



# Java Security

A presentation



By Johnny

# Java Security - Evolution

In JDK 1.0 : The “Sandbox” approach  
(all restrictions on applets, none on local)

JDK 1.1 : full trust for signed applets  
Additional cryptographic APIs

JDK 1.2 : Fine-Grained access control  
The Policy, Permission-Based approach.

In the beginning, Java took a very primitive approach to security.

Applets downloaded from the Web were confined to a “SandBox”, disabling any access to the local filesystem.

In JDK 1.1, cryptography support was introduced, and – most notably, the ability to fully trust applets that were digitally signed by their manufacturer.

JDK 1.2 and above, however, bring the real promise – of a fine grained access control mechanism, using tunable policies and permission files.

# Language Level Security



- No Pointer Arithmetic!
- Garbage Collection
- Other Language Features
  - All heap memory is initialized
  - Uninitialized stack memory cannot be used.
  - Final, is final!

The java Language-Level Security is multi-leveled:

**No Pointer Arithmetic:** In C/C++, it is possible to reach any address in memory by adding/subtracting values from pointers.


**Garbage Collection:** Unused memory is reclaimed by system, preventing memory leaks.

**Other Language Features:**

- Strong type checking (must use explicit type casts)
- Variables may not be used prior to initialization
- keyword “final” disallows all class subclassing and method overriding.



# JVM Security

- The SandBox
- Byte Code Verification:
  - Four pass process, ensuring bytecode validity
- Class Loading:
  - Security checks performed by the classloader
  - Extensible.
- Runtime Checking 
  - JVM ensures only properly assignable operations
  - Array bounds checking (important vs. buffer overflows!)
- Managing Security
  - Security Managers and policy files.

The JVM provides several layers of security, as well:

**The SandBox** – All java bytecode is executed within a self-contained “SandBox” disallowing arbitrary access to the underlying filesystem, and operating system services.

**Byte Code Verification** –

Bytecode is filtered through a four stage process to ensure its integrity and validity.

**Class Loading** –

- No loading of java.\* packages from over the network permitted (preventing trojans)
- Separate namespaces for different download locations (again, trojan prevention)
- Classes loaded from different locations cannot communicate within JVM space (preventing hostile classes from getting information about trusted programs)

**Runtime Checking** –

- Array Bounds Checking - Java will throw an *ArrayIndexOutOfBoundsException* if arrays overflow
- Cannot fool around with class hierarchy

**Managing Security** -

A topic by itself – Java permissions may be tweaked, by using policy files, to enable SPECIFIC operations and permissions, while disallowing others.

# Java Security APIs



Java presents four basic APIs for security:

- JCA – Java Cryptography Architecture
- JCE – Java Cryptography Extension
- JSSE – Java Secure Socket Extension
- JAAS – Java Authentication & Authorization Service

The four Java APIs are:

JCA – Java Cryptography Architecture – The basic cryptographic support

JCE – Java Cryptography Extension – Extensions for “strong” encryption

JSSE – Java Secure Socket Extension – SSL Support

JAAS – Java Authentication & Authorization Service – Permission control

These extensions are all downloadable, but were not included in the JDK 1.3.x releases.

# Cryptography, in a Nutshell

**Symmetric Algorithms** are those in which shared “secrets” are used to encrypt data.

The most common algorithm is the former Data Encryption Standard (DES), which is 56-bit.

Also common are RC2 (40), RC4 (40 or 128) and RC5 (128).

**Assymmetric Algorithms** involve two separate keys.

Diffie-Hellman is a method for key agreement

RSA is common for public/private encryption.

DES has now been officially replaced by AES, which Java will support in its next edition. AES (Rijndael) is a stronger, 128-bit algorithm which is FAR more impervious than DES, which has already been broken, repeatedly.

# Java Cryptography Architecture



- Adds more APIs that providers can support
  - Keystore creation/management
  - Algorithm parameter handling
  - Conversions between different key representations
  - Certificate factory classes
  - Secure PseudoRandom Number Generators

Secure (unpredictable) pseudo-random number generation is imperative to many cryptographic protocol implementations.



## Java Cryptography Extension

An optional extension, providing:

- Encryption
- Key exchange
- Key generation
- Message authentication code (MAC)

Multiple “providers” supported

Keys & certificates in “keystore” database

JCE provides for stronger encryption, as well as extensible cipher support.

