Java Cryptography

Jeff Lawson

Presentation Contents

- 1 Introduction to Cryptography
- 2 Cryptographic Service Providers
- 3 Symmetric Key Cryptography
- 4 Symmetric Key Cryptography for Android and iOS
- 5 Asymmetric Key Cryptography
- 6 Digital Signatures
- 7 Authenticated Encryption

- 8 Digital Certificates
- 9 PKI
- 10 Key Stores and Trust Stores
- 11 SSL and TLS (JSSE)
- 12 Accessing LDAP Servers with JNDI
- 13 Certificate Revocation Lists and OCSP
- 14 Privilege Management Infrastructure

Java Cryptography

Introduction to Cryptography

Jeff Lawson

Contents

- Cryptography
- Encryption and Decryption
- Import / Export / Domestic Laws

Cryptography

- Cryptography names the processes involved in:
    - Keeping information confidential (secret)
    - Establishing the provenance of information
    - Checking the integrity of information
    - Controlling access to information
- Cryptography has become a mandatory feature of many information processing systems
- Consumers and law courts are starting to demand that personal information be handled securely

Encryption and Decryption

- *Encryption* is the process of transforming information from a human-intelligible form ('in the clear') to an unintelligible form (encrypted) i.e. Data can still be read, it just cannot be understood!

- *Decryption* is the reverse process of recovering the original clear data from the encrypted data

- Encryption processes:
    - simple, e.g. Caesar
    - sophisticated, e.g. Enigma machine, computational algorithm

Import / Export / Domestic Laws

- Many governments classify encryption technologies as munitions!

- Consequently, there are a variety of legal ramifications to consider when developing, importing, exporting and travelling with cryptographic software!

- Since Java is held to originate from the U.S.A., there are export considerations

- Since some countries have restrictive legislation regarding import, Java is configured for weak cryptographic keys, by default

- The United States government prohibits export of cryptographic software, (e.g. Java SE!) to:

  Cuba, Iran, Sudan, North Korea, Syria

- Additionally, the U.S. Department of Treasury prohibits dissemination of cryptographic software to:

  - Specially Designated Nationals

  - Specially Designated Terrorists

  - Specially Designated Narcotic Traffickers

- Also, the U.S. Department of Commerce prohibits dissemination of cryptographic software to people on:

  - Table of Denial Orders

- Many other countries control export, import and domestic use of cryptographic software; some follow the Wassenaar Arrangement controls the export of weapons

- The United Kingdom, France, Holland, India, Thailand, Malaysia, Singapore and Australia demand decryption

- France is of particular note in having exceptional restrictions

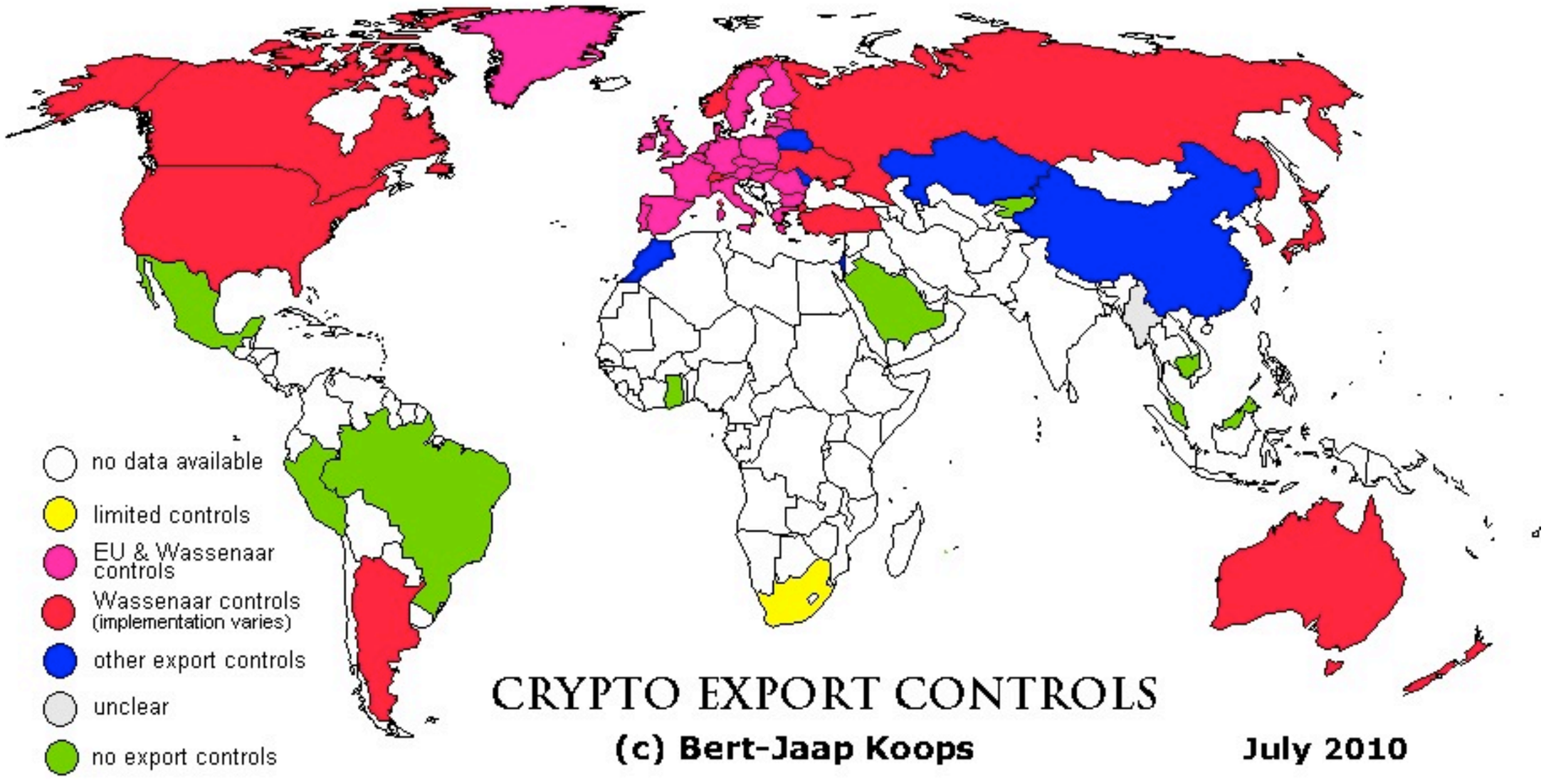
- In Switzerland, crypto import is not controlled and import certificates will be given if the country of origin requires it; export of 'mass-market and public-domain software' is not controlled but re-export is not permitted if the country of origin does not allow the export to the destination country

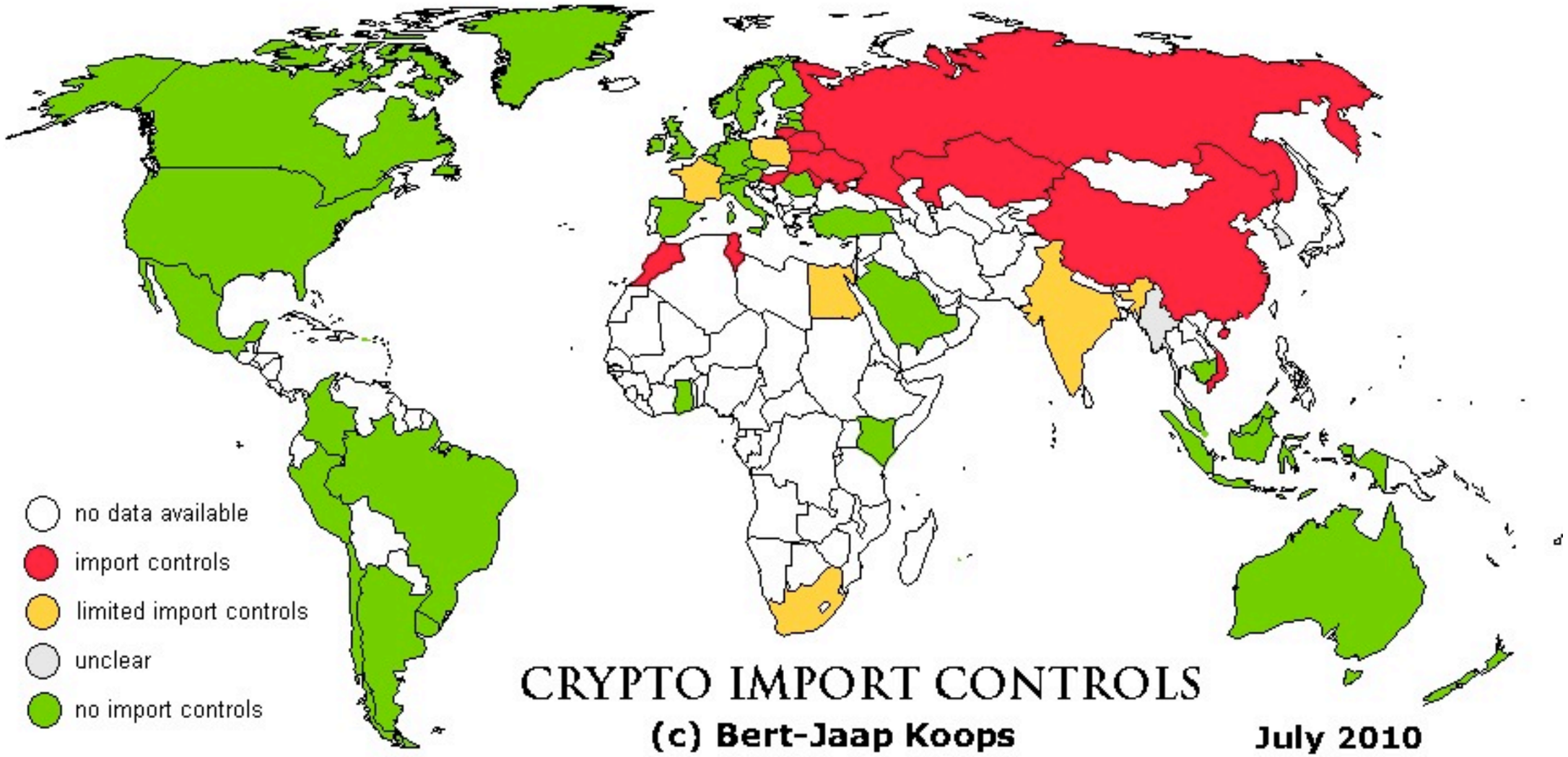
- The number of international laws restricting cryptography is bewildering
- See the Crypto Law Survey:

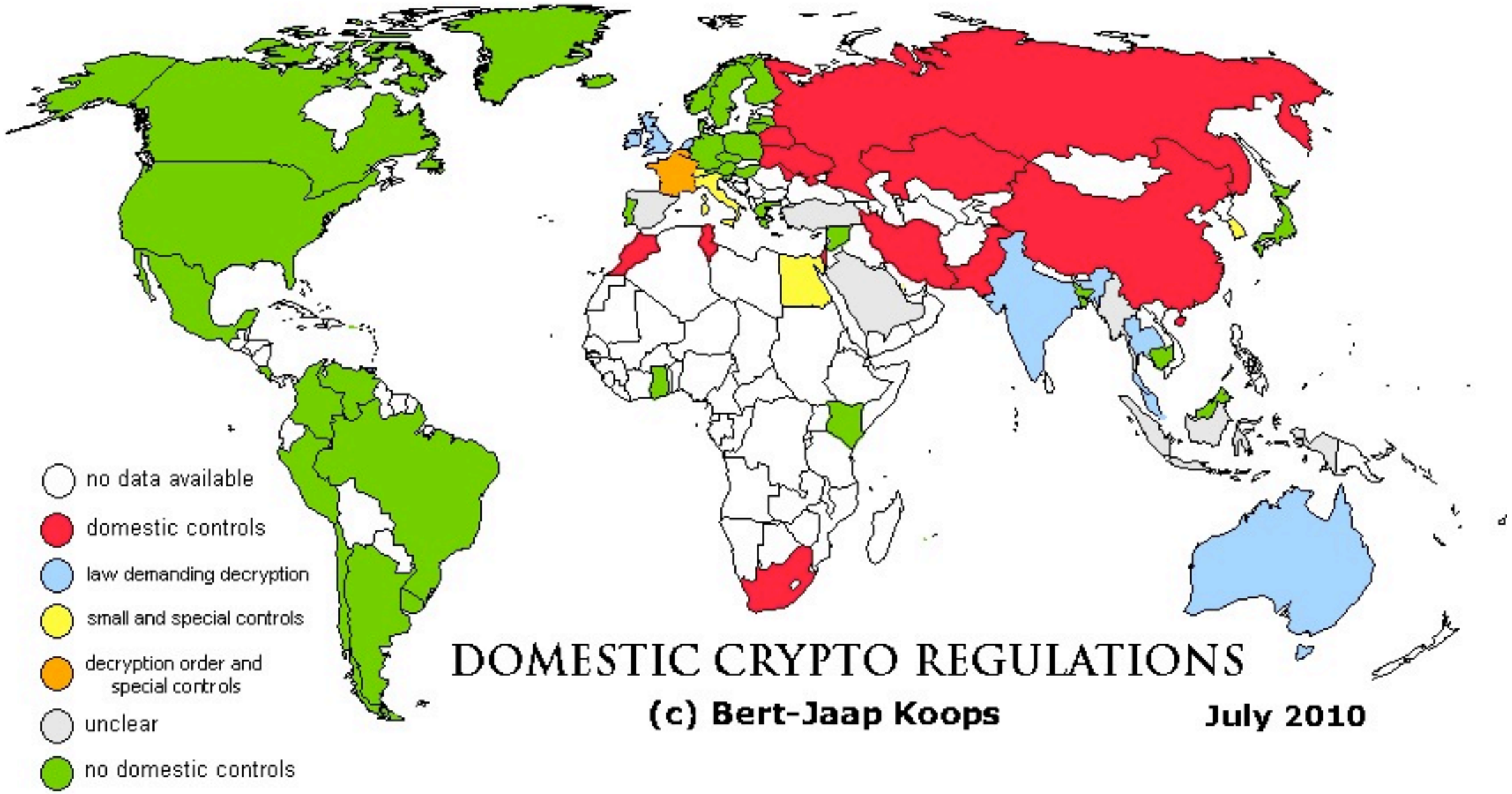
  www.cryptolaw.org

- Ultimately, legal advice may be needed

- The following three images are taken from the Crypto Law Survey web site and are used here with kind permission of Bert-Jaap Koops
  Thank you, Bert-Jaap :-)







