Manual de Seguridad

Sistema de Inscripciones Extraescolares

Versión 1.0

Cliente: [ITESM]

Nombre	Matrícula
Charly Eid	717789
Roberto Domínguez	717836
Rodrigo Tarrats	719169
Armando Betancourt	717343

Materia: *Seguridad en Cómputo*
Dirigido: *Ing. Arturo García*
Fecha: *20/Noviembre/2001*
Elaborado: *Symbiosys*
Validado: *Symbiosys*
Autorizado: *Symbiosys*

Introducción a las Características de Seguridad en Java 2 JDK 1.2 ____ 4

Extensiones de la Arquitectura de Seguridad ____ 4
Extensiones de Arquitectura Criptográfica ____ 6
Servicios de criptografía ____ 7
Interfaces y Clases para Certificados en el Sistema IE ____ 8
Interfaces y Clases para Manejo de Llaves en IE ____ 8
Herramientas Relacionadas con la Seguridad ____ 9

Guía Rápida para Controlar Aplicaciones en IE ____ 10

Libertad de las Aplicaciones ____ 10
Cómo Restringir las Aplicaciones ____ 11
Propiedades Sensibles a la Seguridad ____ 12
El Archivo de Política del Sistema ____ 12
Configurar el Archivo de Política para Conceder los Permisos Requeridos ____ 13
Abrir el Archivo de Política ____ 13
Conceder los Permisos Requeridos ____ 14
Grabar el Archivo de Política ____ 20
Efectos del Archivo de Política ____ 20

API y Herramientas para Código Seguro e Intercambio de Archivos en IE ____ 21

Seguridad de Código y Documentos ____ 21
Firmas Digitales ____ 21
Certificados ____ 21
Keystores (Repositorios de Llaves) ____ 22
Características Del API De Seguridad del JDK que utilizaremos en IE ____ 23
Uso De Herramientas Para Firmar Código en el sistema IE ____ 23
Generar una Petición de Firma de Certificado (CSR) ____ 24
Importar la Respuesta del CA ____ 24
Importar a IE un Certificado desde un CA como un "Certificado confiable" ____ 25
Importar el Certificado Devuelto por el CA ____ 25
Fimar Código y Conceder Permisos ____ 26
Pasos para el que Firma el Código ____ 27
Pasos para el Receptor del Código ____ 30

Glosario de Términos de Seguridad en Java 2 ____ 38

Términos sobre Seguridad ____ 38
Certificado ____ 38
Algoritmo de Criptografía ____ 38
Desencriptación ____ 38
Firma Digital ____ 38
Dominio o Protección de Dominio ____ 39
Encriptación ____ 39
Clase Motor ____ 39
Algoritmo Resumen de Mensaje ____ 40
Representación de Clave Oscura ____ 40
Representación Oscura de Parámetros ____ 41
Permiso ____ 41
Política ____ 41
Archivo de Política ____ 41
Clave Privada ____ 41
Privilegiado ____ 42
Proveedor ____ 42

Clave Pública 42
Controlador de Seguridad 42
Certificado Auto-Firmado 43
Código Firmado 43
Representación de Clave Transparente 43
Representación de Parámetros Transparente 43

Resumen del API de Seguridad en Java 2 44

¿Qué Proporciona el API de Seguridad del JDK en beneficio al sistema IE? 44
Métodos del API 44
Personalización de las Características de Seguridad 44
¿Qué pasa con la Encriptación y la Desencriptación en IE? 45

Sumario de Archivos de Seguridad en Java 2 46

El archivo de Propiedades de Seguridad java.security 46
El archivo de Política del Sistema java.policy 47
El KeyStore de Certificados cacerts 47
El archivo de Política de Usuario.java.policy 47
Keystores 48

