Java security, Part 1: Crypto basics

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

There is perhaps no software engineering topic of more timely importance than application security. Attacks are costly, whether the attack comes from inside or out, and some attacks can expose a software company to liability for damages. As computer (and especially Internet) technologies evolve, security attacks are becoming more sophisticated and frequent. Staying on top of the most up-to-date techniques and tools is one key to application security; the other is a solid foundation in proven technologies such as data encryption, authentication, and authorization.

The Java platform, both the basic language and library extensions, provides an excellent foundation for writing secure applications. This tutorial covers the basics of cryptography and how it is implemented in the Java programming language, and it offers example code to illustrate the concepts.

In this first installment of a two-part tutorial, we cover material in the library extensions -- now part of the JDK 1.4 base -- known as Java Cryptography Extension (JCE) and Java Secure Sockets Extension (JSSE). In addition, this tutorial introduces the CertPath API, which is new for JDK 1.4. In Part 2 (see Resources on page 31), we'll expand the discussion to encompass access control, which is managed in the Java platform by the Java Authentication and Authorization Service (JAAS).

Should I take this tutorial?

This is an intermediate-level tutorial; it assumes you know how to read and write basic Java programs, both applications and applets.

If you are already a Java programmer and have been curious about cryptography (topics such as private and public key encryption, RSA, SSL, certificates) and the Java libraries that support them (JCE, JSSE), this tutorial is for you. It does not assume any previous background in cryptography, JCE, or JSSE.

This tutorial introduces the basic cryptographic building block concepts. Each concept is followed by the Java implementation considerations, a code example, and the results of the example execution.

Tools, code samples, and installation requirements

You'll need the following items to complete the programming exercises in this tutorial:

- *JDK 1.4, Standard Edition*
- The tutorial source code and classes, JavaSecurity1-source.jar, so that you can follow the examples as we go along
- The *Bouncy Castle Crypto library* for the RSA example

- A browser that supports the Java 1.4 plug-in

You can use JDK 1.3.x, but you must install JCE and JSSE yourself.

A note on the code examples

The code examples dump encrypted data directly to the screen. In most cases, this will result in strange-looking control characters, some of which may occasionally cause screen-formatting problems. This is not good programming practice (it would be better to convert them to displayable ASCII characters or decimal representations), but has been done here to keep the code examples and their output brief.

In most cases in the example execution sections, the actual strings have been modified to be compatible with the character set requirements of this tutorial. Also, in most examples, we look up and display the actual security provider library used for a given algorithm. This is done to give the user a better feel of which libraries are called for which functions. Why? Because, in most installations, there are a number of these providers installed.

About the author

Brad Rubin is principal of Brad Rubin & Associates Inc., a computer-security consulting company specializing in wireless network and Java application security and education. Brad spent 14 years with IBM in Rochester, MN, working on all facets of the AS/400 hardware and software development, starting with its first release. He was a key player in IBM's move to embrace the Java platform, and was lead architect of IBM's largest Java application, a business application framework product called SanFrancisco (now part of WebSphere). He was also chief technology officer for the Data Storage Division of Imation Corp., as well as the leader of its R&D organization.

Brad has degrees in Computer and Electrical Engineering, and a Doctorate in Computer Science from the University of Wisconsin, Madison. He currently teaches the Senior Design course in Electrical and Computer Engineering at the University of Minnesota, and will develop and teach the university's Computer Security course in Fall 2002. You can reach Brad at *BradRubin@BradRubin.com*.

Section 2. Java security programming concepts

How the Java platform facilitates secure programming

The Java programming language and environment has many features that facilitate secure programming:

- **No pointers**, which means that a Java program cannot address arbitrary memory locations in the address space.

- **A bytecode verifier**, which operates after compilation on the .class files and checks for security issues before execution. For example, an attempt to access an array element beyond the array size will be rejected. Because buffer overflow attacks are responsible for most system breaches, this is an important security feature.

- **Fine-grained control over resource access** for both applets and applications. For example, applets can be restricted from reading from or writing to disk space, or can be authorized to read from only a specific directory. This authorization can be based on who signed the code (see The concept of code signing on page 26) and the http address of the code source. These settings appear in a java.policy file.

- **A large number of library functions** for all the major cryptographic building blocks and SSL (the topic of this tutorial) and authentication and authorization (discussed in the second tutorial in this series). In addition, numerous third-party libraries are available for additional algorithms.

What are secure programming techniques?

Simply put, there are a number of programming styles and techniques available to help ensure a more secure application. Consider the following as two general examples:

- **Storing/deleting passwords.** If a password is stored in a Java `String` object, the password will stay in memory until it is either garbage collected or the process ends. If it is garbage collected, it will still exist in the free memory heap until the memory space is reused. The longer the password `String` stays in memory, the more vulnerable it is to snooping.

  Even worse, if real memory runs low, the operating system might page this password `String` to the disk's swap space, so it is vulnerable to disk block snooping.

  To minimize (but not eliminate) these exposures, you should store passwords in `char` arrays and zero them out after use. (`String`s are immutable, so you can't zero them out.)

- **Smart serialization.** When objects are serialized for storage or transmission any private fields are, by default, present in the stream. So, sensitive data is vulnerable to snooping. You can use the `transient` keyword to flag an attribute so it is skipped in the streaming.

We'll be discussing these and other techniques in more detail when we encounter a need for

them throughout the tutorial.

Security is integrated in JDK 1.4

Prior to JDK 1.4, many security functions had to be added to the base Java code distribution as extensions. Tight U.S. export restrictions required this separation of function.

Now, new relaxed regulations open the door to tighter integration of security features and the base language. The following packages -- used as extensions prior to the 1.4 release -- are now integrated into JDK 1.4:

- **JCE** (Java Cryptography Extension)
- **JSSE** (Java Secure Sockets Extension)
- **JAAS** (Java Authentication and Authorization Service)

JDK 1.4 also introduces two new functions:

- **JGSS** (Java General Security Service)
- **CertPath API** (Java Certification Path API)

JCE, JSSE, and the CertPath API are the subject of this tutorial. We'll focus on JAAS in the next tutorial in this series. Neither tutorial covers the JGSS (which provides a generic framework to securely exchange messages between applications).