Java Cryptography

Cryptographic Service Providers

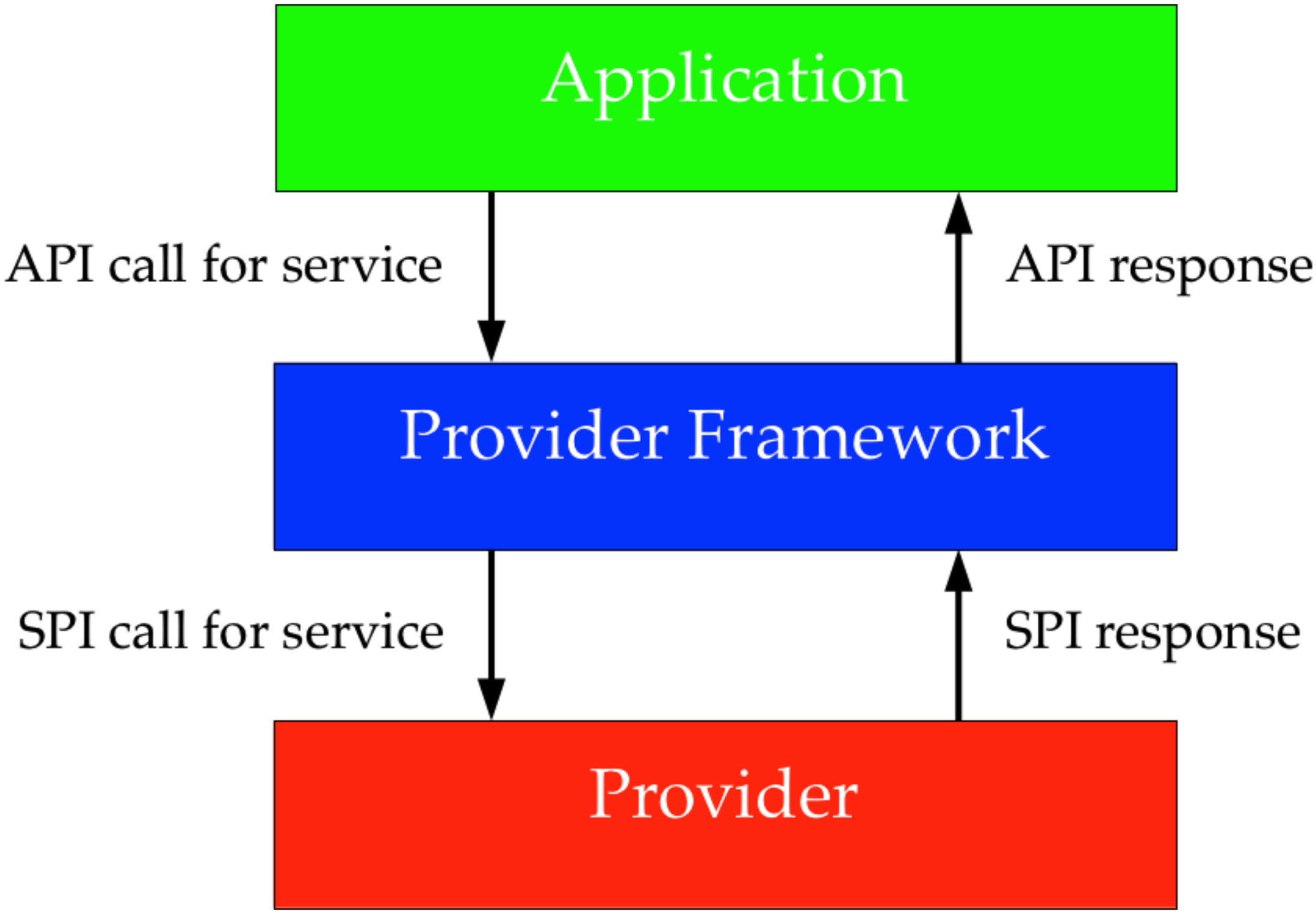
Jeff Lawson

Contents

JCA Cryptographic Service Providers

- In common with many other customisable Java technologies, the JCA/JCE uses a *service provider* architecture:
    - Providers implement a service provider interface
    - Java exposes an application programming interface that calls into zero or more service providers through their service provider interface

...



Available JCA Providers

- Java SE is bundled with several JCA service providers: *

SUN 1.7	SunRsaSign 1.7
SunEC 1.7	SunJSSE 1.7
SunJCE 1.7	SunJGSS 1.7
SunSASL 1.7	XMLDSig 1.0
SunPCSC 1.7	SunMSCAPI 1.7

- These providers implement their own set of cryptographic algorithms, e.g. SunJSSE has specific support for the Java Secure Socket Extension

* JRE 1.7.0_10-b18 on Windows 8

- To discover which providers are available, use:

```
import java.security.Provider;
import java.security.Security;
// ...
Provider[] providers =  Security.getProviders();
Provider provider;
for (int n=0; n<providers.length; n++)
{
  provider = providers[n];
  System.out.println(provider.getName() + " " +
                     provider.getVersion() + " : " +
                     provider.getInfo());
}
```

- Sample output line:

```
SunEC 1.7 : Sun Elliptic Curve provider (EC, ECDSA, ECDH)
```

- To discover which algorithms are available from a provider, use:

```
import java.security.Provider.Service;
// ...
Set<Service> setServices = provider.getServices();
Iterator<Service> itServices = setServices.iterator();
Service service;
while (itServices.hasNext())
{
  service = itServices.next();
  System.out.println("   " + service.getAlgorithm()
                     + " " + service.getType());
}
```

- Sample output...

- For the Sun 1.7 provider:

```
SHA1PRNG SecureRandom
SHA1withDSA Signature
NONEwithDSA Signature
DSA KeyPairGenerator
MD2 MessageDigest
MD5 MessageDigest
SHA MessageDigest
SHA-256 MessageDigest
SHA-384 MessageDigest
SHA-512 MessageDigest
DSA AlgorithmParameterGenerator
DSA AlgorithmParameters
DSA KeyFactory
X.509 CertificateFactory
JKS KeyStore
CaseExactJKS KeyStore
JavaPolicy Policy
JavaLoginConfig Configuration
PKIX CertPathBuilder
PKIX CertPathValidator
LDAP CertStore
Collection CertStore
com.sun.security.IndexedCollection CertStor
```

- For even more information, use:

```
String strKey;
String strValue;
Iterator<Object> itKeys = provider.keySet().iterator();
while (itKeys.hasNext())
{
  strKey = (String)itKeys.next();
  strValue = provider.getProperty(strKey);
  System.out(strKey + "  " + strValue);
}
```

- Sample output...

- For the Sun 1.7 provider:

```
KeyFactory:
  DSA  sun.security.provider.DSAKeyFactory
       ImplementedIn  Software
AlgorithmParameters:
  DSA  sun.security.provider.DSAParameters
       ImplementedIn  Software
AlgorithmParameterGenerator:
  DSA  sun.security.provider.DSAParameterGenerator
       ImplementedIn  Software
       KeySize  1024
Signature:
  NONEwithDSA  sun.security.provider.DSA$RawDSA
       SupportedKeyClasses
           java.security.interfaces.DSAPublicKey |
           java.security.interfaces.DSAPrivateKey

...
```

Accessing JCA Providers

- JCA providers are used implicitly or explicitly
- Implicit, unspecified provider use example:

```
Signature sigDefault = Signature.getInstance("MD5withRSA");
```

Sample result: `SunRsaSign provider`

- Oracle recommends that "General purpose applications SHOULD NOT request cryptographic services from specific providers."

- The system knowns about providers that are listed in the file

  `<java-home>/lib/security/java.security`

  and those that have been loaded from code, e.g.

  ```
  Security.addProvider(new BouncyCastleProvider());
  ```

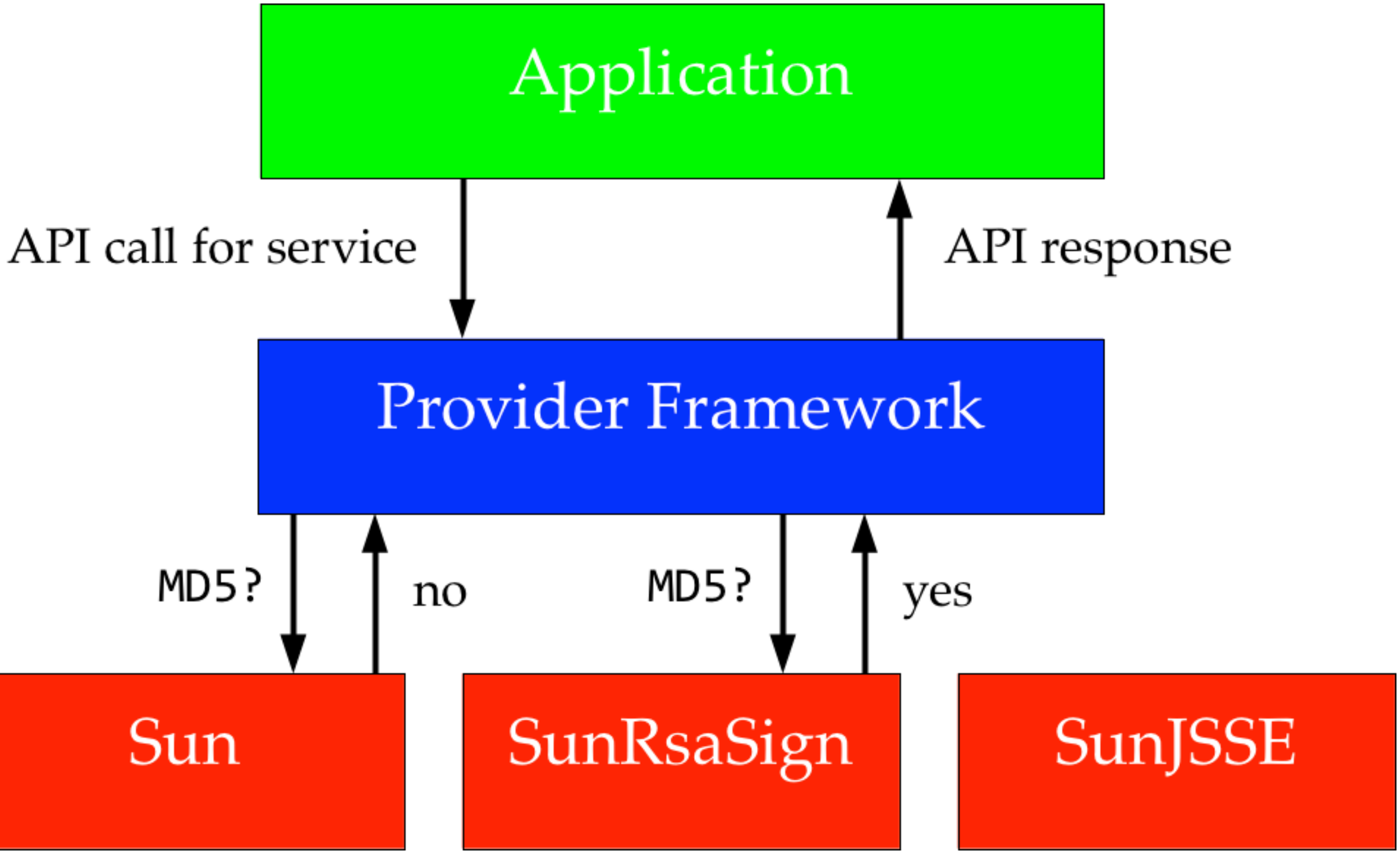
- `java.security` contains precedence numbers

- When a provider is not specified, the JCA queries each known provider in order of precedence do discover whether the desired algorithm is supported

- The first provider to acknowledge support for the algorithm is used by the JCA

...

```
Signature.getInstance("MD5")
```



- You may want your solutions to work with specific providers that you trust!

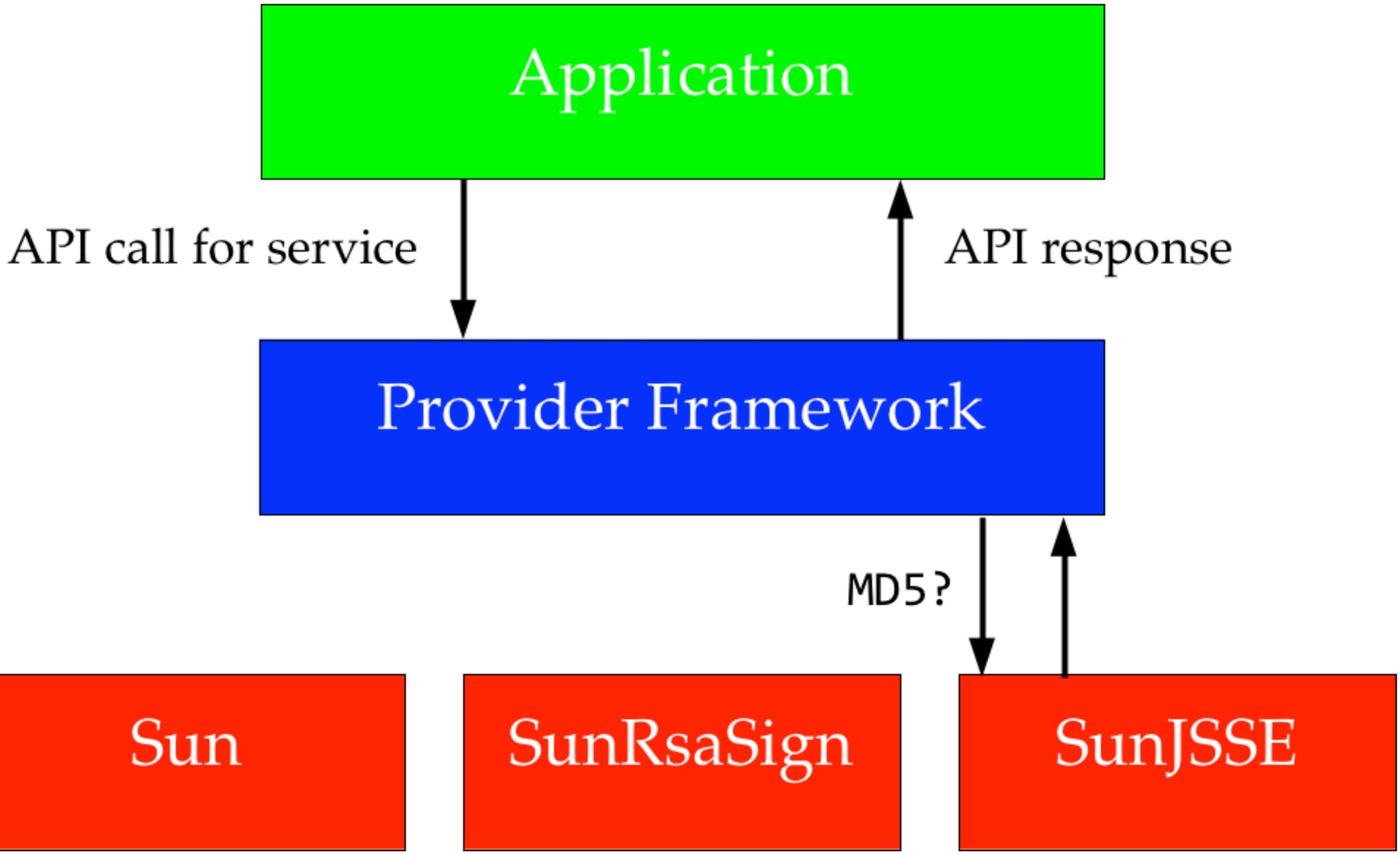
- Explicit, specified provider use example:

```
Signature sigSunJSSE = Signature.getInstance("MD5withRSA",
                                             "SunJSSE");
```