Referencias 49

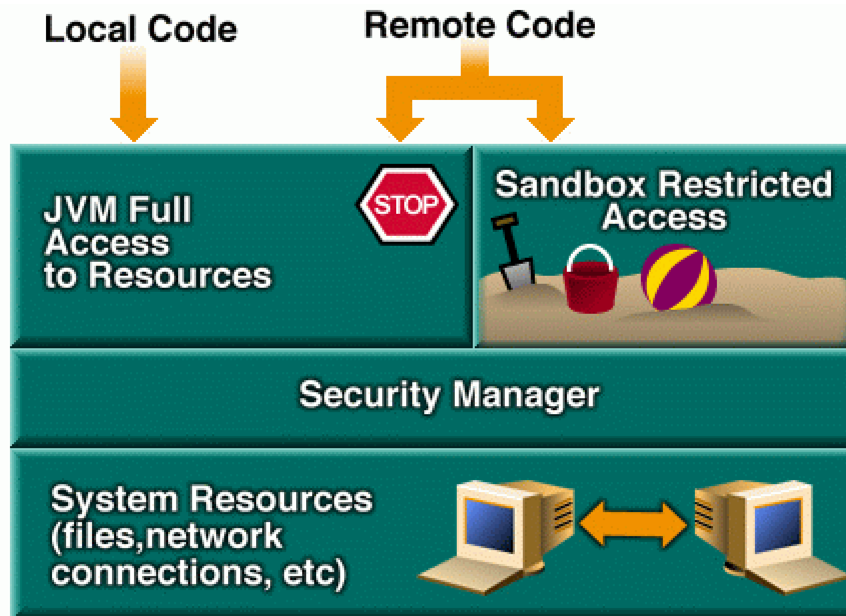
Introducción a las Características de Seguridad en Java 2 JDK 1.2

El JDK 1.2 contiene una mejora sustancial de las características de seguridad: basado en política, fácilmente configurable, concesión de control de acceso, nuevos servicios de criptografía y manejo de certificados y llaves; y tres nuevas herramientas. Estos tópicos se cubren en las siguientes secciones.

Extensiones de la Arquitectura de Seguridad

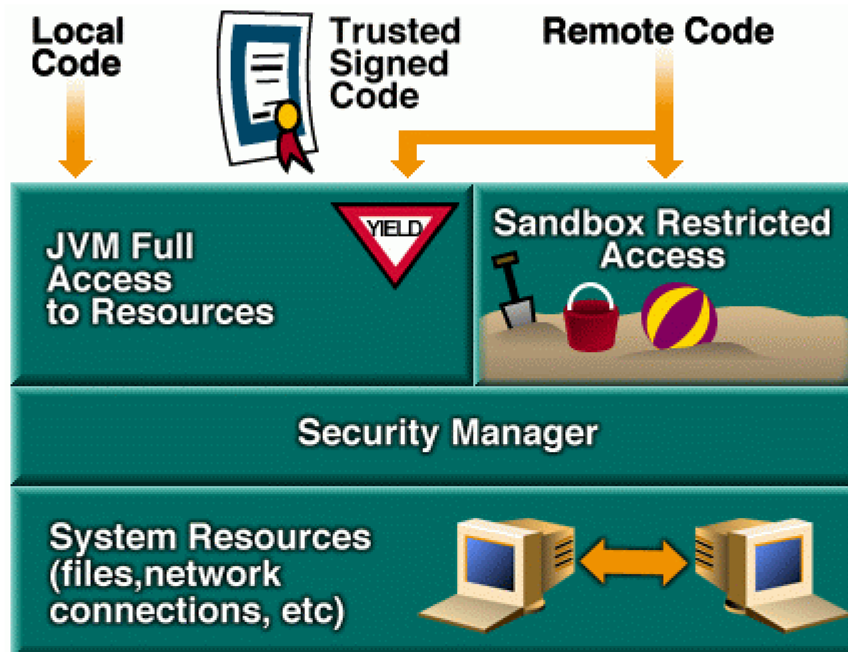
El control de acceso ha evolucionado para ser más fino que en versiones anteriores de Java. El modelo original de seguridad, conocido como el modelo "sandbox", existió para proporcionar un entorno muy restrictivo en el cual ejecutar código no firmado obtenido desde una red abierta. En este modelo, mostrado en el siguiente diagrama, el código local tiene total acceso a los recursos vitales del sistema, como el sistema de archivos, pero el código descargado remotamente (un applet) sólo puede tener acceso a recursos limitados proporcionados dentro del sandbox. Un controlador de seguridad es el responsable en cada plataforma de determinar qué accesos a recursos están permitidos.