Security is enriched with third-party libraries

We can enhance an already rich set of functions in the current Java language with third-party libraries, also called *providers*. Providers add additional security algorithms.

As an example of a library, we'll be working with the Bouncy Castle provider (see Resources on page 31). The Bouncy Castle library provides other cryptographic algorithms, including the popular RSA algorithm discussed in What is public key cryptography? on page 15 and What are digital signatures? on page 18 of this tutorial.

While your directory names and *java.security* files might be a bit different, here is the template for installing the Bouncy Castle provider. To install this library, download the bcprov-jdk14-112.jar file and place it in the j2sdk1.4.0\jre\lib\ext and the Program Files\Java\J2re1.4.0\lib\ext directories. In both java.security files, which are in the same directories as above but use "security" instead of "ext", add the following line:

```
security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider
```

to the end of this group of lines:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
```

```
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Looking ahead

In this section, we've introduced the features the Java language provides, either fully integrated or extension-based, that help to ensure that programming remains secure. We've offered some general examples of secure programming techniques to help you become familiar with the concept. We've covered security technologies that used to be extensions but are now integrated into the version 1.4 release; we've also noted two new security technologies. And we've demonstrated that third-party libraries can enhance security programs by offering new technologies.

In the remainder of this tutorial, we will familiarize you with these concepts designed to provide secure messaging (as they apply to Java programming):

- **Message digests.** Coupled with message authentication codes, a technology that ensures the integrity of your message.
- **Private key encryption.** A technology designed to ensure the confidentiality of your message.
- **Public key encryption.** A technology that allows two parties to share secret messages without prior agreement on secret keys.
- **Digital signatures.** A bit pattern that identifies the other party's message as coming from the appropriate person.
- **Digital certificates.** A technology that adds another level of security to digital signatures by having the message certified by a third-party authority.
- **Code signing.** The concept that a trusted entity embeds a signature in delivered code.
- **SSL/TLS.** A protocol for establishing a secure communications channel between a client and a server. Transport Layer Security (TLS) is the replacement for Secure Sockets Layer (SSL).

As we discuss each of these topics, we'll serve up examples and sample code.

Section 3. Ensuring the integrity of a message

Overview

In this section, we will learn about message digests, which take the data in a message and generate a block of bits designed to represent the "fingerprint" of the message. We will also cover the JDK 1.4-supported algorithms, classes, and methods related to message digests, offer a code example and a sample execution code for both the message digest and message authentication features.

What is a message digest?

A *message digest* is a function that ensures the integrity of a message. Message digests take a message as input and generate a block of bits, usually several hundred bits long, that represents the fingerprint of the message. A small change in the message (say, by an interloper or eavesdropper) creates a noticeable change in the fingerprint.

The message-digest function is a one-way function. It is a simple matter to generate the fingerprint from the message, but quite difficult to generate a message that matches a given fingerprint.

Message digests can be weak or strong. A checksum -- which is the XOR of all the bytes of a message -- is an example of a weak message-digest function. It is easy to modify one byte to generate any desired checksum fingerprint. Most strong functions use hashing. A 1-bit change in the message leads to a massive change in the fingerprint (ideally, 50 percent of the fingerprint bits change).

Algorithms, classes, and methods

JDK 1.4 supports the following message-digest algorithms:

- **MD2** and **MD5**, which are 128-bit algorithms
- **SHA-1**, which is a 160-bit algorithm
- **SHA-256, SHA-383,** and **SHA-512**, which offer longer fingerprint sizes of 256, 383, and 512 bits, respectively

MD5 and SHA-1 are the most used algorithms.

The `MessageDigest` class manipulates message digests. The following methods are used in the Message digest code example on page 8 :

- `MessageDigest.getInstance("MD5")`: Creates the message digest.
- `.update(plaintext)`: Calculates the message digest with a plaintext string.
- `.digest()`: Reads the message digest.

If a key is used as part of the message-digest generation, the algorithm is known as a *message-authentication code* . JDK 1.4 supports the HMAC/SHA-1 and HMAC/MD5 message-authentication code algorithms.

The Mac class manipulates message-authentication codes using a key produced by the KeyGenerator class. The following methods are used in the Message authentication code example on page 9 :

- KeyGenerator.getInstance("HmacMD5") and .generateKey(): Generates the key.
- Mac.getInstance("HmacMD5"): Creates a MAC object.
- .init(MD5key): Intializes the MAC object.
- .update(plaintext) and .doFinal(): Calculates the MAC object with a plaintext string.

Message digest code example

```
import java.security.*;
import javax.crypto.*;
//
// Generate a Message Digest
public class MessageDigestExample {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {

      System.err.println("Usage: java MessageDigestExample text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // get a message digest object using the MD5 algorithm
    MessageDigest messageDigest = MessageDigest.getInstance("MD5");
    //
    // print out the provider used
    System.out.println( "\n" + messageDigest.getProvider().getInfo() );
    //
    // calculate the digest and print it out
    messageDigest.update( plainText);
    System.out.println( "\nDigest: " );
    System.out.println( new String( messageDigest.digest(), "UTF8") );
  }
}
```

Message digest sample execution

```
D:\IBM>java MessageDigestExample "This is a test!"

SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests
; SecureRandom; X.509 certificates; JKS keystore; PKIX CertPathValidator
; PKIX CertPathBuilder; LDAP, Collection CertStores)
```

```
Digest:
D93,.x2%$kd8xdp3di5*
```

Message authentication code example

```
import java.security.*;
import javax.crypto.*;
//
// Generate a Message Authentication Code
public class MessageAuthenticationCodeExample {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {
      System.err.println
        ("Usage: java MessageAuthenticationCodeExample text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // get a key for the HmacMD5 algorithm
    System.out.println( "\nStart generating key" );
    KeyGenerator keyGen = KeyGenerator.getInstance("HmacMD5");
    SecretKey MD5key = keyGen.generateKey();
    System.out.println( "Finish generating key" );
    //
    // get a MAC object and update it with the plaintext
    Mac mac = Mac.getInstance("HmacMD5");
    mac.init(MD5key);
    mac.update(plainText);
    //
    // print out the provider used and the MAC
    System.out.println( "\n" + mac.getProvider().getInfo() );
    System.out.println( "\nMAC: " );
    System.out.println( new String( mac.doFinal(), "UTF8") );

  }
}
```

Message authentication sample execution

```
D:\IBM>java MessageAuthenticationCodeExample "This is a test!"

Start generating key
Finish generating key

SunJCE Provider (implements DES, Triple DES, Blowfish, PBE, Diffie-Hellman,
HMAC-MD5, HMAC-SHA1)

MAC:
Dkdj47x4#.@kd#n8a-x>
```

Note that the key generation takes a long time because the code is generating excellent quality pseudo-random numbers using the timing of thread behavior. Once the first number is generated, the others take much less time.