- When a requested provider is not found, a `NoSuchProviderException` exception is thrown

- When a requested provider is found but it does not support the requested algorithm, a `NoSuchAlgorithmException` exception is thrown

```
Signature.getInstance("MD5", "SunJSSE")
```



Enabling Unlimited Strength

- Java SE (both JRE and JDK) comes with support for unlimited strength cryptography but it is not enabled by default

- To enable unlimited strength cryptography, download and install the Java Cryptography *Extension (JCE) Unlimited Strength Jurisdiction Policy Files* at the bottom of the Java downloads web page

- The `README.txt` document provides instructions

- Essentially: replace the files `local_policy.jar` and `US_export_policy.jar` in `<java-home>/lib/security`

- To test for unlimited strength cryptography, use:

```
byte[] data = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};
SecretKey key256 = new SecretKeySpec(new byte[]
        {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
         0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
         0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
         0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F}, "AES");
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(Cipher.ENCRYPT_MODE, key256);
cipher.doFinal(data);
```

- Throws `java.security.InvalidKeyException` if unlimited strength cryptography is unavailable

Adding JCA Service Providers

- JCA security providers can be added:
    - As part of Java SE for all applications to use
    - As part of a single application
- Adding a provider in a Java SE installation requires:
    - Writing the provider's JAR file to:
      `<java-home>/lib/ext/`
    - Adding an entry to the text file:
      `<java-home>/lib/security/java.security`

      e.g. ...

- Updated `<java-home>/lib/security/java.security` file:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
security.provider.11=
                org.bouncycastle.jce.provider.BouncyCastleProvider
```

- Add providers to the bottom of the list, i.e. use the next available precedence number, because there is a good deal of software that depends on the default order!

- Adding a provider as part of a single application requires:

    - Adding the provider's JAR file application's class path

    - Loading the provider programmatically:

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;

int nPredecence =
        Security.addProvider(new BouncyCastleProvider());
```

- Typically use a static initializer so that the provider always loads before it's needed and loads only once:

```
public class AddBouncyCastleProviderOnce
{
  static
  {
     Security.addProvider(new BouncyCastleProvider());
  }

  public static void main(String[] args)
  {
     Provider provider = Security.getProvider("BC");
     System.out.println((provider == null ?
                         "Bouncy Castle not available" :
                          provider.getName() + " " +
                          provider.getVersion() + " installed"));
  }
}
```

```
BC 1.48 installed
```

The *Legion of the Bouncy Castle* JCA Cryptographic Service

- The Legion of the Bouncy Castlee cryptography software is developed in Australia (since the year 2000) and thereby circumvents the United States' export restrictions, though this is no longer a problem
- The BC software includes a comprehensive JCA service provider that supports all the popular cryptographic algorithms

- The Legion of the Bouncy Castle software can be downloaded from:
  `www.bouncycastle.org/latest_releases.html`

- Choose the `crypto-148.tar.gz` or `crypto-148.zip` file to "grab the lot"!

- Extract the downloaded file; the Bouncy Castle provider comprises the file `bcprov-jdk15on-148.jar` * found in the `jars` subdirectory

* `bcprov-ext-jdk15on-148.jar` supports IDEA and NTRU which you are unlikely to use, due to patent restrictions

Java Cryptography

Symmetric Key Cryptography

Jeff Lawson

Contents

Information in the Clear

- Corporate networks typically contain file servers with shares that:
    - Require authentication and authorization for access
    - Supply data 'in the clear'
- Protocol analyzers (packet sniffers) can log data from networks: most of this data is readily accessed, in fact it's often used for debugging
- Most transmitted information is human-readable

Cryptography

- Encryption is a process intended to make information secret and may be achieved in several ways, e.g.
    - Caesar transposition of letters
    - Enigma machine
    - Computational algorithm (cipher)
- Symmetric key (a.k.a. secret key) cryptography uses the same key to both encrypt and decrypt data
- Virtually all encrypted data uses symmetric keys

- *Cryptographic keys:*
    - Are special numbers required to customize ciphers
    - Longer keys produce more secure cipher-data
- *Attacks* are attempts at compromising security made by unauthorized agents
- Weak security schemes are easily exploited,
  e.g. human deficiencies such as poor key-management
- Brute-force attacks cycle through each possible key-value until the correct one is found…this might take billions of years!
- Side channel attacks exploit physical characteristics
  e.g. timer-based

- Cryptographic algorithms are often called *ciphers*

- The word *cipher* also refers to the data emitted by a cryptographic algorithm

- Common symmetric key ciphers:

    - DES—1970s Data Encryption Standard: 56-bit

    - TripleDES—three keys; used by FBI: 112- and 168-bit

    - Blowfish—Bruce Schneier*: 32- to 448-bit

    - Twofish—Bruce Schneier: 128-, 192-, 256-bit

    - RC2/4/5/6†—Ronald Rivest: various strengths

    * "Applied Cryptography" † Ron's Code

Advanced Encryption Standard

- Advanced Encryption Standard (AES):
  - Algorithm selected by competition
    - publicly disclosed encryption algorithm
    - royalty-free worldwide
    - Rijndael (pronounced "Reign Dahl")
    - Belgian university originators, VincentRijmenand Joan Daemen
- Officially accepted 26 May 2002
- 128-, 192- or 256-bit key lengths

- The National Institute of Standards and Technology (NIST) publishes Federal Information Processing Standards (FIPS)

- AES is the official NIST successor to DES

- FIPS PUB 197 describes a cipher called the Advanced Encryption Standard (AES), developed by Belgian cryptographers, Joan Daemen and Vincent Rijmen

- FIPS PUB 197 can be downloaded from:

csrc.nist.gov/publications/fips/fips197/fips-197.pdf

- AES is a successor to the Data Encryption Standard (DES), the official U.S. Government adopted cipher for protecting classified information
- DES uses 56-bit keys,* AES uses 128-, 192- or 256-bit keys
- (minimum 128-bit for SECRET, 192-bit for TOP SECRET)
- DES handles blocks of 64 bits, AES handles 128 bits

* 168-bit keys for 3DES

Block Cipher Modes

- A block-based cipher encrypts information in a single block
- Attacks can look for patterns across several blocks encrypted with the same key
- To thwart such attacks, random data is introduced to add noise to the input blocks
- Such randomness is codified in a *mode of operation* or *block cipher mode*

- Electronic Code Book (ECB) is the simplest cipher mode, input is simply split into blocks and no randomness is added

- Cipher Block Chaining (CBC) uses:

    - Random data (called an *initialization vector*) to XOR with the first input block

    - The previous output block to XOR with the next input block

        - The same process to decrypt

- Other cipher modes: Cipher Feedback (CFB), Propagating Cipher-Block Chaining (PCBC), Output Feedback (OFB), Counter (CTR)

Block Cipher Padding

- Encryption algorithms typically handle data in blocks
  e.g. DES uses a block size of 8 bytes,
  AES uses a block size of 16 bytes

- Data that does not have a length that is a whole number of blocks must be padded with extra bytes

- PKCS #7 is a commonly-used padding mechanism* in which the last $1 \leq n \leq 8$ bytes have a value of n
  e.g. if 3 bytes of padding are required then the last three bytes will be 3, 3, 3
  (8, 8, 8, 8, 8, 8, 8, 8 used when no padding is required!)

  * Public-Key Cryptography Standard

Cipher Names

- The FIPS PUB 197 document provides *test vectors* (correct output for a given input and key) for AES with several key lengths using ECB mode and no padding (just a single 16-byte block)

- A *cipher name* is used to specify the characteristics of the desired cryptographic operation and takes the form:

```
<algorithm><mode><padding>
```

e.g. `"DES/ECB/NoPadding"`
`"AES/CBC/PKCS7Padding"`

- In Java, cipher objects are obtained from a factory by specifying a cipher name and, optionally, a provider:

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding",
                                   "BC");
```

- The Java documentation from Oracle recommends *not* specifying a provider, thereby enabling the system to be configured independently of an application

- Most developers will want to specify a known provider in which they have confidence!

- A pre-defined symmetric key can encapsulated in a `SecretKeySpec` object

- The key bit-length is implicit from the supplied byte array, e.g. 128-bit (16 byte) key for AES

```
byte[] bytesFIPS197Key128 = new byte[] {0x00, 0x01, 0x02, 0x03,
                                        0x04, 0x05, 0x06, 0x07,
                                        0x08, 0x09, 0x0a, 0x0b,
                                        0x0c, 0x0d, 0x0e, 0x0f};
Key key128AES = new SecretKeySpec(bytesFIPS197Key128, "AES");
```

- A `Cipher` object is initialized with a key and one of the following operations:

    - `ENCRYPT_MODE`

    - `DECRYPT_MODE`

    e.g. `cipher.init(Cipher.ENCRYPT_MODE, key128AES);`

- Encryption can proceed as one or more update() calls followed by a `doFinal()` call thereby allowing repeated use of small buffers, e.g.

```
int nCipherLen = cipher.update(bytesFIPS197PlainText,
                                    0, bytesFIPS197PlainText.length,
                                    cipherText, 0);
nCipherLen += cipher.doFinal(cipherText, nCipherLen);
```

- Symmetric Key Encryption in Java

```
Key key128AES = new SecretKeySpec(bytesFIPS197Key128, "AES");
Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");
byte[] cipherText = new byte[bytesFIPS197PlainText.length];
cipher.init(Cipher.ENCRYPT_MODE, key128AES);
int nCipherLen = cipher.update(bytesFIPS197PlainText, 0,
                               bytesFIPS197PlainText.length,
                               cipherText, 0);
nCipherLen += cipher.doFinal(cipherText, nCipherLen);
```

See the sample project `Symmetric Key AES FIPS-197`

- Symmetric Key Decryption in Java

```
Key key128AES = new SecretKeySpec(bytesFIPS197Key128, "AES");
Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding", "BC");
byte[] bytesDecryptedText = new byte[nCipherLen];
cipher.init(Cipher.DECRYPT_MODE, key128AES);
int nDecryptedTextLen = cipher.update(cipherText, 0, nCipherLen,
                                      bytesDecryptedText, 0);
nDecryptedTextLen += cipher.doFinal(bytesDecryptedText,
                                    nDecryptedTextLen);
```

- When padding is requested, the encrypted output is unlikely to be the same length as the clear input

- A Cipher object will supply the output length, e.g.

```
byte[] byteMessage = s_strMessage.getBytes("UTF-8");
cipher.init(Cipher.ENCRYPT_MODE, key192AES);
byte[] cipherText =
                new byte[cipher.getOutputSize(byteMessage.length)];
...
String strDecryptedText = new String(bytesDecryptedText, 0,
                                     nDecryptedTextLen, "UTF-8");
```

Initialization Vectors

- For block modes that require an initialization vector, this must be generated and used for both encryption and decryption, e.g.

```
byte[] bytesIV = new byte[] {0x00, 0x01, 0x02, 0x03,
                             0x04, 0x05, 0x06, 0x07,
                             0x08, 0x09, 0x0a, 0x0b,
                             0x0c, 0x0d, 0x0e, 0x0f};
IvParameterSpec iv = new IvParameterSpec(bytesIV);
cipher.init(Cipher.ENCRYPT_MODE, key192AES, iv);
...
cipher.init(Cipher.DECRYPT_MODE, key192AES, iv);
```

Random Keys and Initialization

- The keys and initialization vectors we have seen so far have been hard-coded!

- In general, we should generate these values using a cryptographically strong random number generator using unpredictable seeds

- The NIST standard in section 4.7.1 of FIPS PUB 140-2:

  `csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`

- The `SecureRandom` class is typically used to generate pseudo-random numbers

- Each new SecureRandom object uses a new seed and should, therefore, be re-instantiated when convenient

- The seed is generated the first time the SecureRandom object is used, e.g. (see the sample project Symmetric Key Generator)

```
SecureRandom sr = new SecureRandom();
byte[] bytesIV = new byte[16];
sr.nextBytes(bytesIV);                    // seed on first use
```

- Random keys can be generated with a KeyGenerator object, e.g.