Modelo de Seguridad del JDK 1.0:



El JDK 1.1 introdujo el concepto de "applet firmado", éste es tratado como código local, con total acceso a los recursos, si se usa la llave pública para verificar la firma. Los applets no firmados aún se ejecutan dentro del sandbox. Los applets firmados se envían con sus respectivas firmas, en archivos JAR (Java Archives) firmados.

Modelo de Seguridad del JDK 1.1:

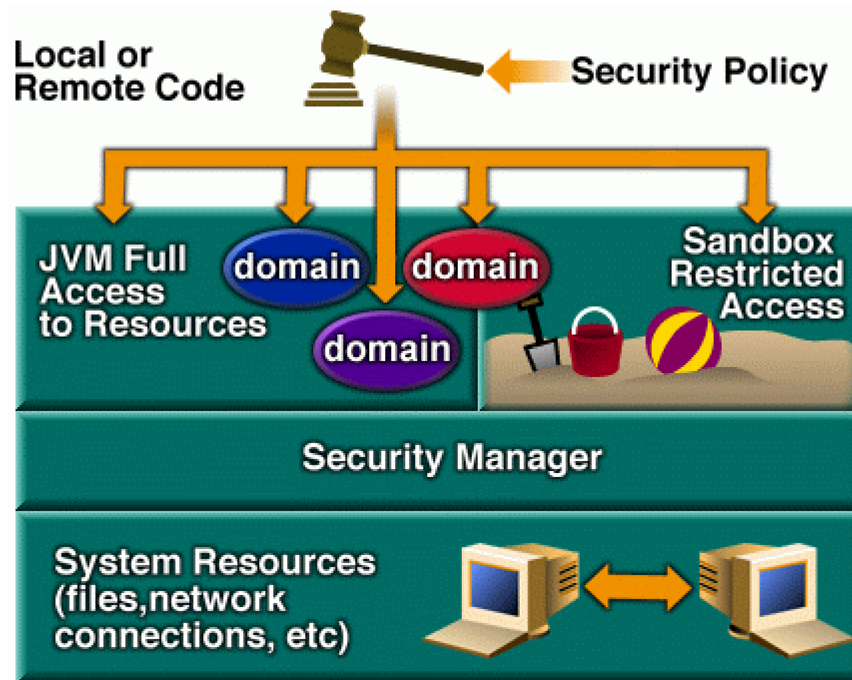


El JDK introduce un gran número de mejoras sobre el JDK 1.1. Primero, todo el código, sin importar si es local o remoto, puede ahora esta sujeto a una **política** de seguridad. Esta política define un conjunto de **permisos** disponibles para el código de varios firmantes o direcciones y puede ser configurado por el usuario o un administrador de sistemas. Cada permiso especifica un acceso permitido a un recurso particular, como accesos de lectura y escritura a un archivo o directorio especifico, o bien el acceso de conexión a un host dado y a un puerto.

El sistema de ejecución organiza el código en **dominios** individuales. Cada uno de ellos encierra un conjunto de clases cuyos ejemplares pueden acceder al mismo conjunto de permisos. Un dominio puede configurarse como un sandbox, por eso los applets aún se pueden ejecutar en entornos restrictivos si el usuario o el administrador lo eligen así. Por defecto, las aplicaciones se ejecutan sin restriciones, pero opcionalmente ahora pueden estar sujetas a una política de seguridad.

La nueva arquitectura de seguridad en el JDK 1.2 es aún más robusta, la flecha de la izquierda se refiere a un dominio cuyo código tiene total acceso a los recursos; la flecha de la derecha se refiere al extremo opuesto: un dominio restringido exactamente igual que en el sandbox original. Los dominios intermedios tienen más accesos permitidos que el sandbox pero menos que el burdo y anticuado acceso total.

