Java & Cryptography - Part 1

Marc Sel - 1999

Abstract

This article provides a high-level overview of the basic concepts governing the deployment of cryptographic functions in Java. It briefly discusses the basic Java API and it introduces the Java security model. It then proceeds to review the Java Cryptographic Architecture, including basic methods and tools.

Java basics

A lot of excitement took place since Sun started shipping Java 1.0 in January 1996. As soon as March 1996, Microsoft licensed Java from Sun, and a number of other initiatives happened since then that helped to promote Java.

In Java, programming is object oriented. Objects are instances of a particular class. A program consists of one or more class definitions, each of which has been compiled into its own "class files" of JVM object code. One of the classes must define a *main(),* which will start running on a JVM. Every class is part of a package (if not explicitly indicated, the 'default' package).

The Java 1.1 API consists of 23 packages, defining classes and interfaces such as: *java.lang* (the very basics of the language), *java.applet* (for applets, obviously), *java.awt* (the abstract windows toolkit), *java.math, java.net* (sockets etc), *java.rmi* (remote method invocation), *java.security, java.sql* and many more.

To create Java programs, you need tools. The starting point for Java tools is Sun's JDK (Java Development Kit). Today both JDK 1.1 and 1.2 (also referred to as Java 2) are both popular. The JDK includes the compiler (*javac*) and interpreter (*java*), a basic tool to manage cryptographic keys (*javakey* / *keytool*), a tool to define security policies (*policytool*) and so on. Platform independent 'bytecode', produced by the compiler, runs under a JVM provided by Sun. Various third parties also provide JVM's and tools, including e.g. Transvirtual's Kaffé and HP's Chai (the arab word for 'tea'). Many IDEs (Integrated Development Environments) are available, from parties such as Sun, Symantec, Borland and even Microsoft.

Today, most popular browsers contain a JVM and are hence capable to execute Java code when you surf the web. Your browser's preferences and pop-up windows will set the limits here. Do you know yours?

Java security

A number of concepts are instrumental in enforcing Java security. These include features such as the lack of support for pointers and strong type checking to name but a few.

Java's security is strongly based on the 'sandbox' concept. Code, including applets, can not get out of the sandbox, unless explicitly authorized by the user. Fundamental in enforcing the sandbox are the Bytecode Verifier, the Classloader and the Security Manager.

The Bytecode Verifier assumes the bytecode could be created by a ‘hostile’ compiler, and hence verifies the format of the code. It then uses a theorem prover to ensure that bytecode does not over/underflow stacks, that instructions have parameters of the correct type (InputStreams is used as InputStreams, and nothing else), that no illegal data conversions ("casts") occur (eg treating an integer as a pointer), that private, public and protected class access are legal (no accesses to restricted interfaces attempted), and that register accesses and stores are valid.

The Class Loader makes sure key parts of the running environment will not be replaced, this includes e.g. that he’s the only allowed Class Loader under a browser.

The Security Manager will perform run time checks on methods. Java code consults the Security Manager whenever a ‘potentially dangerous’ operation is attempted, and the SecurityManager can veto the operation by generating a Security Exception.

Since Java 2, the Security Manager is further assisted by a dedicated Access Controller. When an agent or a JVM loads code from a codebase, the Security Manager and the Access Controller will rely on and co-operate with cryptographic functions to decide whether a particular operation (write to a socket, read a private key, verify a signature,…) is allowed or not.

Java and cryptography

The basic security class is *java.security* which arrived with JDK 1.1. Note that this already partially covered cryptographic concepts such as certificates and signatures.

Architecture

The overall design of cryptographic classes is governed by the Java Cryptographic Architecture (JCA), which specifies design patterns and an extensible architecture. Concepts are encapsulated in classes in the *java.security* (which can be exported from the U.S. without restrictions) and the *javax.crypto* packages (also referred to as the Java Cryptographic Extensions or JCE, where export restrictions apply). In the U.S., the JCE comes with the provider 'sunjce', supporting e.g. DES and 3DES. As could be expected, various companies outside the U.S. do provide alternatives to Sun's JCE. These include e.g. Baltimore (Ireland) and Brokat (Germany).

Together, the *java.security* and *javax.crypto* packages represent what you would expect from a cryptographic programming environment, such as ciphers, keys, key exchange protocols, certificates, key generators, hash and signature functions and much more.