```
KeyGenerator kg = KeyGenerator.getInstance("AES", "BC");
kg.init(256);
Key key256AES = kg.generateKey();
```

Java Cryptography

Symmetric Key Cryptography for Android and iOS

Contents

Cryptography for Android

- Android makes use of a cut-down, old version of the Bouncy Castle provider

- Name-collisions prevent us from using an alternative version of the Bouncy Castle provider

- The Spongy Castle project enables us to get around this problem by re-naming `org.bouncycastle.*` packages to `org.spongycastle.*` :

  `rtyley.github.com/spongycastle/`

- In general, download the Bouncy Castle software and run the `become-spongy.sh` script

- For simplicity in this training course, we will download the pre-built Spongy Castle JAR files:
  - `scprov-jdk15on-1.47.0.2.jar`
  - `sc-light-jdk15on-1.47.0.2.jar`
- In an Android project, copy these JAR files to the libs directory
- Load the Bouncy Castle provider in almost the usual way:

```
import org.spongycastle.jce.provider.BouncyCastleProvider;

static
{
  Security.addProvider(new BouncyCastleProvider());
}
```

- Write the same Java crypto code but specify the provider as `"SC"` rather than `"BC"`:

```
KeyGenerator kg = KeyGenerator.getInstance("AES", "SC");

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding", "SC");
```

Cryptography for iOS

- iOS app development typically uses Apple's Xcode IDE with Objective-C

- We do not have access to the Bouncy Castle software but we can use native libraries and, provided we target the same algorithms, our data will be inter-operable

- Since iOS 5, some algorithms, e.g. AES/CBC have hardware acceleration

- There is no native cryptography implementation that is usable across all versions of iOS

- We will make use of OpenSSL

- OpenSSL was first released (version 0.9.1c) in 1998 and is available from:

  openssl.org

- OpenSSL provides an comprehensive suite of cryptographic algorithms and has been widely adopted and scrutinized by academic researchers

- Like Bouncy Castle, access to the source code ensures that there are no 'back-doors'

- Stefan Arentz makes a pre-built iOS-specific version of OpenSSL available from:

  github.com/st3fan/ios-openssl

- As with all cryptography software, due diligence must be carried out in obtaining the OpenSSL!

- To use OpenSSL in an iOS project:
    - Use the *Add Files* option to copy the `openssl` include folder to your project
    - Identify the folder that contains the openssl folder and add it to the *Header Search Paths* build setting,
      e.g. For `Classes/openssl` add `$(SRCROOT)/Classes`
    - Under *Build Phases*, *Link Binary With Libraries,* add the two OpenSSL libraries `libcrypto.a` and `libssl.a`
- `libcrypto.a` and `libssl.a` from Stefan Arentz target arm7 and i386 so they can be used on ARM 7 iOS devices and in the iOS Simulators
- Typically, write C functions and call them from Obj-C

- OpenSSL provides a high-level API called *envelope encryption* or EVP

- To encrypt:

```
#import <openssl/evp.h>
unsigned char* encryptSymmetric(unsigned char chPlain[],
                                int nInputLength,
                                int* pnOutputLength,
                                unsigned char** ppchKey,
                                unsigned char** ppchIV)
{
  // Prepare key and IV
  // ...

  // Create buffer for output
  // ...

  // Encrypt
  // ...

  // Return cipher text
}
```

- To encrypt with a prepared 256-bit AES key and IV using CBC:

```
// Prepare key and IV

char chKey[] = {"F32A76167C91FE311DD2CE3888BDB0FC
                  BC7FEEB5F1DEB60BFFB584A561733BAF"};
char chIV[] = {"C700C1CDC7DCF34517C66DC06EB5FFC0"};

unsigned char* pchKey = (unsigned char*)malloc(32);
toData(chKey, 32 * 2, pchKey);
*ppchKey = pchKey;

unsigned char* pchIV = (unsigned char*)malloc(16);
toData(chIV, 16 * 2, pchIV);
*ppchIV = pchIV;
```

- To encrypt with a prepared 256-bit AES key and IV using CBC (continued):

```
// Create buffer for output
// -- must be at least one block longer than nInputLength
unsigned char* pchOut = (unsigned char*)malloc(nInputLength + 16);
int nEncryptLen1;
int nEncryptLen2;

// Encrypt
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit(&ctx, EVP_aes_256_cbc(), pchKey, pchIV);
EVP_EncryptUpdate(&ctx, pchOut, &nEncryptLen1,
                  chPlain, nInputLength);
EVP_EncryptFinal(&ctx, pchOut + nEncryptLen1, &nEncryptLen2);
EVP_CIPHER_CTX_cleanup(&ctx);

// Return cipher text
*pnOutputLength = nEncryptLen1 + nEncryptLen2;
return pchOut;
```

- To decrypt:

```
unsigned char* decryptSymmetric(unsigned char chCipher[], int
nInputLength, int* pnOutputLength, unsigned char* pchKey, unsigned
char* pchIV)
{
    // Create buffer for output
    // ...

    // Decrypt cipher text
    // ...

    // Return recovered text
    // ...
}
```

- To decrypt (continue):

```
// Create buffer for output
unsigned char* pchOut = (unsigned char*)malloc(nInputLength);

// Decrypt cipher text
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
EVP_DecryptInit(&ctx, EVP_aes_256_cbc(), pchKey, pchIV);
EVP_DecryptUpdate(&ctx, pchOut, &nDecryptLen1,
                  chCipher, nInputLength);
EVP_DecryptFinal(&ctx, pchOut + nDecryptLen1, &nDecryptLen2);
EVP_CIPHER_CTX_cleanup(&ctx);

// Return recovered text
*pnOutputLength = nDecryptLen1 + nDecryptLen2;
return pchOut;
```

- Invoke the encryption and decryption methods:

```
int nEncryptedLength;
unsigned char* pchKey;
unsigned char* pchIV;

unsigned char* pchEncrypted =
        encryptSymmetric((unsigned char*)"The quick brown fox...",
                         23, &nEncryptedLength, &pchKey, &pchIV);

int nDecryptedLength;
unsigned char* pchDecrypted =
             decryptSymmetric(pchEncrypted, nEncryptedLength,
                              &nDecryptedLength, pchKey, pchIV);

fprintf(stdout, "Decrypted: %s", pchDecrypted);

free(pchDecrypted);
free(pchEncrypted);
free(pchKey);
free(pchIV);
```

- To generate cryptographically random data, OpenSSL provides `RAND_bytes()` which automatically seeds:

```
#import <openssl/rand.h>

unsigned char* pchKey = (unsigned char*)malloc(32);
RAND_bytes(pchKey, 32);
*ppchKey = pchKey;

unsigned char* pchIV = (unsigned char*)malloc(16);
RAND_bytes(pchIV, 16);
*ppchIV = pchIV;
```

Java Cryptography

Asymmetric Key Cryptography

Jeff Lawson

Contents

- Asymmetric Key Cryptography
- Transmission of Confidential Information

Asymmetric Key Cryptography

- Asymmetric key (a.k.a. dual-key) cryptography:
    - Uses matched key-pairs: one key is used to encrypt and the other key is used to decrypt (or vice versa!)
    - One key is designated as a Private Key
        - in principle, the private key never leaves the computer on which the key-pair is generated
    - The other key is designated as the Public Key
        - the public key is freely distributed to anyone

- Simple transmission of confidential data:
    - sender encrypts with the receiver's public key
    - receiver decrypts with their private key
- Asymmetric key ciphers:
    - RSA—Rivest-Shamir-Adleman
    - DSA—Digital Signature Algorithm: NIST
    - Diffie-Hellman

- Key-pair generation in Java:

```
SecureRandom rand = new SecureRandom();

KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "BC");
kpg.initialize(2048, rand);
KeyPair kp = kpg.generateKeyPair();

PublicKey keyPublic = kp.getPublic();
PrivateKey keyPrivate = kp.getPrivate();
```

- Encryption with public key:

```
Cipher cipherEncrypt = Cipher.getInstance("RSA/NONE/OAEPPadding",
                                                    "BC");
cipherEncrypt.init(Cipher.ENCRYPT_MODE, keyPublic, rand);
byte[] bytesEncrypted = cipherEncrypt.doFinal(bytesOriginal);
```

- Decryption with private key:

```
Cipher cipherDecrypt = Cipher.getInstance("RSA/NONE/OAEPPadding",
                                                    "BC");
cipherDecrypt.init(Cipher.DECRYPT_MODE, keyPrivate);
byte[] bytesDecrypted = cipherDecrypt.doFinal(bytesEncrypted);
```

Transmission of Confidential Information

- Since symmetric cryptography presents a difficulty in transmitting the secret key, why not just use only asymmetric cryptography?
    - Because it’s an order of magnitude slower!
- Use both asymmetric and symmetric ciphers:
    - A sender's secret key is encrypted with a recipient's public key

- To transmit a message using symmetric with asymmetric ciphers:
    - create a secret key on the fly for each message
    - encrypt data with secret key
    - encrypt secret key with receiver's public key
    - send encrypted data and encrypted secret key
    - receiver decrypts the secret key with their private key
    - receiver decrypts data with the secret key

Java Cryptography

Digital Signatures

Jeff Lawson

Contents

Message Authenticity and Integrity

- When someone sends you data how do you know:
    - It really came from the declared sender
    - It was not tampered with during transmission
- Possible solution:
    - The sender encrypts the data with their private key
    - If the receiver can successfully decrypt the data with the sender's public key, it must have been sent by the declared sender and it wasn't tampered with!

- Problems:
    - Encryption consumes processor time
    - Anyone with access to the sender's public key can decrypt the message

- Solution:
    - Digital signatures...

...but, first...

Hash Functions

- A hash function is an algorithm that takes any amount of data and generates a relatively small number (a hash value or key) in a way designed to avoid different sets of input resulting in the same output

- When collisions occur, additional steps can be taken to disambiguate the input

- A data structure called a *hash table* contains key/value pairs, where the 'key; is the hash value and the 'value' is the original data

- Hash tables enable us to reference large data values using small keys

Cryptographic Hash Functions

- Cryptographic hash functions are designed to be:
  - Easy to compute
  - Infeasible to construct a message that matches a pre-defined hash
  - Infeasible to modify the input data without changing the hash
  - Impossible to recover the input data from the hash
- Hence, tampering is not achievable
- These hash values are called *messages digests* or, simply, *digest*

- There are several cryptographic hash functions in common use:
    - MD5—Message Digest 5 produces 128-bit digests—flawed: don't use!
    - SHA-1—Secure Hash Algorithm 1 produces 160-bit digests—although used widely, it has been shown to have vulnerabilities and should not be used
    - SHA-2—actually: SHA-244, SHA-256, SHA-384, SHA-512 are thought to be safe
    - SHA-3—in development and not needed (yet!)

Hash-based Message Authentication Codes

- To authenticate messages, we could use a Hash-based Message Authentication Code (HMAC) which is a message digest encrypted with a symmetric key (that may be the same of different from the one used to encrypt the message)

- A MAC enables us to know that a message was not changed and, if the key was correctly agreed upon, that it came from the sender, i.e message integrity and authenticity

- We cannot *prove* that a message came from the sender because the key is not identified with the sender, i.e. We do not have non-repudiation

Digital Signatures

- A digital signature is a message digest that has been encrypted with the private key of the sender
- The digital signature is included as part of the transmitted message
- Because digital signatures can be validated only by using the sender's public key, they provide:
  - Data integrity—the massage was not tampered with
  - Data authenticity—we know who sent it
  - Non-repudiation—the sender cannot deny sending

- To sign a message, we need the private key of the sender and the signature specification, e.g. `SHA512withRSA`:

```
byte[] bytesMessage = strMessage.getBytes("UTF-8");
Signature sigSender = Signature.getInstance("SHA512withRSA",
                                            "BC");
sigSender.initSign(privKeySigning);
sigSender.update(bytesMessage);
byte[] bytesSignature = sigSender.sign();
```

- To verify a digital signature, we need the public key of the sender:

```
Signature sigReceiver = Signature.getInstance("SHA512withRSA",
                                              "BC");
sigReceiver.initVerify(pubKeyVerifying);
sigReceiver.update(bytesMessage);
boolean bValid = sigReceiver.verify(bytesSignature);
```

Java Cryptography

Authenticated Encryption

Jeff Lawson

Contents

- Authenticated Encryption
- GCM Cryptography with Bouncy Castle

Authenticated Encryption

- In the past decade new cryptographic algorithms have emerged that combine:
    - Confidentiality—encryption
    - Integrity—hashing
        - Authenticity—hashing
- These algorithms provide authenticated encryption through Authenticated Encryption with Associated Data (AEAD) block cipher modes

- The most popular AEAD modes are:
    - OCB (Offset Codebook Mode)—fastest; patented but free for use in non-military software, i.e. not for military and not in hardware:
      www.cs.ucdavis.edu/~rogaway/ocb/license.htm
    - CCM (Counter with CBC-MAC)—slow
    - EAX—designed to replace CCM;
      www.cs.ucdavis.edu/~rogaway/papers/eax.pdf
    - CWC (Carter–Wegman + CTR mode)
    - GCM (Galois/Counter Mode)—fast; most popular (GMAC is an authentication-only mode)

- In conclusion:
    - Use OCB if you are sure that your software will not be used for military purposes or if you don't mind paying a licence fee
    - Otherwise, use GCM like most other people do
- Bouncy Castle does not support OCB
- Bouncy Castle's GCM implementation does not support streaming (it's on David Hook's ToDo list)

GCM Cryptography with Bouncy Castle

- The GCM software is easy to write

- Prepare initialization vector and 256-bit AES key:

```
SecureRandom sr = new SecureRandom();
byte[] bytesIV = new byte[16];
sr.nextBytes(bytesIV);
IvParameterSpec iv = new IvParameterSpec(bytesIV);

KeyGenerator kg = KeyGenerator.getInstance("AES", "BC");
kg.init(256);
Key key256AES = kg.generateKey();
```

- Specify the "AES/GCM/NoPadding" configuration—note that PKCS5Padding is not used!

