A Java Cryptography Service Provider Implementing One-Time Pad

Timothy E. Lindquist, Mohamed Diarra, and Bruce R. Millard
Electronics and Computer Engineering Technology
Arizona State University East
http://www.east.asu.edu/ctas/ecet
mailto:Tim@asu.edu

Abstract

Security is a challenging aspect of communications today that touches many areas including memory space, processing speed, code development and maintenance issues. When it comes to dealing with lightweight computing devices, each of these problems is amplified. In an attempt to address some of these problems, SUN's Java 2 Standard Edition version 1.4 includes the Java Cryptography Architecture (JCA). The JCA provides a single encryption API for application developers within a framework where multiple service providers may implement different algorithms. To the extent possible application developers have available multiple encryption technologies through a framework of common classes, interfaces and methods.

The One Time Pad encryption method is a simple and reliable cryptographic algorithm whose characteristics make it attractive for communication with limited computing devices. The major difficulty of the One-Time pad is key distribution.In this paper, we present an implementation of One-Time Pad as a JCA service provider, and demonstrate its usefulness on Palm devices.

1. Problem

Dependence on the communications infrastructure continues to grow as the size of computing devices decreases. The growing dependence on Internet accessibility to services that do not reside in a local machine brings with it the need for secure communications. The target of this work are relatively small devices and their related systems, such as Windows CE, Palm^{TE}, Handspring and cell phones used to access Internet services. While several large computer service organizations have spent millions of dollars recovering from cyber attacks, the potential economic impact of insecure e-commerce communications on limited devices is huge[1], [3].

Java continues to enjoy dominance in server-side technologies, however, a small but growing number of limited device applications are developed in Java. Nevertheless, *Sun Microsystems Inc.*, added Java Cryptography Extension (JCE) and JCA (to the Java™ 2 Development Kit Standard Edition v1.4 (J2SDK), and has created a substantial market for applications running on J2ME (Java 2 Micro Edition). Other vendors are offering Java runtimes for limited devices. These versions bring Java to client application developers [9], [11], and raise the issue of appropriate Java-based security mechanisms.

J2ME does not include JCE and JCA, however *The Legion Of The Bouncy Castle* has developed a lightweight Cryptography API and a Provider for JCE and JCA [14]. Neither provider offers implementation of the One-Time Pad cryptography service [14].

The simplicity of the One-Time Pad method and the fact that it does not require high processor speed, make it ideal for lightweight computing devices.

1.1 Context

This paper focuses on integrating the JCA cryptography service provider, starting by defining the engine classes and then implementing the One-Time Pad method. We include simple evaluation programs to test the provider. The problem of pad distribution is one of the tasks taken-on in order to have successful deployment.

Implementations of the one-time pad encryption-0.9.4 are readily available. For example, one product is available for Windows command line launching. The source code written in ANSI-C and DOS executable are available for download at http://www.vidwest.com/otp/ [1].

The Security documentation provided with J2SDK includes detailed information on the implementation of

the Provider for the JCA [12]. The documentation for "a Provider for JCE and JCA" of *The Legion Of The Bouncy Castle* is also available [14].

The possibility of using the One-Time pad for data encryption and decryption for security purposes on lightweight computing devices was covered at the 35th Hawaii International Conference on System Science 2002 [2].

2. One Time Pad

The one-time pad algorithm is among the simplest in the world of cryptography and is considered by some to be unbreakable. It is nothing more than an exclusive OR between the message (to be encrypted) and the pad (a random key - sequence of bits). The principles that govern the encryption technique are not that simple to apply. First, the key must be random, which by itself is a big challenge. Second, parts of the key that have already been used to do encryption must not be available for other encryption. The key (Pad) must be a sequence of random bits as long as the message to be encrypted. The sender exclusive-OR's the message with the pad and sends the result through a communication channel. The one time requirement that makes it unbreakable and difficult at the same time is that after use, the sender must get rid of part of the pad, and not use it again. At the other end of the communication, the receiver must have an identical copy of the pad. The receiver decrypts the cipher text to obtain the original message by doing an exclusive-OR of the incoming cipher with its copy of the pad [1], [3], [4]. The receiver should also destroy the pad after use. See Figure 1.