Modelo de Seguridad del JDK 1.2:



Extensiones de Arquitectura Criptográfica

La primera versión del API de seguridad del JDK en JDK 1.1 presento la **Java Cryptography Architecture (JCA)**, que se refiere al marco de trabajo para acceder y desarrollar funcionalidades de criptografía para Java. El JCA incluye un proveedor de arquitectura que permite múltiples e interpolables implementaciones criptográficas. El término **Cryptographic Service Provider (CSP)**, o simplemente proveedor, se refiere a un paquete (o conjunto de paquetes) que suministran una implementación concreta de un subconjunto de características de criptografía del API de Seguridad del JDK.

En el JDK, por ejemplo, un proveedor podría contener una implementación de uno o más algoritmos de **firmas digitales**, o de **message digest**, y algoritmos de generación de llaves, el JDK 1.3 añade cinco tipos más de servicios:

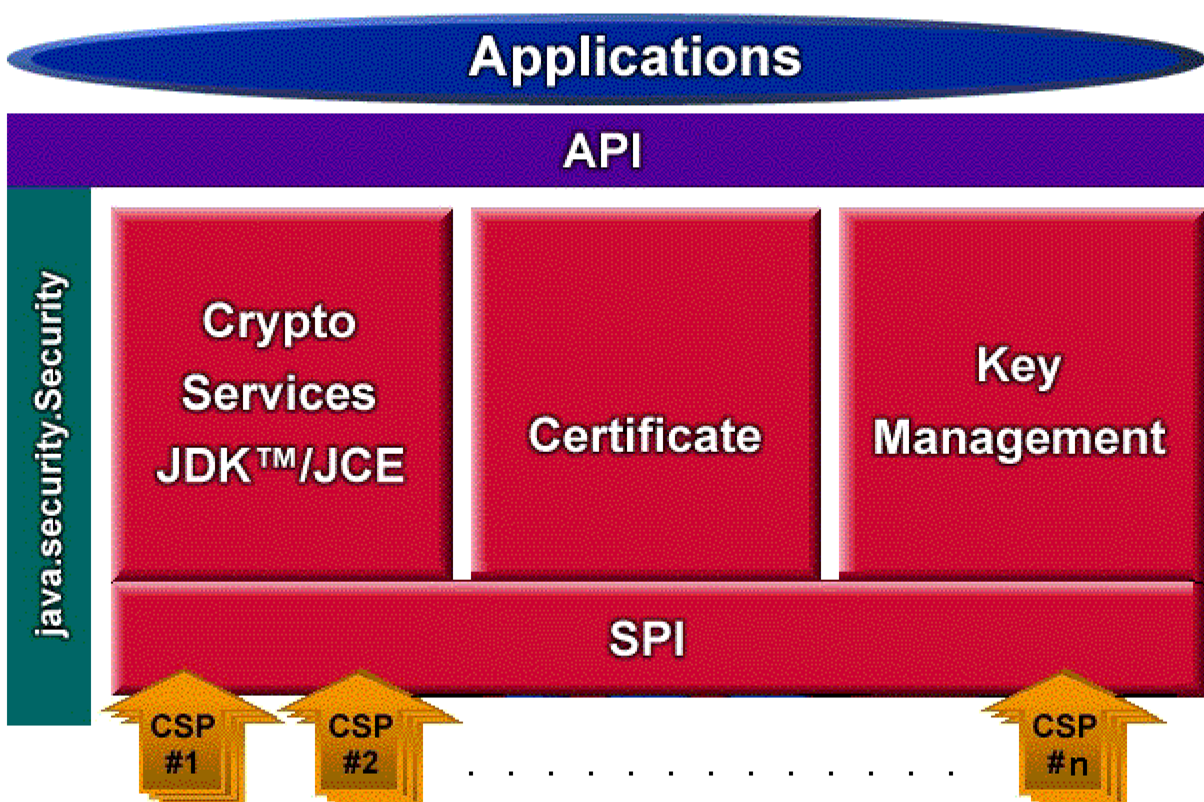
- creación y manejo de bases de llaves
- Algoritmo de manejo de parámetros
- Algoritmo de generación de parámetros
- Fábrica de Llaves para convertir entre las diferentes representaciones de una llave.
- Fábrica de Certificados para generar certificados y listas de revocación de certificados (CRLs) para sus códigos.

