Generating and Verifying Signatures

This lab walks you through the steps necessary to use the JDK Security API to generate a digital signature for some data and to verify that a signature is authentic. This lab is meant to show you how to incorporate security functionality into your programs, should you ever need to.

- So, we will demonstrate the use of the JDK Security API with respect to signing documents.
- We'll see what one program, executed by the person who has the original document, would do to generate keys, generate a digital signature for the document using the private key, and export the public key and the signature to files.
- Then we show an example of another program, executed by the receiver of the document, signature, and public key. It shows how the program could import the public key and verify the authenticity of the signature.
- Finally, we also discusses and demonstrates possible alternative approaches and methods of supplying and importing keys, including in certificates.

So we are going to create two basic applications, one for the digital signature generation and the other for the verification. The lab contains three sections.

- **Generating a Digital Signature:** shows using the API to generate keys and a digital signature for data using the private key and to export the public key and the signature to files. The application gets the data file name from the command line.
- **Verifying a Digital Signature:** shows using the API to import a public key and a signature that is alleged to be the signature of a specified data file and to verify the authenticity of the signature. The data, public key, and signature file names are specified on the command line.
- **Weaknesses and Alternatives:** discusses potential weaknesses of the approach used by the basic programs. It then presents and demonstrates possible alternative approaches and methods of supplying and importing keys, including the use of files containing encoded key bytes and the use of certificates containing public keys.

Generating a Digital Signature

The ***GenSig*** program you are about to create will use the JDK Security API to generate keys and a digital signature for data using the private key and to export the public key and the signature to files.

The following steps create the ***GenSig*** sample program.

1. **Prepare the initial program structure:**
   Create a text file named GenSig.java. Type in the initial program structure (import statements, class name, main method, and so on).

2. **Generate public and private keys**
   Generate a key pair (public key and private key). The private key is needed for signing the data. The public key will be used by the ***VerSig*** program for verifying the signature.

3. **Sign the data**
   Get a Signature object and initialize it for signing. Supply it with the data to be signed, and generate the signature.

4. **Save the signature and the public key in files**
   Save the signature bytes in one file and the public key bytes in another.

5. **Compile and run the program**

Prepare Initial Program Structure

Here's the basic structure of the GenSig program. Place it in a file called GenSig.java

```
import java.io.*;
import java.security.*;
class GenSig {

    public static void main(String[] args) {

        /* Generate a DSA signature */

        if (args.length != 1) {
            System.out.println("Usage: GenSig nameOfFileToSign");
            }
        else try {

        // the rest of the code goes here

        } catch (Exception e) {
            System.err.println("Caught exception " + e.toString());
        }
    }

}
```

Notes:

- The methods for signing data are in the java.security package, so the program imports everything from that package.
- The program also imports the java.io package, which contains the methods needed to input the file data to be signed.
- A single argument is expected, specifying the data file to be signed.
- The code written in following steps will go between the try and the catch blocks.

Generate Public and Private Keys

In order to be able to create a digital signature, you need a private key. (Its corresponding public key will be needed in order to verify the authenticity of the signature.)

In some cases the *key pair* (private key and corresponding public key) are already available in files. In that case the program can import and use the private key for signing.

In other cases the program needs to generate the key pair. A key pair is generated by using the KeyPairGenerator class.

In this example you will generate a public/private key pair for the Digital Signature Algorithm (DSA), and you will generate keys with a 1024-bit length.

Create a Key Pair Generator

The first step is to get a key-pair generator object for generating keys for the DSA signature algorithm.

The way to get a KeyPairGenerator object for a particular type of algorithm is to call the getInstance static factory method on theKeyPairGenerator class.

This method has two forms, both of which hava a String algorithm first argument; one form also has a String provider second argument. The sample code we will use will always specify the default SUN provider built into the JDK.

