Java Cryptographic APIs
Message Digests and Symmetric Ciphers

Simon Foley

October 13, 2008

Message Digests/One way Hash Functions

Output the MD5 hash of a string provided at command line


```
import java.io.*;
import java.security.MessageDigest;
import sun.misc.*;
public class MDExample1 {
  public static void main(String[] args) throws Exception {
    MessageDigest md= MessageDigest.getInstance("MD5");
    if (args.length!=1){
      System.out.println("Usage MDExample1 <text>");
      return;
    }
    byte[] data=args[0].getBytes("UTF8"); // string to bytes
    byte[] rslt = md.digest(data); //generate hash value

    BASE64Encoder encoder= new BASE64Encoder();
    String base64= encoder.encode(rslt);
    System.out.println(base64);              //output in base64 format
  }
}
```

Message Digests in Java: Example Highlights


```
MessageDigest md= MessageDigest.getInstance("MD5");
```

A special *factory* method (static) that returns a MessageDigest object, in this case, MessageDigest uses the algorithm MD5. (others: SHA,...).

```
byte[] data=args[0].getBytes("UTF8");
```

Data to be 'digested' must be a byte array, so we convert the command-line input (string) to a byte array according to the UTF8 standard encoding.

```
byte[] rslt = md.digest(data);}
```

Use MessageDigest md to generate MD5 digest *value* of input data.

```
BASE64Encoder encoder= new BASE64Encoder();
String base64= encoder.encode(rslt);
```

The binary data (**byte** [] rslt) must converted to a base64 encoding for printing in a human-readable ASCII form.

Building Digests in Stages

It may not be sensible to digest a large amount data in 'one go'. For example, we should not read the entire contents of a file into memory (byte array) and then digest it. Suppose args[0] gives the name of a file to be digested; md is a MessageDigest:

```
//open the file as a file input stream:\\
  FileInputStream in = new FileInputStream(args[0]);   \\
  byte[] buffer = new byte[1024]; \\
  int length;

  //read the input stream, one 1024 byte block at a time:\\
  while ((length = in.read(buffer)) != -1)\\
~~~~// pass the 1K block to the Message Digest: \\
~~~~md.update(buffer, 0, length); \\

  // Generate the message digest value: \\
  byte[] rslt = md.digest();
```

If you want to use md to digest another file, it must be reset: md.reset();

Building Digests for Streams

We can wrap a DigestInputStream around a FileInputStream making it really easy to calculate a message digest value on any input stream.
Suppose that args[0] gives the name of a file we want to read and do some processing from; md is a MessageDigest:

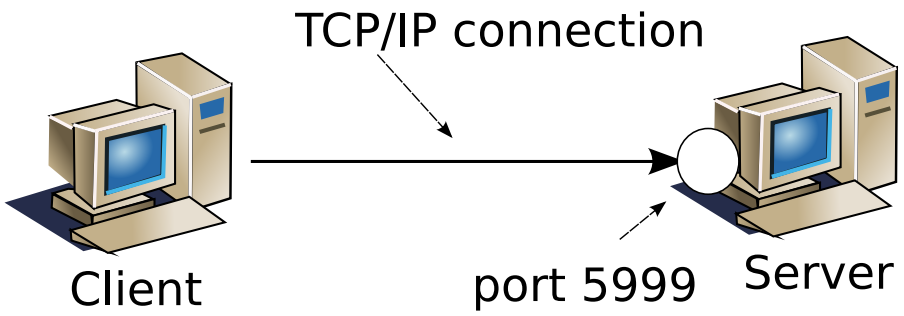
```
//open the file as a file input stream:
FileInputStream fin = new FileInputStream(args[0]);
//wrap it with a digest stream
DigestInputStream din = new DigestInputStream(fin,md);

byte[] buffer = new byte[1024];
//read the file , 1K at a time
while (din.read(buffer)!=-1)
  ; // do the processing

byte[] rslt = md.digest();
```

DigestOutputStream works the same way: all bytes written to the stream are automatically passed to the MessageDigest.

Example: Simple Login for Client Server Application



The server of a client-server application accepts connections on port 5999. Clients connecting to this port must first provide a valid name and password before being allowed to proceed.