Also, notice that unlike the message digest, the message-authentication code uses a cryptographic provider. (For more on providers, see Security is enriched with third-party libraries on page 5 .)

Section 4. Keeping a message confidential

Overview

In this section, we'll examine the uses of private key encryption and focus on such concepts as cipher blocks, padding, stream ciphers, and cipher modes. We'll quickly detail cipher algorithms, classes, and methods and illustrate this concept with a code example and sample executions.

What is private key cryptography?

Message digests may ensure integrity of a message, but they can't be used to ensure the confidentiality of a message. For that, we need to use *private key* cryptography to exchange private messages.

Consider this scenario: Alice and Bob each have a shared key that only they know and they agree to use a common cryptographic algorithm, or cipher. In other words, they keep their key private. When Alice wants to send a message to Bob, she encrypts the original message, known as *plaintext,* to create *ciphertext* and then sends the ciphertext to Bob. Bob receives the ciphertext from Alice and decrypts the ciphertext with his private key to re-create the original plaintext message. If Eve the eavesdropper is listening in on the communication, she hears only the ciphertext, so the confidentiality of the message is preserved.

You can encrypt single bits or chunks of bits, called blocks. The blocks, called *cipher blocks*, are typically 64 bits in size. If the message is not a multiple of 64 bits, then the short block must be *padded* (more on padding at What is padding? on page 11). Single-bit encryption is more common in hardware implementations. Single-bit ciphers are called *stream ciphers*.

The strength of the private key encryption is determined by the cryptography algorithm and the length of the key. If the algorithm is sound, then the only way to attack it is with a brute-force approach of trying every possible key, which will take an average of *(1/2)*2*n* attempts, where *n* is the number of bits in the key.

When the U.S. export regulations were restrictive, only 40-bit keys were allowed for export. This key length is fairly weak. The official U.S. standard, the DES algorithm, used 56-bit keys and this is becoming progressively weaker as processor speeds accelerate. Generally, 128-bit keys are preferred today. With them, if one million keys could be tried every second, it would take an average of many times the age of the universe to find a key!

What is padding?

As we mentioned in the previous panel, if a block cipher is used and the message length is not a multiple of the block length, the last block must be padded with bytes to yield a full block size. There are many ways to pad a block, such as using all zeroes or ones. In this tutorial, we'll be using PKCS5 padding for private key encryption and PKCS1 for public key encryption.

With PKCS5, a short block is padded with a repeating byte whose value represents the number of remaining bytes. We won't be discussing padding algorithms further in this tutorial,

but for your information, JDK 1.4 supports the following padding techniques:

- No padding
- PKCS5
- OAEP
- SSL3

The BouncyCastle library (see Security is enriched with third-party libraries on page 5 and Resources on page 31) supports additional padding techniques.

Modes: Specifying how encryption works

A given cipher can be used in a variety of *modes*. Modes allow you to specify how encryption will work.

For example, you can allow the encryption of one block to be dependent on the encryption of the previous block, or you can make the encryption of one block independent of any other blocks.

The mode you choose depends on your needs and you must consider the trade-offs (security, ability to parallel process, and tolerance to errors in both the plaintext and the ciphertext). Selection of modes is beyond the scope of this tutorial (see Resources on page 31 for further reading), but again, for your information, the Java platform supports the following modes:

- **ECB** (Electronic Code Book)
- **CBC** (Cipher Block Chaining)
- **CFB** (Cipher Feedback Mode)
- **OFB** (Output Feedback Mode)
- **PCBC** (Propagating Cipher Block Chaining)

Algorithms, classes, and methods

JDK 1.4 supports the following private key algorithms:

- **DES.** DES (Data Encryption Standard) was invented by IBM in the 1970s and adopted by the U.S. government as a standard. It is a 56-bit block cipher.

- **TripleDES.** This algorithm is used to deal with the growing weakness of a 56-bit key while leveraging DES technology by running plaintext through the DES algorithm three times, with two keys, giving an effective key strength of 112 bits. TripleDES is sometimes known as DESede (for encrypt, decrypt, and encrypt, which are the three phases).

- **AES.** AES (Advanced Encryption Standard) replaces DES as the U.S. standard. It was invented by Joan Daemen and Vincent Rijmen and is also known as the Rinjdael algorithm. It is a 128-bit block cipher with key lengths of 128, 192, or 256 bits.

- **RC2, RC4,** and **RC5.** These are algorithms from a leading encryption security company, RSA Security.

- **Blowfish.** This algorithm was developed by Bruce Schneier and is a block cipher with variable key lengths from 32 to 448 bits (in multiples of 8), and was designed for efficient implementation in software for microprocessors.

- **PBE.** PBE (Password Based Encryption) can be used in combination with a variety of message digest and private key algorithms.

The `Cipher` class manipulates private key algorithms using a key produced by the `KeyGenerator` class. The following methods are used in the Private key cryptography code example on page 13 :

- `KeyGenerator.getInstance("DES")`, `.init(56)`, and `.generateKey()`: Generates the key.

- `Cipher.getInstance("DES/ECB/PKCS5Padding")`: Creates the `Cipher` object (specifying the algorithm, mode, and padding).

- `.init(Cipher.ENCRYPT_MODE, key)`: Initializes the `Cipher` object.

- `.doFinal(plainText)`: Calculates the ciphertext with a plaintext string.

- `.init(Cipher.DECRYPT_MODE, key)`: Decrypts the ciphertext.

- `.doFinal(cipherText)`: Computes the ciphertext.