A list of classes follows in the next slide.

# Javax.crypto class hierarchy

## **Generic Ciphers:**

Cipher , CipherInputStream , CipherOutputStream – Work with algorithms

## **Implemented Algorithms:**

**DES:** DESedeKeySpec , DESKeySpec , DHGenParameterSpec

**Diffie-Hellman:** DHKey ,DHParameterSpec ,DHPrivateKey  
,DHPrivateKeySpec...

**Password Based Encryption:** PBEKeySpec ,PBESpec

**RCx:** RC2ParameterSpec , RC5ParameterSpec - RC2, RC5 Parameters,  
respectively

**Misc:** IVParameterSpec – Block Cipher CBC mode

**Key exchanges:** KeyAgreement ,KeyGenerator

**Message authentication:** Mac

**Key Creation:** SecretKey , SecretKeyFactory , SecretKeySpec

**NULL encryption:** NullCipher

**Safe Objects:** SealedObject – Creating cryptographically protected objects

# Java Secure Sockets

JSSE provides full SSL support for Java.

- Separate cryptographic algorithms
- Compliant with SSLv3.
- Support for HTTPS
- SSLSocket and SSLServerSocket.



JSSE has been integrated into the JDK as of version 1.4

Providing many features:

- Pure Java implementation
- Exportable from the United States
- Secure Sockets Layer (SSL) v3 support
- Transport Layer Security (TLS) 1.0 support
- Basic utilities for key and certificate management, including securely encrypted storage of private keys and Certificate Authority (CA) support
- SSLSocket and SSLServerSocket classes, which can be instantiated to create secure channels
- Cipher Suite negotiation, which performs SSL "handshaking" to initiate or verify secure communications
- Client and server authentication, as a part of the normal SSL handshaking
- HTTPS support: the ability to access data such as HTML pages using HTTPS
- Server Session Management to manage the cache of sessions.
- RSA cryptography algorithms -- the JSSE implementation includes code licensed from RSA Data Security.

Cryptographic suites, including:

## Using JSSE...

```
/*  
 * Instead of:  
 * Socket socket = new Socket (www.hisown.com, 80);  
 */  
  
SSLSocketFactory factory =  
    (SSLSocketFactory) SSLFactory.getDefault();  
  
SSLSocket socket = (SSLSocket)  
    factory.createSocket("www.hisown.com", 443);  
PrintWriter out = new PrintWriter(  
    new BufferedWriter(  
    new OutputStreamWriter(socket.getOutputStream())));  
out.println("GET /trip.html HTTP/1.1"); /* request */  
out.flush();
```

Using JSSE is almost as simple as using standard java.net.sockets. The SSL negotiations are completely transparent.

# JAAS

Bringing UNIX PAM to Java:

- Pluggable Authentication Module support
- Authenticators may be:
  - Required
  - requisite (stop on failure)
  - sufficient (but not required)
  - or optional
- Adds user-centric (vs. code-centric) control: permissions granted to a ***Principal***
- Implemented through a SecurityManager



JAAS enables support for advanced authentication mechanisms, such as smart cards, fingerprint scanners, and other authentication modules.

Much like UNIX has its own Pluggable Authentication Module, so does JAAS enable “pluggability” by defining the order of authentication mechanisms, prerequisites on each, etc.

JAAS will not be discussed in this lecture.

# PKI in a Nutshell

## Refresher: RSA Encryption

- Generate  $p, q$  – prime numbers. Compute  $n=pq$
- Generate Encryption Exponent,  $e$  (small exponent)
- Find a Decryption Exponent,  $d$ , such that:

$$\mathbf{ed \equiv 1 \pmod{\phi(n)}}$$

Call  $(n, e)$  the public key. Distribute.

Call  $d$  the private key. Keep secret.

To encrypt message  $M$ :  $C = M^e \pmod{n}$

To decrypt:  $M = C^d \pmod{n}$



$$C = M^e \pmod{n}$$

$$\begin{aligned} C^d &= (M^e)^d \pmod{n} = \\ &= M^{ed} \pmod{n} \end{aligned}$$

But remember  $ed \equiv 1 \pmod{\phi(n)}$ .