The Client-Server follow the following authentication protocol.

$$\begin{array}{lll} \text{Msg 1:} & C \rightarrow S & name, random, timestamp, \\ & & h(name\hat{}random\hat{}timestamp\hat{}password) \\ \text{Msg 1:} & S \rightarrow C & result \end{array}$$

where $password$ is a secret shared between the client and server.

Socket Programming is Easy in Java!

Host server listens on socket 5999 for a single message:

```
ServerSocket s = new ServerSocket(5999);// create socket
Socket c = s.accept(); // wait for client to connect
// server will communicate with client via socket c
DataInputStream in= new DataInputStream(c.getInputStream());
// c provides: c.getInputStream(), c.getOutputStream()
String msg = in.readUTF(); //expecting just one string
System.out.println(msg); //and print it out
```

The Client connects and sends the message:

```
s new Socket("server", 5999); //connect to server
// s provides: s.getInputStream(), s.getOutputStream()
DataOutputStream out =
    new DataOutputStream(s.getOutputStream());
out.writeUTF("Hi there"); // send the message
out.close();// and we're all done
```

Client Side Code Fragment


```
Socket s = new Socket("slaptop",5999); // the server
DataOutputStream out = new DataOutputStream(s.getOutputStream());
String cname = "client";  // hardcoded--horrid
String cpass = "cpasswd"; // ditto
long ctim = (new Date()).getTime();// get current time
double cran = Math.random(); // and a random value
MessageDigest md= MessageDigest.getInstance("MD5");
md.update(cname.getBytes("UTF8"));
md.update(cpass.getBytes("UTF8"));
// stream the long ctim and double cran to a ByteArray
ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream(bout);
dout.writeLong(ctim);
dout.writeDouble(cran);
md.update(bout.toByteArray());
// write the details to the socket's stream
out.writeUTF(cname); out.writeLong(ctim);
out.writeDouble(cran); out.write(md.digest());
```

Server Side Code Fragment


```
ServerSocket s = new ServerSocket(5999);// create the socket
Socket c= s.accept(); // wait for client to connect
DataInputStream in = new DataInputStream(c.getInputStream());
// read the expected values from client socket
String cname = in.readUTF(); // the client's name
String cpass = "cpasswd"; // password is not sent in clear!
long ctim = in.readLong(); // the client's time
double cran = in.readDouble(); // the random value
byte[] hashval = new byte[16]; // kludge: MD5 hash is 16 bytes
in.readFully(hashval); // should really be algorithm independent
// recompute the digest value
MessageDigest md= MessageDigest.getInstance("MD5");
md.update(cname.getBytes("UTF8"));
md.update(cpass.getBytes("UTF8"));
// stream ctim and cran to ByteArray and update md
...
if (MessageDigest.isEqual(hashval,md.digest()))
    ... //check freshness of message, etc.
```

Secret Keys

Secret Keys in Java: A Simple Example


```
import java.io.*;
import javax.crypto.SecretKey;
import javax.crypto.Mac
import javax.crypto.spec.SecretKeySpec;
import sun.misc.*;

public class Mac1 {
  public static void main (String[] args) throws Exception {
    // we're going to be using a keyed MD5 hash function: a MAC
    byte[] keyBytes = {12,34,56,78};// ugg, never generate a key this way!
    SecretKey key = new SecretKeySpec(keyBytes, "HmacMD5");

    Mac mac= Mac.getInstance("HmacMD5");
    mac.init(key);  // initialize MAC with this key
    byte[] data=args[0].getBytes("UTF8"); // convert string to bytes
    byte[]  rslt = mac.doFinal(data);  //generate MAC value

    BASE64Encoder encoder= new BASE64Encoder();
    System.out. println (encoder.encode( rslt ));//output in base64 format
    }}
```

Example Highlights

Java interface javax.crypto.SecretKey encapsulates a secret cryptographic key. Java supports different cryptographic algorithms (and keys).

```
byte[] keyBytes = {12,34,56,78};
SecretKey key = new SecretKeySpec(keyBytes, "HmacMD5");
```

makes a secret key (value defined by keyBytes) for use with a one-way MD5 hash function. Object key encapsulates attributes, including kind of key, value, etc. Here, invoking key.getAlgorithm() returns "HmacMD5".

```
Mac mac= Mac.getInstance("HmacMD5");
mac.init(key);
```

We use factory method getInstance to generate the MAC, which must be initialized by the intended key. This means we can easily replace the MAC algorithm (eg, HmacSHA1) without having to change subsequent code!.

```
byte[] rslt = mac.doFinal(data);
```

This computes the MAC value for the data provided (and then resets MAC).