Private key cryptography code example

```
import java.security.*;
import javax.crypto.*;
//
// encrypt and decrypt using the DES private key algorithm
public class PrivateExample {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {
      System.err.println("Usage: java PrivateExample text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // get a DES private key
    System.out.println( "\nStart generating DES key" );
    KeyGenerator keyGen = KeyGenerator.getInstance("DES");
    keyGen.init(56);
    Key key = keyGen.generateKey();
```

```
    System.out.println( "Finish generating DES key" );
    //
    // get a DES cipher object and print the provider
    Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    System.out.println( "\n" + cipher.getProvider().getInfo() );
    //
    // encrypt using the key and the plaintext
    System.out.println( "\nStart encryption" );
    cipher.init(Cipher.ENCRYPT_MODE, key);
    byte[] cipherText = cipher.doFinal(plainText);
    System.out.println( "Finish encryption: " );
    System.out.println( new String(cipherText, "UTF8") );

    //
    // decrypt the ciphertext using the same key
    System.out.println( "\nStart decryption" );
    cipher.init(Cipher.DECRYPT_MODE, key);
    byte[] newPlainText = cipher.doFinal(cipherText);
    System.out.println( "Finish decryption: " );

    System.out.println( new String(newPlainText, "UTF8") );
  }
}
```

Private key cryptography sample execution

```
D:\IBM>java PrivateExample "This is a test!"

Start generating DES key
Finish generating DES key

SunJCE Provider (implements DES, Triple DES, Blowfish, PBE, Diffie-Hellman,
HMAC-MD5, HMAC-SHA1)

Start encryption
Finish encryption:
Kdkj4338*3n1#kxkgtixo4

Start decryption
Finish decryption:
This is a test!
```

Section 5. Secret messages with public keys

Overview

In this section, we'll look at public key cryptography, a feature that solves the problem of encrypting messages between parties without prior arrangement on the keys. We'll take a short walk through the algorithms, classes, and methods that support the public key function, and offer a code sample and execution to illustrate the concept.

What is public key cryptography?

Private key cryptography suffers from one major drawback: how does the private key get to Alice and Bob in the first place? If Alice generates it, she has to send it to Bob, but it is sensitive information so it should be encrypted. However, keys have not been exchanged to perform the encryption.

Public key cryptography, invented in the 1970s, solves the problem of encrypting messages between two parties without prior agreement on the key.

In public key cryptography, Alice and Bob not only have different keys, they each have two keys. One key is private and must not be shared with anyone. The other key is public and can be shared with anyone.

When Alice wants to send a secure message to Bob, she encrypts the message using Bob's public key and sends the result to Bob. Bob uses his private key to decrypt the message. When Bob wants to send a secure message to Alice, he encrypts the message using Alice's public key and sends the result to Alice. Alice uses her private key to decrypt the message. Eve can eavesdrop on both public keys and the encrypted messages, but she cannot decrypt the messages because she does not have either of the private keys.

The public and private keys are generated as a pair and need longer lengths than the equivalent-strength private key encryption keys. Typical key lengths for the RSA algorithm are 1,024 bits. It is not feasible to derive one member of the key pair from the other.

Public key encryption is slow (100 to 1,000 times slower than private key encryption), so a hybrid technique is usually used in practice. Public key encryption is used to distribute a private key, known as a *session key*, to another party, and then private key encryption using that private session key is used for the bulk of the message encryption.

Algorithms, classes, and methods

The following two algorithms are used in public key encryption:

- **RSA.** This algorithm is the most popular public key cipher, but it's not supported in JDK 1.4. You must use a third-party library like BouncyCastle to get this support.

- **Diffie-Hellman.** This algorithm is technically known as a *key-agreement algorithm* . It cannot be used for encryption, but can be used to allow two parties to derive a secret key by sharing information over a public channel. This key can then be used for private key

encryption.

The `Cipher` class manipulates public key algorithms using keys produced by the `KeyPairGenerator` class. The following methods are used in the Public key cryptography code example on page 16 example:

- `KeyPairGenerator.getInstance("RSA")`, `.initialize(1024)`, and `.generateKeyPair()`: Generates the key pair.
- `Cipher.getInstance("RSA/ECB/PKCS1Padding")` Creates a `Cipher` object (specifying the algorithm, mode, and padding).
- `.init(Cipher.ENCRYPT_MODE, key.getPublic())`: Initializes the `Cipher` object.
- `.doFinal(plainText)`: Calculates the ciphertext with a plaintext string.
- `.init(Cipher.DECRYPT_MODE, key.getPrivate())` and `.doFinal(cipherText)`: Decrypts the ciphertext.

Public key cryptography code example

```
import java.security.*;
import javax.crypto.*;
//
// Public Key cryptography using the RSA algorithm.
public class PublicExample {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {
      System.err.println("Usage: java PublicExample text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // generate an RSA key
    System.out.println( "\nStart generating RSA key" );
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair key = keyGen.generateKeyPair();
    System.out.println( "Finish generating RSA key" );
    //
    // get an RSA cipher object and print the provider
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    System.out.println( "\n" + cipher.getProvider().getInfo() );
    //
    // encrypt the plaintext using the public key
    System.out.println( "\nStart encryption" );
    cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());
    byte[] cipherText = cipher.doFinal(plainText);
    System.out.println( "Finish encryption: " );
```

```
    System.out.println( new String(cipherText, "UTF8") );
    //
    // decrypt the ciphertext using the private key
    System.out.println( "\nStart decryption" );
    cipher.init(Cipher.DECRYPT_MODE, key.getPrivate());
    byte[] newPlainText = cipher.doFinal(cipherText);
    System.out.println( "Finish decryption: " );
    System.out.println( new String(newPlainText, "UTF8") );
  }
}
```

Public key cryptography sample execution

```
D:\IBM>java PublicExample "This is a test!"

Start generating RSA key
Finish generating RSA key

BouncyCastle Security Provider v1.12

Start encryption
Finish encryption:
Ajsd843*342l,AD;LKJL;1!*AD(XLKASD498asdjlkkKSFJHDuhpja;d(kawe#kjalfcas,
.asd+,1LKSDJf;khaouiwheyahdsl87458q9734hjfc*nuywe

Start decryption
Finish decryption:
This is a test!
```

Section 6. Signatures without paper

Overview

In this section, we'll examine digital signatures, the first level of determining the identification of parties that exchange messages. We'll illustrate both difficult and easy ways to identify the message source through code samples. We'll also list the digital signature algorithms that JDK 1.4 supports, and look at the classes and methods involved.