Meaning –  $ed = 1 + K(p-1)(q-1)$  for some  $K$ .

$$\begin{aligned} C^d &= (M^{1+K(p-1)(q-1)}) \pmod{p} = M * M^{(K(p-1)(q-1))} \pmod{p} = \\ &= M * (M^{K(q-1)})^{(p-1)} \pmod{p} \text{ (Fermat's little theorem)} \\ &= M \pmod{p}! \end{aligned}$$

And similarly, for  $q$ :

$$\begin{aligned} C^d &= (M^{1+K(p-1)(q-1)}) \pmod{q} = M * M^{(K(p-1)(q-1))} \pmod{q} = \\ &= M * (M^{K(p-1)})^{(q-1)} \pmod{q} = M \pmod{q}! \end{aligned}$$

So  $C^d = M \pmod{n}$ ! (q.e.d)

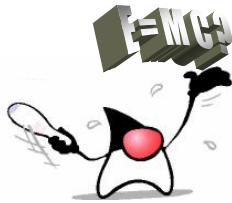
# PKI In a Nutshell (II)

## Refresher: RSA Digital Signatures

Using same (n,e) and d:

To sign message M:  $S = M^d \text{ mod } n$

To Verify:  $M = C^e \text{ mod } n$



This symmetry lays the foundation for Digital signatures

This works because exponentiation is cumulative, of course.

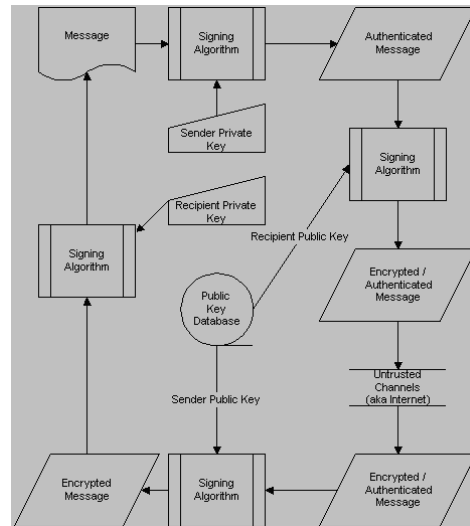
# Authentication and Privacy

Public Keys can be used both to authenticate and to encrypt.

Anything encrypted PRIVATEly can be decrypted PUBLICly

Anything encrypted PUBLICly can only be decrypted PRIVATEly.

Algorithms of choice are RSA, MD5, and SHA1/DSA



PKI helps achieve both privacy and authentication.

As the above illustration shows, if a user wants to encrypt a message to another, he simply needs to know the message recipient's *public* key, and use it. Only the holder of the private key can then decrypt the message successfully.

If a user wishes to sign a message, to prove it is indeed authentic, he or she may use the *private* key to sign a message (or a hash thereof). The recipient may verify authenticity easily by validating the signature, using the sender's public key.



## Java.security key Interface

- public interface **Key** extends [Serializable](#)

### Subinterfaces:

- PrivateKey
- PublicKey
- DSAPrivateKey, DSAPublicKey
- RSAPrivateKey, RSAPublicKey, RSACertPubKey



### All keys have:

- An algorithm
- An encoding (PKCS #8, PKCS #11, X509v3, etc)
- A format (describing the encoding format)

Keys are produced using KeyGenerators.

The Key interface is the top-level interface for all keys. It defines the functionality shared by all key objects. All keys have three characteristics:

**An Algorithm** This is the key algorithm for that key. The key algorithm is usually an encryption or asymmetric operation algorithm (such as DSA or RSA), which will work with those algorithms and with related algorithms (such as MD5 with RSA, SHA-1 with RSA, Raw DSA, etc.) The name of the algorithm of a key is obtained using the `getAlgorithm` method.

**An Encoded Form** This is an external encoded form for the key used when a standard representation of the key is needed outside the Java Virtual Machine, as when transmitting the key to some other party. The key is encoded according to a standard format (such as X.509 or PKCS#8), and is returned using the `getEncoded` method.

**A Format** This is the name of the format of the encoded key. It is returned by the `getFormat` method.

Keys are generally obtained through key generators, certificates, or various Identity classes used to manage keys. Keys may also be obtained from key specifications (transparent representations of the underlying key material) through the use of a key factory.

