Global Information Assurance Certification Paper

Muath Al-Khalaf
February 18, 2004
GIAC Security Essentials Certification (GSEC)
Practical Assignment Version 1.4 b, Option 1

Introduction to Java Cryptography

Abstract

In this paper I am going to introduce cryptography libraries in Java. First I will explain general cryptography concepts from a programmer's point of view. Then I will give a general overview of the architecture of Java cryptographic libraries. After that I will go through well-known cryptographic operations and show how to implement them using Java programs. Finally I will discuss some cryptographic scenarios that a programmer may deal with in his/her programs. This paper is the gateway for any programmer who wants to enter the cryptography world of Java.

Introduction

People are very concerned about information security and how to protect their systems and PCs. There are several kinds of attacks on networks and computers exploiting different types of vulnerabilities. Reasons behind these attacks differ ranging from denial of service only to stealing the organization's or people important data such as credit cards' numbers. Security engineers and experts use a mixture of hardware, software, procedures and practices in a layered approach to build security infrastructure to defend against these attacks in what is known as "Defense in Depth". In the heart of the infrastructure is cryptography but "Cryptography by itself is fairly useless" [1] so it is always used in conjunction with other security tools and practices.

"Cryptography is the art and science of encryption" [1]. Encryption was the start which then leads to other things including digital signatures and authentication. The two major contexts in which cryptography is used are during data exchange or storage. The main goals behind using cryptography in these two contexts are confidentiality, authentication and data integrity [7].

- **Confidentiality** means to hide your important and secret data from unauthorized entities either during exchanging it or when you store it. This is achieved by encryption.
- **Data integrity** means to make sure that data has not been changed by unauthorized entities after transfer or retrieval from storage. This is achieved by digital signature.
- **Authentication** is to assure that data originated from authorized entities when you receive it. This is also achieved by a digital signature.

"Cryptography is an extremely varied field" [1] in which you will find mathematicians, electrical engineers, system analysts and designers, programmers, politicians, ...etc. In this paper we will discuss cryptography

from a programmer point of view. In particular we will discuss Java Cryptography and how to use Java Cryptography Architecture (JCA) to develop cryptography related modules in your programs. We will not discuss in details PKI related topics such as keys and certificates management in Java programs. Also, when we discuss cryptographic operations and methods used to accomplish them, we will only give a subset of methods and you can refer to "Java 2 Standard Edition (J2SE) API Documentation" [6] for more details.

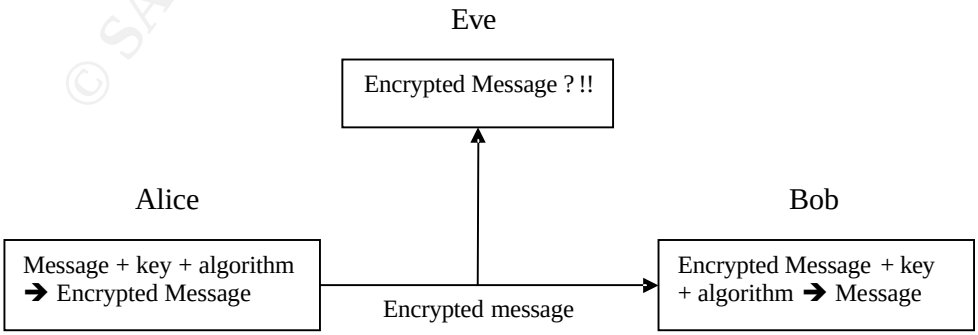
Before discussing Java cryptography we will give a theoretical overview of cryptographic primitives that you could use as a programmer in your programs regardless of the chosen programming language.

Cryptographic Primitives

When you deal with cryptography in any program you will find yourself using a mixture of four cryptographic primitives which are symmetric encryption, asymmetric encryption, message digest and digital signature [7]. These primitives are combination of mathematical cryptographic algorithms and practices. All cryptographic operations are centralized around these primitives so it is very important to understand them.

Symmetric Encryption

Symmetric encryption is a transformation of data bits from an understandable useful plain format into a non understandable and encrypted format. This is done by a combination of mathematical symmetric encryption algorithm and a secret key. You can then use this secret key along with the algorithm to decrypt the data i.e. transform it to the original plain format. Because of the usage of the same key to encrypt and decrypt data it is sometimes called shared or secret key encryption. Suppose that Alice wants to send a message to Bob without letting Eve (who is listening to them) read it. Then Alice will use a secret shared key which they have in common, encrypt this message using symmetric encryption and send it to Bob. Bob then will get the encrypted message from Alice, decrypt the message using the shared key and read it. If Eve intercept the transmission and get the message she can not read it because she does not have the shared key.



Symmetric encryption is very fast. There are many algorithms used here such as RC4, Rijndael and TripleDES which uses different key sizes. They are called block ciphers because they encrypt any amount of data block by block. This means that you can divide a huge amount of data into a stream of chained blocks and then encrypt them one block at a time. The most well-known chaining mode is cipher block chaining (CBC) in which you feed the algorithm data along with an initialization vector (IV) – number of bits fetched randomly - to get more randomization [1] [2] [7]. IV will be used with first block and then the second block will use an IV from the output of the first block. You will get the same size output as the input block after encrypting it so there is no overhead in storage or bandwidth. The only problem with the symmetric encryption is the exchange of the shared key. How will Alice and Bob exchange the shared secret key?