2.1 Advantages of the One Time Pad

If the pad is actually random and has been distributed securely to the receiver, then no third party can decrypt the message. Even guessing part of the key will not allow a third party to determine the remainder. This is why some people claim that one-time pad is unbreakable. While there are a number of very good pseudo-random number generators, so far any attempt to generate a truly random key with computers appears to generate the same sequence after a certain point. Several approaches avoid this problem by personalizing the key. Another advantage of this technique resides in the simplicity of its algorithm. It does not involve complex operations that challenge the computational speed of some relatively small processors.

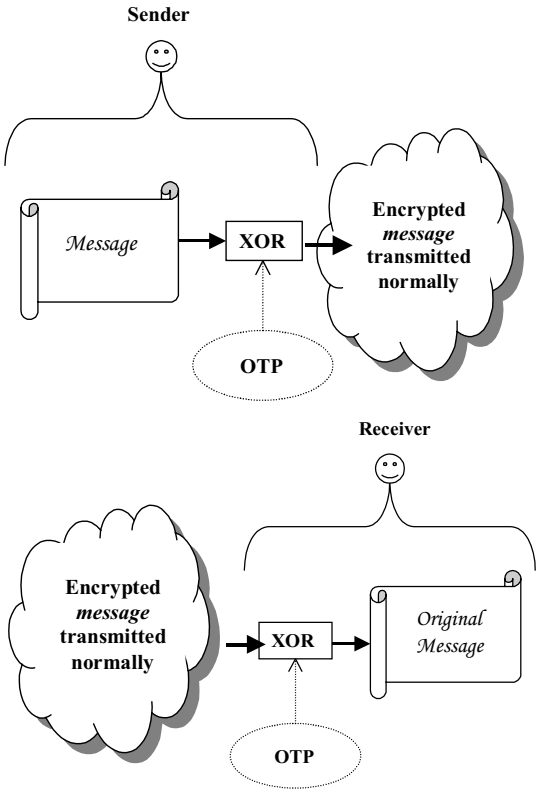

Figure 1. One-time pad (OTP) cryptography.

2.2 Disadvantages of the One Time Pad

The key must be as large as the message being encrypted; this fact is sometimes inconvenient especially in the case of large messages. The principle of the one-time pad is to have a unique key for each communication, which makes the generation and management of keys problematic as the number of recipients and frequency of use escalates. The last challenging aspect of the One Time Pad is the Key distribution. In fact the key should remain undisclosed (secret) to any other party besides the communicating parties. Extensions to the One-Time Pad provider discussed in this paper center on portable memory devices [15][16] that are frequently synchronized with more capable machine (laptops or desktops) for key exchange. However, our implementation is aimed at handheld devices in general. General purpose (non-proprietary) portable memory interfaces for handheld devices don't exist yet, so another approach is necessary. Some possibilities are discussed in the Key Management section below. Each application must deal with this issue in its own effective way(s).

3. Java Cryptography Architecture

The Java security model has been evolving to adjust to new security issues. The JCA is a framework providing cryptography functionality development capabilities for a Java platform. It was introduced early in Java's evolution as an add-on package. The first release of the Security API was an extension of JCA including API's for encryption, key exchange, and coding message authentication. Prior to J2SDK 1.4, JCE was optional, in part due to export restrictions. The "Java Secure Socket Extension" (JSSE) and "Java Authentication and Authorization Service" (JAAS) security features have also been integrated into the J2SDK, version 1.4. Two new security features have been introduced: "Java GSS-API" (Java Generic Security Services Application Program Interface) that can be used for securely exchanging messages between communicating applications using the Kerberos V5 mechanism and "Java Certification Path API" that includes classes and methods in the *java.security.cert* package. These classes allow the developer to build and validate certificate chains.

The java cryptography architecture includes a provider architecture [2], [5], [6], [7], [10], [12]. The notion of Cryptography Service Provider (CSP), or just provider, has been introduced in JCA. The provider architecture allows for multiple and interoperable cryptography implementations. An application developer can create or specify his/her own cryptography service provider. The service provider interface (SPI) presents a single interface for implementors. Classes, methods and properties are accessible to applications through the JCA application program interface (API). The SPI allows a cryptography service provider to plug-in implementations for java applications. A provider can be used to implement any security service. Several providers can be available and they may or may not provide similar cryptography services and algorithms. Figure 2 depicts the layers of Java Cryptography Architecture, and is taken from Sun Java documentation [6].

