DATABASE PROGRAMMING IN JAVA

Session: 5

Digital Signatures

Objectives

- Explain Digital Signatures
- Explain signing and verifying data using Java tools
- Explain signing and verifying data using Security API

Introduction of Digital Signatures

Digital signatures can be used to implement security and prevent unauthorized access of data.

Digital signatures are used to digitally sign messages or objects. Digital signatures not only verify that the content of a message is unaltered, but also help to identify the creator of a message. Following are the two purposes of digital signatures:

- **Ensuring message content integrity**
  - Some mathematical calculations are performed on a message repeatedly to generate a digital signature, which is appended at the end of the message.
- **Verifying the authenticity of the message sender**
  - A digital signature ensures that an encrypted message cannot be deciphered by unintended recipients.

Digital Signature Elements

Digital Signatures are generated by Public Key Cryptography, using public and private keys to encrypt and decrypt messages. Each key is used differently for message signing.

- **Public Key**
  - Each sender has a unique public key which is accessible easily to the recipient. A public key is used by a receiver to decrypt a message, thus establishing the authenticity.
- **Private Key**
  - A sender encrypts a message with his private key. If a receiver can decipher the received message with the senders public key, the data must be from the sender.

Benefits of Digital Signatures

- **Authentication**
- **Integrity**

Digital Signature Basics

Digital signature is used to authenticate any message, which can be anything from an electronic mail to a contract. A digital signature uses three algorithms. They are as follows:

- **Key Generation Algorithm**
  - The key generation algorithm generates a pair of keys for the signer.
- **Signing Algorithm**
  - The algorithm produces a signature on the input of a message and a signing key.
- **Signature Verification Algorithm**
  - The signature verifying algorithm on input of a message, verifying key and a signature will accept or reject the document.

Digital Signature Algorithm

Full Domain Hash (FDH) is based on Rivest-Shamir-Adleman (RSA) algorithm using the hash-and-sign standard.

Digital Signature Algorithm (DSA) has two phases in key generation, choosing the algorithm parameters, key generation.

Elliptic Curve DSA is a variation of the Digital Signature Algorithm and it works on elliptic curve groups.

ElGamal Signature Algorithm allows the verifier to confirm the authenticity of a message, when it is sent using an insecure channel.

Undeniable Signature uses two processes, that is, it has an interactive verification process and denial protocol.

Aggregate Signature supports aggregation. It can aggregate multiple signatures into single signature.

Working of Digital Signatures 1-3

Comparing Digital Signatures with Message Digest

- A message digest alters the content of a message into a fixed length result. The original message content cannot be recovered from the digest.
- A message digest does not provide secrecy, but a digital signature is encrypted.

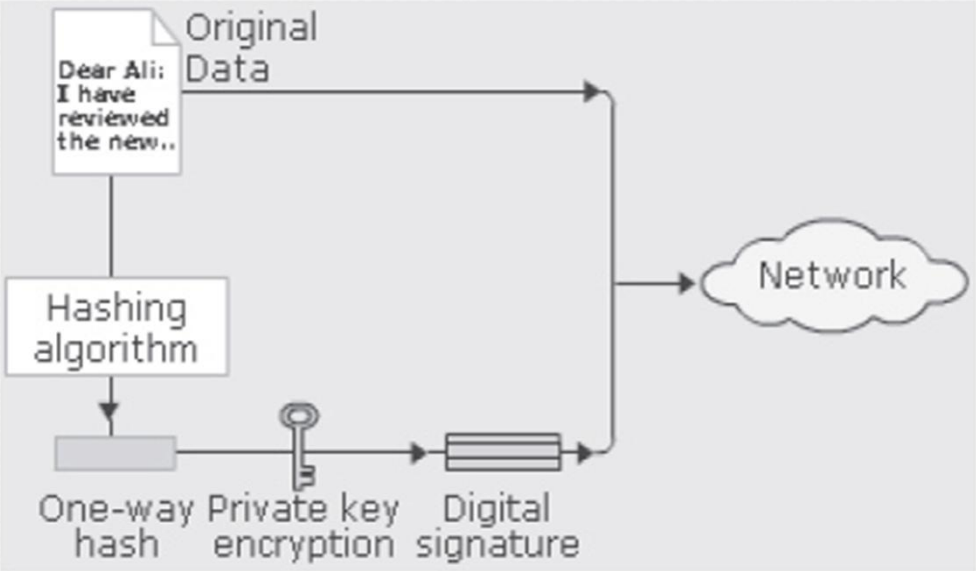
Digital signatures work by using the mechanism of encryption and decryption.

Working of Digital Signatures 2-3

The steps of encryption of digital signatures are as follows:

- A Hash or a message digest is prepared using the Hashing Algorithm.
- The hashed data or message digest is encrypted using the senders private key.
- The digital signature and the senders public key are appended to the end of the message.