Asymmetric Encryption

The major problem of symmetric encryption is sharing the key. This problem is solved by asymmetric encryption. Asymmetric encryption does the same thing as symmetric encryption in which it transforms bits of data from plain format to non-useful encrypted format and then returns them into useful plain one. But the difference here is that it uses two - mathematically linked - keys for encryption and decryption such that when you encrypt a message with one of the keys you can only decrypt it with the other one. These two keys are called public key and private key. So when you encrypt the message with public key you can only decrypt it with private and vice versa. According to this, each entity should have two keys; public and private and one should only distribute his/her public key and keeps his/her private key secret.

Now lets go back again to Alice and Bob. Alice can create a random secret key for each message or set of messages (connection session), encrypt this key with Bob's public key and send it to Bob. Bob, then, decrypts the message with his private key and obtains the secret key to decrypt the future messages. This is called session key exchange and it is the way used today to exchange secret keys through Internet. You may wonder why not to use asymmetric encryption directly and encrypt the whole message with the Bob's public key. This is due to the fact that asymmetric encryption is too slow compared to symmetric and is not designed to be used with large amounts of data. So we encrypt the message with the secret key and then exchange the secret key using asymmetric encryption.

Certificates and CAs

Another problem arises now which is how Alice makes sure that what she has is really Bob's public key? What if Eve sent faked public key to Alice and claimed that it is Bob's? This is solved by digital certificates. Bob's certificate is his public key along with some additional information about him in a file which is digitally signed by a third party (called a Certificate Authority, CA) to be for him. A CA is a well known organization (such as VeriSign or Microsoft) which has its public key distributed to all people by trusted means in a self signed certificate called root certificate or a certificate signed by a higher CA

in the CA chain (hierarchy). Some of the trusted means are web browsers and operating systems which prepackage these CA certificates. So Bob will send his certificate to Alice including his public key which then she could use to encrypt the session key.

Message Digest

There is special one way mathematical function which "takes as input an arbitrarily long string of bits (or bytes) and produces a fixed-size result" [1]. If you change any bit of input then all the output will be changed (but the size will remain fixed). This function is called a hash function or a message digest function and the output is called hash value or digest. SHA-1 and MD5 are the two most well-known hash functions. MD5 for example outputs a 128 bit string as a digest. Hash functions are used mainly for data integrity either alone or with digital signature (along with asymmetric encryption).

Digital Signature

Digital signature is a method used to assure the identity of the entity that sends data (authentication). The idea here is to use the sender private key to encrypt data so that no one can decrypt this data without the sender public key. Suppose that Alice wants to send some data to Bob so that no one including Eve can change this data. When she encrypts this data using her private key Bob will make sure that it is from Alice by using her public key to decrypt it. If Eve intercepts this data, decrypts it and changes it, she can not encrypt it again with Alice private key. If she changes the encrypted version then Bob will not be able to decrypt it.

In real world, when we want to sign a message we do not encrypt them all using private key because asymmetric encryption is too slow. Instead, we hash the message first and then encrypt the digest (which is very small compared to the original message). After the receiver gets the message, he hashes it, decrypts the original digest and compares the two digests.

Java Cryptography Architecture

Java Cryptography Architecture (JCA) is a cryptography framework inside Java Security API (java.security package and its sub packages) which allows Java programmers to implement and use cryptographic operations inside their programs [3]. It was first introduced with in JDK 1.1 and implements only massage digests and digital signatures. Due to some US export laws, encryption, key exchange and some other security operations were implemented separately in another API called Java Cryptography Extension (JCE). JCE was not packaged with JDK prior to 1.4 but now it comes by default with 1.4.x versions. In the following sections I will explain both JCA and JCE and how to use them in your programs.

Security Provider Infrastructure

When you implement cryptography in your programs you will find your self doing it as "a set of consequence cryptographic operations". Cryptographic operations include creating random secret keys, reading certificates and extracting keys from them, producing message digest, producing digital signature, encrypting a message, ...etc. when I say that they are done sequentially I mean that for example you first create a secret key (an operation) and then use it to encrypt a message (another operation). Also, you may first read certificate and extract the public key from it (an operation), hash the received message (second operation), decrypt the digital signature and get the digest (third operation) and then compare the two digests (final one). In JCA these operations are done using "cryptographic engines" such that each cryptographic engine is used to do a cryptographic operation-.

Cryptographic engines

Cryptographic engines are abstract classes which come by default with JCA API and represent the programmer's interface to cryptographic operations [2]. For example, `java.security.MessageDigest` is the cryptographic engine that you could use to implement message digests in JCA. Now you may ask how to use specific algorithm in an engine (to do an operation)? For example, how to use SHA algorithm or MD5 algorithm to do message digest? This is very easy; you only have to mention the name of the cryptographic algorithm (which you want to use) when you create the engine class. For example,

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

creates a message digest engine which uses SHA algorithm implementation. (Notice that you can not instantiate an engine class directly but instead you have to use the `getInstance()` method.) The beauty here is that designers of JCA have separated the conceptual cryptographic operations from their corresponding algorithmic implementation form different vendors to let you use any algorithm implementation from any vendor (either the default form Sun or other third parties such as IBM) through only one interface which is engine class.