```
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");
```

- Encrypt as usual:

```
byte[] byteMessage = s_strMessage.getBytes("UTF-8");
cipher.init(Cipher.ENCRYPT_MODE, key256AES, iv);

byte[] cipherText =
            new byte[cipher.getOutputSize(byteMessage.length)];

int nCipherLen = cipher.update(byteMessage, 0,
                               byteMessage.length,
                               cipherText, 0);
nCipherLen += cipher.doFinal(cipherText, nCipherLen);
```

- Decrypt as usual:

```
byte[] bytesDecryptedText = new byte[nCipherLen];

cipher.init(Cipher.DECRYPT_MODE, key256AES, iv);
int nDecryptedTextLen = cipher.update(cipherText, 0, nCipherLen,
                                      bytesDecryptedText, 0);
nDecryptedTextLen += cipher.doFinal(bytesDecryptedText,
                                    nDecryptedTextLen);

String strDecryptedText = new String(bytesDecryptedText, 0,
                                     nDecryptedTextLen, "UTF-8");
System.out.println("Decrypted text: " + strDecryptedText);
```

Java Cryptography

Digital Certificates

Jeff Lawson

Contents

Public Key Distribution

- If I give you my public key, you can:
    - Send me confidential information (encrypted)
    - Verify that information I send you came from me
- Why do we need these security measures?
    - Because we suspect the communication mechanism is vulnerable to snooping and tampering
- If that is the case, then it must be possible for an attacker to intercept transmission of my public key and forward to you *their* public key—man-in-the-middle attack!

Digital Certificates

- A digital certificate is a collection of information and other data that has been digitally signed by a trusted authority

- A digital certificate typically contains:

    - Public key(s)

    - Identity of the owner (subject)

    - Identity of the issuer

    - Intended use, e.g. DSA public keys can only be used for verifying signatures, not for encryption

- Digital certificates are the main technology used in:
    - Public Key Infrastructure (PKI)
    - Privilege Management Infrastructure (PMI)
- Consequently, there are two types of digital certificate:
    - Public key certificates for PKI
    - Attribute certificates for PMI
- X.509 is an ITU-T* standard for PKI and PMI
- The Public-Key Infrastructure (X.509) working group is known as PKIX—see `datatracker.ietf.org/wg/pkix/`

* International Telecommunication Union standard for telecommunications

X.500 and Distinguished Names

- X.500 is a series of standards for directory services
- Originally, OSI-based, e.g. Directory Access Protocol (DAP) but now TCP/IP-based, e.g. LDAP
- X.500 information forms a single tree (Directory Information Tree)—originally just one but now many
- Each item of information is an entry in the tree, located by and identified by a unique distinguished name (DN)
- Distinguished names are composed of tree nodes called Relative Distinguished Names (RDN)

- Distinguished names identify objects of interest:
    - Principals, e.g. a person, a group or an organization
    - Equipment, e.g. an e-mail server or a printer
    - Anything else! e.g. a document, a movie, a drill-bit
- Example DN:
  `"CN=www.cogentlogic.com, OU=Cogent Logic Ltd., O=Company, C=UK"`
- Each `"X=value"` is a RDN and comprizes:
    - Attribute type, which is an ASN.1* Object Identifier (OID), usually aliased to a keyword e.g. `"OU"`
    - Attribute value, e.g. `"Cogent Logic Ltd."`

    * Abstract Syntax Notation 1

- Directory objects (entries) can have the following relationships with real-world things:
    - One-to-one, e.g. my web server
    - One-to-many, e.g. contacts in my address book
    - Many-to-one, e.g. me as a customer, me as an employee, me as a software developer

- To make Directory Information Trees more manageable, RFC 2253 introduces restrictions on:
    - OID types
    - Values (UTF-8 and escape sequences, e.g. `"\,"` for `","`)
    - DN and RDN element structure

- Common attribute type keywords and their OIDs:
    - CN—commonName, "2.5.4.3"
    - OU—organizationalUnit, "2.5.4.11"
    - O—organizationName, "2.5.4.10"
    - C—country, "2.5.4.6"
    - L—localityName, "2.5.4.7"
    - ST—stateOrProvinceName, "2.5.4.8"
    - DN—distinguishedName, "2.5.4.49"
    - DC—domainComponent, "0.9.2342.19200300.100.1.25"
- Note: a DN identifies an object; the object itself can have zero or more attributes

- The OID structure for the preceding types is:

    Top of OID tree

        2 - ISO/ITU-T jointly assigned OIDs

            2.5 - X.500 Directory Services

                2.5.4 - X.500 attribute types

- For other attribute types, see the drop-down list at:

    `oid-info.com/get/2.5.4`

- A domain name example, cogentlogic.com:

    `"DC=cogentlogic, DC=com"`

- A DN might be:

    `"CN=Cogent Logic Ltd., DC=cogentlogic, DC=com"`

- Organizations can request an OID from oid-info.com

- Assignations follow:

iso(1)
identified-organization(3)
dod(6)
internet(1)
private(4)
enterprise(1)

- E.g. Cogent Logic Corporation : 1.3.6.1.4.1.14981

- Once you have an enterprise OID you can create your own OIDs, e.g.
  MIMUID—myIMinesUserID, "1.3.6.1.4.1.14981.10.1"

X.509

- X.509 is the X500 standard for authentication and describes public key certificates and attribute certificates

- X.509 version 3 certificates use distinguished names to identify principals, e.g. subject and issuer

- X.509v3 public key certificates contain a public key that the certificate issuer (a Certificate Authority) asserts is owned by the subject

- X.509v3 public key certificates are signed by the issuer using their private key; the issuer's public key is held in their self-signed root certificate

- X.509 public key certificates contain:
    - Issuer—distinguished name
    - Subject—distinguished name
    - The subject's public key
    - Serial number: any number as long as it is not re-used for this type of certificate, e.g. a timestamp or counter
    - Date before which the certificate is invalid
    - Date after which the certificate is invalid
    - Signature algorithm, e.g. `SHA512WITHRSA`
    - Usage attributes—what the certificate is intended to be used for, e.g. encrypting information
    - Signature

X500Principle Java Objects

- X.500 DNs are represented in Java as `X500Principle` objects (or Bouncy Castle `X500Name` objects)

- Note: reading a name from a `X500Principle` object (`getName()`) and using the name to create a new `X500Principle` object will not necessarily produce an exact copy of the object!

- Use `getName()` *only* when a human-readable version of the DN is needed

- To transmit a DN, use `getEncoded()`, which returns a DER* encoded string

* Distinguished Encoding Rules

- To create a subject, invoke a `javax.security.auth.x500.X500Principal` constructor, e.g.

```
X500Principal x500PrincipleSubject =
          new X500Principal("CN=My iMine CA, DC=myimine, DC=com,
                            OU=My iMine Trust Authority,
                            O=Cogent Logic Ltd., C=UK");
```

- Alternatively, use `org.bouncycastle.asn1.x500.X500NameBuilder`, e.g.

```
X500NameBuilder nbSubject = new X500NameBuilder(BCStyle.INSTANCE);
nbSubject.addRDN(BCStyle.O, "Cogent Logic Ltd.");
nbSubject.addRDN(BCStyle.L, "Tideswell");
nbSubject.addRDN(BCStyle.ST, "Derbyshire");
nbSubject.addRDN(BCStyle.C, "UK");
X500Name x500Name = nbSubject.build();
```

X.509v3 Certificates in Java

- End user or client certificates are called *end entity certificates*
- To issue our own end entity certificates we must first become a Certificate Authority
- We do this by creating a self-signed root certificate, i.e. a certificate that:
  - Contains our public key
  - Contains our identity as the issuer *and* the subject
  - Is flagged as a CA certificate
  - Is signed with our private key

- In preparation for creating a certificate, we must decide upon a scheme for generating serial numbers

- We generate serial numbers as needed but each one must be unique within the set certificates of a particular type

- For example, all our root certificates must have different serial numbers but this group of numbers is distinct from those used for root certificates by *other* CAs and distinct from the end entity certificates that we create

- It is common for a timestamp to be used to create a serial number; alternatively, a simple sequence will work too:

```
BigInteger biSerialNumber =
                BigInteger.valueOf((System.currentTimeMillis()));
BigInteger biSerialNumber = new BigInteger("1");
```

- Certificates also require dates between which they are valid, e.g

```
int nYearsValid = 20;
Date dateNotBefore = new Date(System.currentTimeMillis());
Date dateNotAfter = new Date(System.currentTimeMillis() +
                        (1000L * 60 * 60 * 24 * 365 * nYearsValid));
```

- For root certificates, the same `X500Principal` can be used for issuer and subject, e.g.

```
X500Principal x500PrincipleRootDN =
      new X500Principal("CN=My iMine CA, DC=myimine, DC=com,
      OU=My iMine Trust Authority,
      O=Cogent Logic Ltd., C=UK");
```

- Certificates can be generated from a `org.bouncycastle.cert.jcajce.JcaX509v3CertificateBuilder` which creates a certificate within a X509CertificateHolder, e.g.

```
JcaX509v3CertificateBuilder x509Builder =
            new JcaX509v3CertificateBuilder(x500PrincipleRootDN,
                                            biSerialNumber,
                                            dateNotBefore,
                                            dateNotAfter,
                                            x500PrincipleRootDN,
                                            keyPublicRoot);
X509CertificateHolder x509Holder =
                x509Builder.build(sb.build(keyPrivateRoot));
```

- The holder has the advantage of making the content of a certificate more easily accessible

- To extract the certificate:

```
CertificateFactory cf = CertificateFactory.getInstance("X.509",
                                                       "BC");
X509Certificate x509Cert =
        (X509Certificate)cf.generateCertificate(
          new ByteArrayInputStream(x509Holder.getEncoded()));
```

- The validity and authenticity of a certificate can easily be checked (exceptions thrown on error):

```
x509Cert.checkValidity();                  // currently within dates?
x509Cert.verify(keyPublicRoot, "BC");   // signature OK?
```

The public key used here belongs to the pair whose private key was used for signing!

This is the whole point of a certificate: confidence that the contained public key belongs to the subject

- Accessing the content of a certificate directly:

```
X500Principal prinIssuer = x509Cert.getIssuerX500Principal();
X500Principal prinSubject = x509Cert.getSubjectX500Principal();
System.out.println("Issuer: " + prinIssuer.getName());
System.out.println("Subject: " + prinSubject.getName());
```

- Accessing the content of a certificate from a holder:

```
X500Name nameIssuer = x509CertHolder.getIssuer();
X500Name nameSubject = x509CertHolder.getSubject();
System.out.println("Issuer: " + nameIssuer);
System.out.println("Subject: " + nameSubject);
```

PKCS #7 Cryptographic Message Syntax

- We need to be able to distribute our certificates
- RSA Security, Inc. has published a group of public-key cryptography standards (PKCS)
- PKCS #7 descibes the Cryptographic Message Syntax used for packaging certificates for distribution, often in text files with a `.p7b` file name extension
- Bouncy Castle provides the class:

  `org.bouncycastle.cms.CMSSignedDataGenerator`

- In preparation for CMS, we need to create a certificate store:

```
ArrayList<X509CertificateHolder> alCert =
                          new ArrayList<X509CertificateHolder>();
                                          // N.B. not X509Certificate
alCert.add(x509CertHolder);               // N.B. not x509Cert
Store storeCerts = new CollectionStore(alCert);
```

- Then we can generate the CMS message

```
CMSSignedDataGenerator cmsGen = new CMSSignedDataGenerator();
cmsGen.addCertificates(storeCerts);
CMSTypedData msg = new CMSProcessableByteArray(new byte[0]);
CMSSignedData signedData = cmsGen.generate(msg, false);
byte[] pkcs7 = signedData.getEncoded();
```

- The message can then be transmitted in a data packet or stored in a `.p7b` file, e.g.

```
File fileP7B = new File("c:\\temp\\RootCert.p7b");
if (!fileP7B.exists())
  fileP7B.createNewFile();

BufferedWriter bw =
   new BufferedWriter(new FileWriter(fileP7B.getAbsoluteFile()));
bw.write("-----BEGIN PKCS7-----");
bw.write(new String(Base64.encode(pkcs7)));
bw.write("-----END PKCS7-----");
bw.close();
```

- `.p7b` files can be imported into web browsers, servers, etc.

Certificate Extensions

- The root certificate we just created was pretty crude!
- We can refine our certificate by adding extensions:
    - Flag for use as a CA certificate!!!
    - Restrictions on its use
- The way a certificate is used depends entirely upon the software that uses it!
- For interoperability with third-party software, we need to ensure the content is acceptable for the intended purpose

- The `X509v3CertificateBuilder` method `addExtension()` takes three parameters:
    - `ASN1ObjectIdentifier`—extension type
    - `Boolean`—critical flag
    - `ASN1Encodable`—value
- If the critical flag is false the intention is that software *may* make use of the certificate without recognizing the extension
- If the critical flag is true the intention is that software *must* recognize the extension and act upon its value
- The `X509Extension` class provides `ASN1ObjectIdentifier`s

- The Basic Constraints extension flags a certificate as belonging to a CA (e.g. root) or to a client (end entity), e.g.

```
x509Builder.addExtension(X509Extension.basicConstraints,
                         true, new BasicConstraints(true));
```

- This means: add a Basic Constraints extension that is critical and indicates that the certificate's subject is a CA

- To re-emphasize: extensions can be seen as:
  - A way to bolster commercial interests
  - A way to check that we are following PKI rules
  - Somewhere in between!

- We can specify restrictions on the use of a certificate by adding *key usage* or *extended key usage* extensions

- The `KeyUsage` class enables us to specify any number of:
  - `digitalSignature`
  - `nonRepudiation`
  - `keyEncipherment`
  - `dataEncipherment`
  - `keyAgreement`
  - `keyCertSign`
  - `cRLSign`
  - `encipherOnly`
  - `decipherOnly`

- If we would like to specify that the public key contained in the certificate can be used to check digital signatures, we add `digitalSignature`

- If we intend the key to be used to encrypt a symmetric key, we add `keyEncipherment`

- E.g.