Secret Key Generation

One must be very careful when generating keys; eg, is the key 12,34,56,78 easily guessed? Poor keys makes our protected data vulnerable to attack. JCE provides classes used to generate appropriate random keys as follows:

1. Obtain a key generator object for the algorithm you want to use.
2. Initialize the key generator.
3. Ask the key generator to generate a key.

For example, to generate a new random HmacMD5 key:

```
KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
kg.init(new SecureRandom());          // initialize key generator
SecretKey key1 = kg.generateKey();    // generate a random key
SecretKey key2 = kg.generateKey();    // and another, ....
```

The important classes for creating/manipulating keys include:
java.security.Key (the basic object class) javax.crypto.SecretKey (for secret keys) javax.crypto.KeyGenerator (for generating keys)
javax.crypto.spec.SecretKeySpec and javax.crypto.spec.SecretKeyFactory (translating between data and keys)

Message Authentication Codes: Example

Once we have created a keyed message digest (MAC) then we can compute the MAC value in one go (as above) or, if it is large, feed it in blocks.

```
// given passphrase in args[0], compute its MD5 hash value
MessageDigest pbmd= MessageDigest.getInstance("MD5");
byte[] keyBytes= pbmd.digest(args[0].getBytes("UTF8"));
// and turn that digest value into an HmacSHA key
SecretKey key = new SecretKeySpec(keyBytes, "HmacSHA1");
Mac mac= Mac.getInstance("HmacSHA1");
mac.init(key);    // dont' forget to initialize it

// read the file stream and compute its MAC value
FileInputStream fin = new FileInputStream("MyFile");
byte[] buffin = new byte[1024];
// read the file , 1K at a time, updating mac
while (fin.read(buffin)!=-1)
  mac.update(buffin);

byte[] rslt = mac.doFinal();     // generate MAC value
```

Aside: Pseudo Random Numbers

java . util .Random is a pseudo-random number generator which is fine for ordinary work, but not suitable for generating random cryptographic keys.

- It uses an algorithm that produces a predictable sequence of numbers. This means that an attacker who knows an earlier random value (key) can guess/predict later keys.
- If you don't provide a random seed value then it uses the value of the system clock: an attacker who knows roughly when the random value (key) was generated can guess as seed values.

These attacks are not that unlikely. We shall see later that keys are often distributed to principles by key distribution centers. An attacker who has been (legally) given one key might guess what other keys are being handed out.

Poor (crypto) PRNG's in the past include SESAME (key=rand());
MIT's Magic Cookie (key=rand()\%256); Kerberos V4:
srandom(time.tv_usec^time. tv_sec ^getpid()^ gethostid ()^counter++);
key=random();