Cryptographic Service Provider

This separation is done using the concept of Cryptographic Service Provider This term refers to a package (or a set of packages) that supply a concrete implementation for all (or a set) of cryptographic engines (operations) [4]. When you use any cryptographic engine you really use an implementation of it from one of the *registered* providers in your Java VM. This gives you a choice among different providers according to your business requirements (budget, HW fast encryption ...etc) **without changing your code** (we will show this). Sun's version of the Java runtime environment (VM) comes standard with a default provider, named "SUN" which implemented many of the engines. There are also other providers out there such as IBM Common Cryptographic Architecture (CCA) which uses a hardware implementation [8]

and open source free provider Bouncy Castle [5] which we will use in this paper (because it is rich with features and free to be used by any one).

Choosing provider implementation

At runtime, the Java VM has a registered set of cryptographic service providers which you can use along with engine classes. When you use a specific algorithm in an engine class (such as SHA for MessageDigest) the implementation for this algorithm from the default registered provider will be used. If this provider does not have an implementation then the other registered providers will be asked, in order according to their priority, if they have implementation (we will see how to set priority later). The first one found to have implementation for this algorithm will be used to accomplish the operation. If no registered provider implements this algorithm then a `NoSuchAlgorithmException` is raised. You, instead, can directly mention the provider to be used for this cryptographic operation along with the algorithm in your code. For example,

```
MessageDigest md = MessageDigest.getInstance("SHA", "BC");
```

means that you want the SHA algorithm implemented by Bouncy Castle provider to be used for message digest.

Adding a new security provider

You can specify the security providers that you want to be registered with your Java VM and their priorities (preferences) through a configuration file called *$JAVA_HOME/jre/lib/security/java.security*. In this file you will find lines containing providers to be registered at runtime which are similar to the following:

```
security.provider.priority_number=provider_master_class_name
```

In my VM, configuration file has the following lines (default with J2SE 1.4):

```
#
#List of providers and their preference orders (see above :(
#
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
```

Each line of these tells the VM to register a provider with the given priority– the number - and the master class. For example, first line means to register SUN security provider (its master class is `sun.security.provider.Sun`) and give it the highest priority (preference). When you have a new provider and want it to be registered just do two steps:

1- Place the package (jar or zip file) you get from vendor inside your CLASSPATH.
2- Insert a line in the previous file `java.security` which has the priority you want for the provider and the master class name. The master class name could be found at documentation.

For example, we will use Bouncy Castle as our provider for examples in this paper so we will register it as following:

1- Download the provider package from www.bouncycastle.org which is named *bcprov-jdk14-122.jar* and add it to your CLASSPATH (Bouncy Castle has additional classes in other jar file which are not part of JCA so we will not discuss them).
2- Insert the following line in `java.security`

```
security.provider.<n>=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Here <n> is the priority and I will give it 1 which is the highest priority. Remember to change other priorities of providers.

From now on, all engine classes in your programs will use the implementation from the Bouncy Castle provider as first choice whenever it is possible.

Security Class

In the previous discussion we mentioned how to add new security provider to be used in your programs, how to specify priorities (preferences) for security providers and how to specify a specific provider during runtime to be used for a security operation (engine). The remaining question is how to register a security provider in the VM at runtime either from `java.security` file or programmatically? (Before we answer this question remember that what we did with `java.security` file is only mentioning the providers to be registered but not registering them.) A class called `java.security.Security` is the one responsible for registering and managing installed providers at runtime (and managing other security properties for your program). When you run your program, Security class will load all the available providers from `java.security` file and register them with the VM. This class is also responsible for choosing the provider at runtime to do a cryptographic operation when you call `getInstance(`*`String algorithm`*`, String` *`provider`*`)` method of the engine class. For example, suppose you have two providers Provider_1 and Provider_2 such that:

1- Provider_1 implements SHA and MD5 for message digest.
2- Provider_2 implements MD5 only but has a higher priority than Provider_1.

When you ask for a message digest with MD5 the engine class will ask the Security class for this and Security class will answer with the implementation from Provider_2. If you ask for SHA algorithm then Security class will return Provider_1 implementation because Provider_2 does not have one. You can enforce using MD5 from provider_1 by telling this to the Security class through `getInstance` method. Finally, you can use Security class to register a provider at runtime even if it is not written in `java.security` file using `addProvider(Provider provider);` method. For example,

