Java Cryptography Extension

Robert Gretzinger

rgretzi@kodak.com
Introduction

“The JavaTM Cryptography Extension (JCE) is a set of packages that provide a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.”

- The Java Cryptography Extension is extension to the Java Cryptography Architecture (JCA)
- The JCE provides the a framework enabling encryption and decryption building blocks
- An extension because of U.S. export rules. But now part of the JDK as of 1.4.0.
Agenda

- Cryptography concepts and terminology
- JCE
  - General architecture
  - Major classes
  - Examples
Overview

- Security vs. Cryptography
Vocabulary

- Ciphers
  - Key Strategy
    - Symmetric
    - Asymmetric
  - Modes
- Key Agreement (Shared Secret)
- Message Authentication Codes (MAC)
Symmetric Key Cipher

- Same key is used for encryption and decryption.
- Examples: DES, Triple DES, RC Family, Blowfish, TwoFish

SAME KEY

Plain text

$k_{\text{secret}}$ Encrypt Cipher text

$k_{\text{secret}}$ Decrypt Plain text
Asymmetric Key Cipher

- Different, but related keys, are used for encryption and decryption.
- Examples: RSA, ElGamal
Modes

- Block ciphers have modes of operation to facilitate different applications
  - ECB – Electronic Codebook
    - Allows parallelism
  - CBC – Cipher Block Chaining
  - CFB – Cipher Feedback
    - Byte at time encryption
  - OFB – Output Feedback
    - Streams: audio/video

- Java Security Handbook provides a decision tree to help select the correct mode
## Modes and Padding from ABA Provider

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mode</th>
<th>Padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>ECB, CBC</td>
<td>PKCS5Padding, NoPadding</td>
</tr>
<tr>
<td>DESede</td>
<td>ECB, CBC</td>
<td>PKCS5Padding, NoPadding</td>
</tr>
<tr>
<td>IDEA</td>
<td>ECB, CBC</td>
<td>PKCS5Padding, NoPadding</td>
</tr>
<tr>
<td>Blowfish</td>
<td>ECB, CBC</td>
<td>PKCS5Padding, NoPadding</td>
</tr>
<tr>
<td>Twofish</td>
<td>ECB, CBC</td>
<td>PKCS7Padding, NoPadding</td>
</tr>
<tr>
<td>RSA</td>
<td>ECB</td>
<td>PKCS1Padding, NoPadding</td>
</tr>
</tbody>
</table>
Key Agreement

- Allows $N$ parties to agree on a shared secret
- Used in Hybrid solutions (SSL for example)
- Example: Diffie-Hellman
Message Authentication Code

- A message digest the is computed with a secret key
- Protects against man-in-the-middle attack
- MACs that use hashing functions are HMAC
  - MD5 128bit (faster on large data sources)
  - SHA-1 160bit (considered more secure)
- Non-Hashing
  - CBC-MAC (FIPS 113)
    - Provided in the ABA provider
    - If DES -> 64bit
  - Message Authentication Algorithm (MAA) created in 1983,
    - 32bit, slow
The U.S. Export

- Changed October 2000
- Motivation was to enable U.S. companies to compete internationally, and to keep up with the policies of EU
- Can export, without limits, to most countries
- Countries under embargo are restricted

Source: http://www.bxa.doc.gov/Encryption/19Oct2KFactsheet.html
JCA/JCE Architecture

Service Provider Interface

- java.security
- javax.crypto
- Provider #1
- Provider #2
- «file»
- Policy

Application
“strong" version of the jurisdiction policy files:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Maximum Key Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES</td>
<td>64</td>
</tr>
<tr>
<td>DESede</td>
<td>*</td>
</tr>
<tr>
<td>RC2</td>
<td>128</td>
</tr>
<tr>
<td>RC4</td>
<td>128</td>
</tr>
<tr>
<td>RC5</td>
<td>128</td>
</tr>
<tr>
<td>RSA</td>
<td>2048</td>
</tr>
<tr>
<td>* (all others)</td>
<td>128</td>
</tr>
</tbody>
</table>

