**Encrypting and Decrypting Data**

Although SSL may handle most of your cryptography needs, you may discover that you need to manually encrypt and decrypt data from time to time. Basically, to encrypt or decrypt data, you need a key and a cipher. You can use the SecretKeyFactory to generate new keys. You can either generate a random key or use an existing array of bytes for the key. In the latter case, you first create a key "spec" from the original bytes and then pass this spec to the key factory.

The following code snippet creates a key spec from an array of bytes and then creates a key. The key is a DESede key, which is commonly known as Triple-DES—a more secure version of the original Data Encryption Standard (DES) algorithm. Some of the other available algorithms are DES (the original DES algorithm), Blowfish (a block cipher invented by Bruce Schneier), and PBEWithMD5AndDES (password-based encryption with MD5 digest mode and DES encryption).

// Create an array to hold the key

byte[] encryptKey = "This is a test DESede key".getBytes();

// Create a DESede key spec from the key

DESedeKeySpec spec = new DESedeKeySpec(encryptKey);

// Get the secret key factor for generating DESede keys

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(

"DESede");

// Generate a DESede SecretKey object

SecretKey theKey = keyFactory.generateSecret(spec);

Once you have a key, you can create a cipher using the Cipher.getInstance method:

Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");

In this case, you create a DESede (Triple-DES) cipher with some extra parameters. The second parameter (CBC) controls how you handle successive blocks of data. The simplest method of encrypting multiple blocks is called Electronic Code Book (ECB). When you use ECB, you simply encrypt one block at a time. For each block, you just take the original data, and encrypt it without any additional information. ECB is rather unsafe, however, because someone may be able to learn something about the content of the data by analyzing identical blocks. You can make your data safer by using Chained Block Cipher (CBC) mode. When you encrypt using CBC mode, you combine the data you want to encrypt with a special array of data called an initialization vector (IV) and then encrypt the combined data. For the second block, however, you combine the encrypted first block with the data you want to encrypt (usually with a simple XOR) and then encrypt the combined block. This eliminates repetitive patterns in the encrypted data.

Two other modes, Output Feedback (OFB) and Cipher Feedback (CFB), create a stream cipher byte generator out of a block cipher; that is, they make a block cipher behave like a stream cipher.

The other parameter—PKCS5Padding, in this case—indicates how to handle incomplete blocks. Remember that a block cipher operates on blocks of a fixed size. By using one of the common padding algorithms, you include the block size in the encrypted data, ensuring that when you decrypt, you get the correct number of bytes back. You can also specify NoPadding if you don't want any padding.

When you use CBC mode, you must create an initialization vector before you encrypt or decrypt data. The contents of the IV don't matter much, but you *must* use the same IV to encrypt and decrypt.

IvParameterSpec IvParameters = new IvParameterSpec(

new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 });

The following program encrypts a string of data and writes it out to a file:

import java.io.\*;

import javax.crypto.\*;

import javax.crypto.spec.\*;

public class EncryptData

{

public static void main(String[] args)

{

try

{

// Create an array to hold the key

byte[] encryptKey = "This is a test DESede key".getBytes();

// Create a DESede key spec from the key

DESedeKeySpec spec = new DESedeKeySpec(encryptKey);

// Get the secret key factor for generating DESede keys

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(

"DESede");

// Generate a DESede SecretKey object

SecretKey theKey = keyFactory.generateSecret(spec);

// Create a DESede Cipher

Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");

// Create an initialization vector (necessary for CBC mode)

IvParameterSpec IvParameters = new IvParameterSpec(

new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 });

// Initialize the cipher and put it into encrypt mode

cipher.init(Cipher.ENCRYPT\_MODE, theKey, IvParameters);

byte[] plaintext =

"This is a sentence that has been encrypted".getBytes();

// Encrypt the data

byte[] encrypted = cipher.doFinal(plaintext);

// Write the data out to a file

FileOutputStream out = new FileOutputStream("encrypted.dat");

out.write(encrypted);

out.close();

}

catch (Exception exc)

{

exc.printStackTrace();

}

}

}

The process of encryption and decryption is almost identical from a coding perspective. To decrypt data, you use Cipher.DECRYPT\_MODE in the init method for the Cipher object. Beyond that, everything else is the same. The following program decrypts the data that was encrypted by the EncryptData program:

import java.io.\*;

import javax.crypto.\*;

import javax.crypto.spec.\*;

import java.security.\*;

public class DecryptData

{

public static void main(String[] args)

{

try

{

// Create an array to hold the key

byte[] encryptKey = "This is a test DESede key".getBytes();

// Create a DESede key spec from the key

DESedeKeySpec spec = new DESedeKeySpec(encryptKey);

// Get the secret key factor for generating DESede keys

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(

"DESede");

// Generate a DESede SecretKey object

SecretKey theKey = keyFactory.generateSecret(spec);

// Create a DESede Cipher

Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");

// Create the initialization vector required for CBC mode

IvParameterSpec ivParameters = new IvParameterSpec(

new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 } );

// Initialize the cipher and put it in decrypt mode

cipher.init(Cipher.DECRYPT\_MODE, theKey, ivParameters);

File encryptedFile = new File("encrypted.dat");

// Create a byte block to hold the entire encrypted file

byte[] encryptedText = new byte[(int) encryptedFile.length()];

FileInputStream fileIn = new FileInputStream(encryptedFile);

// Read the entire encrypted file

fileIn.read(encryptedText);

fileIn.close();

// Decrypt the data

byte[] plaintext = cipher.doFinal(encryptedText);

String plaintextStr = new String(plaintext);

System.out.println("The plaintext is:");

System.out.println(plaintextStr);

}

catch (Exception exc)

{

exc.printStackTrace();

}

}

}

The cryptography libraries in Java 2 SDK 1.4 are pretty robust, and there are many options available. You can digitally sign or encrypt whole objects. You can also encrypt a stream of data without resorting to SSL. Take a look at the javax.crypto package to get an idea of some of the other available features.