```
Java.security. Security.addProvider(new BouncyCastlePr ovider());
```

will add the Bouncy Castle provider at runtime even if we did not write it in `java.security` file. Note that these methods for adding or removing

providers in security class can only be executed by a trusted program. A "trusted program" is either

- a local application not running under a security manager, or
- an applet or application with permission to execute the specified method.

Configuring these permissions is done using Security Manager and policy files and they are part of the Java Security in general which is out of the scope of this paper.

Putting it all together

Now let us see how to prepare for encryption operations you want to do.

1- Choose the cryptography service provider you want to use in your program.
2- Register this provider in your program's VM either by adding it to `java.security` file or at runtime using security class.
3- Import all JCA and JCE packages you will use. These include many packages but the most important are `java.security.*` and `javax.crypto.*`. You may also import some additional packages such as utilities from providers.
4- Decide which operation you will use and use the corresponding engine class and algorithm using *`EngineClass`*`.getInstance(`*`String algorithm`*`, String `*`provider`*`)` method of engine class. (Remember that you can not create an engine object directly but instead you will instantiate one using this method.)

The remaining steps **depend on the specific engine class** and how does it work which we will discuss in the following chapters.

What is happening behind the sense?

Up to now we have completed the discussion of Security Provider Infrastructure that you need to know to start cryptography in Java and you could proceed to next chapters. But you may wonder how this works. How could an engine class have many implementations for algorithms and how does it tell the security class about them? How to instantiate an engine class in my program? All of this is achieved through the interaction between different types of classes including Service Provider Interface (SPI) classes, Engine classes, Security class and Provider class. We will first explain the SPI and then proceed to the process of instantiating an engine class.

Service Provider Interface

Each engine class has a corresponding abstract class called Service Provider Interface (SPI). This class allows the separation between engine classes and their implementations from different vendors. Any provider wants to implement a particular engine class needs to implement the engine's SPI. You may find two different classes implementing SHA algorithm for message digest engine from two different providers. This is achieved by allowing each class to implement SPI of message digest engine class. We will not discuss how to implement an SPI class because this is out of scope of this document (developer does not need to do it but a cryptographic provider needs to).

Instantiating an Engine Class

Engine class is the interface to the programmer to cryptographic operations. You can not declare an engine class directly in you program but instead you have to use `getInstance()` method. When you use it you will give the algorithm to be used to accomplish the operation. After that the engine class name along with the algorithm used will be send as a string by engine class to Security class to do the real instantiation. For example, `MessageDigest.SHA` will be sent to security class to ask for a message digest with SHA algorithm.

Security class is the one responsible for registering providers with the VM and instantiating objects from engine classes. When security class receives the message from engine class, it will search the array of registered providers it manages to find an implementation for the requested algorithm. It will ask them one by one according to their priority if they have an implementation for this engine class using this algorithm. In fact it will send the string from the engine class (such as `MessageDigest.SHA`) to the provider class to see if there is an implementation.

Provider class is the glue between the engine classes and their implementations (of their SPIs). Each vendor implements his own provider subclass (such as `XYZProvider extends Provider`) and indexes all of his implementations of algorithms of engine classes in this class. Provider class extends properties class and it contains a table of engine-algorithm pairs along with their implementation classes. For example,

```
MessageDigest.SHA → XYZProvider.MessageDigets.SHA
```

says that the class that implements message digest engine using SHA algorithm in this provider is `XYZProvider.MessageDigets.SHA`.
So, when provider finds an implementation it will send the name of the class that implements it. Then security class will instantiate an object from this class and give it back to the programmer as an engine class instance. For example, when security class sends `MessageDigest.SHA` to XYZProvider it will answer with `XYZProvider.MessageDigets.SHA` class. Then Security will create an instance of this class and return it to programmer through engine class.

Creating a Message Digest

The first engine class we will discuss is the message digest. The operations we want to do here is either to create a message digest or verify a given digest. Before we do any of the operations we first have to prepare as following:

1- Import the message digest engine class.
```
import java.security.MessageDigest
```

2- Create an instance of this engine class with the algorithm you want to use using

```
public static MessageDigest getInstance(String algorithm)   or
public   static   MessageDigest   getInstance(String   algorithm,   String
provider)
```

If `algorithm` was not found then `NoSuchAlgorithmException` is raised. If `provider` was not found then `NoSuchProviderException` is raised.

Now if we want to create a message digest then we do the following:

1- Input the message that we want to hash using

```
public void update(byte input)
public void update(byte[] input)
public void update(byte[] input, in offset, int length)
```

The first one input a single byte, the second an array of bytes and the third one a subset of an array. Consecutive calls to these methods append data to the internal one.

2- Get the digest of the message using

```
public byte[] digest()
public byte[] digest(byte[] input)
```

the second one allows you optionally to load the message you want to hash.

Here a code snippiest which creates a message digest.

```
package GSECExamples;
import java.security.MessageDigest;
public class CreateMessageDigest {
        public static void main(String args[]) {
                try {
                        String message = new String("This is a sample message");
                        MessageDigest md = MessageDigest.getInstance("SHA");
                        md.update(message.getBytes());
                        System.out.println(md.digest());
                } catch (Exception e) {
                        System.out.println(e);
                }
        }
}
```

Normally when verifying a message digest you will receive the original message along with the computed digest. If you want to verify the message digest you do the following:

1- Create the digest of the received message that you want to verify using the same algorithm which created the first as explained previously.

2- Compare the two digests, the received one and the created one using