Following figure demonstrates the encryption:

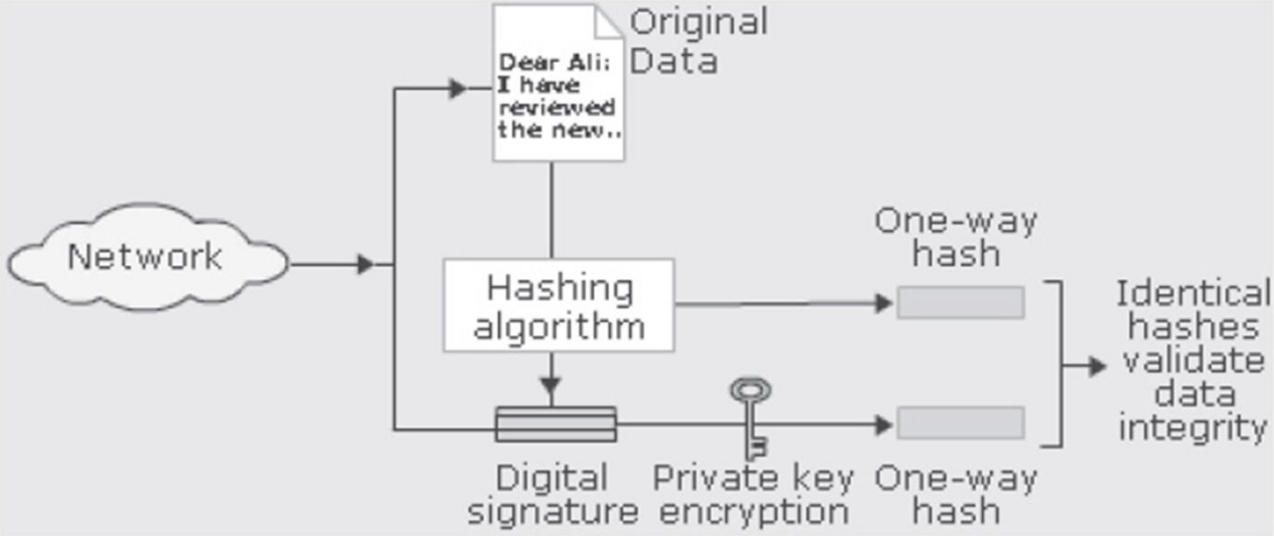


Working of Digital Signatures 3-3

The steps of decryption are as follows:

- The receiver receives the message and a digitally signed message digest.
- The receiver separately calculates a message digest for the received message.
- The receiver uses the senders public key to decrypt the signed message digest that was received and compares this to the independently calculated message digest.
- If the two digests do not match, the data may have been tampered with or the data may not be authentic or may not have been intended for the receiver.

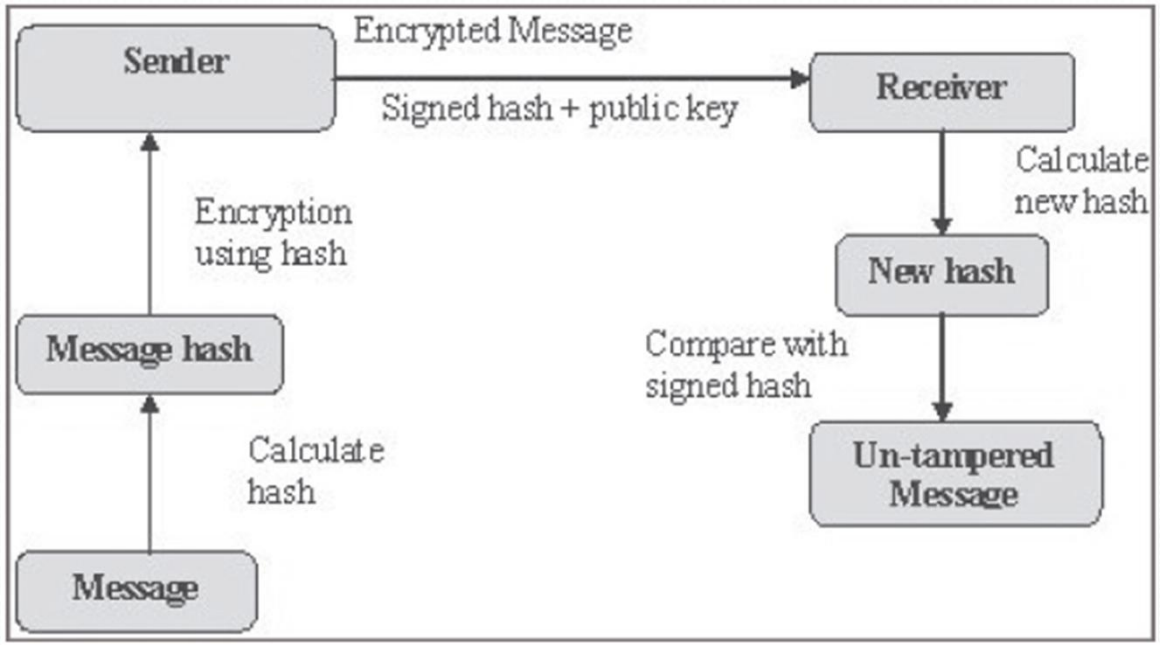
Following figure demonstrates the decryption process:



Validating Data Integrity

- Data integrity is said to be maintained, if there is no difference between data that is sent and received.
- A receiver checks data integrity by calculating a new hash value on the received message and decrypts the signed hash received.
- The computed hash is compared with the hash decrypted from digital signature, if both values match, it implies the data integrity is validated.