La versión de SUN del JRE viene de serie con un proveedor por defecto, llamado SUN. Este paquete incluye implementaciones de un número de servicios de DSA (Digital Signature Algorithm), implementaciones de los algoritmos de MD5 (RFC 1321) y SHA-1 (NIST FIPS 180-1), una fábrica de

certificados para certificados X.509 y una lista de revocación de certificados, un algoritmo de generación de números pseudo-aleatorios, y una implementación de un almacén de llaves.

El Java Cryptography Extension (JCE) amplía el JDK para que incluya el API para **encripción**, intercambio de llaves, y código de autentificación de mensajes (MCA). Junto con el JCE y los aspectos de criptografía del JDK proporciona un API completo de criptografía independiente de la plataforma.

La infraestructura del JCA se delimita en tiers (capas), por lo que podemos observar básicamente 3 de ellas, las cuales con los servicios de encripción (Crypto Services), los certificados (Certificate) y la administración de llaves (Key Management).



Servicios de criptografía

Se ha añadido un gran número de engines al JDK 1.2 para las clases **Signature**, **MessageDigest**, y **KeyPairGenerator** disponibles en el JDK 1.1. Una clase motor define un servico criptográfico de una forma abstracta. Una maquinaria define métodos del API que permiten a las aplicaciones acceder a tipos específicos de servicios criptográficos que proporciona, como un algoritmo de firma digital. Las interfaces de aplicación suministradas por la engine son implementadas en términos de un service provider interface (SPI). Es decir, cada engine tiene una correspondiente clase abstracta SPI que define los métodos de la interfase que provee del servicio, que el proveedor del servicio criptográfico debe implementar.

Por ejemplo, un cliente API podría pedir y usar un ejemplar de la clase **Signature** para acceder a la funcionalidad de un algoritmo de firma digital para firmar digitalmente un archivo. La implementación real suministrada en una subclase **SignatureSpi** será aquella para un tipo de algoritmo de firma específico, como SHA-1 con DSA o MD5 con RSA. Cada ejemplar de un engine, encapsula un ejemplar de su correspondiente clase SPI como implementada por un proveedor de servicio criptográfico. Cada método API de dicho motor invoca al correspondiente método SPI del objeto SPI encapsulado.

Interfaces y Clases para Certificados en el Sistema IE

El JDK 1.2 presenta interfaces para manejar y analizar certificados y proporciona una implementación X.509 v3 de las interfaces de certificados. Un certificado es básicamente una sentencia firmada digitalmente desde una entidad (persona, compañía, etc.) diciendo que la llave pública de otra entidad tiene un valor particular.

A continuación podemos ver las diferentes interfases que podemos utilizar en el IE para implementar mecanismos de seguridad.

Algunas de las clases relacionadas con certificados (todas del paquete **java.security.cert**) son las siguientes.

- **Certificate** - Esta clase es una abstracción para certificados que tienen varios formatos pero tienen usos comunes importantes. Por ejemplo, varios tipos de certificados, como X.509 y PGP, comparten funcionalidades de certificado generales, como codificación y verificación, y algunos tipos de información como una llave pública. Los certificados X.509, PGP, y SDSI pueden ser implementados con una subclase de **Certificate**, incluso aunque contengan diferentes conjuntos de información y almacenen y recuperen la información de formas diferentes.
- **CertificateFactory** - Esta clase define la funcionalidad de la fábrica de certificados que se usa para generar objetos certificados y listas de revocación de certificados (CRL) de sus códigos.
- **X509Certificate** - Esta clase abstracta para certificados X.509 proporciona una forma estándar de acceso a todos los atributos de un certificado de este tipo.