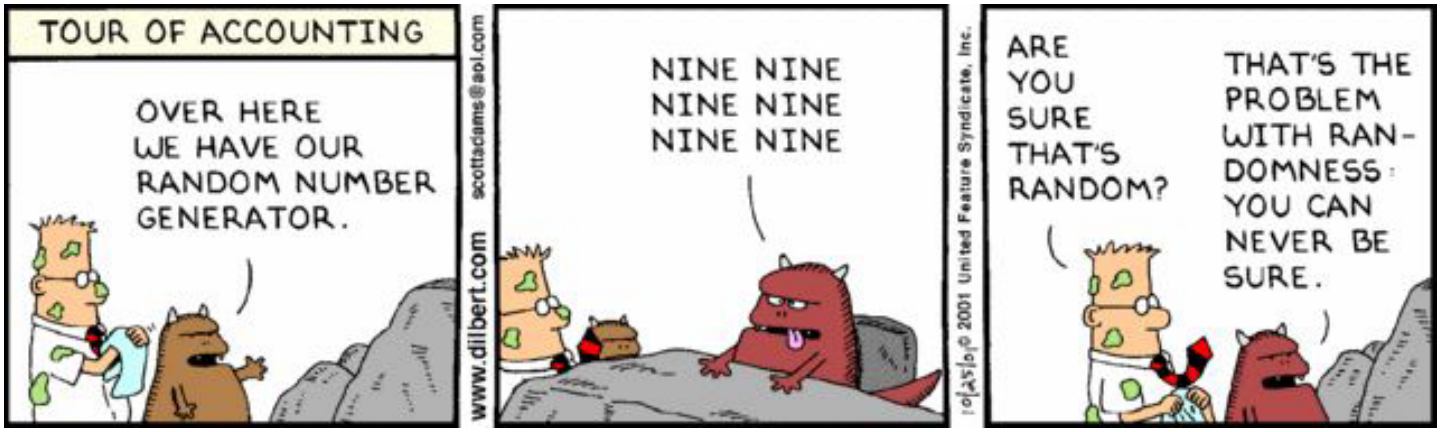
Aside: Pseudo Random Numbers

java . security .SecureRandom implements a stronger pseudo-random number generator that can be used to generate random cryptographic keys:

- SecureRandom(s) generates a PRNG object using seed s. The seed value is digested (SHA1) and resulting value stored as part of SecureRandom's state. An internal counter is set to zero.
- Given sr = SecureRandom(s), then sr .nextBytes(b) generates more pseudo-random numbers and stores them in **byte** b.. The message digest is updated with the internal state and the counter, which is incremented. The resulting data is digested and returned as the new pseudo-random number.

If you don't provide a seed when initializing a SecureRandom object then one gets generated from within the JVM based on the timing of threads. This has not been thoroughly studied; it might have weaknesses that could be exploited.

