Global Information Assurance Certification Paper

Security Features Overview of Merlin (J2SE Version 1.4)
Craig Walker
December 1, 2001

Introduction

All the safeguards that we, as security professionals, employ are rendered useless if the foundation upon which they are laid is not sound. That is why Java™ has become the language of choice for the security minded application developer. From its inception, security was one of the primary tenets of the Java™ distributed computing platform. The language implemented several features to enforce secure programming including range checking on strings and arrays, garbage collection and automatic memory management. Runtime code legitimacy is insured by the byte code verifier and the Java™ Virtual Machine (JVM). The security manager in conjunction with the class loader enforce strict access policy for code operating on a machine. The Java™ 'sandbox' of JDK 1.0 created a new trust model for distributed and potentially malicious code. That model has been extended and redefined in subsequent JDK versions.

Building security in is preferred to bolting something on to protect an inherently insecure application. By providing the tools and infrastructure to easily create secure applications, Java will help 'sell' security as a cost effective and marketable solution. By making security cheap and easy, it is easier to justify in today's rapid time to market software economy. Consider how our jobs would change if we no longer had to worry about defending against the easily avoidable buffer overflow exploits; one attack method with so many targets.

Java™ 2 Standard Edition Security Features

The soon to be released Java™ 2 Standard Edition version 1.4 takes security to a new plateau by combining, in the language core, security features to support confidentiality, integrity, and availability. The following security features renew Sun's commitment to secure computing through Java.

- Java Cryptographic Extensions (JCE)
- Java Secure Socket Extensions (JSSE)
- Java Authentication and Authorization Service (JAAS)
- Generic Security Services Application Program Interface (GSS-API)
- Java Certification Path API

Java Cryptographic Extensions (JCE)

The JCE provides an extensible framework for implementing various cryptographic operations and algorithms. JCE provides functionality to support encryption, key generation and agreement, and message authentication code (MAC).

The Java Cryptographic Extension provider implementations that ship with J2SE 1.4 include:

- Key generation algorithms for DES, Triple DES, Blowfish, MD5, and SHA1
- Symmetric encryption algorithms for DES, Triple DES, and Blowfish
- Diffie-Hellman implementation that includes key agreement algorithm, key pair generator, and an algorithm parameter generator.
- Hashing algorithms MD5 and SHA1.
- A padding scheme for PKCS#5
- A keystore implementation type named JCEKS

JCE uses the notion of a service provider architecture to create a very open and flexible cryptographic infrastructure. This architecture which is depicted in Figure 1, has become the norm for providing independent and extensible services. The cryptographic services are defined very generically in the service provider interface (SPI) layer and then implemented as a provider of that service. This allows for the simple insertion or removal of cryptographic providers. Clients may configure their runtimes with different providers (acceptable algorithms for instance) and specify a preference order. This preference order can be used to help the application select the strongest supported algorithm. As new algorithms are created, they simply need to be implemented as a service provider and plugged in to the architecture, having little or no effect on the applications that are using the service.

