



Java Cryptography – Dealing with Practical Issues

Paul Tremblett
AudioAudit, Inc.



Sorting Out the Alphabet Soup

- Java Cryptography Architecture (JCA)
 - Framework for accessing and developing cryptographic functionality for the Java platform
 - Introduced in JDK 1.1
 - Enhanced and extended in subsequent releases
 - Encompasses the parts of the J2SDK Security API related to cryptography
 - Includes the provider architecture



Sorting Out the Alphabet Soup₍₂₎

- Java Cryptography Extension (JCE)
 - Extends the JCA API to include APIs for:
 - Encryption
 - Key exchange
 - Message authentication
 - An extension to versions 1.2 and 1.3
 - Integrated into JDK 1.4



JCA Design Principles

- Implementation independence and interoperability
 - Achieved using a provider-based architecture
- Algorithm independence and extensibility
 - Achieved by defining classes that provide the functionality of well-defined “engines”
- Extensibility
 - New algorithms can be easily added



Cryptographic Service Provider

- A package (or set of packages) that supplies a concrete implementation of a subset of the cryptography aspects of the Security API
- Each SDK installation has one or more provider packages installed
- New providers may be added statically or dynamically



Practical Issue 1

- Determining which providers are available and the services they offer

- The architecture was designed in such a manner that you can achieve this goal using code, so you do not have to rely on external documentation



Listing Providers

- The static method `getProviders()` of the `Security` class returns an array of `Provider` objects
- The `Provider` class defines the methods:
 - `getName()`
 - `getVersion()`
 - `getInfo()`
 - `toString()`



Listing Providers

```
import java.security.Provider;
import java.security.Security;
public class ListProviders {
    public static void main(String[] args) {
        Provider[] p = Security.getProviders();
        for (int i = 0; i < p.length; ++i) {
            System.out.println("-----");
            System.out.println("PROVIDER # " + (i + 1));
            System.out.println(p[i]);
            System.out.println("info: " + p[i].getInfo());
            System.out.println("name: " + p[i].getName());
            System.out.println("version: " + p[i].getVersion());
            System.out.println("string: " + p[i].toString());
            System.out.println();
        }
        System.exit(0);
    }
}
```



Partial List of Providers

PROVIDER # 1

SUN version 1.2

info: SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS keystore; PKIX CertPathValidator; PKIX CertPathBuilder; LDAP, Collection CertStores)

name: SUN

version: 1.2

string: SUN version 1.2

PROVIDER # 2

SunJSSE version 1.41

info: Sun JSSE provider(implements RSA Signatures, PKCS12, SunX509 key/trust factories, SSLv3, TLSv1)

name: SunJSSE

version: 1.41

string: SunJSSE version 1.41



Determining Available Services

- Can be approached at two levels:
 - Use the **keySet ()** method of **Provider**
 - Returns a **Set**
 - Use **Iterator** to retrieve elements
 - Use the **getAlgorithms ()** method of **Security**
 - Takes as argument a **String** specifying one of:
 - ✓ "Signature", "MessageDigest", "Cipher", "Mac", "Keystore"
 - Returns a **Set**
 - Use **Iterator** to retrieve elements



ListProviderProperties

```
import java.security.Provider;
import java.security.Security;
import java.util.Iterator;
import java.util.Set;
public class ListProviderProperties {
    public static void main(String[] args) {
        Provider[] providers = Security.getProviders();
        for (int i = 0; i < providers.length; ++i) {
            System.out.println("Provider " + providers[i]
                .getName() + " has the following properties:");
            System.out.println();
            Set keyset = providers[i].keySet();
            Iterator it = keyset.iterator();
            while (it.hasNext()) {
                System.out.println((String)it.next());
            }
        }
    }
}
```



Partial List of Properties

Provider SunJCE has the following properties:

Cipher.DES

KeyStore.JCEKS

Alg.Alias.SecretKeyFactory.TripleDES

SecretKeyFactory.DES

KeyGenerator.HmacSHA1

Alg.Alias.KeyFactory.DH

KeyGenerator.DESede

Mac.HmacMD5

Cipher.Blowfish

Mac.HmacSHA1

Cipher.DESede

AlgorithmParameters.DESede

Alg.Alias.AlgorithmParameters.TripleDES

KeyPairGenerator.DiffieHellman

KeyFactory.DiffieHellman

Alg.Alias.AlgorithmParameters.PBEWithMD5AndDES

AlgorithmParameters.PBE

AlgorithmParameterGenerator.DiffieHellman



ListAlgorithms

```
public class ListAlgorithms {  
    public static void main(String[] args) {  
        String[] services = {"Signature", "MessageDigest",  
            "Cipher", "Mac", "KeyStore"};  
        for (int i = 0; i < services.length; ++i) {  
            System.out.println("\n" + services[i] + ":");  
            Set s = Security.getAlgorithms(services[i]);  
            Iterator it = s.iterator();  
            while (it.hasNext()) {  
                System.out.println("\t" + (String)it.next());  
            }  
        }  
    }  
}
```