# Java.security classes

## Key Manipulation Classes:

<b>KeyFactory</b>	Create/convert key representations
<b>KeyPair</b>	Public/Private Key pair
<b>KeyPairGenerator</b>	Create KeyPairs.
<b>KeyStore</b>	in-memory collection of keys and certificates

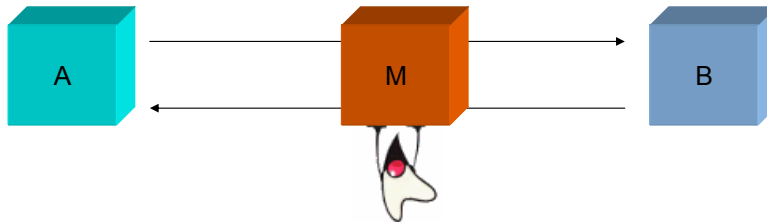
## Other Interesting Classes:

<b>MessageDigest</b>	Compute MD5 hashes
<b>SecureRandom</b>	PseudoRandom # Generators
<b>Signature</b>	Digitally sign a message
<b>Permission</b>	Code Permissions (complex(!))

Additionally, a service provider interface (SPI) is defined for each class. Discussion will be omitted, for brevity.

# Problem: MiM

All public key protocols are highly vulnerable to a “Man in the Middle” attack.



The attacker impersonates B to A, and vice versa.  
Communication may be secure, but with the wrong party!

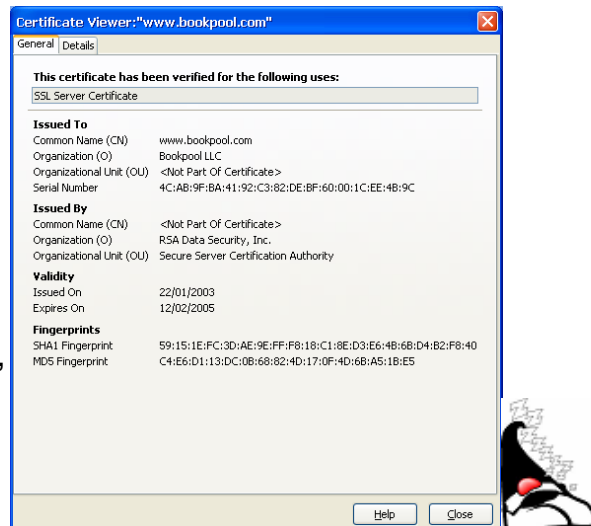
The “Man in the Middle” attack breaks many cryptographic protocols, such as Diffie Hellman key exchanges, and RSA. It is a case of what mathematicians coin the “Chess GrandMaster Problem”, in which a simultaneous game with two chess players can lead to mutual draw, or a victory over one of them. This is accomplished by the player going back and forth, and simulating moves, so as to transparently allow the two opponents to battle amongst themselves..

# Solution: X509v3 Certificates

SSL relies heavily on the X509v3 standard:

These certificates ensure that the identity of a remote peer is indeed authentic.

Once the identity is verified, and the public key can be trusted, a secure channel can be established.



Certificates are easily viewable by using any web browser upon establishing an SSL (https) connection. The excerpt in the picture shows Mozilla's certificate viewer.

A certificate is issued by a CA (which is the trusted entity).

A certificate consists of a public key, detailed information about the certificate owner (such as name, e-mail address, and so on) and other, application-specific data. To ensure the integrity of the certificate, it is signed by a certification authority (CA), a trusted entity whose public key is widely known and distributed. In essence, the CA vouches for the validity of the public key contained in the certificate, and ensures that the identity of the certificate owner reflects the actual person or entity.


# Generating Keys

To generate keys, use `KeyPairGenerator`:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(1024); // Initialize cipher strength

KeyPair kp = kpg.generateKeyPair();
```

Or a specific subclass:



```
DSAKeyPairGenerator dkpg =
    (DSAKeyPairGenerator) KeyPairGenerator.getInstance("DSA");

DSAParams params = new DSAParams(aP, aQ, aG);

dkpg.initialize(1024, params, new SecureRandom());
```

The advantages of using a specific subclass are evident only when specific parameters to the encryption algorithm need tweaking.

# Signing messages with JCA

**Step I:** Acquire a signature object

```
Signature sig = Signature.getInstance("DSA"); // Or "SHA", etc.
```

**Step II:** Retrieve private key (necessary to sign)

```
PrivateKey priv = kp.getPrivate(); // Created previously..  
sig.initSign(priv); // Initialize signature object...
```

**Step III:** Provide message as input to signature object

```
sig.update(message); // message is a byte[]...
```

**Step IV:** Sign, and retrieve message digest..