Following figure demonstrates the validation process:



Drawbacks of Digital Signatures

Drawbacks of Digital Signatures

- **Non-Repudiation**
  - Repudiation means disclaiming responsibility for a sent message. A digital signature makes non-repudiation difficult.
- **Timestamping**
  - Digital Signatures do not contain any record of the date and time when a particular document was signed.

Digital Certificates

- Impersonation using a false public key can be reduced with the use of digital certificates.
- Digital Certificates prevent impersonation by storing a widely known and distributed public key, information such as name, e-mail address, and other application-specific data about the certificate owner.
- A Certification Authority (CA) issues these certificates and ensures the validity of information in the certificate.

Standards and Features of Digital Certificates

- **X.509**

  The X.509 standard was created by the international telephone standards body, to authenticate the originator of Internet objects.

- **PGP (Pretty Good Privacy)**

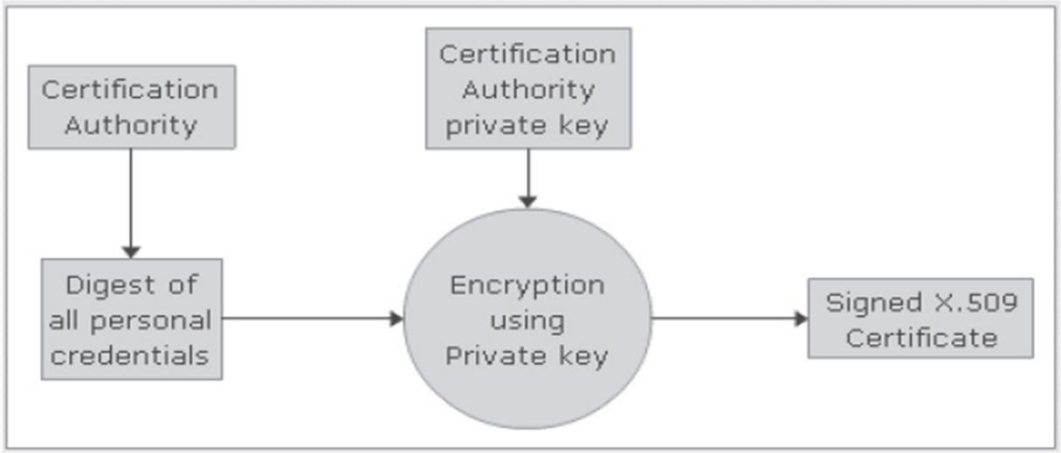
  The PGP standard which stands for Pretty Good Privacy was developed by Phil Zimmermann, is used for encrypting, compressing, and authenticating e-mail messages and attachments.

Following figure shows standards and features of digital signatures:

Verifying the Authenticity of the Sender 1-3

The steps to ensure the authenticity of a sender, using a Digital Certificate are:

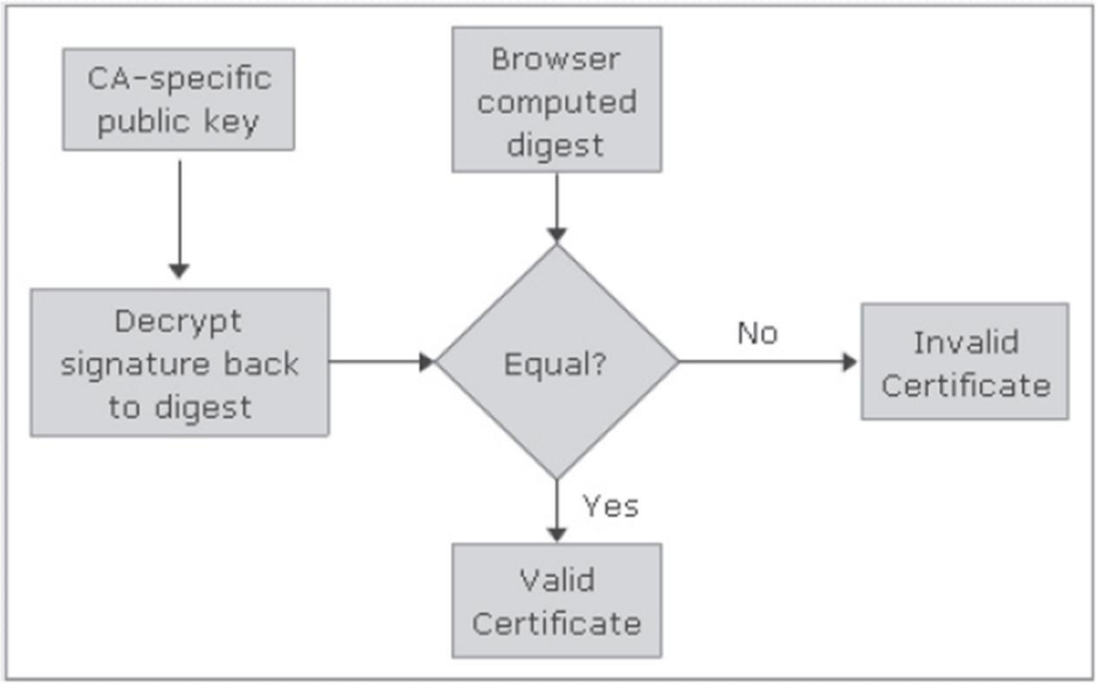
- **Validation of the credential of the party applying for digital certificate**
  - A digital ID is used along with a public key encryption system for credentials verification.
  - The certification authority verifies that the public key belongs to the specific individual or company.
- **Creation of the certificate**
  - If the validation process is completed successfully, the CA creates an X.509 certificate.
  - Following figure demonstrates the process of issuing a digital certificate:



Verifying the Authenticity of the Sender 2-3

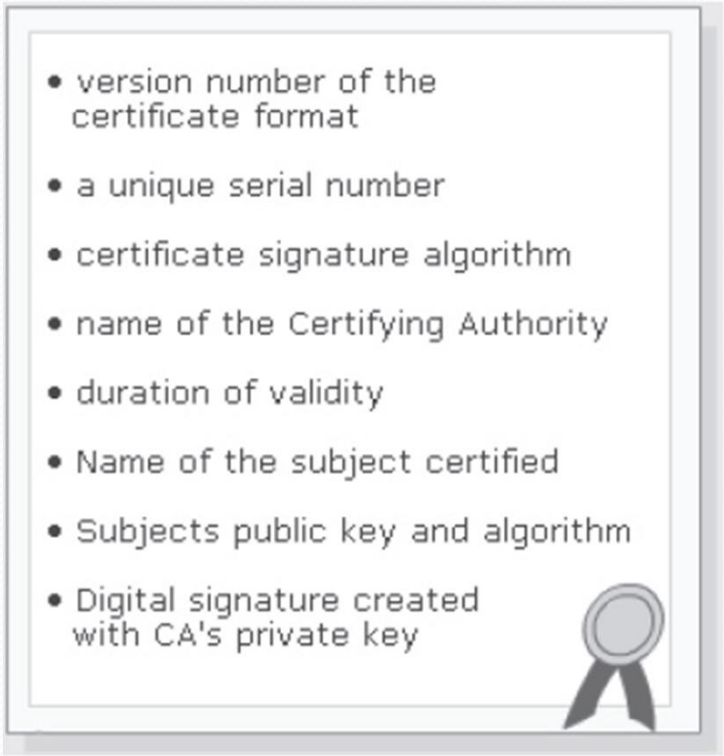
- **Verifying the certificate**
  - The recipient verifies the certificate through Web browser.
  - The browser re-computes its own digest from the plain text in the certificate and compares it with the decrypted digest.
  - If the two digests match, then the certificate is said to be valid.

Following figure shows the verification of the certificate:



Verifying the Authenticity of the Sender 3-3

- **Generating the certificate**
  - Certificate is generated based on the validated information. Following figure shows a digital certificate with information:



- **Sign and verify the digital certificate**

Trusted Timestamping

Trusted Timestamping

- Timestamp refers to a time code where the signer guarantees the existence of the signed document at the time specified as part of the digital signature.
- Useful for logging events.
- A trusted third-party acts as a timestamping authority and provides the timestamp.

Creation of Timestamp

- Timestamp is created using digital signature and hash function techniques, which in turn is validated through TimeStamp Authority(TSA).

Verifying Timestamp

- The digital signature of TSA is checked by decrypting the signed hash given by the TSA with the public key of the TSA.

Signing and Verifying Data Using Java Tools 1-3

Data is usually transmitted in the form of a JAR file which is a Java Archive file. The JAR file is usually digitally signed through cryptographic keys and authenticated using digital certificates.

Digital Keys and Key Store

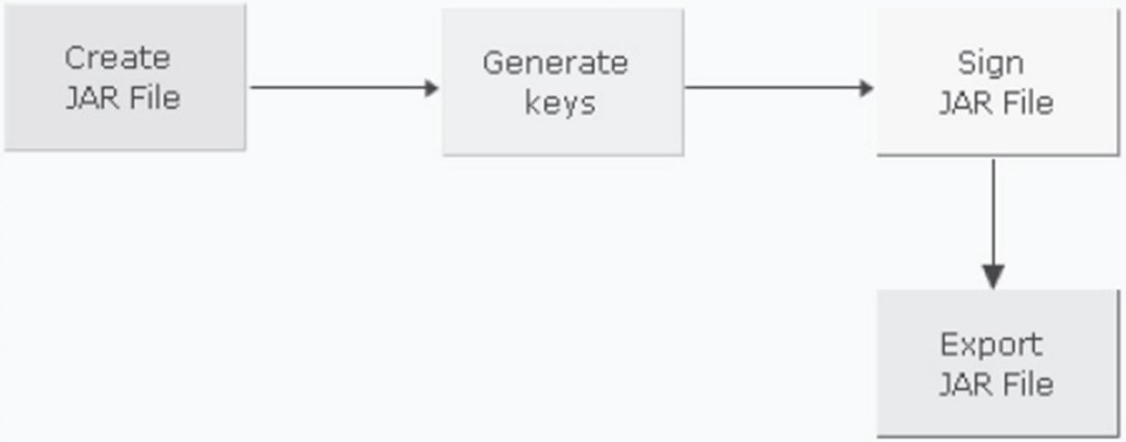
- A digital key is a kind of a password or a passphrase used in cryptography calculations.
- A file containing a collection of public and private keys is known as a keystore.

