level security that delivers the core security principles of confidentiality, integrity, and non-repudiation to a distributed message-based architecture.

Shared key technologies are the key to confidentiality. Shared key is also referred to as symmetric key or secret key. In shared key encryption, the same key is used to encrypt and decrypt. This means the sender and recipient both need the same key but must find a way to share it and still keep it absolutely secret. Positive attributes of shared key encryption are that it is fast and can handle unlimited message size. However, getting the shared secret key to both ends of the conversation securely is very hard.

Kerberos is an alternative approach to shared keys that is useful only in a closed trust domain where all identities are known. Its importance to Web services security is that when services cross organizational boundaries and therefore cross trust domains, provisions have to be made to map between Kerberos and other trust environments.

Public key technologies are the key to integrity and non-repudiation. Public key encryption is also referred to as asymmetric encryption. Keys come in pairs, where one is used to encrypt and only its mate is used to decrypt messages. This technology is much slower than shared key encryption and is limited to small message sizes. However, it is of critical importance in key exchange to enable shared key encryption, and it is the basis for digital signatures. Public key encryption is used to securely establish secret shared keys in XML Encryption and is the core cryptographic technology used in XML Signature.
Public key encryption is at the heart of digital signatures. Hashing is also a critical security technology in digital signatures. A digital signature is an encrypted message digest of a hashed plaintext message sent along with the signer’s public key and the original plaintext message. Signature verification involves using the sender’s public key to decrypt the signed message digest and verifying that this exactly matches the locally computed message digest from the plaintext message also sent along. An XML Signature is a digital signature expressed in XML.

Trust issues are prevalent and fundamental to public key technologies. The private key must be kept secret and in the control of its owner without exception. The public key of an unknown identity must be vouched for by a trusted third party. Public keys need containers called digital certificates to transport them. Trusted third parties that issue digital certificates to identified individuals are called certificate authorities (CA). The infrastructure for dealing with CAs, certificates, and keys is called the Public Key Infrastructure (PKI). Numerous complex issues revolve around trusting the trusted third-party CAs. In a certification chain, one CA vouches for and signs the certificate of another CA. Root certificates are self-signed by a CA that is at the root of a certification chain and is so well known that its public keys are pre-installed in servers, browsers, and other repositories where applications can easily access them. A major issue for PKI in the past has been the lack of effective revocation in which the up-to-date validity of a certificate can be checked. This issue becomes even more critical for Web services.

Numerous Web services designed to deliver trust services are emerging. These services will include, but not be limited to, key management services, digital signature services, single sign-on services, access control services, hardware accelerators, and billing and metering.

Secure Socket Layer (SSL) is a mature, tried-and-true transport layer security. It works on HTTP, so it “just works” for Web services. All application servers support SSL. SSL supports two-way authentication, although its more common usage with browsers is one-way. SSL should always be considered first for Web services security because it is effective for message confidentiality in simple point-to-point Web services.

Resources