```
public static boolean isEqual(byte[] digestA, byte[] digestB);
```

3- If they are equal then the message has not been changed else you have to receive this message again.

The message digest engine has other methods for additional work and you can get the documentation of the full API from JCA documentation and javadoc.

Keys and Certificates Creation and Usage

The remaining three other cryptographic primitives (which are symmetric encryption, asymmetric encryption and digital signature) depend on the presence of keys and certificate before they can be used. So, before we discuss their usage in JCA and JCE, we have first to discuss how to create keys and certificates, how to manage them in our programs and how to prepare them for encryption or signing. There are two operations which developers need when dealing with keys and certificates:

1- Creating new keys and certificates from scratch.
2- Importing or exporting keys and certificates from and to different types of storage (special type files or special key stores).

When you deal with keys, two concepts should be understood well; the difference between creating new keys and importing or exporting ones at runtime and the difference between the internal representation of keys which is used by code implementation and external portable one which is used to store and exchange keys between different systems.

Internal and external representation of keys

There are two types of key classes:

1- Internal representation classes (object classes) which are used with encryption and digital signature engines.
2- External ones (specification classes) which are used to read or write keys from and to storage and then transform them into internal representation.

`java.security.Key` interface is the top-level interface for all internal representation classes (key objects). When you want to use a key with an encryption or digital signature engine class you have to use a key object or a subclass of it. These types of keys are also called opaque keys because they are not transparent and you can not get material that constitutes the key. These keys have three characteristics which are:

1- The name of the algorithm of this key (such as RSA, DSA). This is retrieved using `public String getAlgorithm()` method.
2- An external encoded form which is used to represent the key outside Java VM such as PKCS#8 encoding. This is retrieved using `public byte[] getEncoded()`
3- The name of the format for external encoding which is retrieved using `public String getFormat()`

There are subclasses or sub interfaces for Key interface. Some of them are for asymmetric keys such as `PublicKey`, `PrivateKey`, and others are for symmetric keys such as `SecretKey`.

`Java.security.spec.KaySpec` interface is the root for all classes used to store keys from external storage. It is called Key Specification class and it has two types of subclasses `EncodedKeySpec` class and `AlgorithmParameterSpec` interface. Some keys are encoded using some of the encoding standards such as PEM and PKCS#8. With these we use `EncodedKeySpec` class or its subclasses. Some other keys may be stored using the parameters which they consists of such as public key which is a result of multiplying some mathematical numbers and these numbers are stored. With these keys we use `AlgorithmParameterSpec` interface or its subinterfaces.

Creating a New Key and loading it from an External Storage

When you want to use a key in an encryption or digital signature engine you have to either create a new `Key` object from scratch or load one from storage using `KeySpec` and then transform it into `Key` object. To create a new key from scratch you have to use a generator engine class. Generator engine classes are used only with `Key` objects. There are two generators:

1- `java.security.KeyPairGenerator` engine class which is responsible for creating two asymmetric keys public and private.
2- `javax.crypto.KeyGenerator` engine class which is responsible for creating a secret random key.

To load a key from storage you have to use `KeySpec` class. Then you have to transform the `KeySpec` (which you can not use with engine classes) into a `key` object. To do so you have to use one of two classes; `java.security.KeyFactory` for asymmetric (public and private) keys or `javax.crypto.SecretKeyFactory` for secret keys.

Random Numbers

Before generating a key we have to use a random number generator to generate a random number to be used when generating keys. The class responsible for this is `java.security.SecureRandom`. This class works as a wrapper for the generator either hardware or a pseudo one depending on provider.

Generating asymmetric keys

To generate two asymmetric keys you have to use `java.security.KeyPairGenerator` class. This class will give a `java.security.KeyPair` object which is a simple data structure used to store public and private keys. The reason behind creating both keys at once is that public and private keys are mathematically related (they are created by applying different mathematical operations on a shared parameters). You instantiate this class using `getInstance()` like other engines. Two important methods are used here which are:

1-
```
public void initialize(int strength)
public abstract void initialize(int strength, SecureRandom random)
```

this method initializes the generator with the `strength` which is number of bits that are used as input to the engine to create keys and `random` which is an optional random number. Strength varies among algorithms. For example, RSA strength could be 512, 1024, 2048 …etc and this depends on the provider for RSA. Sometimes you may not create key objects from scratch but instead initialize a generator from storage through key specification using

```
public void initialize(AlgorithmParameterSpec params)
public void initialize(AlgorithmParameterSpec params, SecureRandom
random)
```

We will talk about these specification classes later.

```
2- public abstract KeyPair generateKeyPair()
```

This method will generate a `KeyPair` object containing the two keys (public and private) parameters.

Now, using the previous two methods it is very easy to create two asymmetric keys from scratch (remember to import the corresponding classes). For example,

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);
KeyPair kp = kpg.generateKeyPair();
```

will generate a pair of public and private keys stored into `KeyPair` object. Using `KeyPair` object you can use the only two methods it provides to get the public and private keys which are:

```
1- public PublicKey getPublic()
2- public PrivateKey getPrivate()
```

Now to complete the previous example we extract the two keys as following

```
PublicKey PuK = kp.getPublic()
PrivateKey Prk = kp.getPrivate()
```

Generating symmetric keys

To generate a symmetric key you have to use `javax.crypto.KeyGenerator` class. This class is very similar to the previous one except that it outputs one key only which is a `javax.crypto.SecretKey`. You instantiate this class using `getInstance()` like other engines and feed it the block cipher you want a key for (like AES). Two important methods are used here which are:

```
1- public final void init (int strength)
   public final void init (int strength, SecureRandom random)
```

this method initializes the generator with the `strength` which is number of bits of the created key and `random` which is an optional random number (if not specified it will be created internally by the VM). Strength varies among algorithms and providers. For example, AES strength could be

0..256 using Bouncy Castle provider with 192 as a default length [5]. Sometimes you may not create key objects from scratch but instead initialize a generator from storage through key specification using

```
public final void init(AlgorithmParameterSpec params)
public final void init(AlgorithmParameterSpec params, SecureRandom
random)
```

but not all algorithms keys could use this (DES key for example).

```
2- public final SecretKey generateKey()
```

This method will generate a `SecretKey` object.

Now, using the previous two methods it is very easy to create a symmetric key from scratch (remember to import the corresponding classes). For example,

```
KeyGenerator kg = KeyGenerator.getInstance("AES","BC");
kg.init(256);
SecretKey sk = kg.generateKey();
```

will generate a 256 secret key for AES block cipher using Bouncy Castle provider (Note that Sun does not have an implementation for AES).