Interfaces y Clases para Manejo de Llaves en IE

El JDK 1.1 presentó las interfaces abstractas **Key**. El JDK 1.2 añade:

- Una clase **KeyStore** (otro engine) que suministra interfaces bien definidas para acceder y modificar información en un almacén de llaves, que es un repositorio de llaves y certificados. Son posibles múltiples implementaciones concretas, donde cada una de éstas son para un tipo diferente de keystore (llavero). Un tipo de keystore define el almacenamiento y el formato de los datos de la información de las llaves.
- Una implementación de **KeyStore** por defecto, que implementa el keystore como un archivo, usando un formato de keystores propietario llamado JKS. La implementación de keystore protege cada llave privada con su password particular y también protege la integridad del keystore completa con otro password.
- **Interfaces de Especificación de Llaves**, que son usadas para representaciones "transparentes" del material llave que constituye la llave. Este material podría ser, por ejemplo, la propia llave y los parámetros del algoritmo usado para calcularla. Una representación "transparente" de llaves significa que podemos acceder al valor material de cada llave de manera individual.
- Una herramienta (**keytool**) para manejar llaves y certificados.

Estos mecanismos pueden ser utilizados para administrar las llaves en el sistema de IE ya que permiten un mejor manejo de la seguridad al tener de manera ordenada las llaves que se generen y al mismo tiempo se asegura la integridad de las mismas.

Herramientas Relacionadas con la Seguridad

El JDK presenta tres nuevas herramientas que son de gran utilidad en IE.

- **Keytool**- Esta herramienta puede ser muy útil en el sistema IE para crear pares de llaves pública/privada, para importar y mostrar cadenas de certificados, para exportar certificados y peticiones de certificados que pueden ser enviados a una autoridad de certificación.
- **Jarsigner** usada para firmar archivos JAR y verificar la autenticidad de las firmas de estos archivos.
- **Policy Tool** crea y modifica la configuración de los archivos de política que definen la política de seguridad de nuestra instalación.

Guía Rápida para Controlar Aplicaciones en IE

Libertad de las Aplicaciones

En el sistema IE, cuando se ejecuta una aplicación **no** se instala automáticamente un controlador de seguridad. En el siguiente paso, veremos como aplicar las mismas políticas de seguridad a una aplicación encontrada en el sistema local de archivos de un applet de la red. Primero, demostraremos que no se instala ningún controlador de seguridad para las aplicaciones, y así las aplicaciones tienen acceso total a los recursos, como fue siempre en el caso del JDK 1.1.

Creamos un archivo llamado **GetProps.java** basándonos en el mismo ejemplo de la documentación de SUN:

```
import java.lang.*;
import java.security.*;

class GetProps
{
    public static void main(String[] args)
        {
        String s;
        try
            {
            System.out.println("Getting os.name property value");
            s = System.getProperty("os.name", "not specified");
            System.out.println("Your OS is: " + s);
            System.out.println("Getting java.version propertyvalue");
            s = System.getProperty("java.version", "not specified");
            System.out.println("  The JVM you are running is: " + s);
            System.out.println("Getting user.home property value");
            s = System.getProperty("user.home", "not specified");
            System.out.println("  Your user home directory is: " + s);
            System.out.println("Getting java.home property value");
            s = System.getProperty("java.home", "not specified");
            System.out.println("Your JRE install directory is: " + s);
        }
        catch (Exception e)
            {
            System.err.println("Caught exception " + e.toString());
        }
        }
}
```

Estos ejemplos deben asumirse bajo los siguientes directorios: **C:\Test** si estamos usando un sistema Windows o: **~/test** para UNIX.

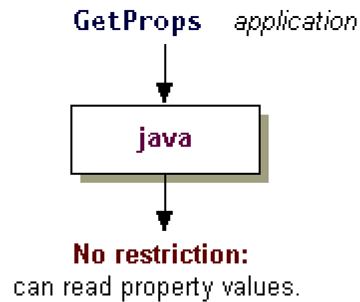
Como vemos en el código fuente, este programa trata de leer valores de varias propiedades cuyos nombres son: **"os.name"** , **"java.version"**, **"user.home"**, y **"java.home"**.

