Java SE Security

Java security technology includes a large set of APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols. The Java security APIs span a wide range of areas, including cryptography, public key infrastructure, secure communication, authentication, and access control. Java security technology provides the developer with a comprehensive security framework for writing applications, and also provides the user or administrator with a set of tools to securely manage applications.

Java Cryptography Architecture (JCA)

The JCA is a major piece of the platform, and contains a "provider" architecture and a set of APIs for digital signatures, message digests (hashs), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation, to name a few. These APIs allow developers to easily integrate security into their application code.

Other cryptographic communication libraries available in the JDK use the JCA provider architecture, but are described elsewhere. The Java Secure Socket Extension (JSSE) provides access to Secure Socket Layer (SSL) and Transport Layer Security (TLS) implementations. The Java Generic Security Services (JGSS) (via Kerberos) APIs, and the Simple Authentication and Security Layer (SASL) can also be used for securely exchanging messages between communicating applications.

Design Principles

The JCA was designed around these principles:

- implementation independence and interoperability
- algorithm independence and extensibility

You can use cryptographic services, such as digital signatures and message digests, without worrying about the implementation details or even the algorithms that form the basis for these concepts. While complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate a specific implementation.

Algorithm independence is achieved by defining types of cryptographic "engines" (services), and defining classes that provide the functionality of these cryptographic engines. These classes are called engine classes, and examples are the MessageDigest, Signature, KeyFactory, KeyPairGenerator, and Cipher classes.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptographic Service Provider (CSP) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services. A program may simply request a particular type of object (such as a Signature object) implementing a particular service (such as the DSA signature algorithm) and get an implementation from one of the installed providers. If desired, a program may instead request an implementation from a specific provider. Providers may be updated transparently to the application, for example when faster or more secure versions are available.

Implementation interoperability means that various implementations can work with each other, use

each other's keys, or verify each other's signatures. This would mean, for example, that for the same algorithms, a key generated by one provider would be usable by another, and a signature generated by one provider would be verifiable by another.

Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily.

JCA Concepts

Engine Classes and Algorithms

An engine class provides the interface to a specific type of cryptographic service, independent of a particular cryptographic algorithm or provider. The engines either provide:
cryptographic operations (encryption, digital signatures, message digests, etc.),
generators or converters of cryptographic material (keys and algorithm parameters), or
objects (keystores or certificates) that encapsulate the cryptographic data and can be used at higher layers of abstraction.

The following engine classes are available:
SecureRandom: used to generate random or pseudo-random numbers.

MessageDigest: used to calculate the message digest (hash) of specified data.

Signature: initilialized with keys, these are used to sign data and verify digital signatures.

Cipher: initialized with keys, these used for encrypting/decrypting data. There are various types of algorithms: symmetric bulk encryption (e.g. AES, DES, DESede, Blowfish, IDEA), stream encryption (e.g. RC4), asymmetric encryption (e.g. RSA), and password-based encryption (PBE).

Message Authentication Codes (MAC): like MessageDigests, these also generate hash values, but are first initialized with keys to protect the integrity of messages.

KeyFactory: used to convert existing opaque cryptographic keys of type Key into key specifications (transparent representations of the underlying key material), and vice versa.

SecretKeyFactory: used to convert existing opaque cryptographic keys of type SecretKey into key specifications (transparent representations of the underlying key material), and vice versa. SecretKeyFactorys are specialized KeyFactorys that create secret (symmetric) keys only.

KeyPairGenerator: used to generate a new pair of public and private keys suitable for use with a specified algorithm.

KeyGenerator: used to generate new secret keys for use with a specified algorithm.

KeyAgreement: used by two or more parties to agree upon and establish a specific key to use for a particular cryptographic operation.

AlgorithmParameters: used to store the parameters for a particular algorithm, including parameter encoding and decoding.

AlgorithmParameterGenerator: used to generate a set of AlgorithmParameters suitable for a specified algorithm.

KeyStore: used to create and manage a keystore. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.

CertificateFactory: used to create public key certificates and Certificate Revocation Lists (CRLs).

CertPathBuilder: used to build certificate chains (also known as certification paths).

CertPathValidator: used to validate certificate chains.

CertStore: used to retrieve Certificates and CRLs from a repository.

Java Secure Socket Extension (JSSE)

The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol, such as Hypertext Transfer Protocol (HTTP), Telnet, or FTP, over TCP/IP.

JSSE provides both an application programming interface (API) framework and an implementation of that API. The JSSE API supplements the "core" network and cryptographic services defined by the java.security and java.net packages by providing extended networking socket classes, trust managers, key managers, SSLContexts, and a socket factory framework for encapsulating socket creation behavior. Because the socket APIs were based on a blocking I/O model, in JDK 5.0, a non-blocking SSLEngineAPI was introduced to allow implementations to choose their own I/O methods.

The JSSE API is capable of supporting SSL versions 2.0 and 3.0 and Transport Layer Security (TLS) 1.0. These security protocols encapsulate a normal bidirectional stream socket and the JSSE API adds transparent support for authentication, encryption, and integrity protection. The JSSE implementation shipped with Sun's JRE supports SSL 3.0 and TLS 1.0. It does not implement SSL 2.0.