Signing a Jar file

- Create a JAR file containing the data file to be sent.
- Generate the keys for signing the JAR file. The key generator tool also creates a keystore to store the keys generated.
- Sign the JAR file using the jarsigner tool and the private key generated in the previous step.
- Export the public key so that message recipients can authenticate the senders signature. The public key is exported as a copy of the certificate obtained from the keystore.

Signing and Verifying Data Using Java Tools 2-3

Following figure shows the process of signing a JAR file:



The syntax to sign a JAR file is as follows:

Syntax:

```
jar cvf Data.jar data.txt
```

where,
`data.txt` – file containing the data
`Data.jar` – JAR file created to be signed and transmitted

Signing and Verifying Data Using Java Tools 3-3

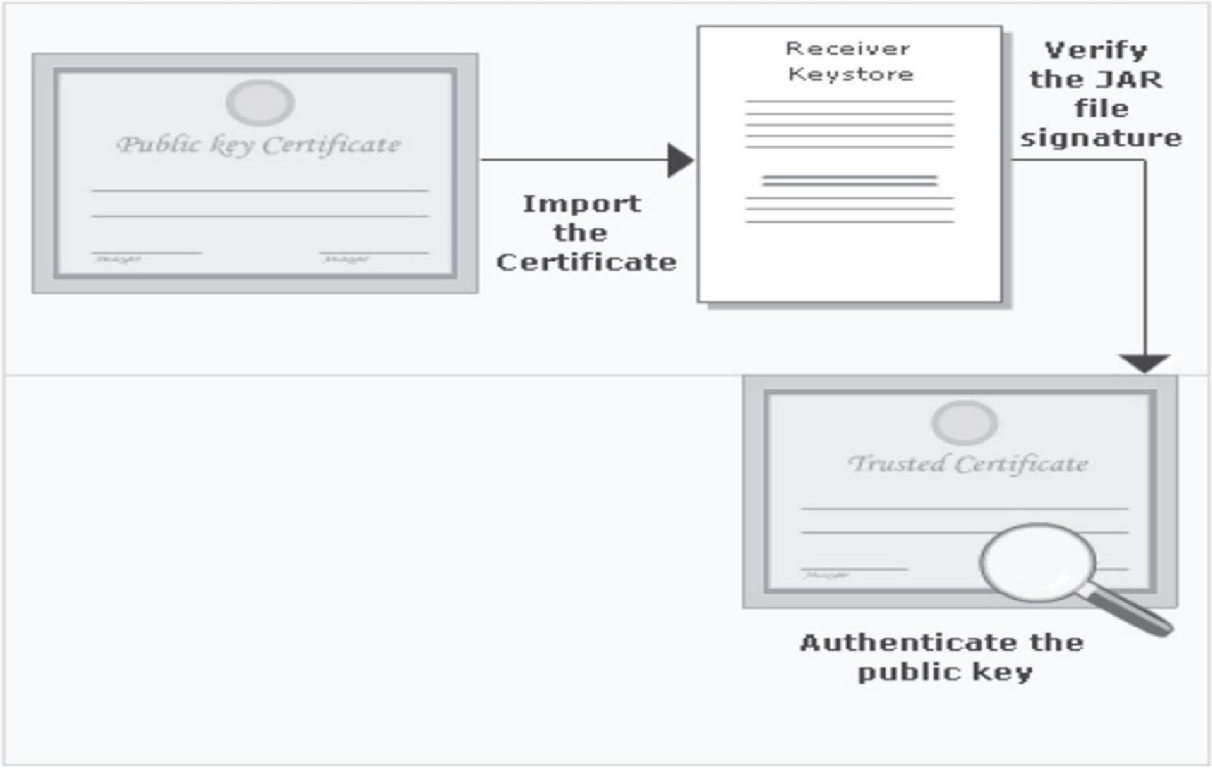
- The following command generates a `keystore` file and also generates the public and private keys for an organization **Financial**. It then stores the keys in the `keystore` file.
  - `keytool -genkey -alias signFinancial -keystore senderStore`
- The following command signs the JAR file and a new signed JAR file is created:
  - `jarsigner -keystore senderStore -signedjar sData.jar Data.jar signFinancial`
- The signed JAR file and the certificate containing a copy of the public key have to be exported to the client for authentication.
- The following command exports the public key certificate:
  - `keytool -export -keystore senderStore -alias signFinancial -file Email.cer`

Verification of a Digital Certificate 1-2

The recipient performs the following steps to verify a digital signature:

- Import the certificate as a Trusted Certificate
- Verify the JAR file signature

Following figure shows verification of digital certificate:



Verification of a Digital Certificate 2-2

- Import the certificate as a trusted certificate and store it with an alias named sender.
- The following command accomplishes this:
  - `keytool -import -alias sender -file Email.cer -keystore receiverStore`
- The following command verifies if the signature of the JAR file is valid:
  - `jarsigner -verify -verbose -keystore receiverStore sData.jar`

Signing and Verifying Data Using Security API 1-2

Java provides a Security API which has various classes and interfaces implementing various security mechanisms.