Si compilamos el archivo que llamamos **GetProps.java** veremos la siguiente salida:

```
C:\TEST>java GetProps
```

```
Getting os.name property value
  Your OS is: Windows 95
    Getting java.version property value
  The JVM you are running is: 1.2.2
Getting user.home property value
  Your user home directory is: C:\WINDOWS
Getting java.home property value
  Your JRE install directory is: C:\JDK1.2.2\JRE
```

Como se puede ver, las aplicaciones tienen acceso total a los valores de las propiedades, como se vemos en este diagrama:



Cómo Restringir las Aplicaciones

Cuando se ejecuta una aplicación **no** se instala automáticamente un controlador de seguridad. Para aplicar la misma política de seguridad a una aplicación encontrada en el sistema local de archivos que a los applets descargados de la red, compilamos con el nuevo argumento de la línea de comandos **-Djava.security.manager**.

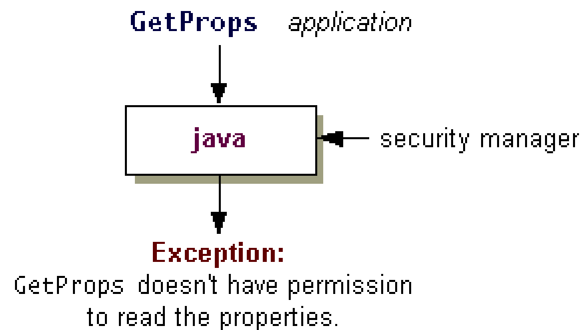
Para ejecutar la aplicación **GetProps** con el controlador de seguridad por defecto, tecleamos lo siguiente.

```
java -Djava.security.manager GetProps
```

Aquí podemos ver la salida del programa:

```
C:\TEST>java -Djava.security.manager GetProps
  Getting os.name property value
    Your OS is: Windows 95
  Getting java.version property value
    The JVM you are running is: JDK 1.2.2
  Getting user.home property value
  Caught exception java.security.AccessControlException.
  access denied (java.util.PropertyPermission user.home read)
```

El proceso se puede apreciar en el diagrama:



Como vemos en el diagrama anterior, de esta manera es fácil restringir el acceso en el uso de aplicaciones, lo cual es bastante benéfico si estamos hablando de un sistema de inscripciones IE al cual tienen acceso una cantidad impresionante de usuarios, los cuales deben tener diferentes niveles de seguridad.

Propiedades Sensibles a la Seguridad

El archivo de política del sistema, cargado por defecto, concede a todo código permiso para acceder a algunas de las propiedades más comunes como **"os.name"** y **"java.version"**. Estas propiedades no son sensibles a la seguridad, por eso la concesión de dichos permisos no supone un problema.

Las otras propiedades a las que **GetProps** intenta acceder, **"user.home"** y **"java.home"**, no están en la lista de propiedades para las que el archivo de política del sistema concede permiso de lectura. Así, tan pronto como **GetProps** intenta acceder a la primera de estas propiedades (**"user.home"**), el controlador de seguridad evita el acceso y reporta una **AccessControlException**. Esta excepción indica que la política actualmente activa que consiste en entradas de uno o más archivos de política, no concedan permiso para leer la propiedad **"user.home"**.

El Archivo de Política del Sistema

El archivo de política del sistema se encuentra por defecto en.

Windows.
java.home\lib\security\java.policy

UNIX.
java.home/lib/security/java.policy

Como podemos observar, java.home representa el valor de la propiedad **"java.home"**, que es una propiedad del sistema que especifica el directorio en que se instaló el JRE. Así si el JRE fue instalado en el directorio llamado **C:\jdk1.2.2\jre** sobre Windows y **/jdk1.2.2/jre** sobre UNIX, el archivo de política del sistema estará en:

Windows.
C:\jdk1.2.2\jre\lib\security\java.policy

UNIX.
/jdk1.2.2/jre/lib/security/java.policy

Configurar el Archivo de Política para Conceder los Permisos Requeridos

Vamos a mostrar un ejemplo a continuación, independientemente del funcionamiento del sistema IE, para dejar muy clara la política de permisos que se utiliza y el funcionamiento de la misma.