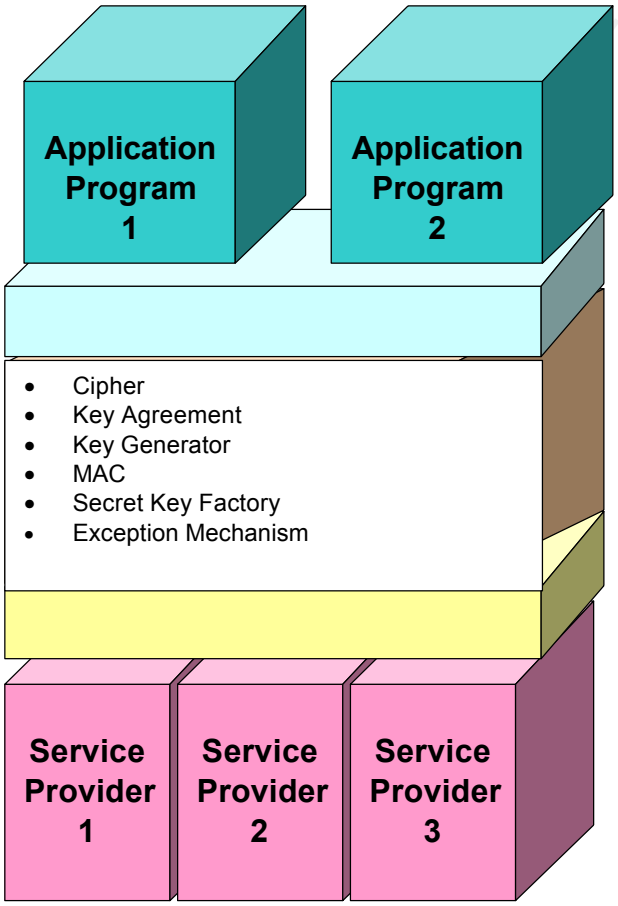

Figure 1

The Java Secure Sockets Extension

J2SE 1.4 will also include the Java Secure Sockets Extension (JSSE). JSSE provides implementations and support for Secure Sockets Layer version 3 as well as Transport Layer Security (TLS) version 1. The inclusion of the SSLSocket and SSLServerSocket classes provides an easy transition to secure channel programming. As you can see from the code fragments in Code Sample 1, it is a trivial matter to convert a socket based application to a secure communication channel.[1]

```
// A sample socket program fragment to get a page from port 80
Socket socket = new Socket "www.someNONsecuresite.com", 80);
PrintWriter out = new PrintWriter(
          new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())));
out.println("GET http://www.someNONsecuresite.com/index.html HTTP/ 1.1"
out.flush();

// The same program fragment over a secure socket
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) factory.createSocket("www.someSECUREsite.com", 443);
PrintWriter out = new PrintWriter(
          new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())));
out.println("GET http://www.someSECUREsite.com/index.html HTTP/ 1.1"
out.flush();
```

Code Sample 1

JSSE additionally provides support utilities for key and certificate management, cipher negotiation, and client and server authentication. Table 1 outlines the cryptographic suites along with the cipher strengths supported.

Cryptographic Suite	Key Lengths (Bits)
RSA (authentication and key exchange)	2048 (authentication) 2048 (key exchange) 512 (key exchange)
RC4 (bulk encryption)	128 128 (40 effective)
DES (bulk encryption)	64 (56 effective) 64 (40 effective)
Triple DES (bulk encryption)	192 (112 effective)
Diffie-Hellman (key agreement)	1024 512
DSA (authentication)	1024

Table 1

JSSE implements the mechanism for cipher negotiation through the SSL 'handshake'.

This process involves:

1. The client sends its version number, cipher settings, and some session specific information.
2. The server responds with the same information along with its certificate.
3. The client authenticates the server certificate with the appropriate Certificate Authority.
4. Using the agreed upon information thus far, the client generates the pre-master secret key and encrypts it with the server public key. It then sends the encrypted key back to the server to be used in subsequent transmittals.
5. The server can then optionally validate the client (if it has requested a client certificate in the previous step).
6. Both the client and the server then generate the session key from the pre-master secret. This session key will be used to encrypt all further secure communications during this session.
7. Both systems then exchange an SSL handshake complete message and begin secure communications.

Java Authentication and Authorization Service (JAAS)

JAAS, as the name implies, is intended to be used for the authentication of users as well as resource access control (authorization). It is implemented (similar to the JCE described above) in a modular fashion, isolating applications from the underlying authentication services.

Authentication involves a user providing some proof that they are indeed who they claim to be. This proof may take the form of something they have (smartcard or token), something they know (password), something they are (biometric), or a combination of these methods. As authentication methods change, it is important to minimize the impact on dependent application. The modular nature of JAAS provides just such an independence. Dual (or even triple) mode authentication can also be implemented easily in JAAS through the notion of stackable authentication modules, similar to the UNIX Pluggable Authentication Modules (PAM). These modules use a two phase commit to ensure that either all authentication methods pass or they all fail. In the first phase (login phase), each module is instructed to do authentication only. The second phase (commit phase) is only invoked if all modules pass authentication. In the commit phase all modules are called once again to provide the appropriate credentials for the subject.

Once identity has been established through authentication, a system needs to determine whether the principal has the appropriate authority to use the protected resources. This authorization mechanism is implemented through the Java SecurityManager when untrusted code requires sensitive system resources. The security policy for JAAS is consistent with but extends the Java 2 Codesource-based security model (code signing) by adding principal based authorization. The current and default implementation uses a local file to define the security policy.

Generic Security Services Application Program Interface (GSS-API)

The Java GSS-API is the mechanism for secure message exchange between applications providing principal authentication, delegation, and message confidentiality and integrity assurance. The GSS-API is defined in RFC 2853. The two mechanisms defined in RFC 2853 are the simple public-key GSS-API mechanism and the Kerberos version 5 GSS-API mechanism.