```
x509Builder.addExtension(X509Extension.keyUsage, true,
                         new KeyUsage(KeyUsage.digitalSignature |
                                      KeyUsage.keyEncipherment));
```

- The `ExtendedKeyUsage` class requires `KeyPurposeId` values to be set

- For example, to specify that a key can be used for code signing and for encrypted e-mail:

```
KeyPurposeId[] keyPurposes = {KeyPurposeId.id_kp_codeSigning,
                              KeyPurposeId.id_kp_emailProtection};
jcaX509v3CertificateBuilder.addExtension(
                               X509Extension.extendedKeyUsage,
                               true,
                               new ExtendedKeyUsage(keyPurposes));
```

- Certificate code to this point is in the `X509DigitalCertificate` project

End Entity Certificates

- Now we are a Certificate Authority and we have a root certificate, we can issue end entity (client) certificates

- An end entity certificate is distinguished by:

    - A CA issuer
    - A client subject
    - Basic Constraints that indicate it is not a CA
    - Client-specific key usage
    - Signed by a CA, i.e. not authenticated by the subject's public key

Java Cryptography

PKI

Jeff Lawson

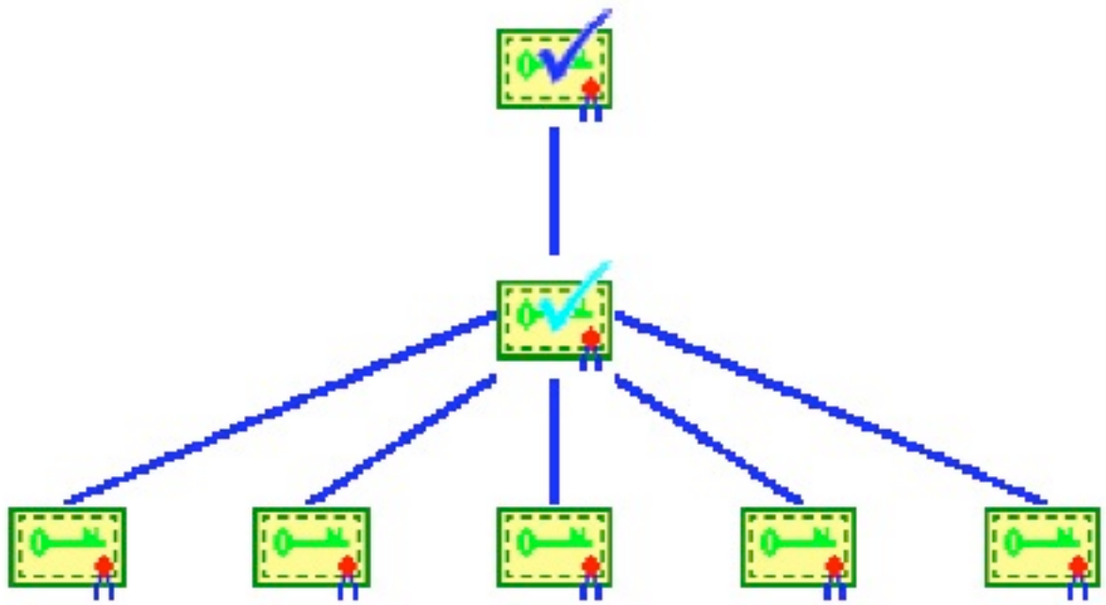
Contents

- Public Key Infrastructure

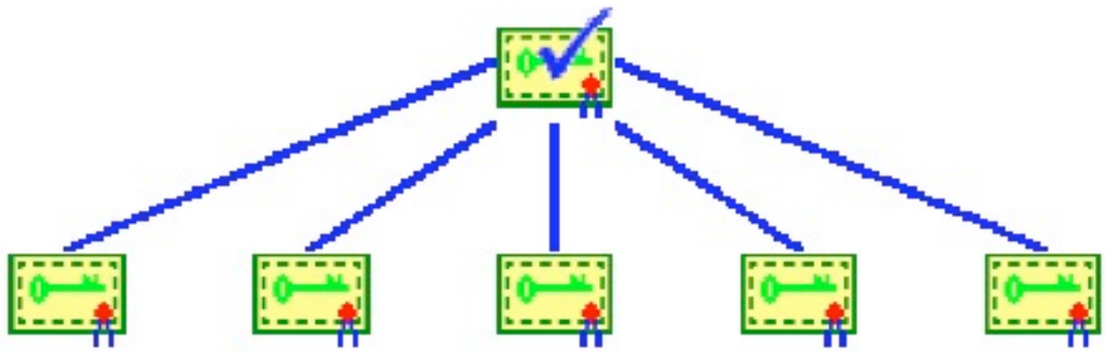
Public Key Infrastructure

- Many companies and organizations may wish to issue their own certificates
- Departments may wish to issue certificates
- Problem:
    - Dependant on communicating with a Certificate Authority for issuance of certificates
- Solution:
    - Act as an Intermediate CA
    - Act as a Root CA

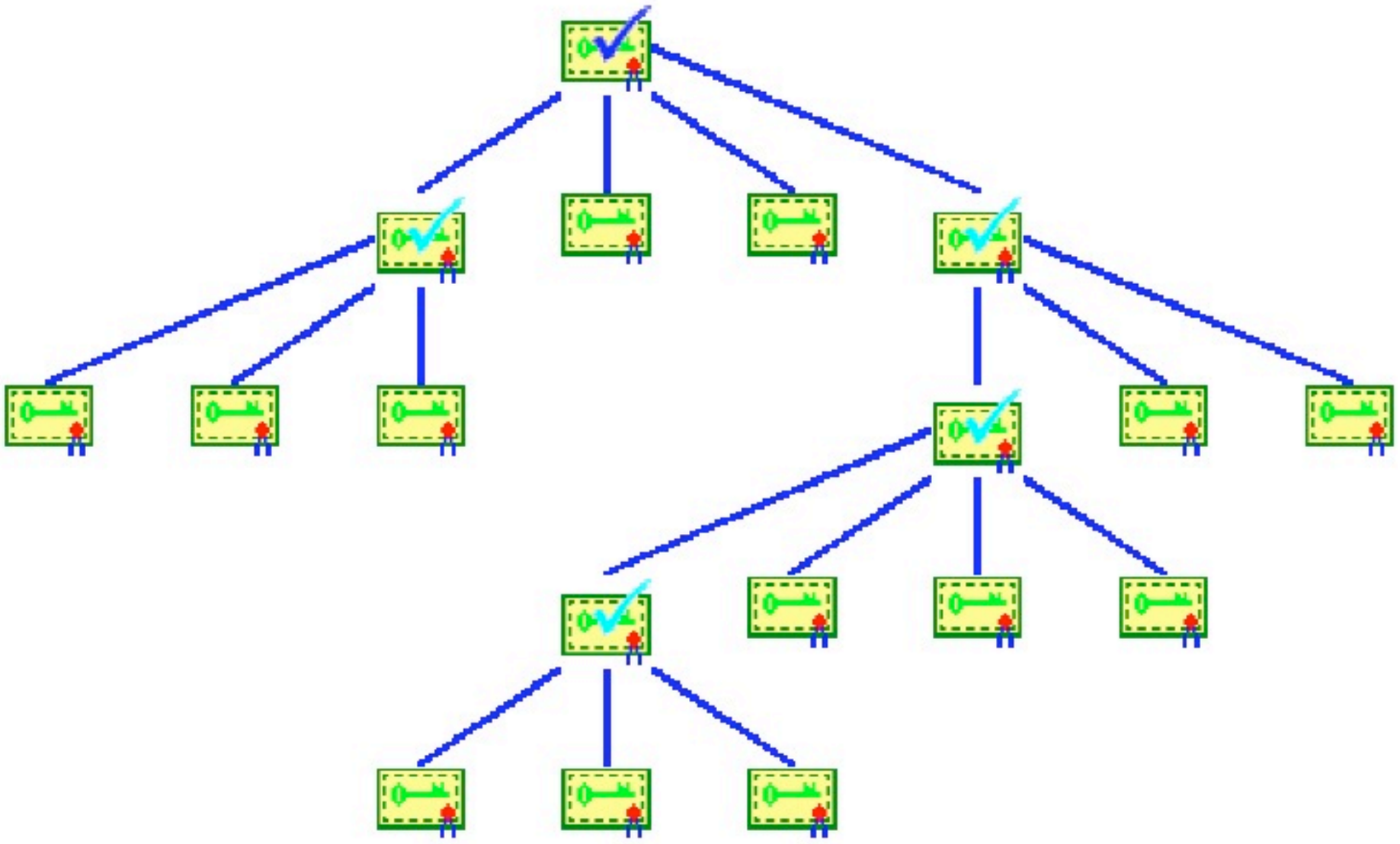
- Intermediate CA:



- Root CA:



- Root CA and with many Intermediate CAs:



- Hence: establish an appropriate Public-Key Infrastructure (PKI)

- In PKI, each certificate is issued by the subject of the next certificate higher up in the tree except for the root which is self-signed

- A *certificate path,* or *certificate chain,* is a collection of certificates: an end-entity certificate following issuing CA certificates to the root

- Certificate paths are often handled in sequence from the end-entity certificate onwards but omitting the root certificate, it being obtained in an especially careful way

- Root certificate are the guarantors of trust that are usually installed in a careful, deliberate step

Java Cryptography

Key Stores and Trust Stores

Jeff Lawson

Contents

Key Stores and Trust Stores

- A key pair is generally handled as:
    - A X.509 Public-Key Certificate Path and
    - A Private Key
- JCA provides Key Stores for holding this data

Some key stores support storage of symmetric keys but this is not ordinarily necessary

- There are two categories of key store:
    - Those that hold a subject's private key and their X.509 certificate path
    - Those that hold the certificates of trusted third-parties The latter does not hold any private keys and is usually referred to as a Trust Store

- Key stores make use of passwords for:
    - The store as a whole
    - Each private key
      (it is usual to keep these passwords synchronized)
- Each entry in a key store is identified by an alias (name)
- Some key store use case-sensitive aliases and other use case-insensitive aliases—recommendation: do not use aliases that differ only by case
- Some certificate tools, e.g. Internet Explorer *Certificates* dialog, show the alias as a 'friendly name'
- The Mac OS X *Keychain Access* shows the CN as 'name' and use this as an alias

Types of Key Store

- Different providers support different types of key store so it's easy for software to encounter key store types it does not recognise!

- Each JVM has a default key store type, set in the java.security file as the `keystore.type` property and retrievable with `KeyStore.getDefaultType()`

- (`"jks"` is not set in the `java.security` file)

- Security providers are required to support the `"PKCS12"` key store type

- PKCS #12 key stores are Java-independent and are usually stored in files with `.p12` or `.pfx` name extensions

Types of *JVM* Key Store

- The SUN 1.7 security provider supports*:
    - *JKS KeyStore*—ci—original Sun-format key store
    - *CaseExactJKS KeyStore*—cs—variant of JKS
- Both use a weak cipher to store data
- The SunJSSE 1.7 security provider supports:
    - *PKCS12 KeyStore*—ci—PKCS #12
- The SunJCE 1.7 security provider supports:
    - *JCEKS KeyStore*—ci—keys protected with Triple-DES

* ci => case-insensitive aliases

Types of *System-Specific* Key Store

- The SunMSCAPI 1.7 security provider supports:
    - *Windows-MY KeyStore*
    - *Windows-ROOT KeyStore*

      both used to access Windows CryptoAPI certificate stores
- The Apple 1.1 security provider supports:
    - *KeychainStore KeyStore*—enables Mac OS X keychain access from Java

Types of *Legion of the Bouncy Castle* Key Store

- The BC 1.48 security provider supports:
    - *BouncyCastle KeyStore*
    - *BKS KeyStore*—cs—keys protected with Triple-DES
    - *BCPKCS12 KeyStore*
    - *PKCS12 KeyStore*
    - *PKCS12-DEF KeyStore*
    - *PKCS12-3DES-40RC2 KeyStore*
    - *PKCS12-3DES-3DES KeyStore*
    - *PKCS12-DEF-3DES-40RC2 KeyStore*
    - *PKCS12-DEF-3DES-3DES KeyStore*

Creating a PKCS #12 Key Store

- Create and initialize the key store:

```
KeyStore ksPKCS12 = KeyStore.getInstance("PKCS12", "BC");
ksPKCS12.load(null, null);
```

- Add entries to the key store:

```
ksPKCS12.setCertificateEntry("Cogent Logic Root CA",
                             cw.getX509RootCA());

ksPKCS12.setCertificateEntry("My iMine Intermediate CA",
                             cw.getX509IntermediateCA());

ksPKCS12.setCertificateEntry("Jeff Lawson End Entity",
                             cw.getX509EndEntity());
```

- Save the key store:

```
FileOutputStream fos =
          new FileOutputStream("c:\\temp\\KeyStorePKCS12.p12");
ksPKCS12.store(fos, "obscure".toCharArray());   // password
fos.close();
```

- The `.p12` file can be imported into PKCS #12 compliant certificate management tools such as web browsers and the Mac OS X *Keychain Access* utility

Reading a PKCS #12 Key Store

- Read the key store:

```
KeyStore ksFromDisk = KeyStore.getInstance("PKCS12", "BC");
FileInputStream fis =
  new java.io.FileInputStream("c:\\temp\\KeyStorePKCS12.p12");
ksFromDisk.load(fis, "obscure".toCharArray());   // password
fis.close();
```

- Read entries from the key store:

```
X509Certificate x509RootCA = (X509Certificate)
            ksFromDisk.getCertificate("Cogent Logic Root CA");

X509Certificate x509IntermediateCA = (X509Certificate)
        ksFromDisk.getCertificate("My iMine Intermediate CA");

X509Certificate x509EndEntity = (X509Certificate)
          ksFromDisk.getCertificate("Jeff Lawson End Entity");
```

- Make use of the entries:

```
System.out.println("X509 Root CA -- " +
                   x509RootCA.getSubjectDN());

System.out.println("X509 Intermediate CA -- " +
                   x509IntermediateCA.getSubjectDN());

System.out.println("X509 End Entity -- " +
                   x509EndEntity.getSubjectDN());
```

Reading a Mac OS X Keychain

- The Apple JCA security provider enable Mac OS X keychains to be accessed, e.g.

