Java Cryptography Architecture (JCA)

Introduction to Cryptography

Saeed Sadeghian

Java Cryptographic Overview

- Java cryptography uses 2 main APIs
  - Java Cryptographic Architecture (JCA)
  - Java Cryptographic Extensions (JCE)
- Robust and Extensible
- Platform independent
- Interoperable among vendor implementations

Introduction to JCA and JCE

- Java Cryptography Architecture (JCA) is part of Java 2 run-time environment.
  → **java.security.***

- The Java Cryptography Extension (JCE) extends the JCA API to include *encryption* and *key exchange* and is integrated into Java 2 SDK since the 1.4 release.

- JCE adds encryption and decryption APIs to JCA.
  → **javax.crypto.***

Architecture Overview

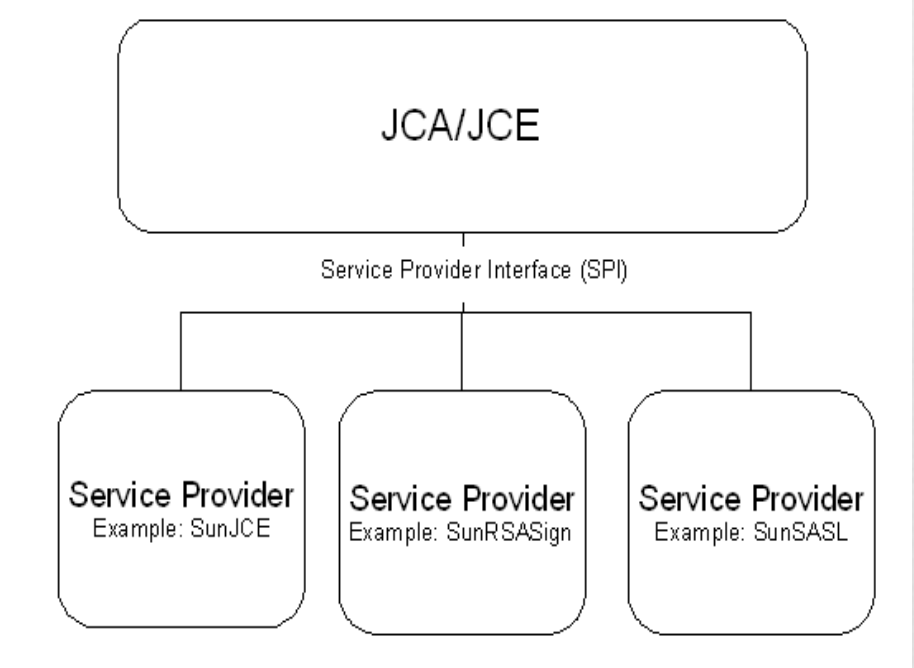
- In order to create an extensible framework for implementing cryptographic services
  - Separates engine classes from service providers
- JCA and JCE implement the engine classes which provide a standard interface to the service providers
- The service providers are the APIs which actually implement the cryptographic algorithms and types

More on that

- JCA will implement a class such as a message digest,
- We know what a message digest is, but just having a generic message digest doesn't tell us anything
- The cryptographic service provider will implement the actual algorithm, such as MD5 or SHA-1
- JCA implements the generic class
- The service provider implements the actual algorithm or type of cryptographic service that will be used

Java Cryptographic Architecture

- JCA/JCE define the types and functionalities of different cryptographic services
- The actual cryptographic implementation is done by service providers
- JCA/JCE is made up of mostly "engine" classes which provide a standard interface into the service providers
- This makes the overall implementation extensible since new service providers can be "plugged in"



Installing new service provider

- Somewhat convoluted

1. Jar file needs to be put into a directory that is in the CLASSPATH environment variable
2. If the application is an applet or is otherwise running underneath a security manager then appropriate permissions must be granted
3. The code must be signed by a trusted Certificate Authority (CA)

JCA

- Two Provider Classes
  - **Provider:** is used to query information about the installed service providers such as name and version.
  - **Security:** is used to add, remove, and modify the service providers
- Engine Classes: interface between JCA and the actual implementation of the service classes

Engine Class

- They (very) basically provide an interface to a specific type of cryptographic operation, eg. Cryptographic operations, generators, converters, objects.
- Example:
  - Signature
  - SecureRandom
  - Cipher
  - MessageDigest
  - …
- They don't have public constructors so you have to use getinstance() to get one

Example

- **Creating a Message Digest**

- First, the data that we are going to get the message digest of is created.

```
byte[] dataBytes = "This is test data".getBytes();
```

- Secondly, the MessageDigest class, "md" in instantiated creating a concrete object using the "SHA" algorithm.