The basic idea behind the JCA is that you use a factory method to request e.g. an algorithm, and a provider will supply it. The *javax.crypto.Cipher* class encapsulates both symmetric and asymmetric ciphers. *Cipher* is an abstract class, providing a factory method that returns useful instances such as actual implementations of RSA, DSA and so forth.

Here's how simple it is to generate a 1024 bit key pair suitable for DSA signatures in Java:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA"); find an implementation of DSA

kpg.initialize(1024); each algorithm has its specific initialization methods

KeyPair pair = kpg.genKeyPair(); and here we go.
```

It is the *java.security.Security* class' role to manage and keep track of those providers. Providers can be configured statically (by editing the */$javahome/lib/security/java.security* file) or dynamically (by calling *Security.addProvider()).*

Key management models

Generating keys is nice, but you need to be able to store and use them. JDK 1.1 relied on the Identity Key Management paradigm, which evolved around the *Identity* class, which represented something that possesses a public key. A *Signer* is an extension of an Identity, capable of signing with the corresponding private key. Maybe not really elegant, but it did the basic trick. JDK 1.1 came with j*avakey,* allowing you to maintain a database of identities and their associated keys and certificates. A.o. you can:

- create normal identities (which can be trusted or not - remember that trusted means the code can run OUTSIDE the sandbox...) (e.g. *'javakey -c Sanne true'* (i.e. Sanne is trusted but not a signer));
- create signers (which can again be trusted or not) (e.g. *'javakey -cs Doukje true'* (i.e. Doukje is both trusted and a signer));
- generate key pairs for signers (e..g. *'javakey -gk Doukje DSA 1024 DoukjePublic.x509 DoukjePrivate.x509'*);
- import keys (e.g. *'javakey -ik Trijntje TrijntjePublic.x509'*);
- create a self-signed certificate for starting up a CA such as C4root (e.g. *'javakey -gc C4root.directive'*) - here the "directive" file stores all the required statements supporting the signing process;
- create certificates for others (e.g. *'javakey -gc C4rootOverSanne.directive'*).

Private keys are stored in the *javakey* database. This file resides by default in the JDK installation directory and is called *'identitydb.obj'*. It is good practise to protect the *javakey* tool and the *java.security* and *identitydb.obj* files. Private keys can also be written out to files. Obviously, such files need to be protected then, typically by encrypting them, e.g. under a passphrase.

Java 2 (as 1.2 is now called by Sun) introduced the *KeyStore* Management paradigm, based on *java.security.Keystore*, a box that holds keys and certificates. *Keystores* have two types of entries: private keys and chains of certificates corresponding to the matching public key (referred to as a "private key entry") and certificates from parties you trust (referred to as "trusted certificate entry"). These entries co-operate with the *policytool* to define a security policy for a trusted code signer.

The major interface to the *Keystore* is the new *keytool*, handling the corresponding database. Its basic functionality is similar to its predecessor (*javakey)*, but it includes e.g. support for X.500 distinguished names (DN), and better database protection. It also allows you to create a Certificate Signing Request (CSR).

It is fair to state that Java provides an excellent foundation for professionally dealing with security in

an Internet-enabled world. But of course creating a full-fledged Certification Authority, an SSL implementation or a secured Client/Server solution still requires some hard work. Only think about supporting all PKCS standards (Public Key Cryptographic Standards - the de-facto standards created by RSA). How about the quality of the key generation? How about the quality of the randomness which is used?

Conclusion

It looks as if Java is here to stay. The elegance of the language led already to a fairly widespread acceptance. The publicity given in the context of the AES contest (Advanced Encryption Standard - the NIST initiative to establish a successor to DES) helped here too. Obviously, Java is still slower than C++, but as a trade-off, the maintainability of its code is much higher. And it remains to be see how far Sun (or other parties) will get in providing either hardware support of JIT (Just-In-Time) compilers.

Companies that were quick in adopting Java into their range of products and services were not the least. They include RSA (U.S.), Brokat (Germany), and Baltimore (Ireland), to name but a few.

Taking into account the ever increasing acceptance of Java, it can be assumed that it will play a role on the field of Internet-cryptography in general.

Look out for Part 2 of this article, which will address many items which have not yet been discussed here, including:

- the relationship between a typical browser, its cryptographic functions and Java;
- how to use Java to implement cryptographic algorithms;
- Java and SSL.