```
// N.B. KeychainStore, not PKCS12
KeyStore ksKeychain = KeyStore.getInstance("KeychainStore",
                                           "Apple");
FileInputStream fis =
    new java.io.FileInputStream("/Users/jeff/Library/
                                Keychains/login.keychain");
ksKeychain.load(fis, null);
fis.close();
```

- A password is not required if the software runs in the security context of the logged-on user

- Read entries:

```
// N.B. The 'alias' created by the Mac OS X keychain is the CN
// (not the 'friendly name' from the Java keystore)
X509Certificate x509RootCA = (X509Certificate)
                ksKeychain.getCertificate("Cogent Logic Root CA");

X509Certificate x509IntermediateCA = (X509Certificate)
            ksKeychain.getCertificate("My iMine Intermediate CA");

X509Certificate x509EndEntity = (X509Certificate)
                        ksKeychain.getCertificate("Jeff Lawson");
```

- Make use of the entries:

```
System.out.println("X509 Root CA -- " +
                        x509RootCA.getSubjectDN());

System.out.println("X509 Intermediate CA -- " +
                        x509IntermediateCA.getSubjectDN());

System.out.println("X509 End Entity -- " +
                        x509EndEntity.getSubjectDN());
```

- Note: when data from a keychain is loaded into a Java key store, certificates are parsed. If an unrecognized certificate extension is encountered an internal exception is thrown (KeychainStore Ignored Exception) that appears as "Duplicate extensions not allowed"

Reading a Windows CryptoAPI Certificate Store

- Importing a `.p12` file into Internet Explorer should work fine but intermediate and end entity certificates may not appear in the Certificates dialog!

- To access these certificates, use the provider that interfaces with the Windows CryptoAPI:

```
KeyStore ksCAPI = KeyStore.getInstance("Windows-ROOT",
                                       "SunMSCAPI");
ksCAPI.load(null,null);
```

- Notice that a file path or password is not provided

- Read entries:

```
X509Certificate x509RootCA = (X509Certificate)
                    ksCAPI.getCertificate("Cogent Logic Root CA");

X509Certificate x509IntermediateCA = (X509Certificate)
                ksCAPI.getCertificate("My iMine Intermediate CA");

X509Certificate x509EndEntity = (X509Certificate)
                  ksCAPI.getCertificate("Jeff Lawson End Entity");
```

- Note: if there is a 'friendly name' this will be used as the alias else ('friendly name' shown as <None>) the CN is used as the alias

- Make use of the entries:

```
System.out.println("X509 Root CA -- " +
                    x509RootCA.getSubjectDN());

System.out.println("X509 Intermediate CA -- " +
                    x509IntermediateCA.getSubjectDN());

System.out.println("X509 End Entity -- " +
                    x509EndEntity.getSubjectDN());
```

Storing a Private Key and a Keychain

- A user my wish to store their private key and its associated keychain
- Separate passwords are needed for the key store and the private key within the key store (though they often have the same value)
- Create and initialize a key store in the usual way:

```
KeyStore ksPrivateKeyAndCertChain =
                              KeyStore.getInstance("PKCS12", "BC");
ksPrivateKeyAndCertChain.load(null, null);
```

- Create the key store entry:

```
// the root certificate is typically not included
Certificate[] certChain = {cw.getX509EndEntity(),
                           cw.getX509IntermediateCA()};

ksPrivateKeyAndCertChain.setKeyEntry("End Entity",
                           cw.getKeyPairEndEntity().getPrivate(),
                           "obscureEE".toCharArray(), certChain);
```

- Note: `java.security.cert.X509Certificate` is a `java.security.cert.Certificate`

- Save the key store:

```
fos = new FileOutputStream("c:\\temp\
                           \PrivateKeyAndCertChainPKCS12.p12");

ksPrivateKeyAndCertChain.store(fos,
                           "obscure2".toCharArray());
fos.close();
```

- To read the key store:

```
KeyStore ksFromDisk2 = KeyStore.getInstance("PKCS12", "BC");
fis = new java.io.FileInputStream("c:\\temp\
                                  \PrivateKeyAndCertChainPKCS12.p12");
ksFromDisk2.load(fis, "obscure2".toCharArray());
fis.close();
```

- Discover aliases, if desired:

```
Enumeration<String> enumAliases = ksFromDisk2.aliases();
while (enumAliases.hasMoreElements())
  System.out.println(enumAliases.nextElement());
```

- Read entries from the key store:

```
PrivateKey privKey = (PrivateKey)ksFromDisk2.getKey("End Entity",
                                    "obscureEE".toCharArray());
Certificate[] certChainFromDisk =
                      ksFromDisk2.getCertificateChain("End Entity");
```

- Make use of the entries:

```
System.out.println("Private Key -- " + privKey.getAlgorithm());
X509Certificate x509Cert;
for (int n=0; n<certChainFromDisk.length; n++)
{
  x509Cert = (X509Certificate)certChainFromDisk[n];
  System.out.println("X.509 Certificate in chain -- " +
                     x509Cert.getSubjectDN());
}
```

Java Cryptography

SSL and TLS (JSSE)

Jeff Lawson

Contents

SSL and TLS

- Secure Sockets Layer is an application protocol created by Netscape to:
    - Encrypt Internet links
    - Ensure data integrity through a HMAC
    - Authenticate servers to clients
    - Optionally authenticate clients to servers
- SSL 3.0 published in 1996
- Transport Layer Security replaces SSL; currently TLS 1.2
- HTTPS runs over SSL/TLS: install certificate in Apache

TLS 1.2 with Java

- Java Virtual Machine SSL and TLS support used to be called the Java Secure Socket Extension (JSSE) but became integrated in Java 1.4

- Java 1.7 (Java Platform, Standard Edition 7), adds the possibility of mandating the use of desired protocols and cipher schemes

- To implement SSL/TLS with both server-side and client-side authentication, we first need to create:

    - A trust store for verifying certificates: contains the root CA certificate

    - A server-based key store: private key and root CA certificate

    - A client-based key store: private key and certificate chain

- JSSE makes use of `KeyManagers` and `TrustManagers`
- A `KeyManager` object encapsulates a key store:
    - Local public-key certificate or certificate chain
    - Local private key
- A `TrustManager` object encapsulates a trust store:
    - Remote public-key certificates:
        - On a client to authenticate the server
        - On a server to authenticate a client

- First generate key pairs and create certificates

- The root CA certificate needs the extended usage `id_kp_serverAuth`

```
KeyPair kpRootCA = generateKeyPair();
X500Principal x500PrincipleRootCA =
                    new X500Principal("CN=Cogent Logic Root CA,
                                       DC=cogentlogic, DC=com,
                                       O=Cogent Logic Ltd., C=UK");
int nKeyUsageRootCA = KeyUsage.keyCertSign |
                      KeyUsage.digitalSignature |
                      KeyUsage.keyEncipherment;
KeyPurposeId[] keyPurposesRootCA =
                                  {KeyPurposeId.id_kp_scvpServer};
X509CertificateHolder x509CertHolderRootCA =
     createCertificateHolder(x500PrincipleRootCA,
                              new BigInteger("1"), 20,
                              x500PrincipleRootCA,
                              kpRootCA.getPublic(), true, 1,
                              nKeyUsageRootCA, keyPurposesRootCA,
                              kpRootCA.getPrivate());
```

- The intermediate CA certificate:

```
KeyPair kpIntermediateCA = generateKeyPair();

X500Principal x500PrincipleIntermediateCA =
                    new X500Principal("CN=My iMine Intermediate CA,
                                       DC=myimine, DC=com,
                                       OU=My iMine Trust Authority,
                                       O=Cogent Logic Ltd., C=UK");

int nKeyUsageIntermediateCA = KeyUsage.keyCertSign |
                              KeyUsage.digitalSignature |
                              KeyUsage.keyEncipherment;

KeyPurposeId[] keyPurposesIntermediateCA = {};

X509CertificateHolder x509CertHolderIntermediateCA =
   createCertificateHolder(x500PrincipleRootCA,
                           new BigInteger("10"), 2,
                           x500PrincipleIntermediateCA,
                           kpIntermediateCA.getPublic(), true, 0,
                           nKeyUsageIntermediateCA,
                           keyPurposesIntermediateCA,
                           kpRootCA.getPrivate());
```

- The end entity certificate needs the extended usage `id_kp_clientAuth`

```
KeyPair kpRootCA = generateKeyPair();

X500Principal x500PrincipleRootCA =
                  new X500Principal("CN=Cogent Logic Root CA,
                                     DC=cogentlogic, DC=com,
                                     O=Cogent Logic Ltd., C=UK");

int nKeyUsageRootCA = KeyUsage.keyCertSign |
                      KeyUsage.digitalSignature |
                      KeyUsage.keyEncipherment;

KeyPurposeId[] keyPurposesRootCA =
                                    {KeyPurposeId.id_kp_scvpServer};

X509CertificateHolder x509CertHolderRootCA =
     createCertificateHolder(x500PrincipleRootCA,
                              new BigInteger("1"), 20,
                              x500PrincipleRootCA,
                              kpRootCA.getPublic(), true, 1,
                              nKeyUsageRootCA, keyPurposesRootCA,
                              kpRootCA.getPrivate());
```

- Prepare to use the key pairs and certificates:

```
CryptoWrapper cw = generateCrypto();
KeyPair kpRootCA = cw.getKeyPairRootCA();
KeyPair kpEndEntity = cw.getKeyPairEndEntity();
X509Certificate x509RootCA = cw.getX509RootCA();
X509Certificate x509IntermediateCA = cw.getX509IntermediateCA();
X509Certificate x509EndEntity = cw.getX509EndEntity();
```

- Create a trust store:

```
KeyStore ksTrustStoreJKS = KeyStore.getInstance("JKS");
ksTrustStoreJKS.load(null, null);
ksTrustStoreJKS.setCertificateEntry(Constants.TRUST_STORE_KS_NAME,
                                    x509RootCA);
FileOutputStream fos =
     new FileOutputStream(Constants.TRUST_STORE_KS_NAME + ".jks");
ksTrustStoreJKS.store(fos, Constants.TRUST_STORE_KS_PASSWORD);
fos.close();
```

- Create the server-based key store:

```
KeyStore ksServerJKS = KeyStore.getInstance("JKS");

ksServerJKS.load(null, null);

ksServerJKS.setKeyEntry(Constants.SERVER_KS_NAME,
                        kpRootCA.getPrivate(),
                        Constants.SERVER_KS_PASSWORD,
                        new Certificate[] {x509RootCA});
```

```
fos = new FileOutputStream(Constants.SERVER_KS_NAME + ".jks");

ksServerJKS.store(fos, Constants.SERVER_KS_PASSWORD);

fos.close();
```

- Create the client-based key store:

```
KeyStore ksClientPKCS12 = KeyStore.getInstance("PKCS12", "BC");

ksClientPKCS12.load(null, null);

ksClientPKCS12.setKeyEntry(Constants.CLIENT_KS_NAME,
                           kpEndEntity.getPrivate(),
                           Constants.CLIENT_KS_PASSWORD,
                           new Certificate[] {x509EndEntity,
                                              x509IntermediateCA,
                                              x509RootCA});

fos = new FileOutputStream(Constants.CLIENT_KS_NAME + ".p12");

ksClientPKCS12.store(fos, Constants.CLIENT_KS_PASSWORD);

fos.close();
```

SSLContext Objects

- On both the server and the client, we need to encapsulate the trust manager and the appropriate key store in an `SSLContext` object

- The trust store initialises a `TrustManagerFactory`:

```
TrustManagerFactory tmFactory =
                        TrustManagerFactory.getInstance("SunX509");
KeyStore ksTrustStore = KeyStore.getInstance("JKS");
ksTrustStore.load(
   new FileInputStream(Constants.TRUST_STORE_KS_NAME + ".jks"),
                        Constants.TRUST_STORE_KS_PASSWORD);
tmFactory.init(ksTrustStore);
```

- A key store initializes a `KeyManagerFactory`—server:

```
KeyManagerFactory kmFactory =
                    KeyManagerFactory.getInstance("SunX509");
KeyStore ksServer = KeyStore.getInstance("JKS");
ksServer.load(new FileInputStream(Constants.SERVER_KS_NAME + ".jks"),
                                        Constants.SERVER_KS_PASSWORD);
kmFactory.init(ksServer, Constants.SERVER_KS_PASSWORD);
```

- A key store initializes a `KeyManagerFactory`—client:

```
KeyManagerFactory kmFactory =
                    KeyManagerFactory.getInstance("SunX509");
KeyStore ksClient = KeyStore.getInstance("PKCS12");
ksClient.load(new FileInputStream(Constants.CLIENT_KS_NAME + ".p12"),
                                      Constants.CLIENT_KS_PASSWORD);
kmFactory.init(ksClient, Constants.CLIENT_KS_PASSWORD);
```

- Finally, the `SSLContext` can be created from the `TrustManagerFactory` and the `KeyManagerFactory`:

```
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");

sslContext.init(kmFactory.getKeyManagers(),
                tmFactory.getTrustManagers(), null);
                                        // use default SecureRandom
```

- Now we have `SSLContext` objects we can create a `SSLServerSocket` on the server and a `SSLSocket` on the client

- Both these classes have methods `getSSLParameters()` and `setSSLParameters()` for additional configuration

Server-Side TLS 1.2

- Server-side TLS 1.2 code:

```
SSLServerSocketFactory sslServerFactory =
                                sslContext.getServerSocketFactory();
SSLServerSocket sslServerSock = (SSLServerSocket)
        sslServerFactory.createServerSocket(Constants.PORT_NO);
SSLParameters sslParams = sslServerSock.getSSLParameters();
sslParams.setProtocols(new String[] {"TLSv1.2"});
sslParams.setCipherSuites(new String[]
                               {"TLS_RSA_WITH_AES_256_CBC_SHA256"});
sslParams.setNeedClientAuth(true);  // c.f. SetWantClientAuth()
sslServerSock.setSSLParameters(sslParams);
```

- ```
  SSLSocket sslSock = (SSLSocket)sslServerSock.accept();
  sslSock.startHandshake();
          // throws SSLHandshakeException "No appropriate protocol"
          // or "no cipher suites in common"
  if (isEndEntityAuthorized(sslSock.getSession()))
  {
    InputStream is = sslSock.getInputStream();
    OutputStream os = sslSock.getOutputStream();
    // ...
  }
  sslServerSock.close();
  ```

```
private static boolean isEndEntityAuthorized(SSLSession session)
                                        throws SSLPeerUnverifiedException
{
  Principal prin = session.getPeerPrincipal();
  if (!(prin instanceof X500Principal))
     return false;
  X500Principal x500Prin = (X500Principal)prin;
  return x500Prin.getName().equals("CN=Jeff Lawson,
              OU=Software Development,O=Cogent Logic Ltd.,C=UK");
}
```
```

Client-Side TLS 1.2

- Client-side TLS 1.2 code:

```
SSLSocketFactory sslFactory = sslContext.getSocketFactory();

SSLSocket sslClientSock =
        (SSLSocket)sslFactory.createSocket(Constants.HOST,
                                           Constants.PORT_NO);

SSLParameters sslParams = sslClientSock.getSSLParameters();

sslParams.setProtocols(new String[] {"TLSv1.2"});

sslParams.setCipherSuites(new String[]
                          {"TLS_RSA_WITH_AES_256_CBC_SHA256"});

sslClientSock.setSSLParameters(sslParams);

OutputStream os = sslClientSock.getOutputStream();
InputStream is = sslClientSock.getInputStream();
// ...
sslClientSock.close();
```

SSL/TLS Protocols and Cipher Suites

- We need to be sure that the protocols and cipher suites we are happy to support on the server is available to all clients