Put the following statement after the *else try {* line in the file we created in the previous step,

```
KeyPairGenerator keyGen =
    KeyPairGenerator.getInstance("DSA", "SUN");
```

Initialize the Key-Pair Generator

The next step is to initialize the key-pair generator. All key-pair generators share the concepts of a keysize and a source of randomness. The KeyPairGeneratorclass has an initialize method that takes these two types of arguments.

The keysize for a DSA key generator is the key length (in bits), which you will set to 1024.

The source of randomness must be an instance of the SecureRandom class. This example requests one that uses the SHA1PRNG pseudo-random-number generation algorithm, as provided by the built-in SUN provider. The example then passes this SecureRandom instance to the key-pair generator initialization method.

```
SecureRandom random =
             SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
```

Generate the Pair of Keys

The final step is to generate the key pair and to store the keys in PrivateKey and PublicKey objects.

```
KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

Sign the Data

Now that you have created a public key and a private key, you are ready to sign the data. In this example you will sign the data contained in a file. GenSig gets the file name from the command line. A digital signature is created (or verified) using an instance of the Signature class.

Signing data, generating a digital signature for that data, is done with the following steps.

Get a Signature Object:

The following gets a Signature object for generating or verifying signatures using the DSA algorithm, the same algorithm for which the program generated keys in the previous step.

```
Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
```

Note:

- When specifying the signature algorithm name, you should also include the name of the message digest algorithm used by the signature algorithm. SHA1withDSA is a way of specifying the DSA signature algorithm, using the SHA-1 message digest algorithm.

Initialize the Signature Object

Before a Signature object can be used for signing or verifying, it must be initialized. The initialization method for signing requires a private key. Use the private key placed into the PrivateKey object named priv in the previous step.

```
dsa.initSign(priv);
```

Supply the Signature Object the Data to Be Signed

This program will use the data from the file whose name is specified as the first (and only) command line argument. The program will read in the data a buffer at a time and will supply it to the Signature object by calling the update method.

```
FileInputStream fis = new FileInputStream(args[0]);
BufferedInputStream bufin = new BufferedInputStream(fis);
byte[] buffer = new byte[1024];
int len;
while ((len = bufin.read(buffer)) >= 0) {
    dsa.update(buffer, 0, len);
};
bufin.close();
```

Generate the Signature

Once all of the data has been supplied to the Signature object, you can generate the digital signature of that data.

```
byte[] realSig = dsa.sign();
```

Now that you have generated a signature for some data, you need to save the signature bytes in one file and the public key bytes in another so you can send (via modem, CD, Email, and so on) to someone else

- the data for which the signature was generated,
- the signature, and
- the public key

The receiver can verify that the data came from you and was not modified in transit by running the VerSig program you will generate in the upcoming steps.

That program uses the public key to verify that the signature received is the true signature for the data received.

Recall that the signature was placed in a byte array named realSig. You can save the signature bytes in a file named sig via the following.

```
        /* save the signature in a file */
        FileOutputStream sigfos = new FileOutputStream("sig");
        sigfos.write(realSig);
        sigfos.close();
```

Recall from the **Generate Public and Private Keys** step that the public key was placed in a PublicKey object named pub. You can get the encoded key bytes by calling thegetEncoded method and then store the encoded bytes in a file. You can name the file whatever you want. If, for example, your name is Susan, you might name it something likesuepk (for "Sue's public key"), as in the following:

```
        /* save the public key in a file */
        byte[] key = pub.getEncoded();
        FileOutputStream keyfos = new
FileOutputStream("suepk");
        keyfos.write(key);
        keyfos.close();
```

Now that we have added all the source code for GenSig.java. Compile and run it. Remember, you need to specify the name of a file to be signed, as in

java GenSig data

After executing the program, you should see the saved suepk (public key) and sig (signature) files.

Verifying a Digital Signature

If you have data for which a digital signature was generated, you can verify the authenticity of the signature. To do so, you need

- the data
- the signature
- the public key corresponding to the private key used to sign the data

In this example you write a VerSig program to verify the signature generated by the GenSig program. This demonstrates the steps required to verify the authenticity of an alleged signature.

VerSig imports a public key and a signature that is alleged to be the signature of a specified data file and then verifies the authenticity of the signature. The public key, signature, and data file names are specified on the command line.

The steps to create the VerSig sample program to import the files and to verify the signature are the following.

1. **Prepare Initial Program Structure**
   Create a text file named VerSig.java. Type in the initial program structure (import statements, class name, main method, and so on).

2. **Input and Convert the Encoded Public Key Bytes**
   Import the encoded public key bytes from the file specified as the first command line argument and convert them to a PublicKey.

3. **Input the Signature Bytes**
   Input the signature bytes from the file specified as the second command line argument.

4. **Verify the Signature**
   Get a Signature object and initialize it with the public key for verifying the signature. Supply it with the data whose signature is to be verified (from the file specified as the third command line argument), and verify the signature.

5. **Compile and Run the Program**

Prepare Initial Program Structure

Here's the basic structure of the VerSig program. Place this program structure in a file called VerSig.java.

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

class VerSig {

    public static void main(String[] args) {

        /* Verify a DSA signature */

        if (args.length != 3) {
            System.out.println("Usage: VerSig publickeyfile
signaturefile datafile");
            }
        else try{

        // the rest of the code goes here

        } catch (Exception e) {
            System.err.println("Caught exception " +
e.toString());
        }
    }

}
```

Notes

- Three arguments are expected, specifying the public key, the signature, and the data files.

Input and Convert the Encoded Public Key Bytes

Next, VerSig needs to import the encoded public key bytes from the file specified as the first command line argument and to convert them to a PublicKey. A PublicKey is needed because that is what the Signature initVerify method requires in order to initialize the Signature object for verification.

First, read in the encoded public key bytes.

```
        FileInputStream keyfis = new FileInputStream(args[0]);
        byte[] encKey = new byte[keyfis.available()];
        keyfis.read(encKey);

        keyfis.close();
```

Now the byte array encKey contains the encoded public key bytes.

So, first you need a key specification. You can obtain one via the following, assuming that the key was encoded according to the X.509 standard, which is the case, for example, if the key was generated with the built-in DSA key-pair generator supplied by the SUN provider:

```
X509EncodedKeySpec pubKeySpec =
       new X509EncodedKeySpec(encKey);
```

Now you need a KeyFactory object to do the conversion. That object must be one that works with DSA keys.

```
KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
```

Finally, you can use the KeyFactory object to generate a PublicKey from the key specification.

```
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
```

Next, input the signature bytes from the file specified as the second command line argument.

```
FileInputStream sigfis = new FileInputStream(args[1]);
byte[] sigToVerify = new byte[sigfis.available()];
sigfis.read(sigToVerify);
sigfis.close();
```

Now the byte array sigToVerify contains the signature bytes.

Verify the Signature

You've added code to the VerSig program to

- Input the encoded key bytes and converted them to a PublicKey named pubKey
- Input the signature bytes into a byte array named sigToVerify

You can now proceed to do the verification.

Initialize the Signature Object for Verification

As with signature generation, a signature is verified by using an instance of the Signature class. You need to create a Signature object that uses the same signature algorithm as was used to generate the signature. The algorithm used by the GenSig program was the SHA1withDSA algorithm from the SUN provider.

```
Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
```

Next, you need to initialize the Signature object. The initialization method for verification requires the public key.

```
sig.initVerify(pubKey);
```

Supply the Signature Object With the Data to be Verified

You now need to supply the Signature object with the data for which a signature was generated. This data is in the file whose name was specified as the third command line argument. As you did when signing, read in the data one buffer at a time, and supply it to the Signature object by calling the update method.

```
FileInputStream datafis = new FileInputStream(args[2]);
BufferedInputStream bufin = new BufferedInputStream(datafis);

byte[] buffer = new byte[1024];
int len;
while (bufin.available() != 0) {
    len = bufin.read(buffer);
    sig.update(buffer, 0, len);
};

bufin.close();
```

Verify the Signature

Once you have supplied all of the data to the Signature object, you can verify the digital signature of that data and report the result. Recall that the alleged signature was read into a byte array called sigToVerify.

```
boolean verifies = sig.verify(sigToVerify);

System.out.println("signature verifies: " + verifies);
```

The verifies value will be true if the alleged signature (sigToVerify) is the actual signature of the specified data file generated by the private key corresponding to the public key pubKey.