What are digital signatures?

Did you notice the flaw in the public key message exchange described in What is public key cryptography? on page 15 ? How can Bob prove that the message *really* came from Alice? Eve could have substituted her public key for Alice's, then Bob would be exchanging messages with Eve thinking she was Alice. This is known as a *Man-in-the-Middle attack* .

We can solve this problem by using a *digital signature* -- a bit pattern that proves that a message came from a given party.

One way of implementing a digital signature is using the reverse of the public key process described in What is public key cryptography? on page 15 . Instead of encrypting with a public key and decrypting with a private key, the private key is used by a sender to sign a message and the recipient uses the sender's public key to decrypt the message. Because only the sender knows the private key, the recipient can be sure that the message really came from the sender.

In actuality, the message digest (What is a message digest? on page 7), not the entire message, is the bit stream that is signed by the private key. So, if Alice wants to send Bob a signed message, she generates the message digest of the message and signs it with her private key. She sends the message (in the clear) and the signed message digest to Bob. Bob decrypts the signed message digest with Alice's public key and computes the message digest from the cleartext message and checks that the two digests match. If they do, Bob can be sure the message came from Alice.

Note that digital signatures do not provide encryption of the message, so encryption techniques must be used in conjunction with signatures if you also need confidentiality.

You can use the RSA algorithm for both digital signatures and encryption. A U.S. standard called DSA (Digital Signature Algorithm) can be used for digital signatures, but not for encryption.

Algorithms

JDK 1.4 supports the following digital signature algorithms:

- **MD2/RSA**
- **MD5/RSA**
- **SHA1/DSA**

- **SHA1/RSA**

We'll examine two examples in this section. The first, the hard way (see Digital signature code example: The hard way on page 19), uses the primitives already discussed for message digests and public key cryptography to implement digital signatures. The second, the easy way (see Digital signature code example: The easy way on page 20), uses the Java language's direct support for signatures.

Digital signature code example: The hard way

```
import java.security.*;
import javax.crypto.*;
//
// This program demonstrates the digital signature technique at the
// primative level by generating a message digest of the plaintext

// and signing it with an RSA private key, to create the signature.
// To verify the signature, the message digest is again generated from
// the plaintext and compared with the decryption of the signature
// using the public key.  If they match, the signature is verified.
public class DigitalSignature1Example {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {
      System.err.println("Usage: java DigitalSignature1Example text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // get an MD5 message digest object and compute the plaintext digest
    MessageDigest messageDigest = MessageDigest.getInstance("MD5");
    System.out.println( "\n" + messageDigest.getProvider().getInfo() );
    messageDigest.update( plainText );
    byte[] md = messageDigest.digest();
    System.out.println( "\nDigest: " );
    System.out.println( new String( md, "UTF8") );
    //
    // generate an RSA keypair
    System.out.println( "\nStart generating RSA key" );
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair key = keyGen.generateKeyPair();
    System.out.println( "Finish generating RSA key" );
    //
    // get an RSA cipher and list the provider
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    System.out.println( "\n" + cipher.getProvider().getInfo() );
    //
    // encrypt the message digest with the RSA private key
    // to create the signature
    System.out.println( "\nStart encryption" );
    cipher.init(Cipher.ENCRYPT_MODE, key.getPrivate());
    byte[] cipherText = cipher.doFinal(md);
    System.out.println( "Finish encryption: " );
    System.out.println( new String(cipherText, "UTF8") );
    //
    // to verify, start by decrypting the signature with the
```

```
    // RSA private key
    System.out.println( "\nStart decryption" );
    cipher.init(Cipher.DECRYPT_MODE, key.getPublic());

    byte[] newMD = cipher.doFinal(cipherText);
    System.out.println( "Finish decryption: " );
    System.out.println( new String(newMD, "UTF8") );
    //
    // then, recreate the message digest from the plaintext
    // to simulate what a recipient must do
    System.out.println( "\nStart signature verification" );
    messageDigest.reset();
    messageDigest.update(plainText);
    byte[] oldMD = messageDigest.digest();
    //
    // verify that the two message digests match
    int len = newMD.length;
    if (len > oldMD.length) {
      System.out.println( "Signature failed, length error");
      System.exit(1);
    }
    for (int i = 0; i < len; ++i)
      if (oldMD[i] != newMD[i]) {
        System.out.println( "Signature failed, element error" );
        System.exit(1);
      }
    System.out.println( "Signature verified" );
  }
}
```

Sample execution

```
D:\IBM>java DigitalSignature1Example "This is a test!"

SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests
; SecureRandom; X.509 certificates; JKS keystore; PKIX CertPathValidator
; PKIX CertPathBuilder; LDAP, Collection CertStores)

Digest:
D647dbdek12*e,ad.?e

Start generating RSA key
Finish generating RSA key

BouncyCastle Security Provider v1.12

Start encryption
Finish encryption:
Akjsdfp-9q8237nrcas-9de8fn239-4rb[*[OPOsjkdfJDL:JF;lkjs;ldj

Start decryption
Finish decryption:
iNdf6D213$dcd(ndz!0)

Start signature verification
Signature verified
```

Digital signature code example: The easy way

The `Signature` class manipulates digital signatures using a key produced by the `KeyPairGenerator` class. The following methods are used in the example below:

- `KeyPairGenerator.getInstance("RSA")`, `.initialize(1024)`, and `.generateKeyPair()`: Generates the keys.
- `Cipher.getInstance("MD5WithRSA")`: Creates the `Signature` object.
- `.initSign(key.getPrivate())`: Initializes the `Signature` object.
- `.update(plainText)` and `.sign()`: Calculates the signature with a plaintext string.
- `.initVerify(key.getPublic())` and `.verify(signature)`: Verifies the signature.

```
import java.security.*;
import javax.crypto.*;
//
// This example uses the digital signature features to generate and
// verify a signature much more easily than the previous example
public class DigitalSignature2Example {

  public static void main (String[] args) throws Exception {
    //
    // check args and get plaintext
    if (args.length !=1) {
      System.err.println("Usage: java DigitalSignature1Example text");
      System.exit(1);
    }
    byte[] plainText = args[0].getBytes("UTF8");
    //
    // generate an RSA keypair
    System.out.println( "\nStart generating RSA key" );
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);

    KeyPair key = keyGen.generateKeyPair();
    System.out.println( "Finish generating RSA key" );
    //
    // get a signature object using the MD5 and RSA combo
    // and sign the plaintext with the private key,
    // listing the provider along the way

    Signature sig = Signature.getInstance("MD5WithRSA");
    sig.initSign(key.getPrivate());
    sig.update(plainText);
    byte[] signature = sig.sign();
    System.out.println( sig.getProvider().getInfo() );
    System.out.println( "\nSignature:" );
    System.out.println( new String(signature, "UTF8") );
    //
    // verify the signature with the public key
    System.out.println( "\nStart signature verification" );
    sig.initVerify(key.getPublic());
    sig.update(plainText);
    try {
      if (sig.verify(signature)) {
        System.out.println( "Signature verified" );
```

```
      } else System.out.println( "Signature failed" );
    } catch (SignatureException se) {
      System.out.println( "Signature failed" );
    }
  }
}
```

Sample execution

```
Start generating RSA key
Finish generating RSA key
Sun JSSE provider(implements RSA Signatures, PKCS12, SunX509 key/trust
factories, SSLv3, TLSv1)

Signature:
Ldkjahasdlkjfq[?owc42093nhasdk1a;sn;a#a;lksjd;fl@#kjas;ldjf78qwe09r7

Start signature verification
Signature verified
```

Section 7. Proving you are who you are

Overview

In this section, we'll discuss digital certificates, the second level to determining the identity of a message originator. We'll look at certificate authorities and the role they play. We'll examine key and certificate repositories and management tools (keytool and keystore) and discuss the CertPath API, a set of functions designed for building and validating certification paths.

What are digital certificates?

As you likely noticed, there is a problem with the digital signature scheme described in What are digital signatures? on page 18 . It proves that a message was sent by a given party, but how do we know for sure that the sender *really* is who she says she is. What if someone claims to be Alice and signs a message, but is actually Amanda? We can improve our security by using *digital certificates* which package an identity along with a public key and is digitally signed by a third party called a *certificate authority* or CA.

A certificate authority is an organization that verifies the identity, in the real-world physical sense, of a party and signs that party's public key and identity with the CA private key. A message recipient can obtain the sender's digital certificate and verify (or decrypt) it with the CA's public key. This proves that the certificate is valid and allows the recipient to extract the sender's public key to verify his signature or send him an encrypted message. Browsers and the JDK itself come with built-in certificates and their public keys from several CAs.

JDK 1.4 supports the X.509 Digital Certificate Standard.

Understanding keytool and keystore

The Java platform uses a *keystore* as a repository for keys and certificates. Physically, the keystore is a file (there is an option to make it an encrypted one) with a default name of .keystore. Keys and certificates can have names, called *aliases*, and each alias can be protected by a unique password. The keystore itself is also protected by a password; you can choose to have each alias password match the master keystore password.

The Java platform uses the *keytool* to manipulate the keystore. This tool offers many options; the following example (keytool example on page 24) shows the basics of generating a public key pair and corresponding certificate, and viewing the result by querying the keystore.

The keytool can be used to export a key into a file, in X.509 format, that can be signed by a certificate authority and then re-imported into the keystore.

There is also a special keystore that is used to hold the certificate authority (or any other trusted) certificates, which in turn contains the public keys for verifying the validity of other certificates. This keystore is called the *truststore*. The Java language comes with a default truststore in a file called *cacerts*. If you search for this filename, you will find at least two of these files. You can display the contents with the following command:

```
keytool -list -keystore cacerts
Use a password of "changeit"
```

keytool example

In this example, using the default keystore of *.keystore*, we generate a self-signed certificate using the RSA algorithm with an alias of *JoeUserKey* and then view the created certificate. We will use this certificate in The concept of code signing on page 26 to sign a JAR file.

```
D:\IBM>keytool -genkey -v -alias JoeUserKey -keyalg RSA
Enter keystore password: password
What is your first and last name?
  [Unknown]:  Joe User
What is the name of your organizational unit?
  [Unknown]:  Security
What is the name of your organization?
  [Unknown]:  Company, Inc.
What is the name of your City or Locality?
  [Unknown]:  User City
What is the name of your State or Province?
  [Unknown]:  MN
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Joe User, OU=Security, O="Company, Inc.", L=User City, ST=MN, C=US
correct?
  [no]:  y

Generating 1,024 bit RSA key pair and self-signed certificate (MD5WithRSA)
        for: CN=Joe User, OU=Security, O="Company, Inc.", L=User City,
        ST=MN, C=US
Enter key password for <JoeUserKey>
        (RETURN if same as keystore password):
[Saving .keystore]


D:\IBM>keytool -list -v -alias JoeUserKey

Enter keystore password:  password
Alias name: JoeUserKey
Creation date: Apr 15, 2002
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Joe User, OU=Security, O="Company, Inc.", L=User City, ST=MN,
C=US
Issuer: CN=Joe User, OU=Security, O="Company, Inc.", L=User City, ST=MN,
C=US
Serial number: 3cbae448
Valid from: Mon Apr 15 09:31:36 CDT 2002 until: Sun Jul 14 09:31:36
CDT 2002
Certificate fingerprints:

       MD5:  35:F7:F7:A8:AC:54:82:CE:68:BF:6D:42:E8:22:21:39
       SHA1: 34:09:D4:89:F7:4A:0B:8C:88:EF:B3:8A:59:F3:B9:65:AE:CE:7E:C9
```

CertPath API

The Certification Path API is new for JDK 1.4. It is a set of functions for building and validating certification paths or chains. This is done implicitly in protocols like SSL/TLS (see What is Secure Sockets Layer/Transport Layer Security? on page 28) and JAR file signature verification, but can now be done explicitly in applications with this support.

As mentioned in What are digital certificates? on page 23 , a CA can sign a certificate with its private key, and if the recipient holds the CA certificate that has the public key needed for signature verification, it can verify the validity of the signed certificate.

In this case, the chain of certificates is of length two -- the anchor of trust (the CA certificate) and the signed certificate. A self-signed certificate is of length one -- the anchor of trust is the signed certificate itself.