Importing and Exporting Keys from and to Storage

In the previous section we have seen how to create a key from scratch. Now we want to export and import a key. To export an asymmetric key we have two choices:

1- Use the `getEncoded()` method of the key object. For example, for the public key created in the previous example `PuK` we use `PuK.getEncoded()`. This method will output an X509 encoding of the public key in a byte array. We then could store this array in a file using `FileOutputStream`. Here is the code for this

```
//We have created the public key PuK using KeyPairGenerator
FileOutputStream fos = new FileOutputStream("c:/GSECSamples/key");
fos.write(PuK.getEncoded());
```

2- Transform a `Key` object into a `KeySpec` object which is an exportable format using `KeyFactory` engine class. The method you will use from the `KeyFactory` class is

```
public final KeySpec getKeySpec(Key key, Class keySpec)
```

For example, for the public key from the previous example

```
//This is the class pf the KeySpec that we will use to export the key
Class spec = Class.forName("java.security.spec.RSAPublicKeySpec");
KeyFactory kf = KeyFactory.getInstance("RSA");
//Change the Key object into its correspondence KeySpec
RSAPublicKeySpec PukSpec = (RSAPublicKeySpec)kf.getKeySpec(PuK, spec);
//Now PukSpec could be exported using its components (parameters)
```

To import an asymmetric key into your program you have only one choice which is to use `KeySpec` along with key factory (you may find some special

utilities to import keys but we here are talking about standard way). The methods from `KeyFactory` engine class to be used here are

```
public final PublicKey generatePublic(KeySpec ks)
public final PrivateKey generatePrivate(KeySpec ks)
```

For example,

```
//Reading key from storage
FileInputStream fis = new FileInputStream("c:/GSECSamples/key");
byte[] key = new byte[400];
fis.read(key);
// Loading key into KeySpec object
X509EncodedKeySpec keyspec = new X509EncodedKeySpec(key);
//Transforming KeySpec into Key object
KeyFactory kf = KeyFactory.getInstance("RSA");
PublicKey PuK = kf.generatePublic(keyspec);
```

Importing and exporting symmetric keys uses `SecretKeyFactory` with the same concept as the asymmetric. But it is rare to export or import symmetric keys compared to asymmetric ones.

Working with certificates

As we mentioned before, certificates are used to store public keys along with the public key owner's information signed from a trusted third party. The most well-known and the standard certificate format which we will deal with is X.509 (which is downloaded normally from secure web sites and distributed by CAs). To use certificate in programs we will use `java.security.cert` package. This package contains three main abstract classes which are:

1- `java.security.cert.Certificate` to deal with certificate objects. We will deal with an X.509 implementation of it which is `java.security.cert.X509Certificate` which is used to store X.509 format certificates.
2- `java.security.cert.CRL` to deal with revocation lists of certificates. An implementation class for X.509 is `java.security.cert.X509CRL` which is used to deal with X.509 CRL. We will not discuss this issue as it is more PKI related and considered advanced one.
3- `java.security.cert.CertificateFactory` which an engine class to import certificates from storage.

Now using the first and last classes it is very easy to generate an X.509 certificate object from storage (using the same concepts as in importing keys from storage). You have to use

```
public final Certificate generateCertificate(InputStream inStream)
```

method from CertificateFactory engine class to generate a certificate. For example,

```
CertificateFactory cf = CertificateFactory.getInstance("x509");
FileInputStream fis = new FileInputStream("c:/GSECSamples/certificate");
X509Certificate x509cert = (X509Certificate) cf.generateCertificate(fis);
```

will generate an X.509 certificate object which you can use to mainly get the public key. You can do this using

```
public abstract  PublicKey getPublicKey()
```

method. Before using a certificate you have to check its validity which means you have to check its date and time to make sure that it has not expired and check its issuer to make sure that it is trusted. These topics are out of scope of this paper but you can go to documentation from Sun ti get detailed information.

Key Storage in Java

Most of times when you work a lot with asymmetric keys and certificates, especially in a PKI system, you need to use some kind of key and certificate storage designed especially for this purpose. It is almost encrypted and very secure and it has secure access rights. Java comes with a default key storage from Sun which is called key store and a tool called keytool to manipulate this key store. You can access this key store from your programs using `KeyStore` class and you can store or retrieve private keys and certificates into and from it. Key store is out of scope of this document and you can refer to its documentation to get more information.

What is next?

Now after discussing keys and certificates; their generation, importing and exporting, it is time to discuss when to use them and how. This includes discussion of digital signature, asymmetric and symmetric encryption and how to use keys and certificates there.

Digital Signature

Digital signature is used to authenticate the source of received data and check its integrity by first hashing the message and then encrypting the hash value with asymmetric encryption and then verifying it when received using the reverse. Digital signature operations are done using `java.security.Signature` engine class. This class does three things:

1- It hashes any message implicitly before signing it using a hashing function such as SHA1 or MD5 (without a need for `MessageDigest`).
2- It signs the entered message.
3- Or it verifies a digital signature for an entered message.

When you want to use the digital signature engine class you have to create a new instance of it (as previous engines) using `getInstance()`. This method should be given the hashing function along with the asymmetric algorithm to do the signature in the following format *`hashfunction`*`with`*`algorithm`*. For example, the following

```
Signature  s = Signature.getInstance("SHA1withRSA", "BC");
```

will give a digital signature engine instance from Bouncy Castle provider using SHA1 as a hash function and RSA as an encryption algorithm. After this you have two choices; either to sign a message or to verify the signature.