Compile and Run the Program

Compile and run the program. Remember, you need to specify three arguments on the command line:

- The name of the file containing the encoded public key bytes
- The name of the file containing the signature bytes
- The name of the data file (the one for which the signature was generated)

Since you will be testing the output of the GenSig program, the file names you should use are

- suepk
- sig
- data

Here's a sample run; the bold indicates what you type.

%java VerSig suepk sig data
signature verifies: true

File GenSig.java

```
import java.io.*;
import java.security.*;

class GenSig {

    public static void main(String[] args) {

        /* Generate a DSA signature */

        if (args.length != 1) {
            System.out.println("Usage: GenSig nameOfFileToSign");
            }
        else try{

            /* Generate a key pair */

            KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA", "SUN");
            SecureRandom random = SecureRandom.getInstance("SHA1PRNG",
"SUN");

            keyGen.initialize(1024, random);

            KeyPair pair = keyGen.generateKeyPair();
            PrivateKey priv = pair.getPrivate();
            PublicKey pub = pair.getPublic();
```

```
            /* Create a Signature object and initialize it with the
private key */

            Signature dsa = Signature.getInstance("SHA1withDSA",
"SUN");

            dsa.initSign(priv);

            /* Update and sign the data */

            FileInputStream fis = new FileInputStream(args[0]);
            BufferedInputStream bufin = new BufferedInputStream(fis);
            byte[] buffer = new byte[1024];
            int len;
            while (bufin.available() != 0) {
                len = bufin.read(buffer);
                dsa.update(buffer, 0, len);
                };

            bufin.close();

            /* Now that all the data to be signed has been read in,
                    generate a signature for it */

            byte[] realSig = dsa.sign();


            /* Save the signature in a file */
            FileOutputStream sigfos = new FileOutputStream("sig");
            sigfos.write(realSig);

            sigfos.close();


            /* Save the public key in a file */
            byte[] key = pub.getEncoded();
            FileOutputStream keyfos = new FileOutputStream("suepk");
            keyfos.write(key);

            keyfos.close();

        } catch (Exception e) {
            System.err.println("Caught exception " + e.toString());
        }

    };

}
```

File VerSig.java

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

class VerSig {

    public static void main(String[] args) {

        /* Verify a DSA signature */

        if (args.length != 3) {
            System.out.println("Usage: VerSig publickeyfile
signaturefile datafile");
            }
        else try{

            /* import encoded public key */

            FileInputStream keyfis = new FileInputStream(args[0]);
            byte[] encKey = new byte[keyfis.available()];
            keyfis.read(encKey);

            keyfis.close();

            X509EncodedKeySpec pubKeySpec = new
X509EncodedKeySpec(encKey);

            KeyFactory keyFactory = KeyFactory.getInstance("DSA",
"SUN");
            PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

            /* input the signature bytes */
            FileInputStream sigfis = new FileInputStream(args[1]);
            byte[] sigToVerify = new byte[sigfis.available()];
            sigfis.read(sigToVerify );

            sigfis.close();

            /* create a Signature object and initialize it with the
public key */
            Signature sig = Signature.getInstance("SHA1withDSA",
"SUN");
            sig.initVerify(pubKey);

            /* Update and verify the data */

            FileInputStream datafis = new FileInputStream(args[2]);
            BufferedInputStream bufin = new
BufferedInputStream(datafis);

            byte[] buffer = new byte[1024];
            int len;
            while (bufin.available() != 0) {
                len = bufin.read(buffer);
                sig.update(buffer, 0, len);
                };
```