Chains can be of arbitrary length, so in a chain of three, a CA anchor of trust certificate can sign an intermediate certificate; the owner of this certificate can use its private key to sign another certificate. The CertPath API can be used to walk the chain of certificates to verify validity, as well as to construct these chains of trust.

Certificates have expiration dates, but can be compromised before they expire, so *Certificate Revocation Lists* (CRL) must be checked to really ensure the integrity of a signed certificate. These lists are available on the CA Web sites, and can also be programmatically manipulated with the CertPath API.

The specific API and code examples are beyond the scope of this tutorial, but Sun has several code examples available in addition to the API documentation.

Section 8. Trusting the code

Overview

In this section, we'll review the concept of code signing, focusing on the tool that manages the certification of a JAR file, Jarsigner.

The concept of code signing

JAR files are the Java platform equivalent of ZIP files, allowing multiple Java class files to be packaged into one file with a .jar extension. This JAR file can then be digitally signed, proving the origin and the integrity of the class file code inside. A recipient of the JAR file can decide whether or not to trust the code based on the signature of the sender and can be confident that the contents have not been tampered with before receipt. The JDK comes with a *jarsigner* tool that provides this function.

In deployment, access to machine resources can be based on the signer's identity by putting access control statements in the policy file.

Jarsigner tool

The jarsigner tool takes a JAR file and a private key and corresponding certificate as input, then generates a signed version of the JAR file as output. It calculates the message digests for each class in the JAR file and then signs these digests to ensure the integrity of the file and to identify the file owner.

In an applet environment, an HTML page references the class file contained in a signed JAR file. When this JAR file is received by the browser, the signature is checked against any installed certificates or against a certificate authority public signature to verify validity. If no existing certificates are found, the user is prompted with a screen giving the certificate details and asking if the user wants to trust the code.

Code signing example

In this example, we first create a JAR file from a .class file and then sign it by specifying the alias for the certificate in the keystore that is used for the signing. We then run a verification check on the signed JAR file.

```
D:\IBM>jar cvf HelloWorld.jar HelloWorld.class
added manifest
adding: HelloWorld.class(in = 372) (out= 269)(deflated 27%)

D:\IBM>jarsigner HelloWorld.jar JoeUserKey
Enter Passphrase for keystore: password

D:\IBM>jarsigner -verify -verbose -certs HelloWorld.jar

         137 Mon Apr 15 12:38:38 CDT 2002 META-INF/MANIFEST.MF
         190 Mon Apr 15 12:38:38 CDT 2002 META-INF/JOEUSERK.SF
```

```
         938 Mon Apr 15 12:38:38 CDT 2002 META-INF/JOEUSERK.RSA
           0 Mon Apr 15 12:38:00 CDT 2002 META-INF/
smk      372 Mon Apr 15 12:33:02 CDT 2002 HelloWorld.class

      X.509, CN=Joe User, OU=Security, O="Company, Inc.", L=User City,
      ST=MN, C=US (joeuserkey)


  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.
```

Code signing example execution

Here is the HTML for this program:

```
<HTML>
<HEAD>
<TITLE> Hello World Program </TITLE>
</HEAD>
<BODY>
<APPLET CODE="HelloWorld.class" ARCHIVE="HelloWorld.jar"
  WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

When this example is executed with a browser that uses the Java plug-in as the Java virtual machine, a dialog box pops up asking if the user wants to install and run the signed applet distributed by "Joe User", and says that the publisher authenticity is verified by "Company, Inc.", but that the security was issued by a company that is not trusted. The security certificate has not expired and is still valid. It cautions that "Joe User" asserts that this content is safe and should only be installed or viewed if you trust "Joe User" to make that assertion. The user is given the following options:

- Grant this session
- Deny
- Grant always
- View certificate

Section 9. SSL/TLS: Securing C/S communication

Overview

In this section, we'll examine the building blocks of the Secure Sockets Layer (and its replacement, Transport Layer Security), the protocol used to authenticate the server to the client. We'll offer a few code examples as illustrations.

What is Secure Sockets Layer/Transport Layer Security?

Secure Sockets Layer (SSL) and its replacement, Transport Layer Security (TLS), is a protocol for establishing a secure communications channel between a client and a server. It is also used to authenticate the server to the client and, less commonly, used to authenticate the client to the server. It is usually seen in a browser application, where the lock at the bottom of the browser window indicates SSL/TLS is in effect.

TLS 1.0 is the same as SSL 3.1.

SSL/TLS uses a hybrid of three of the cryptographic building blocks already discussed in this tutorial, but all of this is transparent to the user. Here is a simplified version of the protocol:

- When a request is made to a site using SSL/TLS (usually with an *https://* URL), a certificate is sent from the server to the client. The client verifies the identify of the server from this certificate using the installed public CA certificates, then checks that the IP name (machine name) matches the machine that the client is connected to.

- The client generates some random info that can be used to generate a private key for the conversation, known as a session key, and encrypts it with the server's public key and sends it to the server. The server decrypts the message with its private key and uses the random info to derive the same private session key as the client. The RSA public key algorithm is usually used for this phase.

- The client and server then communicate using the private session key and a private key algorithm, usually RC4. A **message-authentication code**, using yet another key, is used to ensure the integrity of the message.

SSL/TLS code sample

In this example, we write an HTTPS daemon process using an SSL server socket that returns an HTML stream when a browser connects to it. This example also shows how to generate a machine certificate in a special keystore to support the SSL deployment.

In Java programming, the only thing that needs to be done is to use an SSL Server Socket Factory instead of a Socket Factory, using lines like the following:

```
SSLServerSocketFacctory sslf =
  (SSLServerSocketFactor)SSLServerSocketFactory.getDefault();
ServerSocket serverSocket = sslf.createServerSocket(PORT);
```

The complete code example is listed below:

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;
//
// Example of an HTTPS server to illustrate SSL certificate and socket
public class HTTPSServerExample {

  public static void main(String[] args) throws IOException {

    //
    // create an SSL socket using the factory and pick port 8080
    SSLServerSocketFactory sslsf =
      (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    ServerSocket ss = sslsf.createServerSocket(8080);
    //
    // loop forever
    while (true) {
      try {
        //
        // block waiting for client connection
        Socket s = ss.accept();
        System.out.println( "Client connection made" );
        // get client request
        BufferedReader in = new BufferedReader(
          new InputStreamReader(s.getInputStream()));
        System.out.println(in.readLine());
        //
        // make an HTML response
        PrintWriter out = new PrintWriter( s.getOutputStream() );
        out.println("<HTML><HEAD><TITLE>HTTPS Server Example</TITLE>" +
                    "</HEAD><BODY><H1>Hello World!</H1></BODY></HTML>\n");
        //
        // Close the stream and socket
        out.close();
        s.close();
      } catch (Exception e) {
          e.printStackTrace();
      }
    }
  }
}
```

HTTPS server sample execution

In this example, we create an HTTPS server daemon that waits for a client browser connection and returns "Hello, World!". The browser connects to this daemon via *https://localhost:8080*.