Available Algorithms

Signature:

MD2WITHRSA

SHA1WITHRSA

SHA1WITHDSA

MD5WITHRSA

MessageDigest:

SHA

MD5

Mac:

HMACSHA1

HMACMD5

**Cipher:**

PBEWITHMD5ANDTRIPLEDES

DESEDE

PBEWITHMD5ANDDES

BLOWFISH

DES

KeyStore:

PKCS12

JKS

JCEKS



Standard Names

- JCE API requires and utilizes standard names for algorithms, algorithm modes and padding schemes
- Standard names defined in Appendix A of *Java Cryptography Architecture API Specification and Reference*
- Standard name list is supplemented in Appendix A of *Java Cryptography Extension (JCE) Reference Guide*



Examples of Standard Names

- Message digest:

- MD2

- MD5

- SHA-1

- Cipher:

- AES

- Blowfish

- DES



Practical Issue 2

- Obtaining sample code, or at least code snippets
- The *Java Cryptography Extension 1.2.2 API Specification & Reference* available from <http://java.sun.com> provides several useful examples
- The slides that follow offer additional snippets (full code will be available on the conference CD)



The Signature Class

- Based on algorithms whose mathematical properties are such that output produced by applying the algorithm using a public key is identical to output produced by applying the algorithm using the corresponding private key
- Used to authenticate and ensure integrity



Generating a Digital Signature

- Generate a public/private key pair
- Obtain a **Signature** object
- Generate the signature
- Transmit the document, signature and public key to the recipient



Generating a Key Pair

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");  
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");  
  
keyGen.initialize(1024, random);  
  
KeyPair pair = keyGen.generateKeyPair();  
  
PrivateKey priv = pair.getPrivate();  
PublicKey pub = pair.getPublic();
```



Signing a Document

```
Signature siginst = Signature.getInstance("SHA1withDSA");
siginst.initSign(priv);
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);
    siginst.update(buffer, 0, len);
};
bufin.close();
byte[] signature = siginst.sign();
```



Verifying a Signature

- Get the public key
- Get the signature bytes
- Obtain a **Signature** object
- Update **Signature** object with bytes from document that was originally signed
- Pass signature bytes to `verify()` method of **Signature** object
- Method returns **true** or **false**



Retrieving a 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");  
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
```

Obtaining and Initializing a Signature Object

```
Signature signature = Signature.getInstance("SHA1withDSA");  
signature.initVerify(pubKey);
```



Updating the Signature Object

```
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);
    signature.update(buffer, 0, len);
};
bufin.close();
```

Performing Signature Verification

```
FileInputStream sigfis = new FileInputStream(args[1]);
byte[] candidateSignature = new byte[sigfis.available()];
sigfis.read(candidateSignature );
sigfis.close();
boolean verifies = signature.verify(candidateSignature);
System.out.println("signature is" +
    ((verifies) ? " " : " not ") + "valid");
```



The Cipher Class

- Provides the functionality of a cryptographic cipher
- Two types:
 - Symmetric
 - Relies on a shared secret
 - Same key used for encryption and decryption
 - Asymmetric
 - Uses public/private key pair for encryption and decryption



Creating a Cipher Object

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



Creating a Secret Key

```
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");  
DESKeySpec spec = new DESKeySpec(passPhrase.getBytes(), 0);  
key = desFactory.generateSecret(spec);
```



Using a Cipher

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

```
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
```

```
DESKeySpec spec = new DESKeySpec(passPhrase.getBytes(), 0);
```

```
key = desFactory.generateSecret(spec);
```

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

```
byte[] cipherText = c.doFinal(clearText.getBytes());
```

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

```
byte[] recoveredTextFromArray = c.doFinal(cipherText);
```



The Mac Class

- Provides the functionality of a “Message Authentication Code” (MAC)
- Used to check integrity of information
- If based on a hash algorithm, called HMAC



Generating a Message Digest

```
public byte[] generateMac(String message, SecretKey key)
    throws NoSuchAlgorithmException, InvalidKeyException {
    Mac mac = Mac.getInstance("HmacMD5");
    mac.init(key);
    return mac.doFinal(message.getBytes());
}
```