```
            bufin.close();


            boolean verifies = sig.verify(sigToVerify);

            System.out.println("signature verifies: " + verifies);


        } catch (Exception e) {
            System.err.println("Caught exception " + e.toString());
};

    }

}
```

Weaknesses and Alternatives

The GenSig and VerSig programs in this lesson illustrate the use of the JDK Security API to generate a digital signature for data and to verify that a signature is authentic. However, the actual scenario depicted by those programs, in which a sender uses the JDK Security API to generate a new public/private key pair, the sender stores the encoded public key bytes in a file, and the receiver reads in the key bytes, is not necessarily realistic, and has a potential major flaw.

In many cases the keys do not need to be generated; they already exist, either as encoded keys in files or as entries in a keystore.

The potential major flaw is that nothing guarantees the authenticity of the public key the receiver receives, and the VerSig program correctly verifies the authenticity of a signature only if the public key it is supplied is *itself* authentic!

Working with Encoded Key Bytes

Sometimes encoded key bytes already exist in files for the key pair to be used for signing and verification. If that's the case the GenSig program can import the encoded private key bytes and convert them to a PrivateKey needed for signing, via the following, assuming that the name of the file containing the private key bytes is in the privkeyfileString and that the bytes represent a DSA key that has been encoded by using the PKCS #8 standard.

```
FileInputStream keyfis = new FileInputStream(privkeyfile);
byte[] encKey = new byte[keyfis.available()];
keyfis.read(encKey);
keyfis.close();
```

```
PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey = keyFactory.generatePrivate(privKeySpec);
```

GenSig no longer needs to save the public key bytes in a file, as they're already in one.

In this case the sender sends the receiver

- the already existing file containing the encoded public key bytes (unless the receiver already has this) and
- the data file and the signature file exported by GenSig.

The VerSig program remains unchanged, as it already expects encoded public key bytes in a file.

But what about the potential problem of a malicious user intercepting the files and replacing them all in such a way that their switch cannot be detected?

In some cases this is not an issue, because people have already exchanged public keys face to face or via a trusted third party that does the face-to-face exchange. After that, multiple subsequent file and signature exchanges may be done remotely (that is, between two people in different locations), and the public keys may be used to verify their authenticity. If a malicious user tries to change the data or signature, this is detected by VerSig.

If a face-to-face key exchange is not possible, you can try other methods of increasing the likelihood of proper receipt. For example, you could send your public key via the most secure method possible prior to subsequent exchanges of data and signature files, perhaps using less secure mediums.
In general, sending the data and the signature separately from your public key greatly reduces the likelihood of an attack. Unless all three files are changed, and in a certain manner discussed in the next paragraph, VerSig will detect any tampering.

If all three files (data document, public key, and signature) were intercepted by a malicious user, that person could replace the document with something else, sign it with a private key, and forward on to you the replaced document, the new signature, and the public key corresponding to the private key used to generate the new signature. Then VerSig would report a successful verification, and you'd think that the document came from the original sender. Thus you should take steps to ensure that at least the public key is received intact (VerSig detects any tampering of the other files), or you can use certificates to facilitate authentication of the public key, as described in the next section.

Working with Certificates

It is more common in cryptography to exchange *certificates* containing public keys rather than the keys themselves.

One benefit is that a certificate is signed by one entity (the *issuer*) to verify that the enclosed public key is the actual public key of another entity (the *subject* or *owner*). Typically a trusted third-party *certification authority* (CA) verifies the identity of the subject and then vouches for its being the owner of the public key by signing the certificate.

Another benefit of using certificates is that you can check to ensure the validity of a certificate you received by verifying its digital signature, using its issuer's (signer's) public key, which itself may be stored in a certificate whose signature can be verified by using the public key of that certificate issuer; that public key itself may be stored in a certificate, and so on, until you reach a public key that you already trust.

If you cannot establish a trust chain (perhaps because the required issuer certificates are not available to you), the certificate **fingerprint(s)** can be calculated. Each fingerprint is a relatively short number that uniquely and reliably identifies the certificate. (Technically it's a hash value of the certificate information, using a message digest, also known as a one-way hash function.) You can call up the certificate owner and compare the fingerprints of the certificate you received with the ones sent. If they're the same, the certificates are the same.

It would be more secure for GenSig to create a certificate containing the public key and for VerSig to then import the certificate and extract the public key. However, the JDK has no public certificate APIs that would allow you to create a certificate from a public key, so the GenSig program cannot create a certificate from the public key it generated. (There *are* public APIs for extracting a public key from a certificate, though.)

If you want, you can use the various security tools, not APIs, to sign your important document(s) and work with certificates from a keystore, as was done in the Exchanging Fileslesson.

Alternatively you can use the API to modify your programs to work with an already existing private key and corresponding public key (in a certificate) from your keystore. To start, modify the GenSig program to extract a private key from a keystore rather than generate new keys. First, let's assume the following.

- The keystore name is in the String ksName
- The keystore type is "JKS", the proprietary type created by Sun Microsystems
- The keystore password is in the char array spass
- The alias to the keystore entry containing the private key, and the public key certificate is in the String alias
- The private key password is in the char array kpass

Then you can extract the private key from the keystore via the following.