```
byte digest[] = sig.sign(); // digest can now be appended..
```

Once the keypair is generated, and the private key is at hand, we can get to work.

Signing messages is a four-stage process, as shown here.

The `sign()` method returns the digital signature, which may be appended to the message, or sent by some other channel.

# Verifying messages with JCA



**Step I:** Acquire a signature object

```
Signature sig = Signature.getInstance("DSA"); // Or "SHA", etc.
```

**Step II:** Retrieve public key from certificate..

```
PrivateKey priv = Certificate.getPublicKey();  
sig.initVerify(priv); // Initialize signature Object
```

**Step III:** Provide message as input to signature object

```
sig.update(message); // message is a byte[]...
```

**Step IV:** Verify signature against proposed digest..

```
if (sig.verify(digest))  
{ /* Yay! */ }  
else { /* Boo! */ }
```

Verification of messages is straightforward. Given a digest, it is handed as input to the verify() method, which returns TRUE or FALSE.

# Handling Certificates



```
KeyStore store = KeyStore.getInstance ("jks");  
    // jks is the default keystore provider  
  
    FileInputStream fis = new FileInputStream ("certs");  
    // "certs" is the file name, of course..  
  
store.load (fis, "password". toCharArray());  
    // password provided for keystore..  
  
Certificate cert = store.getCertificate ("Required_Cert");  
    // Where "required_Cert" is the certificate you wish to retrieve...  
  
PublicKey pub = cert.getPublicKey();  
    // Retrieve the public key using the getPublicKey() method..
```



## Code Signing

### **X509v3 certificates can be used to sign code:**

- Create a JAR file containing the document or class file, using the jar tool.
- Generate keys using ***keytool -genkey***.

### **Optionally, request a certificate for key pair:**

- Use ***keytool -certreq*** ; then send the resulting request to a CA (VeriSign, Thawte, etc ).
- Use the ***keytool -import*** command to import the CA's response.

Code signing makes little sense if there is no certificate attached to it.

In order for a certificate to be creditable, you need to purchase one, from Thawte, VeriSign (In Israel, ComSign), or any other known and trusted root CA.

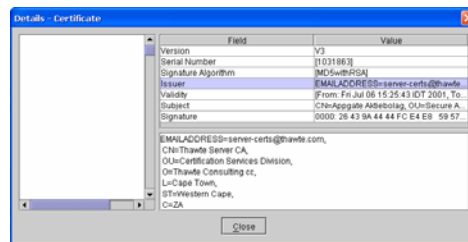
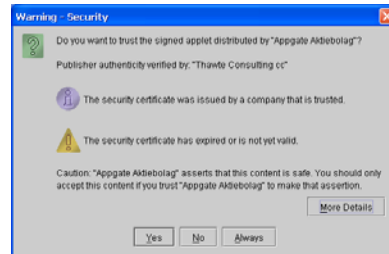


## Code Signing (II)

- Sign the JAR file, using the ***jarsigner***
- Export the certificate using

***keytool -export***

- Present the signed JAR and certificate



The bad news is, most people will simply click “OK” no matter what the box says.

The code signing approach has also been adopted by Microsoft, in ActiveX and in .Net (Strong Names)

## Code Signing (III)

- When used with policy files, code-signing can prove to be an invaluable security mechanism.

(Using `java.security.permission` –  
But that's besides our scope...)

- Microsoft .Net has also adopted this approach, in their “Secure Assemblies” mechanism.





## References

**O'reilly: Java Security, 2<sup>nd</sup> Ed.**

**IBM RedBook: Java 2 Security**

**(<http://www.redbooks.ibm.com>)**

**JSSE: <http://java.sun.com/products/jsse>**

**Java.sun.com – plenty of docs there.**

**<http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/>**

The End!