GSS-API and JSSE contain very similar security functionality. There are, however, several factors that determine the best choice for a given application. If your target platform includes a Kerberos Version 5 implementation that you want to use for authentication, GSS-API may be a better choice since JSSE does not support it. If you are converting a socket based application to a secure socket version, JSSE is probably the right choice. If, on the other hand, you application communicates via UDP or it passes both encrypted and non-encrypted messages, GSS-API is probably more appropriate. GSS-API allows client delegation and impersonation, features not in JSSE.

There are four steps involved in setting up a secure communication with GSS-API. In the first step, the application acquires a credential that will identify it to another application. These two applications then develop a shared security context using GSSContext. This context contains mutual information including cryptographic keys and message sequence numbers. The third step involves using the methods of GSSContext to invoke per message services for authentication, integrity, and confidentiality. The final step 'cleans up' any resources that were used during the communication and releases the context.

Java Certification Path API

The Java Certification Path API provides certificate validation and mapping functions. It can be used to create, build, and validate certification paths. It uses a similar modular architecture to the JCE, employing a provider-based architecture. This is an important API because the entire trust model upon which our e-commerce world is based upon relies on the effective traversal of a list of certificates until an appropriate authority is located.

The API includes interfaces and implementations for four certificate functions, Basic, Validation, Building, and Storage. The basic certification path classes contain the core functionality for representing and encoding certificate paths. The validation classes, as the name implies, handle certificate path validation. The building classes are for creating or automating the discovery of certificate paths. And finally, the storage classes allow for the storage of certificates and revocation list discovery.

Conclusion

Included in the upcoming release of the J2SE version 1.4, the security provider architecture creates a desirable independence between system services and

applications. It will also provide application programmers with many of the tools necessary to implement secure services. We can only hope that this heralds a new era of secure applications.

References

1. Sun Microsystems Inc., "Java™ Secure Socket Extensions" http://java.sun.com/products/jsse/index-14.html (Jun 29, 2001)
2. Sun Microsystems Inc., "Java™ Cryptography Extension (JCE)" http://java.sun.com/products/jce/index-14.html (September 11, 2001)
3. Sun Microsystems Inc., "Java™ Certification Path API Programmer's Guide" http://java.sun.com/j2se/1.4/docs/guide/security/certpath/CertPathProgGuide.html (October 22, 2001)
4. The Internet Engineering Task Force (IETF), "Generic Security Service API Version 2 : Java Bindings" http://www.ietf.org/rfc/rfc2853.txt (June, 2000)
5. Sun Microsystems Inc., "When to use Java GSS-API vs. JSSE" http://java.sun.com/j2se/1.4/docs/guide/security/jgss/tutorials/JGSSvsJSSE.html (November 30, 2001)
6. Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers, "USER AUTHENTICATION AND AUTHORIZATION IN THE JAVA(TM) PLATFORM" http://java.sun.com/security/jaas/doc/acsac.html (December, 1999)
7. Microsoft Corporation, "Description of the Secure Sockets Layer (SSL) Handshake" http://support.microsoft.com/default.aspx?scid=kb;EN-US;q257591 (April 6, 2000)

[1] The code is meant to highlight the similarities between socket and secure socket programs. There are many underlying details that have been excluded for clarity.

Last Updated: May 28th, 2017

Upcoming Training

SANS Stockholm 2017	Stockholm, Sweden	May 29, 2017 - Jun 03, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
Security Operations Center Summit & Training	Washington, DC	Jun 05, 2017 - Jun 12, 2017	Live Event
Community SANS Ottawa SEC401	Ottawa, ON	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC401: Security Essentials Bootcamp Style	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
Community SANS Portland SEC401	Portland, OR	Jun 12, 2017 - Jun 17, 2017	Community SANS
SANS Secure Europe 2017	Amsterdam, Netherlands	Jun 12, 2017 - Jun 20, 2017	Live Event
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Minneapolis SEC401	Minneapolis, MN	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Phoenix SEC401	Phoenix, AZ	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Mentor Session - SEC401	Ventura, CA	Jul 12, 2017 - Sep 13, 2017	Mentor
Mentor Session - SEC401	Macon, GA	Jul 12, 2017 - Aug 23, 2017	Mentor
Community SANS Atlanta SEC401	Atlanta, GA	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Colorado Springs SEC401	Colorado Springs, CO	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANSFIRE 2017 - SEC401: Security Essentials Bootcamp Style	Washington, DC	Jul 24, 2017 - Jul 29, 2017	vLive
Community SANS Charleston SEC401	Charleston, SC	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Fort Lauderdale SEC401	Fort Lauderdale, FL	Jul 31, 2017 - Aug 05, 2017	Community SANS
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event