```
KeyStore ks = KeyStore.getInstance("JKS");
FileInputStream ksfis = new FileInputStream(ksName);
BufferedInputStream ksbufin = new BufferedInputStream(ksfis);

ks.load(ksbufin, spass);
PrivateKey priv = (PrivateKey) ks.getKey(alias, kpass);
```

You can extract the public key certificate from the keystore and save its encoded bytes to a file named suecert, via the following.

```
java.security.cert.Certificate cert = ks.getCertificate(alias);
byte[] encodedCert = cert.getEncoded();

/* save the certificate in a file named "suecert" */
FileOutputStream certfos = new FileOutputStream("suecert");
certfos.write(encodedCert);
certfos.close();
```

Then you send the data file, the signature, and the certificate to the receiver. The receiver verifes the authenticity of the certificate by first getting the certificate's fingerprints, via thekeytool -printcert command.

```
keytool -printcert -file suecert
  Owner: CN=Susan Jones, OU=Purchasing, O=ABC, L=Cupertino, ST=CA, C=US
  Issuer: CN=Susan Jones, OU=Purchasing, O=ABC, L=Cupertino, ST=CA, C=US
  Serial number: 35aaed17
  Valid from: Mon Jul 13 22:31:03 PDT 1998 until:
  Sun Oct 11 22:31:03 PDT 1998
  Certificate fingerprints:
      MD5:  1E:B8:04:59:86:7A:78:6B:40:AC:64:89:2C:0F:DD:13
      SHA1: 1C:79:BD:26:A1:34:C0:0A:30:63:11:6A:F2:B9:67:DF:E5:8D:7B:5E
```

Then the receiver verifies the fingerprints, perhaps by calling the sender up and comparing them with those of the sender's certificate or by looking them up in a public repository.

The receiver's verification program (a modified VerSig) can then import the certificate and extract the public key from it via the following, assuming that the certificate file name (for example, suecert) is in the String certName.

```
FileInputStream certfis = new FileInputStream(certName);
java.security.cert.CertificateFactory cf =
            java.security.cert.CertificateFactory.getInstance("X.509");
java.security.cert.Certificate cert = cf.generateCertificate(certfis);
PublicKey pub = cert.getPublicKey();
```

Ensuring Data Confidentiality

Suppose that you want to keep the contents of the data confidential so people accidentally or maliciously trying to view it in transit (or on your own machine or disk) cannot do so. To keep the data confidential, you should encrypt it and store and send only the encryption result (referred to as *ciphertext*). The receiver can decrypt the ciphertext to obtain a copy of the original data.