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

- Next, "md" is updated with the input data and finally the message digest is created

```
md.update(dataBytes);
byte[] digest = md.digest();
```

JCA Classes

- **MessageDigest:** used to implement one-way hash functions such as MD5 or SHA
- **Signature:** used to implement digital signatures
- **KeyPairGenerator:** used to create public/private key pairs for different algorithms
- **KeyFactory:** used to convert keys into key specifications and then vice-versa
- **CertificateFactory:** used to generate certificates
- **KeyStore:** used to create a keystore which maintains keys and certificates in memory for later usage
- **AlgorithmParameters:** used to maintain the security parameters for specific algorithms
- **AlgorithmParameterGenerator:** used to create a set of parameters to be used for specific algorithms
- **SecureRandom:** used to create random or pseudo-random numbers

JCE

- JCE provides many of the functions that JCA does not provide such as the actual *encryption/decryption* and *symmetric key generation*
- Uses the Provider classes of JCA

JCE Core Classes

- **Cipher Class**
  - Provide the functionality of encryption and decryption
- **KeyGenerator Class**
  - Generate secret keys for encryption and decryption
- **The SealedObject Class**
  - Create an object and protect its confidentiality
- **The Mac Class**
  - Provide integrity protection with Message Authentication Code (MAC).

Reference: http://java.sun.com/j2se/1.5.0/docs/guide/security/jce/JCERefGuide.html

Classes

Class	Description
java.security.Key java.security.PrivateKey java.security.PublicKey javax.crypto.SecretKey	Use to encrypt and sign messages
java.crypto.Cipher	Cipher
java.security.MessageDigest	Message digest function
java.security.Signature	Digital signature
java.security.cert.Certificate	Authentication
java.security.KeyFactory javax.crypto.KeyAgreement javax.crypto.KeyGenerator javax.crypto.SecretKeyFactory	Symmetric Keys and Asymmetric Keys management
java.security.SecureRandom	Secure random number generator
Javax.crypto.Mac	Message Authentication Code

Class: java.crypto.KeyGenerator

Methods:

- getInstance(String algorithm)
  - Creates an instance of KeyGenerator for a specific algorithm such as
  - “AES”,“DES”,”HMACSHA1”
- generateKey()
  - Generate a key for the algorithm specified in the KeyGenerator instance

JCE Examples

- Generate Secret Key
  - KeyGenerator kg = KeyGenerator.getInstance("DES")
  - SecretKey sKey = kg.generateKey();
- A secret key is used for symmetric encryption/decryption
- First, create a concrete key generator object; in this case a DES key
- Second, create a SecretKey object and call the **generateKey** method of the KeyGenerator object to retrieve the key.

JCE Examples

- Encrypt
  - byte[] testdata = “Understanding Java Cryptography”.getBytes();
  - Cipher myCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
  - myCipher.init(Cipher.ENCRYPT_MODE, sKey);
  - byte[] cipherText = myCipher.doFinal(testdata);

- First, load some test data.
- Second, create a concrete Cipher object
- Third, initialize the cipher with the secret key for encryption
- Finally, the **doFinal** method actually encrypts the data

References

- http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
- http://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/guide/security/CryptoSpec.html
- http://www.cs.umd.edu/~jkatz/security/s08/lectures/JCA.pdf
- Engine Classes
  - http://docstore.mik.ua/orelly/java-ent/security/ch08_04.htm
- http://www.cs.cityu.edu.hk/~cs4288/

JCA Examples

- Create Message Digest
  - byte[] dataBytes = “This is test data”.getBytes();
  - MessageDigest md = MessageDigest.getInstance("SHA1");
  - md.update(dataBytes);
  - byte[] digest = md.digest();
- **First, the test data is populated.**
- **Second, a concrete message digest object is created with SHA1 as the cryptographic algorithm**
- **Third, the message digest object is updated; i.e. the digest is updated using the current bytes**
- **Finally, the digest method completes the algorithm**

JCA Examples

- Create Keystore
  - KeyStore ks = KeyStore.getInstance("JCEKS");
  - ks.load(null,password.toCharArray());
  - java.io.FileOutputStream fos = new java.io.FileOutputStream(keyFilePath);
  - ks.store(fos, password.toCharArray());
  - fos.close();
- **First, create the concrete KeyStore object.**
- **Second, load "ks" with a null input**
- **Third, create the output stream to save the file.**
- **Fourth, the store method saves the KeyStore to the file specified and protects it with the password**
- **Finally, close the output stream.**