- **`KeyPairGenerator` Class**

  `KeyPairGenerator` class is an abstract class and is derived from `KeyPairGeneratorSpi` class. Public and private keys are generated using `KeyPairGenerator` class.

Following table shows some of the methods used in the class:

Method	Description
`KeyPairGenerator(String alg)`	The constructor creates a `KeyPairGenerator` object with the specified algorithm.
`KeyPair generateKeyPair()`	The method generates a key pair.
`KeyPair genKeyPair()`	This method generates a key pair and a new key pair will be generated every time the method is invoked.
`String getAlgorithm()`	The method retrieves the algorithm name for this key pair generator.
`KeyPairGenerator getInstance (String alg)`	The method creates an object of the `KeyPairGenerator` class with the specified digest algorithm.

Signing and Verifying Data Using Security API 2-2

Method	Description
`KeyPairGenerator getInstance(String alg, String provider)`	The method creates an object of the `KeyPairGenerator` class with the specified digest algorithm as provided by the specified provider.
`void initialize(AlgorithmParameterSpec parameter)`	The method uses the specified parameter set to initialize the key pair generator. The method uses the highest priority installed provider or the system provided as the source of randomness.
`void initialize (int size)`	The method uses the key size for initializing a key pair generator. The default parameter set and SecureRandom implementation of the highest priority installed provider is used as the source of randomness.
`void initialize (int size, SecureRandom rand)`	The method uses the specified key size and the specified source of randomness for initializing the key pair generator.

Generating Public and Private Keys 1-2

A key pair is generated by using the `KeyPairGenerator` class.

- Following are the steps performed by using JDK Security API to generate keys and a digital signature:
- **Create a Key-Pair Generator**
  - A key-pair generator object is obtained by invoking the `getInstance()` static method on the `KeyPairGenerator` class.
  - The following Code Snippet shows creation of a `KeyPairGenerator` object:

Code Snippet:

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

Generating Public and Private Keys 2-2

- **Initialize the Key-Pair Generator**
  - The following Code Snippet sets the key size and the source of randomness for a key-pair generator:

Code Snippet:

```
SecureRandom random=SecureRandom.getInstance("SHA1PRNG",
"SUN");
keyGenerator.initialize(1024, random);
KeyPair pair=keyGenerator.generateKeyPair();
PrivateKey privKey=pair.getPrivate();
PublicKey pubKey=pair.getPublic();
```

Cryptographic Keys 1-7

In symmetric key cryptography, a single key is used to encrypt and decrypt data.

In public or asymmetric key cryptography, a key is used to encrypt the data whereas another key is used to decrypt the data.

Java provides a number of services and tools for generating, storing, and exchanging cryptographic keys.

The `java.security.Key` interface has methods for algorithm and type independent representation of keys.

Generating Keys:

- A key object can be instantiated by generating it within the program or through another program which makes use of the underlying bit stream.
- The `generateKey()` method is invoked on `KeyGenerator` object and is used to generate the secret key.
- The `getInstance()` factory method accepts the algorithm name and the provider name as arguments and is used to create an object of the `KeyGenerator` class.

Cryptographic Keys 2-7

- The following Code Snippet generates a secret key using DES algorithm:

Code Snippet:

```
...
// table to convert a nibble to a hex char.
static char[] hexChar = {
'0' , '1' , '2' , '3' ,
'4' , '5' , '6' , '7' ,
'8' , '9' , 'a' , 'b' ,
'c' , 'd' , 'e' , 'f'};
StringBuffer strCode = new StringBuffer();
String algorithm = keyObj.getAlgorithm();
String fmtKey = keyObj.getFormat();
byte[] encoded = keyObj.getEncoded();
strCode.append("Key[algorithm=" + algorithm + ", format=" +
fmtKey
+ ", bytes=" + encoded.length + "]\n");
if (fmtKey.equalsIgnoreCase("RAW")){
strCode.append("Key Material (in integer):: ");
```

Cryptographic Keys 3-7

```
strCode.append(convert(encoded));
strCode.append("\nKey Material (in hex):: ");
strCode.append(toHexString(encoded));
}
return strCode.toString();
}
public static int convert(byte[] v) {
return (0xff & v[0]) |
(0xff & v[1]) << 8 |
(0xff & v[2]) << 16 |
v[3] << 24;
}
public static String toHexString ( byte[] b ) {
StringBuffer sb = new StringBuffer( b.length * 2 );
for ( int i=0; i<b.length; i++ )
{
// look up high nibble char
sb.append( hexChar [( b[i] & 0xf0 ) >>> 4] );
// look up low nibble char
sb.append( hexChar [b[i] & 0x0f] );
}
return sb.toString();
```

Cryptographic Keys 4-7

```
public static void main(String[] unused) throws Exception {
KeyGenerator keyObj = KeyGenerator.getInstance("DES");
keyObj.init(56); // 56 is the key size. Fixed for DES
SecretKey key = keyObj.generateKey();
System.out.println("Generated Key:: " + formatKey(key));
}
...
```

- The following Code Snippet shows the usage of `generateKeyPair()` method:

Code Snippet:

```
import java.security.KeyPairGenerator;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.Key;
public class KeyPairTest {
private static String formatKey(Key keyObj){
```