A given installation of J2SDK may have several cryptography service providers installed, which may provide implementations of different algorithms and/or may provide multiple implementations of a single cryptography algorithm. Each provider has a name that is used by application programmers to specify the desired provider. It is also possible to specify the order of preference of providers. The default provider that comes with the J2SDK is the Sun provider, which includes a wide variety of cryptographic algorithms and tools [2], [5], [6], [7], [10], [12].

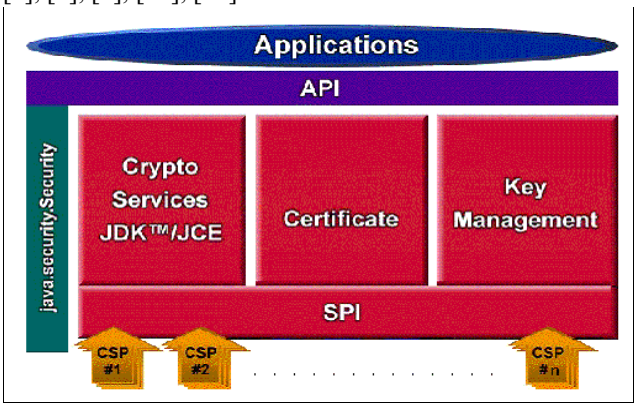

Figure 2. Java cryptography architecture

3.1 Java Cryptographic Service Providers

The Java Cryptography Architecture, which includes the provider(s), has two main design principles, First, is independence from implementation and interoperability: This derives from using the services without knowing their implementation details [12]. Second, is algorithm independence and extensibility; meaning that new service providers and/or algorithms can be added without effecting existing providers. Together these provide a modular architecture that allows for encryption to be done by an implementation of a specific algorithm and subsequent decryption to be done by another implementation.

3.2 Provider Implementing One Time Pad

The primary question when building such a provider is: does the nature of the one-time pad allow it to be implemented in a pluggable architecture? Figure 1 shows the interoperability between Sender and Receiver as independent systems in communication. The element they are required to have in common is the key. The implementation of the algorithm does not matter. As far as extensibility is concerned, it is up to the provider programmer to remain independent of other cryptographic services.

A Cryptographic Service provider is a package or set of packages providing concrete implementations of a subset of the cryptography portion of the Java security SPI. The Java Security Guide in J2SDK, v1.4 documentation lists a series of nine steps to follow for implementing a Provider. This paper follows those steps as guidelines for its development.

Two aspects in the structures of cryptographic service were needed to write the implementation code: the Engine Class and the Service Provider Interface (SPI).

3.3 Engine Class

An Engine Class is an abstraction of a cryptographic service. It defines the service without a concrete implementation of the particular associated algorithms. Applications access instances of the engine class through the API, to carry out available operations. Every engine class has a corresponding service provider interface, which provides abstract classes accessing the engine class features. The service provider interface indicates all the methods that the actual cryptographic service provider should implement for a particular cryptographic algorithm or type. A service provider interface is named with its engine class name followed by "Spi" [12].

For each service that a provider implements, we must define the engine class, and then write its service provider interface. For the One Time Pad technique, the service provider interface's abstract class is called *OneTimePadSpi.* The engine class for *OneTimePadSpi* in compliance to the nomenclature of JCA is called *OneTimePad.* The engine class is a concrete subclass of the service provider interface, implementing all the abstract methods.

The provider class is a final subclass of ***java.security.provider***. Our provider is named *ASUEcetProvider*. The provider name is used by applications to access our one-time pad service [12].