Source: Java 1.4.0 JCE Ref Guide
grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
    permission javax.crypto.CryptoPermission "RC2", 128,
        "javax.crypto.spec.RC2ParameterSpec", 128;
    permission javax.crypto.CryptoPermission "RC4", 128;
    permission javax.crypto.CryptoPermission "RC5", 128,
        "javax.crypto.spec.RC5ParameterSpec", *, 12, *;
    permission javax.crypto.CryptoPermission "RSA", 2048;
    permission javax.crypto.CryptoPermission *, 128;
};
grant {
    // There is no restriction to any algorithms.
    permission javax.crypto.CryptoAllPermission;
};
Major Classes

- java.security
  - Provider
  - KeyPairGenerator
  - SecureRandom
There are many more, but the above are important to the user of JCE.

- javax.crypto (JCE)
  - Cipher
  - CipherInputStream
  - CipherOutputStream
  - MAC
  - KeyAgreement
  - KeyGenerator
  - SealedObject
Using the JCE

- Install JCE
- Configure Provider
- Application Examples
Install

- For 1.4.0
  - (optional) Install unlimited policy jar files in `<java-home>\lib\security`
    - US_export_policy.jar
    - local_policy.jar
  - Otherwise nothing to do

- For 1.3.1
  - Need to download JCE1.2.1
  - Drop jce jar files in `..\jre\lib\ext` (or CLASSPATH)
    - jce1_2_1.jar
    - US_export_policy.jar
    - local_policy.jar
    - sunjce_provider.jar (unless you have other)
Configure Provider

● Static
  – Add to java.security
    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
    security.provider.6=au.net.aba.crypto.provider.ABAProvider

● Dynamic
  Provider provider =
    new au.net.aba.crypto.provider.ABAProvider();
  Security.addProvider(provider);
Cipher Class

- **Create**
  - `transformation` -> "algorithm/mode/padding" or "algorithm"
  
  ```java
  public static Cipher getInstance(String transformation);
  ```
  
  ```java
  public static Cipher getInstance(String transformation,
                                 String provider);
  ```

- **Init (not all of the init methods listed)**
  - `Opmode` [DECRYPT_MODE, ENCRYPT_MODE, WRAP_MODE, NOWRAP_MODE]
  
  ```java
  public void init(int opmode, Key key);
  ```
  
  ```java
  public void init(int opmode, Certificate certificate)
  ```
  
  ```java
  public void init(int opmode, Key key,
                   AlgorithmParameterSpec params,
                   SecureRandom random);
  ```

- **Encrypt/Decrypt (not all methods listed)**
  
  ```java
  public byte[] update(byte[] input);
  ```
  
  ```java
  public byte[] doFinal(byte[] input);
  ```
Cipher (Triple DES Example)

- Key Gen
- Construct Cipher Object
- Init Cipher Object
- Encrypt
- Mess with the key
- Decrypt
Provider provider = new com.sun.crypto.provider.SunJCE();
System.out.println("Provider: " + provider.getInfo());
Security.addProvider(provider);

KeyGenerator keygen = KeyGenerator.getInstance("DESede");
keygen.init(168);
SecretKey desKey = keygen.generateKey();
System.out.println("Key Generated");

// Create the cipher
Cipher desCipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
// Initialize the cipher for encryption
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
// Our cleartext
byte[] cleartext = "This is some text".getBytes();

// Encrypt the cleartext
byte[] ciphertext = desCipher.doFinal(cleartext);

for (int i = 0; i < ciphertext.length; i++ )
    System.out.println("C:" + i + " " + Integer.toHexString((byte)ciphertext[i]));
// now make sure we can regenerate the key from a byte array
SecretKey sameKey = new SecretKeySpec(desKey.getEncoded(),
    "DESede");

// Initialize the same cipher for decryption
desCipher.init( Cipher.DECRYPT_MODE,sameKey);

// Decrypt the ciphertext
byte[] cleartext1 = desCipher.doFinal(ciphertext);
for (int i = 0; i < cleartext1.length; i++)
    System.out.println("T:" + i + " " + (char) cleartext1[i]);
Demo 1
Using Stream Cipher IO