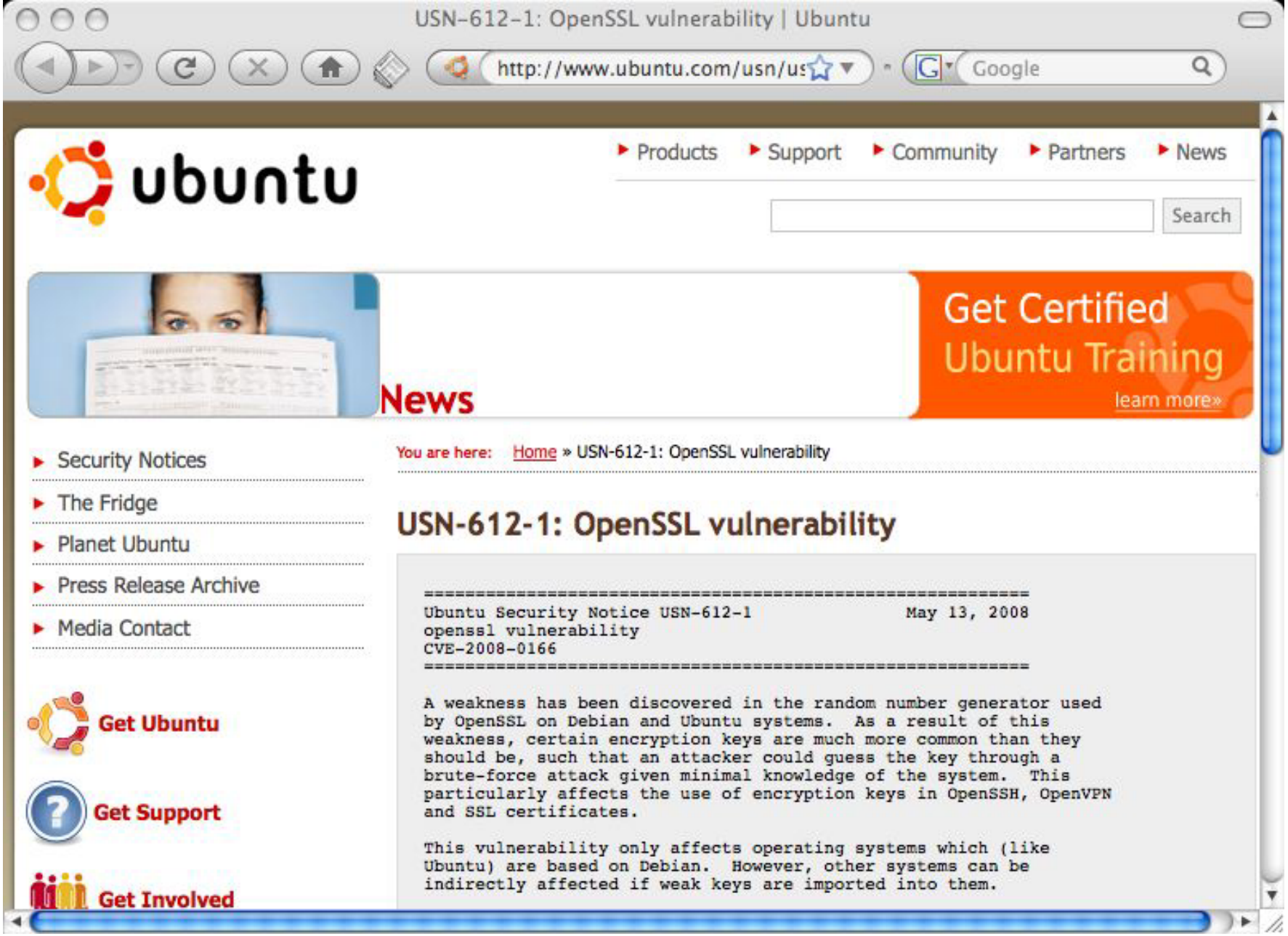
Sources of Entropy



An alternative and accepted technique for seeding pseudo-random numbers is based on measuring the timing between keyboard events. If using a fast timer (1ms or better) then using the low-order bits of timing information will provide the basis of a random seed for the generator: no typist will be able to type with millisecond precision. As used by PGP.

Linux kernel (and windows) entropy from keyboard, mouse movements and IDE timings.

Can also get entropy from hardware: sound from computer microphone, radioactive source, lava lamps



Developer accidently removed lines that referenced un-initialized memory that provided entropy for random number generation (for secret keys) and making it possible to brute force ‘guess’ keys.

Block Ciphers in Java: ECB Example

A code fragment to encrypted a small amount of plaintext using DES-ECB mode. Assume that we already have a suitable DES key.

Cipher factory method is used to generate a DES cipher (ECB and PKCS5Padding) and we initialize the cipher for encryption:

```
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, key);
```

As with Mac, we can use cipher to encrypt in one go. If we had generated a padding cipher then the cipher would expect plaintext to be a multiple of the DES blocksize:

```
byte[] cBytes= cipher.doFinal(data);
```

To decrypt, we just initialize a DES cipher for decryption.

```
....
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] pBytes= cipher.doFinal(cBytes);
```

Block Ciphers in Java: CBC Example

Code fragment to encrypt a small amount of plaintext using DES-CBC mode. Again, we assume that we already have a suitable DES key. To create the cipher (uninitialized):

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

Since cipher is a feedback cipher we need a random initialization vector:

```
SecureRandom sr = new SecureRandom();
byte[] iv = new byte[8];
sr.nextBytes(iv);
```

Now we can initialize the cipher (encryption) with its key and IV:

```
Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
IvParameterSpec spec = new IvParameterSpec(iv);
cipher.init(Cipher.ENCRYPT_MODE, key,spec);
```

And feed it in one go, as before:

```
byte[] ctext= cipher.doFinal(data);
```

Feeding Ciphers

As noted when we looked at MACs, it is not practical to encrypt a large amount of data 'in one go'. We should 'feed it' a blocks of data at a time. Assume we are given a suitably initialized DES/ECB/PKCS5Padding encryption cipher and we execute the following:

```
byte[] cBytesA = cipher.update("Simon".getBytes());
byte[] cBytesB = cipher.update("Foley".getBytes());
```

The block size for DES is 8 Bytes and since "Simon" is only 5 bytes then the length of cbytesA is 0. The second call to update adds 5 more bytes and cBytesB contains the first 8 bytes of ciphertext. The cipher keeps track of the leftover 2 bytes. If we done encrypting, then a call to doFinal() will generate the last block of ciphertext, suitably padded:

```
byte[] cBytesC = cipher.doFinal();
```

There are many (four) overloaded versions of the update() operation and six overloaded versions of doFinal() operation. Details are in the API specs, use them carefully!

Encrypting Streams

If your data to be encrypted/decrypted is a stream then things become easy. Encrypt (using key) contents of srcFile and store result in dstFile :

```
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE,key);
FileInputStream fin = new FileInputStream(srcFile);          // source file
FileOutputStream fout = new FileOutputStream(dstFile); // destination file
CipherOutputStream out = new CipherOutputStream(fout, cipher);
byte[] buffin new byte[1024];
int length;
while ((length = fin.read(buffin))!=-1)
  out.write(buffer,0, length);
fin.close;
fout.close;
```

How might we decrypt the contents of the destination file? Initialize with:

```
cipher.init(Cipher.DECRYPT_MODE,key);
```

Don't forget, streams are much more than just files!