3.4 Provider's Information

The provider class provides access to various properties of the service, including the version, and other information about the service(s) it provides such as algorithm, type, and techniques [2], [12]. The value provided for this argument in this project is: *"ASUEcetProvider v1.0, implementing One Time Pad (OTP) cryptographic technique, Arizona State University East, Electronic and Computer Engineering Technology. May 2003"*

3.5 Install and Configure the Provider

The provider needs to be correctly installed and configured for the application program to utilize its cryptographic service(s). There are two different ways to install a provider. The first method consists of creating a JAR file (Java Archive File) or ZIP file containing all the class files belonging to the Cryptography Service Provider. The JAR file is added to the CLASSPATH environment variable. The exact steps of doing this last action, depends upon the local operating system [2], [12]. The second approach deploys the provider's JAR file of classes as an extension (optional package) to Java. The file can be bundled with a particular application (with a manifest indicating relative URLs), or it can be installed in the Java Runtime Environment to be shared by all running applications.

3.6 Registering the Service

Configuring the service provider enables client access to the service(s) by registering the provider and defining default preferences where more than one provider is registered for the same service algorithm.

Static Registration consists of editing the *java.security* file (located in "*lib\security*" subdirectory of the Java Runtime Environment) to add the provider name to the list of approved providers. For each available provider for a given algorithm, there is a corresponding line in the *java.security* file with the form:
security.provider.<n>=<providerClassName>
Where "**n**" is the preference number for the provider. For example the line:
security.provider.2=asue.provider.ASUEcetProvider
registers *our provider* with an order of preference 2.

Dynamic Registration can be done by a client application upon requesting service(s) from a provider. The client application calls a class method, such as:
Security.addProvider (Provider providerName).

3.7 Test Programs and Documentation

Several test programs were written to exercise three aspects of the service provider. For client applications to be able to request service(s) provided by a specific provider, the provider should be successfully registered with the security API. A simple test program can verify registration by creating an instance of the provider and accessing its name, version, and info (*getName (), getVersion ()*, and *getInfo ()* methods.

After making sure that the provider is accessible from the security API, we need to retrieve the provided service(s) by calling its "*getAlgorithm ()*" method.

Finally, we check the functionality provided by the service by writing sender and receiver applications that use the service.

In addition to providing sample service test programs, our implementation provides documentation. Our documentation is generated by the *Javadoc* tool from the source code and it targets application programmers.

4. Key Management

The generation and distribution of the random keys for this method is of primary concern. Since the size of the key must match the size of the message being

encrypted, we limit our application to transmission of relatively small messages. The key can be any random array of bits, the key type used in JCA has not been used.

The *OneTimePad* class has been designed, so that when a new random key is generated, it is stored in an external file that could be later sent to the receivers. There are many approaches to providing random keys for the One-Time Pad. The primary problem with using the One-Time Pad is the amount of random bits needed. Every message sent needs a key of the same length. If every message is sent encrypted, the required key space becomes large very quickly. The alternative of only encrypting sensitive data suffers from forcing the application to make these choices, and only delays the issue of when and how to replenish the key once all the bits have been used on earlier messages.

Handheld devices, the target for this work, generally only send small messages requiring encryption. This allows for many messages using a small key of say 1 MB. A nightly synchronization using a recharging cradle can be used to also replenish the One-Time Pad key. For devices that don't have a connection to a host while charging, another approach is portable memory devices [15][16]. With a 2 GB pad, the replenishment cycle would be much longer. Generally long enough to add a few jpeg or gif pictures a day. The primary problem with portable memory and handheld devices is that of interfacing requirements. Currently, no general-purpose interfaces (e.g., USB) are available for handhelds. Portable memory devices, to be useful, must come with general purpose interfaces, such as USB. An alternative to the cradle and portable memory approach is to update or replenish the pad on-line. In this approach, a One-Time Pad is used until almost exhausted. Then the remaining pad is used to exchange a block-cipher key for a more computationally complex cryptography algorithm, such as RC4 or AES. The One-Time Pad can then be generated by the server and sent to the handheld using the complex cipher. The encryption process would then return to the One-Time Pad methodology. This approach is expensive on both network and CPU utilization. It may only be acceptable on larger handhelds such as Windows CE (Pocket PC) machines. A final alternative would be to use PRNG (pseudo random number generator) that is cryptographically appropriate. An example PRNG is ISAAC [17]. ISAAC is a relatively fast random number generator with a very long cycle (i.e., guaranteed 2^{40} with an average 2^{8295}). Generating random pads then requires knowing the seed. Using the prior approach of a trailing seed from the last pad as a seed for the new pad would permit, possibly, near real-time generation of One-Time Pads.