- Available options can be discovered:

```
String[] strProtocols = sslParams.getProtocols();
if (strProtocols != null)
  for (int n=0; n<strProtocols.length; n++)
     System.out.println("Protocol: " + strProtocols[n]);

String[] strCipherSuites = sslParams.getCipherSuites();
if (strCipherSuites != null)
  for (int n=0; n<strCipherSuites.length; n++)
     System.out.println("Cipher Suite: " +
                        strCipherSuites[n]);
```

- Sample protocols:

```
SSLv3
TLSv1
TLSv1.1
TLSv1.2
```

- Sample cipher suites:

```
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA  ...
```

Java Cryptography

Accessing LDAP Servers with JNDI

Jeff Lawson

Contents

- Naming and Directory Servers
- Lightweight Directory Access Protocol
- LDAP Access Using JNDI
- Creating an inetOrgPerson LDAP Entry
- Reading an inetOrgPerson LDAP Entry

Naming and Directory Servers

- A *naming service* holds entries that are name/value pairs following a suitable organizational scheme such that each entry can be located by a path
  e.g. DNS, telephone directory

- A *directory service* is like a naming service but each entry may have zero or more attributes
  e.g. Window Active Directory, Apache Directory Server,
  X.500 directory services

- A X.500 directory service identifies entries with distinguished names and is accessed with Directory Access Protocol (DAP) over a OSI network

Lightweight Directory Access Protocol

- Lightweight Directory Access Protocol is used to access X.500 directory services over TCP/IP

- LDAP has become so popular that it is now used to access non-X.500 directory services, e.g. Active Directory, and even has directory services implementations based more on LDAP than X.500, e.g. OpenLDAP

- OpenLDAP uses the well-known port 389 and, for OpenLDAP over TLS/SSL, 636

- OpenLDAP is available for all platforms and may be downloaded from

  www.openldap.org

- Specifically, for packaged versions of OpenLDAP:

  www.openldap.org/faq/data/cache/108.html

- For Windows, use:

  www.userbooster.de/en/download/openldap-for-windows.aspx

- JXplorer is a useful LDAP browser available for all platforms from:

  jxplorer.org

- Note: we do not cover secure access to the OpenLDAP server on this course but by now you ought to be able to generate CA certificates and use them to configure OpenLDAP!

LDAP Access Using JNDI

- The Java Naming and Directory Interface (JNDI) enables Java software to access naming and directory services, including those with an LDAP interface

- A JNDI *context* abstracts *entries* in an LDAP-based server

- JNDI may be configured programmatically or declaratively (using system variables)

- To gain access to the root of an OpenLDAP service:

```
Hashtable<String, String> htEnv =
                                new Hashtable<String, String>(11);
htEnv.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.ldap.LdapCtxFactory");
htEnv.put(Context.PROVIDER_URL, "ldap://localhost:389");
htEnv.put(Context.SECURITY_AUTHENTICATION, "simple");
htEnv.put(Context.SECURITY_PRINCIPAL,
          "cn=Manager,dc=maxcrc,dc=com");
htEnv.put(Context.SECURITY_CREDENTIALS, "secret");
htEnv.put("java.naming.ldap.version", "3");
DirContext dirctx = new InitialDirContext(htEnv);
```

- To gain access to an entry at a given DN:

```
DirContext dirctxPeople =
        (DirContext)dirctx.lookup("ou=People,dc=maxcrc,dc=com");
```

- Note: `"dc=com,dc=maxcrc,ou=People"` will not work

- We can record details of a person, including their end entity X.509 certificate, using an `inetOrgPerson` entry

- We create the desired attributes then make the following call against the containing context:

  ```
  createSubcontext(String name, Attributes attrs)
  ```

  where `name` is the common name

Creating an inetOrgPerson LDAP Entry

- To create an inetOrgPerson entry:

```
String strGivenName = "Jeffrey";
String strSurname = "Lawson";
String strEmail = "jdl@cogentlogic.com";

BasicAttributes attrs = new BasicAttributes(true);
Attribute attrObjclass = new BasicAttribute("objectClass");
attrObjclass.add("top");
attrObjclass.add("inetOrgPerson");
attrs.put(attrObjclass);

Attribute attrGivenName = new BasicAttribute("givenName");
attrGivenName.add(strGivenName);
attrs.put(attrGivenName);
```

- To create an `inetOrgPerson` (continued):

```
Attribute attrSurname = new BasicAttribute("sn");
attrSurname.add(strSurname);
attrs.put(attrSurname);

Attribute attrEmail = new BasicAttribute("mail");
attrEmail.add(strEmail);
attrs.put(attrEmail);

Attribute attrCertificate =
                    new BasicAttribute("userCertificate;binary",
                                       x509EndEntity.getEncoded());
attrs.put(attrCertificate);

DirContext dirctxPerson =
    dirctxPeople.createSubcontext("cn=" + strGivenName +
                                  " " + strSurname, attrs);
```

- Notice the use of the attribute subclass `;binary`

Reading an inetOrgPerson LDAP Entry

- To read an `inetOrgPerson` entry from a root context:

```
DirContext dirctxPerson =
      (DirContext)dirctx.lookup("cn=Jeffrey Lawson,
                                 ou=People,dc=maxcrc,dc=com");
Attributes attrs = dirctxPerson.getAttributes("");
Attribute attr = attrs.get("userCertificate;binary");
byte[] bytesCert = (byte[])attr.get();
if (bytesCert == null || bytesCert.length == 0)
  throw new Exception("Cannot read X.509 certificate");
CertificateFactory cf = CertificateFactory.getInstance("X.509",
                                                       "BC");
InputStream is = new ByteArrayInputStream(bytesCert);
X509Certificate x509Cert =
                  (X509Certificate)cf.generateCertificate(is);
System.out.println(x509Cert.toString());
```

- We ought to verify the certificate with the issuer's public key:

```
x509Cert.checkValidity();
x509Cert.verify(keyPublicCA, "BC");
```

Java Cryptography

Certificate Revocation Lists and Online Certificate Status Protocol

Jeff Lawson

Contents

- Subject Key Identifiers
- Authority Key Identifiers
- Certificate Revocation Lists
- Online Certificate Status Protocol

Subject Key Identifiers

- Processing certificate chains involves identifying which certificate validates which other certificate, i.e. knowing where to find the CA public key to validate a certificate

- To facilitate certificate ordering, we can add to certificates optional X.509 extensions called *key identifiers*

- *Subject key identifiers* can be any value but they are usually a monotonically increasing number or, more likely, the hash of the subject's public key—the benefit of such a value is that it can be referenced from *another* certificate, e.g. (`false` => optional)

```
x509Builder.addExtension(X509Extension.subjectKeyIdentifier,
        false,
        jcaUtils.createSubjectKeyIdentifier(pubKeySubject));
```

Authority Key Identifiers

- An *authority key identifier* in a certificate enables us to reference the subject key identifier of the issuing CA certificate and contains:
    - Key identifier of the authority
    - Authority certificate issuer, i.e. the DN of the authority that issued the authority certificate
    - Authority certificate serial number
- So, an end entity certificate might contain an authority key identifier extension that holds:
    - Key identifier of the issuing intermediate CA
    - DN of the root CN (issuer of the intermediate certificate)
    - The serial number of the intermediate certificate

- Creating an authority key identifier (`false` => optional):

```
x509Builder.addExtension(X509Extension.subjectKeyIdentifier,
                         false,
                         jcaUtils.createSubjectKeyIdentifier(pubKeySubject));
```

- See how this code is used in the sample project `Certificate Revocation Lists`

Certificate Revocation Lists

- In PKI environments very many public-key certificates are generated, expire and are replaced

- It is important for users to have a reliable way to acquire certificates and to known when they have been revoked

- Certificate Revocation Lists are used to inform users of invalid end-entity certificates

- Authority Revocation Lists are used to inform users of invalid CA public-key certificates

- These lists contain one or more entries of revocation

- Possible reasons for revoking a certificate:
  - `unspecified`
  - `keyCompromise`
  - `cACompromise`
  - `affiliationChanged`
  - `superseded`
  - `cessationOfOperation`
  - `certificateHold`
  - `removeFromCRL`
  - `privilegeWithdrawn`
  - `aACompromise`

- To create a CRL using Bouncy Castle:

```
// Issue a CRL now

Date dateNow = new Date();

X509v2CRLBuilder cb =
                new JcaX509v2CRLBuilder(x509CrlIssuer, dateNow);

cb.addExtension(X509Extension.cRLNumber, false,
                 new CRLNumber(biCrlSerialNumber));

// Update (i.e. becomes available) one minute from now
cb.setNextUpdate(new Date(dateNow.getTime() + 60000));
                //  milliseconds since the standard base time
```

- To add an entry to a CRL using Bouncy Castle:

```
cb.addCRLEntry(biRevokedCertificateSerialNumber, dateNow,
                CRLReason.keyCompromise, dateInvalid);

cb.addExtension(X509Extension.authorityKeyIdentifier, false,
                 new JcaX509ExtensionUtils()
                   .createAuthorityKeyIdentifier(x509CrlIssuer));
```

- `dateInvalid` is optional and refers to the event that gave rise revocation, e.g. when the key was compromised

- To generate a CRL:

```
JcaContentSignerBuilder signerBuilder =
        new JcaContentSignerBuilder("SHA256withRSAEncryption");

signerBuilder.setProvider("BC");

ContentSigner cs = signerBuilder.build(privKeyIssuer);

X509CRLHolder crlHolder = cb.build(cs);

JcaX509CRLConverter crlConvert = new JcaX509CRLConverter();

crlConvert.setProvider("BC");

X509CRL x509CRL = crlConvert.getCRL(crlHolder);
```

- To verify a CRL:

```
x509CRL.verify(x509IntermediateCA.getPublicKey(), "BC");
                                        // throws SignatureException
```

- To check if a certificate has been revoked:

```
X509CRLEntry crlEntry =
  x509CRL.getRevokedCertificate(x509EndEntity.getSerialNumber());
if (crlEntry == null)
  System.out.println("Certificate still valid.");
else
  System.out.println(x509CRL.getIssuerX500Principal().getName() +
                     " has revoked " + crlEntry.getSerialNumber() +
                     " on " + crlEntry.getRevocationDate());
```

- Typically, store CRLs in LDAP servers as `cRLDistributionPoint` entries

Online Certificate Status Protocol

- CRLs operate like black-lists, identifying certificates that are invalid
- In systems with a large number of revocations, CRLs become difficult to manage
- A better approach is to have a white-list system that knows about certificates that are valid
- Online Certificate Status Protocol does this
- Clients send requests to a OCSP Responder (server)
- The OCSP Responder informs the client of a certificate's validity

Java Cryptography

Privilege Management Infrastructure

Jeff Lawson

Contents

Privilege Management Infrastructure

- PKI uses X.509 public-key certificates to provide distributed platform-independent subject *authentication*

- The X.509 standard enables extension data to be added to public-key certificates to specify subject *authorization* capabilities such a resource permissions.

- This is a poor way to use public-key certificates because:
    - Certificates must be reissued when permissions change
    - Resource managers must have influence over the issuing CA
- Thankfully, X.509 provides for *attribute certificates* that cite public-key certificates and specify capabilities

- A distributed system of authorization through X.509 Attribute Certificates is known as a Privilege Management Infrastructure (PMI)

- Imagine you are managing resources (printers, file shares, hockey sticks, whatever):

    - You can look up X.509 public-key certificates in you company's (department's, club's) LDAP server and decide who to grant access to which resources and at what level (read-only, 'playable', whatever)

    - When subjects present themselves for resource access, your wonderful software (Security Manager?) will automatically grant/deny access

- After a while you realize that some resources that you have granted access to are being passed on to non-authorized individuals!

- To combat this you decide to institute a classification scheme that requires:

    - Classified resources to display a notice,
      e.g. “Confidential:…”, “Secret:…”, etc.

    - Subjects to be assigned *clearance levels* that determine which classified resources they access and how

- Hence, you have *rule-based access control*

- Things are going great until you realize that other resource managers in other departments have had similar ideas but their notion of “Confidential” is not the same as yours!

- At this point the librarian suggests that each department have its own Security Policy so that resources belong to a Security Domain or Realms

- Hence, you have Partitioned Rule-Based Access Control (PRBAC)

- Unfortunately, few implementations of PMI currently exist but PMI is part of X.509 and the Bouncy Castle security service provider does support attribute certificates

- The U.S. National Security Agency produced a document in 1999 that states:

  SDN.801: ACCESS CONTROL CONCEPT AND MECHANISMS
  This document provides guidance for implementing access control concepts using both public key certificates and attribute certificates.

- SDN.801 provides guidance for X.509-based PRBAC

- What's the problem with Partitioned Rule-Based Access Control?

- Do you really want to create attribute certificates on each type of resource within a security realm for each subject's public-key certificate?

- Probably not. Instead, define user *roles* within each X.509 Security Policy then:

    - Only ever set resource permissions for *roles*
      e.g. '*clean-access* for *offices* is assigned exclusively to *managers* on Saturday evening'

    - Assign subjects to roles

- Hence, Partitioned *Role*-Based Access Control!

- X.509 provides two types of attribute certificates for PRBAC:
    - *Role-Specification Attribute Certificates* describe roles and their privileges—for resources.
    - *Role-Assignment Attribute Certificates* describe entities and their assigned roles—for subjects.
- plus:
    - *Security Policy Information Files (SPIF)*
    - *Security Labels*
    - *Public-Key Certificates*
- and:
    - Security Domain is administered by an *Attribute Authority* (AA)
    - CAs issue public-key certificates
    - AAs issue attribute certificates

Cogent Logic Ltd.

High Quality Hands-On Training
for
Software Developers

- Android Training Courses
  - *Developing Mobile Applications with Android* is for Java programmers wishing to get up to speed on Android development.
  - *Software Development with Java* is for programmers wishing acquire a thorough grounding in Java.
- iOS Training Courses
  - *Developing Mobile Applications with iOS* is for Objective-C programmers wishing to get up to speed on iOS development.
  - *Software Development with Objective-C* is for programmers wishing acquire a thorough grounding in Objective-C.

- Ruby on Rails Training Courses

  - *Developing Web Applications with Ruby on Rails* is for Ruby programmers wishing to get up to speed on Rails development.

  - *Software Development with Ruby* is for programmers wishing acquire a thorough grounding in Ruby.