Generating Secret Keys from Data: Key Translators

We can generate keys randomly using a suitable key generator, or we can build a key specification and construct the key from that. Key translators allow us to translate a Key object to bytes and vice-versa. This is useful, not just for (dubious) key generation, but also for converting keys to a form that can be transmitted over networks, stored locally on disk (and translated back again). We have seen one technique for translating Mac keys. SecretKeyFactory is used for translating the more complex secret cipher keys. From bytes to a key:

```
byte [] keyb = ...;
SecretKeyFactory desF = SecretKeyFactory.getInstance("DES");
KeySpec ks = new DESKeySpec(keyb);
SecretKey key = desF.generateSecret(ks);
```

From a key to bytes:

```
SecretKeyFactory desF = SecretKeyFactory.getInstance("DES");
DESKeySpec ks = (DESKeySpec) desF.getKeySpec(key,DESKeySpec.class);
byte [] keyb = ks.getKey();
```

Using Passphrases for Key Generation

Instead of having to store a secret key somewhere (eg. a file), a passphrase can be used to to generate a key. The passphrase is something a user can remember and type and the key is constructed by calculating a message digest of the passphrase.

```
// given passphrase in args[0], compute its MD5 hash value
MessageDigest pbmd= MessageDigest.getInstance("MD5");
byte[] pass=args[0].getBytes("UTF8"); // convert passphrase to bytes
byte[] keyBytes= pbmd.digest(pass);   //generate hash value

// and turn that into an HmacSHA1 key
SecretKey key = new SecretKeySpec(keyBytes, "HmacSHA1");
Mac mac= Mac.getInstance("HmacSHA1");
mac.init(key);
```

Obvious attacks include the use of easy to remember passphrases, dictionary attacks, and the user using bits of paper to remember the phrases. javax.crypto.spec.PBEKeySpec represents a key that is used with a passphrase. It encapsulates dictionary-foiling techniques such as salt.

Passphrase Encryption PKCS#5

Passphrases allow us to rely on the user to remember/provide keys.
PKCS#5 is an RSA 'standard' that describes how to use passphrases safely.
Given a password P and a salt value S then the derived key using a one-way hash based function H as:

$$K = H(P : S)$$

As we already know, the advantages of using salt are that it will be difficult for an attacker to precompute all keys from a dictionary of passwords, and that it will be unlikely that the same key will be selected twice.
PKCS#5 makes various recommendations on how the salt should be selected.
Providing support for repeated hashing of the passphrase/salt will make a dictionary based attack even harder. PKCS#5 recommends that the *iteration count* should be more that 1,000.
PKCS#5 specifies standard support for various ciphers and hash functions.

PKCS#5 in Java

To create a DES key from a passphrase we use a SecretKeyFactory:

```
String passphrase = args[0];      // assume provided by user
KeySpec ks = new PBEKeySpec(passphrase.toCharArray());
SecretKeyFactory skf = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey key = skf.generateSecret(ks);
```

We can create a cipher object for password-based encryption:

```
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
```

Initialize the cipher, letting JCE supply suitable salt and iteration values:

```
c.init(Cipher.ENCRYPT_MODE, key);
```

And encrypt some data:

```
byte[] cipherText = c.doFinal("This is just an example".getBytes());
```

Details of the algorithm, salt and iteration values, etc., are encapsulated within the passphrase based Cipher c object. These details could extracted for viewing as:

```
AlgorithmParameters algParams = c.getParameters();  // the parameters
algParams.toString();    // returns formatted string describing params
```

which, if printed, would give something like:

```
salt : [0000: D8 1C 2D 6F A3 08 AF 72]
iterationCount : 0a
```

These details (salt and iteration) should to be added to the front of the encrypted data, so that a principle who knows the passphrase can reconstruct the key specification before decrypting the data. Given salt and iter then:

```
AlgorithmParameterSpec aps = new PBEParameterSpec(salt,iter);
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
c.init(Cipher.ENCRYPT_MODE, key, aps);
```