Detecting Tampering

```
String message = "The time has come the Walrus said";
String alteredMessage = "The time has come the Walrus said";
try {
    KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
    SecretKey sk = kg.generateKey();
    MacDemo demo = new MacDemo();
    byte[] result1 = demo.generateMac(message,sk);
    byte[] result2 = demo.generateMac(alteredMessage,sk);
    System.out.println("The string was " +
        (demo.isIdentical(result1,result2) ? "not " : " ") +
        " altered");
}
catch (InvalidKeyException e) {
}
catch (NoSuchAlgorithmException e) {
}
```

Program output: The string was not altered



An Eye Test

```
String message = "The time has come the Walrus said";
String alteredMessage = "The time has come the walrus said";
try {
    KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
    SecretKey sk = kg.generateKey();
    MacDemo demo = new MacDemo();
    byte[] result1 = demo.generateMac(message,sk);
    byte[] result2 = demo.generateMac(alteredMessage,sk);
    System.out.println("The string was " +
        (demo.isIdentical(result1,result2) ? "not " : " ") +
        " altered");
}
catch (InvalidKeyException e) {
}
catch (NoSuchAlgorithmException e) {
}
```

Program output: The string was altered



Practical Issue 3

- The JCE jurisdiction policy files shipped with JDK 1.4 allow “strong” but limited cryptography to be used
- Residents of eligible countries can download and install an “unlimited strength” version
- Installation consists of unzipping download file and using contents to replace existing policy files*

* BACK UP ORIGINAL FILES



Practical Issue 4

- You might want to add a provider if:
 - Default provider(s) delivered with JDK do not provide all of the algorithms you need
 - Alternative providers offer algorithms that perform better



Finding Additional Providers

- Check out:
 - Legion of the Bouncy Castle
 - ❖ <http://www.bouncycastle.org>
 - Cryptix
 - ❖ <http://www.cryptix.org>
- Some others are listed at:
 - ❖ http://java.sun.com/products/jce/jce122_providers.html
- And remember, google.com is a developer's best friend



Adding a Provider

- A provider can be added in two ways:
 - Dynamically
 - Loaded by the program that requires it
 - Available only to the program for the life of the program
 - Statically
 - Loaded when the JVM starts up
 - Available to all programs
 - Persists



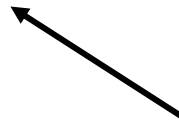
The java.security File

- The master security properties file
- Where static Cryptography Package Providers are registered
- One entry per provider
 - Format is:
 - security.provider.<n> = <className>
 - <className> is Provider subclass name
 - <n> is preference order



Entries in java.security

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsa.jca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
```



Note position of last provider



Adding a Provider Dynamically

- Create an instance of the `Provider` subclass
- Pass the object to the static `addProvider()` method of the `Security` class
- Method returns the preference position at which the provider was added or -1 if the provider was already installed

Code to Load a Provider

```
import java.security.Security;
import java.security.Provider;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class LoadProvider {

    public static void main(String args[]) {
        Provider provider = new BouncyCastleProvider();
        int position = Security.addProvider(provider);
        if (position >= 0) {
            System.out.println("Provider added at position " +
position);
        }
        else {
            System.err.println("Unable to add provider");
        }
    }
}
```

Program Output: Provider added at Position 6

(remember position of last provider two slides back)



Adding a Provider Staticly

- Place JAR file containing provider package where it can be located by JVM at startup (preferably, make it an installed extension)

- Create the appropriate entry in the `java.security` file



New Provider in java.security

```
security.provider.1=sun.security.provider.Sun
```

```
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
```

```
security.provider.3=com.sun.rsa.jca.Provider
```

```
security.provider.4=com.sun.crypto.provider.SunJCE
```

```
security.provider.5=sun.security.jgss.SunProvider
```

```
security.provider.6=org.bouncycastle.jce.provider.BouncyCastleProvider
```



Practical Issue 5

- If you want to add a new algorithm or provide your own implementation of an existing algorithm, you must write your own provider
- The ten steps required to implement a provider are documented at:

<http://java.sun.com/products/jce/doc/guide/HowToImplAProvider.html>



A Matter of Trust

- Authorized providers and users of authorized providers must mutually authenticate
- If you develop your own provider, it must be signed
- The two trusted Certification Authorities are:
 - Sun Microsystems JCE Code Signing CA
 - IBM JCE Code Signing CA



Conclusion

- Hopefully you will leave knowing:
 - How to determine what is in the default JCE and what is offered by installed providers
 - How to use the services provided by JCE
 - How to locate, obtain and install a provider
 - Where to obtain the blueprint for developing your own provider