Cryptographic Keys 5-7

```
StringBuffer strObj = new StringBuffer();
String algorithm = keyObj.getAlgorithm();
String fmtObj = keyObj.getFormat();
byte[] encoded = keyObj.getEncoded();
strObj.append("Key[algorithm=" + algorithm + ", format=" +
  fmtObj + ", bytes=" + encoded.length + "]\n");
if (fmtObj.equalsIgnoreCase("RAW")) {
strObj.append("Key Material (in hex):: ");
strObj.append(convert(encoded));
strObj.append("\nKey Material (in hex):: ");
strObj.append(toHexString(encoded));
}
return strObj.toString();
}
public static int convert(byte[] v) {
return (0xff & v[0]) | (0xff & v[1]) << 8 | (0xff & v[2])
  << 16 | v[3] << 24;
}
```

Cryptographic Keys 6-7

```
public static String toHexString ( byte[] b ) {
StringBuffer sb = new StringBuffer( b.length * 2 );
for ( int i=0; i<b.length; i++ ) {
// look up high nibble char
sb.append( hexChar [( b[i] & 0xf0 ) >>> 4] );
// look up low nibble char
sb.append( hexChar [b[i] & 0x0f] );
}
return sb.toString();
}
static char[] hexChar = {
'0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' , 'a' ,
   'b' , 'c' , 'd' , 'e' , 'f'};
public static void main(String[] unused) throws Exception
KeyPairGenerator kpgObj = KeyPairGenerator.getInstance("DSA");
kpgObj.initialize(512); // 512 is the key size.
KeyPair kpObj = kpgObj.generateKeyPair();
PublicKey pubKeyObj = kpObj.getPublic();
PrivateKey prvKeyObj = kpObj.getPrivate();
```

Cryptographic Keys 7-7

```
System.out.println("Generated Public Key is: " +
   formatKey(pubKeyObj));
System.out.println("Generated Private Key is: " +
   formatKey(prvKeyObj)); } }
```

Storing Keys

The `java.security.KeyStore` class is used to store the key. An object of the `KeyStore` class stores the key and certificate details in an in-memory table.

Signature Class 1-2

- The `Signature` class has functions for digital signature algorithm which is used for integrity assurance and authentication of the digital data.
- Following table shows some methods in `Signature` class:

Method	Description
`Signature getInstance (String algo)`	This method creates a `Signature` object with the specified algorithm.
`Signature getInstance (String algo, String prov)`	This method creates a `Signature` object with the specified algorithm and provider. The provider should have the algorithm.
`void initVerify (Certificate cert)`	The method is used to initialize the object for verification with the public key present in the certificate.
`void initSign (PrivateKey pk)`	The method is used to initialize the object for verification.
`void update(byte b)`	The method is used to update the data by a byte which has to be signed or verified.
`byte [] sign()`	The method returns the signature bytes of all the updated data.

Signature Class 2-2

Method	Description
`boolean verify(byte [] sign)`	The method is used to verify the signature which has been passed-in.
`String getAlgorithm()`	The method is used to return the name of the algorithm used for the specified `Signature` object.
`String toString()`	The method returns the `Signature` object in a String format.
`setParameter(AlgorithmParameterSpec param)`	The method is used to initialize the `Signature` engine with the specified set of parameters.
`AlgorithmParameter getParameters()`	The method retrieves the parameters used with the specified `Signature` object.
`Provider getProvider()`	The method returns the Provider for this `Signature` object.

Digitally Signing the Data 1-4

- The following steps are performed to sign data and generate a digital signature for the signed data:
  - Generate a `Signature` object
    - An instance of the `Signature` class is created to generate or verify signatures using the DSA algorithm.
    - The following Code Snippet demonstrates creating a `Signature` object:

Code Snippet:

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

  - Initialize the `Signature` object
    - The instance of the `Signature` class must be initialized before it can be used to sign or verify a signature.
    - The following Code Snippet shows how to initialize the `Signature` object with the private key object **`privkey`**:

Code Snippet:

```
dsa.initPriv(privKey);
```

Digitally Signing the Data 2-4

- Supply the data to be signed to the `Signature` object
  - The data to be signed is read into a byte array of length 1024 bits.
  - The following Code Snippet supplies data from a text file named **data.txt** to the `Signature` object **`dsa`**:

Code Snippet:

```
FileInputStream input = new FileInputStream(data.txt);
BufferedInputStream inputBuf = new
BufferedInputStream(input);
byte[] dataBuffer = new byte[1024];
int len;
while (inputBuf.available() != 0)
{
len = inputBuf.read(dataBuffer);
dsa.update(databuffer, 0, len);
}
```

Digitally Signing the Data 3-4

- **Generate the signature**
  - When all of the data has been supplied to the `Signature` object, a digital signature can be generated. The following Code Snippet demonstrates the same:

Code Snippet:

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

Saving the Digital Signature and Public Key

The digital signature bytes and the public key bytes need to be saved in two different files to be transmitted to the recipient of a message. Following are the steps:

- Write the signature bytes to a file.
- Obtain the public key bytes from the `PublicKey` object by invoking the `getEncoded()` method.
- Write the public key bytes to a file.