// Create the cipher
Cipher desCipher = Cipher.getInstance("DESede/ECB/PKCS5Padding","SunJCE");

// Initialize the cipher for encryption
desCipher.init(Cipher.ENCRYPT_MODE, desKey);
FileOutputStream f = new FileOutputStream("out.cipher");
CipherOutputStream o = new CipherOutputStream(f, desCipher);

byte[] cleartext = "This is just an example".getBytes();
o.write(cleartext);
o.close();
RSA

- Difficulty in factoring large numbers
- Public Domain September 2000

Compute $n = pq$ where $p$ and $q$ are large distinct primes. $n$ is the modulus.

Select $e$ relatively prime to $(p-1)(q-1)$

Compute $d$ such that $ed \equiv 1 \pmod{(p-1)(q-1)}$

Public Key is $(n,e)$

Private Key is $d$

Encrypt: $c = m^e \mod n$

Decrypt: $m = c^d \mod n$
RSA Example

- Generate Key Pair
- Examine public key
- Init Cipher
- Encrypt
- Decrypt
provider = new au.net.aba.crypto.provider.ABAProvider();
System.out.println("Provider: " + provider.getInfo());
Security.addProvider(provider);
SecureRandom random = new SecureRandom();

// Create the cipher
Cipher cipher = Cipher.getInstance("RSA/ECB/NoPadding");

KeyPairGenerator keygen = KeyPairGenerator.getInstance("RSA");
keygen.initialize(512, random);
out.print("  Generating keypair ");
KeyPair pair = keygen.generateKeyPair();
PrivateKey privateKey = pair.getPrivate();
PublicKey publicKey = pair.getPublic();
RSAPublicKey rsapk = (RSAPublicKey) publicKey;

out.println(rsapk.getModulus());
out.println(rsapk.getPublicExponent());

publicKey = new RSAPubKey (rsapk.getModulus(),
rsapk.getPublicExponent());
out.println("Encrypt");
cipher.init(Cipher.ENCRYPT_MODE, publicKey);

byte[] cleartext = "This is just an example".getBytes();
out.println("...Done
");

// Encrypt the cleartext
byte[] ciphertext = cipher.doFinal(cleartext);

for (int i = 0; i < ciphertext.length; i++)
    System.out.println("C:" + i + " " +
        Integer.toHexString((int)ciphertext[i]));
// Initialize the same cipher for decryption
    cipher.init(Cipher.DECRYPT_MODE, privateKey);

    // Decrypt the ciphertext
    byte[] cleartext1 = cipher.doFinal(ciphertext);
    for (int i = 0; i < cleartext1.length; i++)
        System.out.println("T: " + i + "  "+ Integer.toHexString((int)cleartext1[i]));
Demo 2
MAC Example

- HMAC using MD5 Hashing (from Sun Doc)

```
// Generate secret key for HMAC-MD5
KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
SecretKey sk = kg.generateKey();

// Get instance of Mac object implementing HMAC-MD5, and
// initialize it with the above secret key
Mac mac = Mac.getInstance("HmacMD5");
mac.init(sk);
byte[] result = mac.doFinal("Hi There").getBytes();
```
References

Where to Get More Information

• RSA Security Faqs (http://www.rsasecurity.com/rsalabs/faq)
  • Great resource!
• CRC Handbook of Applied Cryptography (http://www.cacr.math.uwaterloo.ca/hac/)  Complete Book is on-line!
• Java Security Handbook, J. Jaworski, P. J. Perrone
• Bureaucracy
  • NIST (http://www.nist.gov)
    • Info on AES (http://csrc.nist.gov/encryption/aes/)
  • Bureau of Export Administration (http://www.bxa.doc.gov)
• NSA (http://www.nsa.gov/)
  • History (online Museum)
• Some JCE Providers
  • List of Providers (http://www.nue.et-inf.uni-siegen.de/SignStreams/csp/overview_provider.html)
  • CryptixJCE (http://www.cryptix.org) (FREE)
• IAIK-JCE (http://jcewww.iaik.tu-graz.ac.at)
• Crypto-J (http://www.rsasecurity.com)