5. Running on Limited Devices

The fact that JCA and JCE are not part of J2ME, limits its applicability to our intended application space. One approach is to configure limited client applications by embedding the provider directly in the deployed J2ME application.

Another approach is to use the lightweight cryptography API defined by *The Legion Of The Bouncy Castle* to develop a provider based on their design principle of *A provider for the JCE and JCA [14]*. This solution results with a provider not fitting exactly the Java Cryptography Architecture, but which is usable on J2ME devices, such as PalmOS [14].

6. Conclusions and Enhancements

As long as electronic communication continues to expand, security will remain an issue. Cryptography is one of the most effective tools available to address these issues. One-time pad is among the most powerful existing cryptographic techniques, providing it is used within the constraints of its applicability. Primarily the constraints are key management, compute limited devices and encryption tasks that do not require encrypting large files. The Java Cryptography Architecture offers the potential of a single interface for applications that allows plug-in of any number of participating service providers. The approach allows evolution of security approaches with the promise of minimal impact on applications.

Security remains an open field on every computing platform. As platforms continue to evolve to smaller and better connected devices, they will meet the information needs of a broader range of consumers. Its importance to provide frameworks for developing secure distributed and web-based applications on such mobile devices.

7. References

[1.] d@vidwest.net (2000, November 17). "One Time Pad Encryption v0.9.4" Retrieved January 25, 2002 from the World Wide Web: http://www.vidwest.com/otp/

[2.] Gong, L. (1998). "Java™ 2 Platform Security Architecture Version 1.1". *Sun Microsystems, Inc.* Retrieved February 20, 2003 from the World Wide Web: http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-spec.doc.html

[3.] Jenkin M. & Dymond P. (2002) "Secure communication between lightweight computing devices over the Internet". HICSS 35 January 2002.

[4.] Kahn D. (1967) The Codebreakers, New York, NY, MacMillan.

[5.] Knudsen J. (1998) Java Cryptography, Sebastopol, CA,

O'Reilly.

[6.] Lindquist T. (2003, January 14). "Cet427/598 Distributed Object Systems". Retrieved February 02, 2003 from the World Wide Web: http://pooh.east.asu.edu/Cet427/ClassNotes/Security/cnSecurity.html

[7.] McGraw, G. and Felten, E. (1996) Java Security: Hostile Applets, Holes, and Antidotes. New York, NY. John Wiley & Sons.

[8.] McGraw, G. (1998) "Testing for security during development: why we should scrap penetrate and patch". *IEEE Aerospace and Electronic Systems*, 13(4):13-15, April 1998.

[9.] Muchow J. (2001) Core J2ME™ Technology and MIDP, Upper Saddle River, NJ, Prentice Hall.

[10.] Oaks, S. (1998) Java Security, Sebastopol, CA, O'Reilly & Associates.

[11.] Rubin, A, Geer, D. and Ranum, M. (1997) The Web Security Sourcebook. New York, NY, John Wiley & Sons.

[12.] Sun Microsystems, Inc (2002). "Java 2 Platform, Standard Edition, v 1.4.0 API Specification". Retried January 12, 2002 from World Wide Web: http://java.sun.com/docs/

[13.] Sundsted T. (2001). "Java, J2ME, and Cryptography" Retrieved March 20, 2003 from the World Wide Web: http://www.itworld.com/nl/java_sec/10262001/

[14.] The Legion Of The Bouncy Castle (2000). "The Bouncy Castle Crypto APIs" Retrieved March 20, 2003 from the World Wide Web: http://www.bouncycastle.org/index.html

[15.] Zbar, Jeff, "Portable Memory Gets Small," retrieved Sept. 2003; http://www.beststuff.com/article.php3?story_id=4395.

[16.] Lexar Media, "Samsung Sampling 2Gb NAND Flash Memory Devices to Lexar Media," retrieved Sept. 2003; http://www.digitalfilm.com/newsroom/press/press_02_25_02a.html.

[17.] Jenkins, Bob, "ISAAC: a fast cryptographic random number generator," retrieved Sept. 2003; http://www.burtleburtle.net/bob/rand/isaacafa.html.