Signing a message

To sign a message you have first to prepare the Signature engine class. Preparation includes two methods:

1- `public final void initSign(PrivateKey privateKey)`

this method is used to initialize the signature engine class with the private key for encryption.

```
2- public final void update(byte  b)
   public final void update(byte[]  data)
   public final void update(byte[]  data,int off,int len)
```

this method is used to give the message to be signed to the signature class. It is the same as `update` method of the `MessagDigest`.
Now it is time to sign the message using

```
public final byte[] sign()
```

method which will return the signature as a byte array. The following example will illustrate the whole method of singing.

```
//creating a signature instance for SH A1 has function and RSA algorithm
Signature s = Signature.getInstance( "SHA1withRSA", "BC");
//use the previous created private key Prk for encryption
s.initSign(Prk);
//inserting message to be signed
s.update(new String("This is a message to be signed  and verified using SHA1
and RSA").getBytes());
//singing the message
byte[] signature = s.sign();
```

Verifying a Signature

To verify a signature of a message you also have to prepare the Signature engine class. Preparation includes two methods:

```
1- public final void initVerify(Certificate certificate)
   public final void initVerify( PublicKey publicKey)
```

this method is used to initialize the signature engine class with the public key for decryption either directly or from a certificate.

2- The same `update()` method used before to enter the message to be verified.

Now to verify the message signature we use

```
public final boolean verify(byte[]  signature)
```

method which will return a boolean indicating wether the signature matches the message or not. The following example will illustrate how to verify a message digital signature.

```
//use the previous created public key PuK for decryption
s.initVerify(PuK);
s.update(new String("This is a message to be signed and verified using SHA1
and RSA").getBytes());
//verifying the signature  from the previous example  which is an array of
bytes
if (s.verify(signature))
       System.out.println("Message has been authenticated" );
else
       System.out.println("SORRY, this message has been corrupted" );
```

Encrypting and Decrypting Data

Now it is time to talk about the final cryptographic operation in this paper and the most complex one which is encryption. The engine used here and all discussion could be applied to both symmetric and asymmetric encryption as well but with some small differences. For example, asymmetric algorithms such as RSA are only used with small amount of data (RSA block size must be less than modulus size [9]) such as symmetric keys and digests. Also, there is only one mode used with RSA which is ECB (see next section for more information about modes). Before talking about encryption engine and how to use we have to discuss some information necessary for initializing it which are modes and padding schemes.

Different modes and padding schemes

As we mentioned before, symmetric algorithms are called block cipher algorithms because they encrypt data in blocks one by one (i.e. 8 bytes at a time). There are many ways to divide the original data into blocks independent from the block cipher and they are called modes. Also, these blocks should be of the same size so we have to add some bits to some of the blocks to achieve this and this is called padding scheme. Some of the modes used are:

1- ECB, electronic cookbook mode. This is the simplest one and “it takes a simple block of data and encrypt the entire block at once” [2]. It does not try to add any additional data to the original and rearranging blocks will not affect it (RSA uses this mode because it has only one block of data). The standard block size is 8 bytes (64 bits). The problem arising here is lack of randomness especially when data has patterns such as text (binary data such as compressed may not have). When you encrypt two similar blocks with the same block cipher and key then it is easier for a cryptanalysis attack to find the key using some statistical calculations.

2- CBC, cipher block chaining. This mode will divide data into chain of blocks (8 bytes each) so that the next block will get some of input data from the output of the previous one (which is encrypted). This will help hiding patterns because even if two blocks have the same data they will have additional data different from the each other. For the first block we use a random data called initialization vector (IV). This mode can only operate on full blocks of data so you have to use a padding scheme with it [2].

One of the padding schemes used is PKCS5Padding which "ensures that the input data is padded to a multiple of 8 bytes" [2]. You can use NoPadding but you have to make sure that the input is a multiple of 8 byte or you will have an error.

Using Cipher Engine Class for Encryption and Decryption

The encryption and decryption engine used in JCE is `javax.crypto.Cipher` class. Using this class is similar to using Signature class; first you create an instance of an encryption algorithm, initialize it with key and other parameters and then do the encryption or decryption.

When you want to create an instance of Cipher engine you feed the `getInstance()` method with the algorithm name using the following format *algorithm/mode/padding* such as DES/CBC/PKCS5Padding [4]. Mode and padding here are not necessary and if you leave them the default ones will be used. To get the whole available algorithms, modes, padding schemes and their different combinations refer to the documentation of the specific JCE provider you are using (for example, [5] will give full reference information about Bouncy Castle provider).

After that you initialize the engine with the type of operation (either encryption or decryption), key, and algorithm parameters using the following method [6]

```
public void init(int opmode, Key key)
public void init(int opmode, Certificate certificate)
public void init(int  opmode, Key key, AlgorithmParameters params)
```

`opmode` is either `Cipher.ENCRYPT_MODE` or `Cipher.DECRYPT_MODE` which means either to encrypt or decrypt. Algorithm parameters include parameters that are used to initialize the algorithm such as IV. These parameters may not be necessary for encryption (if you did not supply them, random or provider specific ones will be created) but they are necessary for decryption because you have to use the same that were used for encryption [4].