Digitally Signing the Data 4-4

- Following Code Snippet shows how to save the byte array to a file named **DataSig**:

Code Snippet:

```
FileOutputStream sigfos = new FileOutputStream("DataSig");
sigfos.write(dataSignature);
sigfos.close();
```

- The following Code Snippet generates the byte array for the public key and then stores it in a file named **SenderPk**:

Code Snippet:

```
byte[] key=pubKey.getEncoded();
FileOutputStream keyfos= new
FileOutputStream("SenderPk");keyfos. write(key);
keyfos.close();
```

Verifying the Signature 1-3

- Following are the steps to verify a digital signature:

Create a `Signature` instance using the same signature algorithm as that used to generate the signature.

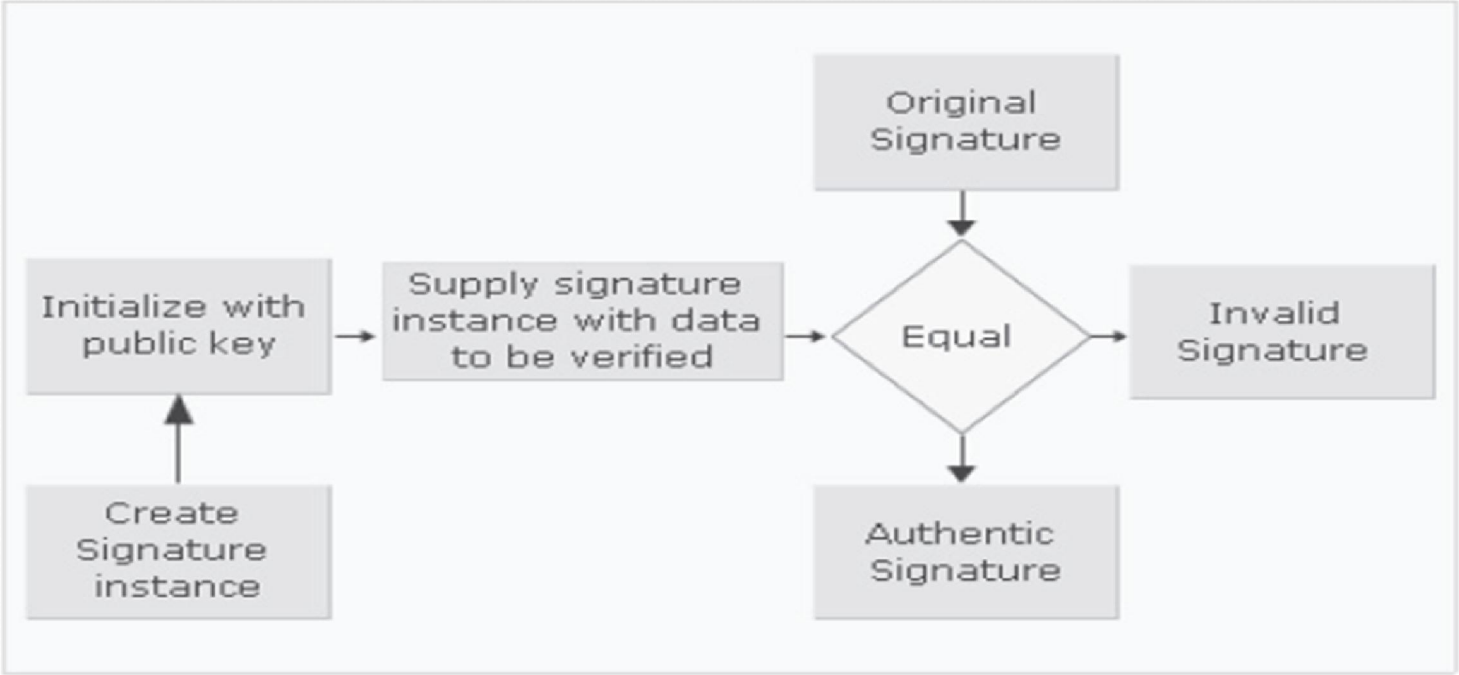
Initialize the `Signature` instance with the public key.

Supply the `Signature` object with the data to be verified by reading one byte array at a time. Supply this to the `Signature` object by calling the update method.

Invoke the `verify()` method on the newly created `Signature` object to compare the two signatures.

Verifying the Signature 2-3

- Following figure shows verifying a signature:



- The following Code Snippet creates a new `Signature` instance using the same algorithm that was used to generate the signature originally:

Code Snippet:

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

Verifying the Signature 3-3

- The `Signature` instance is initialized with the public key as follows:
  - `sig.initVerify(pubKey);`

- The data to be verified is supplied to the `Signature` object through a byte array as shown in the following Code Snippet:

Code Snippet:

```
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();
```

Summary

- Digital signatures are used to digitally sign messages or objects to identify their creators.
- A message with a digital signature encrypts the data thus, providing integrity and authenticity of data.
- Digital certificates positively help to verify the authenticity of the sender.
- A key is a kind of a password used in cryptography calculations.
- A keystore stores public and private keys, users certificate, and third-party public key certificates.
- A digital certificate must be first imported as a trusted certificate and then the signature has to be verified to authenticate the sender.
- Public and private keys are generated from the KeyPairGenerator class.
- Digital signatures are generated by using an instance of the Signature class and a private key.
- A private key is also used to sign a digital certificate.