Benefits connected with using JSSE:
1) Java provides a safe and secure platform for developing and running applications. Compile-time data type checking and automatic memory management leads to more robust code and reduces memory corruption and vulnerabilities. Bytecode verification ensures code conforms to the JVM specification and prevents hostile code from corrupting the runtime environment. Class loaders ensure that untrusted code cannot interfere with the running of other Java programs.

2) Comprehensive API with support for a wide range of cryptographic services including digital signatures, message digests, ciphers (symmetric, asymmetric, stream & block), message authentication codes, key generators and key factories

Support for a wide range of standard algorithms including RSA, DSA, AES, Triple DES, SHA, PKCS#5, RC2, and RC4.

PKCS#11 cryptographic token support

3) Enables single sign-on of multiple authentication mechanisms and fine-grained access to resources based on the identity of the user or code signer. Recent support (in JDK 5) for timestamped signatures makes it easier to deploy signed code by avoiding the need to re-sign code when the signer's certificate expires.

4) APIs and implementations for the following standards-based secure communications protocols: Transport Layer Security (TLS), Secure Sockets Layer (SSL), Kerberos (accessible through GSS-API), and the Simple Authentication and Security Layer (SASL). Full support for HTTPS over SSL/TLS is also included.

5) Tools for managing keys and certificates and comprehensive, abstract APIs with support for the following features and algorithms:

Certificates and Certificate Revocation Lists (CRLs): X.509

Certification Path Validators and Builders: PKIX (RFC 3280), On-line Certificate Status Protocol (OCSP)

KeyStores: PKCS#11, PKCS#12

Certificate Stores (Repositories): LDAP, java.util.Collection

As you can see, Java provides almost everything, what you need while creating applications which need cryptographical tools. But this advantage is also disadvatage because of its complexity.

Java Card versus Java

Language

At the language level, Java Card is a precise subset of Java: all language constructs of Java Card exist in Java and behave identically. This goes to the point that as part of a standard build cycle, a Java Card card program is compiled into a Java class file by a Java compiler, without any special option (the class file is post-processed by tools specific to the Java Card platform). However, many Java language features are not supported by Java Card (in particular types char, double, float and long; the transient qualifier; enums; arrays of more than one dimension; finalization; object cloning; threads); and some features are a runtime option missing in many actual smart cards (in particular type int which is the default type of a Java expression; and garbage collection of objects).

Bytecode

Java Card bytecode run by the Java Card Virtual Machine is a functional subset of Java [Java 2 - Standard Edition] bytecode run by a Java Virtual Machine, but uses a different encoding optimized for size. A Java Card applet thus typically uses less bytecode than the hypothetical Java applet obtained by compiling the same Java source code. This conserves memory, a necessity in resource constrained devices like smart cards. As a design tradeoff, there is no support for some Java language features (as mentioned above), and size limitations. Techniques exist for overcoming the size limitations, such as dividing the application's code into packages below the 64 KB limit.

Library and runtime

Standard Java Card class library and runtime support differs at lot from that in Java, and the common subset is minimal. For example, the Java Security Manager class is not supported in Java

Card, where security policies are implemented by the Java Card Virtual Machine; and transients (non-persistent, fast RAM variables that can be class members) are supported via a Java Card class library, while they have native language support in Java.

Development

Coding techniques used in a practical Java Card program differ significantly from that used in a Java program. Still, that Java Card uses a precise subset of the Java language speeds up the learning curve, and enables using a Java environment to develop and debug a Java Card program (caveat: even if debugging occurs with Java bytecode, make sure that the class file fits the limitation of Java Card language by converting it to Java Card bytecode; and test in a real Java Card smart card early on to get an idea of the performance); further, one can run and debug both the Java Card code for the application to be embedded in a smart card, and a Java application that will be in the host using the smart card, all working jointly in the same environement.

Java Card 3.0

The version 3.0 of the JavaCard specification (draft released in March 2008) is separated in two editions: the Classic Edition and the Connected Edition.
The Classic Edition is an evolution of the Java Card Platform Version 2.2.2 and supports traditional card applets on more resource-constrained devices.
The Connected Edition provides a new virtual machine and an enhanced execution environment with network-oriented features. Applications can be developed as classic card applets requested by APDU commands or as servlets using HTTP to support web-based schemes of communication (HTML, REST, SOAP ...) with the card. The runtime supports volatile objects (garbage collection), multithreading, inter-application communications facilities, persistence, transactions, card management facilities ...)

Provided algorithms and functions

Each card type has its own specification and available functions, which makes programming for Java Cards much more difficult. For example, from Java Card 2.2.2 are provided these additional cryptography algorithms : HMAC-MD5, HMAC-SHA1, SHA-256 and Korean Seed.

In comparison with Java SE Security, Java for SmartCards has these differences:
Security is better

- No dynamic class loading
- No threading
- Applet firewall
- Applet sources controlled

Security is worse

- No on-board byte code verifier
- No sandbox
- Applets are persistent
- Uploading

Assignments

1. Create a program (in Java) that connects to a POP3s server and outputs the server headers (certificate does not have to be verified). [Enclose the source code] {5}
2. Complete method *public boolean verifySignature(byte[] original, byte[] sig, PublicKey*

pub) in the SignHash class. [Enclose the source code] {5}

References

Documentation
http://java.sun.com/javase/6/docs/technotes/guides/security/index.html

Java Cryptography Architecture
http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

Java Secure Socket Extension
http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

Wikipedia
http://en.wikipedia.org/wiki/Java_Card
http://en.wikipedia.org/wiki/Java_(programming_language)