Finally you do the encryption/decryption using one of the two ways: [2] [4]

- One step single-part operation using

  ```
  public final byte[] doFinal(byte[]  input)
  ```

  This method will do the operation (encrypt/decrypt) on the input byte array and output the result to a byte array at one step. One step here means to operate on the input data (and any data buffered before) at once and give all the output immediately (Note that this data is still divided into blocks). If there is any padding scheme then it will be used here when length of data is not an integral number of blocks.
- Multiple steps or multiple part operation using

  ```
  public final byte[] update(byte[]  input) throws IllegalStateException
  ```

  This method will be used to encrypt the input data in input byte array along with data remaining from the previous call. If data length is not an integral number of blocks then no padding will be used, but instead the

remaining will be buffered for the next call (step) of this method. When you want to finish encryption you call

```
public final byte[ ] doFinal()
```

which will encrypt/decrypt the remaining and pad it if necessary.

The following example will encrypt a small message using DES/CBC/PKCS5Padding and then decrypt it. We will not give any IV for encryption but instead let the engine itself create it. For decryption we have to get this IV and give it to the engine.

```
//This is message to be encrypted using Bouncy Castle provider
String msg = new String("This is the sample message");

//generating a 256bit DES random symmetric key (for more info revise the previous
chapter for creating and managing keys)
KeyGenerator kg = KeyGenerator.getInstance("DES","BC");
kg.init(256);
SecretKey sk = kg.generateKey();

//Instantiating a cipher object for encryption/decryption using DES algorithm
//with CBC mode and PKCS5Padding scheme
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding","BC");
//Initializing the cipher object for encryption with secret key. Note that
//IV will be created by the engine internally depending on Bouncy Castle
//implementation
c.init(Cipher.ENCRYPT_MODE, sk);
//Encrypting message in one step
byte[] EncryptedMsg = c.doFinal(msg.getBytes());
System.out.println("Encrypted Message:   " + new String(EncryptedMsg));

//Getting the IV to use it with decryption
byte[] IV = c.getIV();

//Now we have to use IV along with the key to decrypt data

//We put IV in an algorithm parameter spec object to feed it to decryption engine
IvParameterSpec IvParamSpec = new IvParameterSpec(IV);

//Initializing the cipher object for decryption with key and IV
c.init(Cipher.DECRYPT_MODE, sk, IvParamSpec);
//Decrypting the message in one step and printing it out
System.out.println("Decrypted Message:    " + new String(c.doFinal(EncryptedMsg)));
```

Other Encryption and Decryption Operations in JCE

There are some other encryption types you can use with cipher engine. Password based encryption (PBE) [2] is used to extract the key from a password you feed to the engine so you can create this key again in any other place using this password. This is used to protect private keys by encrypting them using your password so if somebody has the key he can not use it without the password. Also you can wrap (encrypt with for example PBE) a key into bytes so that key can be securely transported and then unwrapped (decrypted). Also you can use cipher streams [4] which are similar to normal input/output streams but with a cipher engine associated with them to do the encryption. This should make encrypting large amount of data from files or network done while you are reading or writing them.

Cryptographic scenarios

When you want to use cryptography in your system you will find many usage scenarios depending on your requirements. Sometimes you will use digital signature to authenticate messages between different clients in your system. Sometimes you will use symmetric encryption to encrypt some important data. A typical scenario that you may find in many systems is called "Signed-and-enveloped-data" which is part of PKCS#7 standard [11]. The scenario is as following:

- We will have an important message which should be signed, encrypted and then sent to another client through network.
- The first step is to sign the message using digital signature and sender's private key.
- The second step is to encrypt the signature using the symmetric encryption.
- The third step is to encrypt the message using the symmetric encryption.
- The forth step is to encrypt the random secret key using receiver's public key.
- The forth step is to combine these all; encrypted digital signature, encrypted message and encrypted key and send them to the receiver.
- On the receiver side we will do the reverse steps to decrypt and authenticate the message.

References

[1] Schneier, Bruce, Niels Ferguson. Practical Cryptography. Indianapolis: Wiley Publishing, Inc. 2003.

[2] Oaks, Scott. Java Security. Sebastopol: O'Reilly & Associates, Inc. 2001

[3] Sun Microsystems, Inc. JavaTM Cryptography Architecture API Specification & Reference. 8 February 2002. URL: http://babbage.clarku.edu/java/docs/guide/security/CryptoSpec.html (10 December 2003).

[4] Sun Microsystems, Inc. JavaTM Cryptography Extension (JCE) Reference Guide
10 January 2002. URL: http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html (25 December 2003).

[5] Legion of Bouncy Castle. Bouncy Castle Crypto Package Specification. URL: http://www.bouncycastle.org/specifications.html (10 January 2004).

[6] Sun Microsystems, Inc. JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification. Version 1.4.2. URL: http://java.sun.com/j2se/1.4.2/docs/api/ (12 December 2003)

[7] Microsoft Corp. Cryptography Overview. .NET Framework Developer's Guide. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconCryptographyOverview.asp (20 November 2003)

[8] Benjamin, Tom. JavaTM Cryptography Architecture using Hardware cryptography. 12 August 2001. URL: http://www-106.ibm.com/developerworks/eserver/articles/java_crypto.html (15 January 2004)

[9] OpenSSL Project. RSA_public_encrypt man page. OpenSSL library documentation. 14 July 2003. URL: http://www.openssl.org/docs/crypto/RSA_public_encrypt.html (10 Febuary 2004)

[10] RSA Laboratories. PKCS #7 - Cryptographic Message Syntax Standard. PKCS Standards. Version 1.5. 1 November 1993. URL: http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/ (11 Febuary 2004)