We first create a machine certificate. The name must match the machine name of the computer where the daemon runs; in this case, *localhost*. In addition, we cannot use the same *.keystore* we have used in the past. We must create a separate keystore just for the machine certificate. In this case, it has the name *sslKeyStore*.

```
D:\IBM>keytool -genkey -v -keyalg RSA -alias MachineCert
```

```
  -keystore sslKeyStore
Enter keystore password:  password
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  Security
What is the name of your organization?
  [Unknown]:  Company, Inc.
What is the name of your City or Locality?
  [Unknown]:  Machine Cert City
What is the name of your State or Province?
  [Unknown]:  MN
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=localhost, OU=Security, O="Company, Inc.", L=Machine Cert City,
ST=MN, C=US correct?
  [no]:  y

Generating 1,024 bit RSA key pair and self-signed certificate (MD5WithRSA)
   for: CN=localhost, OU=Security, O="Company, Inc.", L=Machine Cert City,
ST=MN, C=US
Enter key password for <MachineCert>
        (RETURN if same as keystore password):
[Saving sslKeyStore]
```

Then, we start the server daemon process specifying the special keystore and its password:

```
D:\IBM>java -Djavax.net.ssl.keyStore=sslKeyStore
 -Djavax.net.ssl.keyStorePassword=password HTTPSServerExample
```

After waiting a few seconds, fire up a browser and point it to *https://localhost:8080* and you should be prompted on whether or not to trust the certificate. Selecting "yes" should display "Hello World!", and clicking on the lock in Internet Explorer will give the certificate details.

Section 10. Wrapup and resources

Summary

This tutorial introduced the major cryptographic building blocks that can be used to provide a vast array of application security solutions. You've become familiar with such Java security topics as:

- **Built-in features that facilitate secure programming** (no pointers, a bytecode verifier, fine-grained control over resource access for both applets and applications, a large number of library functions for all the major cryptographic building blocks, and SSL).
- **Secure programming techniques** (proper storage and deletion of passwords and intelligent serialization).
- **Features newly integrated in JDK 1.4** (JCE, JSSE, JAAS, JGSS, and CertPath API).
- **Enriching, third-party security offerings**.

And the following concepts:

- Message digests
- Message authentication codes
- Private key cryptography
- Public key cryptography
- Digital signatures
- Digital certificates
- Certification authorities and paths
- Code signing
- SSL/TLS

You should be well poised to explore Java security in more detail (see the Resources on page 31 section) and to take the next tutorial, *Java security, Part 2: Authentication and authorization*.

Resources

Downloads

- Download the complete source code and classes used in this tutorial, javasecurity1-source.jar.
- See *BouncyCastle* (http://www.bouncycastle.org) for the third-party provider library used in this tutorial.

Articles, tutorials, and other online resources

- Sun's *Java Security Web site* (http://java.sun.com/security) is the definitive source for Java security.

- Read Brad Rubin's second tutorial in this series, "*Java security, Part 2: Authentication and authorization*" (*developerWorks*, July 2002, http://www-106.ibm.com/developerworks/education/r-jsec2.html).

- Michael Yuan demonstrates how to digitally sign and verify XML documents on wireless devices using the Bouncy Castle Crypto APIs in his article "*Securing your J2ME/MIDP apps*" (*developerWorks*, June 2002, http://www-106.ibm.com/developerworks/library/j-midpds.html).

- Greg Travis offers a practical look at JSSE in his tutorial "*Using JSSE for secure socket communication*" (*developerWorks*, April 2002, http://www-106.ibm.com/developerworks/education/r-jsse.html).

Books

- For an overall discussion of Web security and Java technology, see *Web Security, Privacy, and Commerce, 2nd Edition* (http://www.oreilly.com/catalog/websec2/), by Simson Garfinkel and Gene Spafford, O'Reilly, 2002.

- If you want to focus more on Java security, see *Professional Java Security* (http://www.amazon.com/exec/obidos/ASIN/1861004257/104-8739833-1347930), by Jess Garms and Daniel Somerfield, Wrox Press, 2001.

- Another great resource for learning about Java security is *Java Security* (http://www.amazon.com/exec/obidos/ASIN/0596001576), by Scott Oaks, O'Reilly & Associates, 2001.

- Find out what everyone needs to know about security in order to survive and be competitive in *Secrets and Lies: Digital Security in a Networked World* (http://www.counterpane.com/sandl.html), by Bruce Schneier, 2000.

- Boasting new algorithms, more information on the Clipper Chip and key escrow, dozens of new protocols, more information on PGP, detailed information on key management and modes of operation, and new source code, this book should be a security winner: *Applied Cryptography, Second Edition* (http://www.counterpane.com/applied.html), by Bruce Schneier, 1995.

Additional resources

- The *IBM Java Security Research* page (http://www.research.ibm.com/javasec/) details various security projects in the works.

- Visit the *Tivoli Developer domain* (http://www-106.ibm.com/developerworks/tivoli/) for help in building and maintaining the security of your e-business.

- The *developerWorks Security special topic* (http://www-106.ibm.com/developerworks/security/) offers developers hands-on technical information covering the general topic of security.

- Participate in the *developerWorks Java security forum* hosted by Paul Abbott.

- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks Java technology zone* (http://www-106.ibm.com/developerworks/java/).

- See the *developerWorks tutorials page* (http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle) for a complete listing of free Java technology-related tutorials from *developerWorks*.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.