Añadiremos una nueva entrada de política concediendo permiso para el código del directorio donde se almacena **GetProps.class** para leer los valores de las propiedades **"user.home"** y **"java.home"**, como se muestra en el siguiente diagrama:

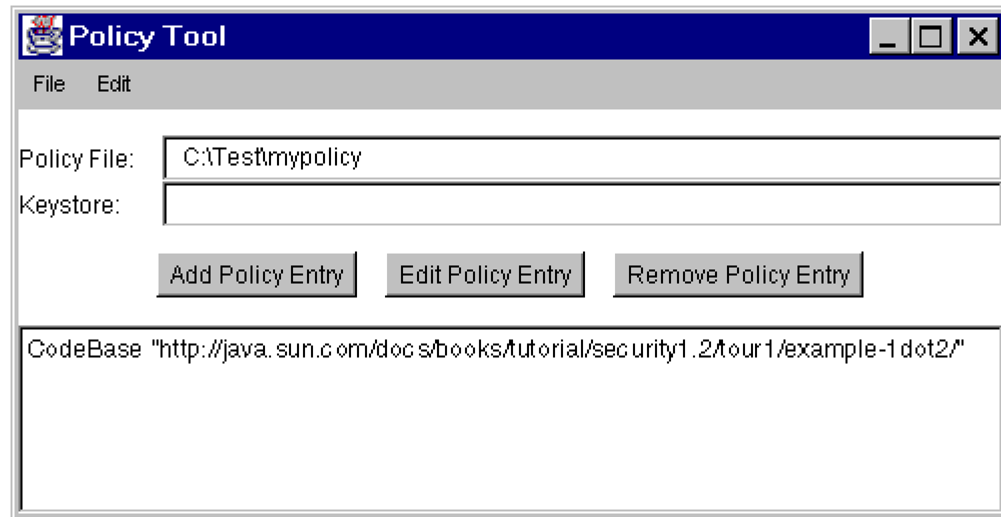
Abrir el Archivo de Política

Arrancamos Policy Tool tecleando lo siguiente en la línea de comandos.

```
policytool
```

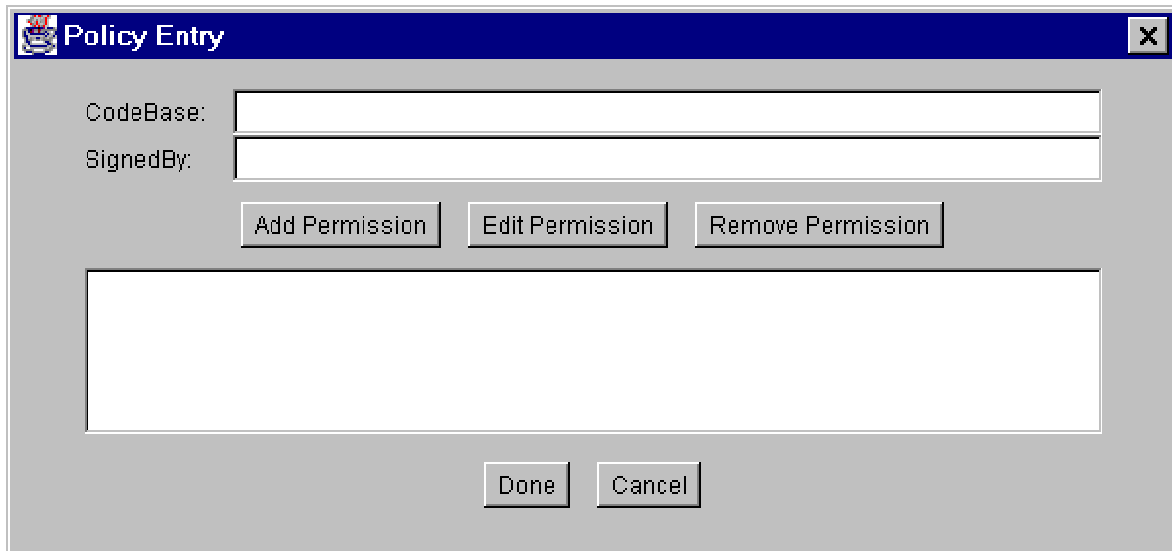
Esto trae la ventana de Policy Tool. Para abrir el archivo de política, usamos el comando **Open** en el menú **File**. Como en cualquier aplicación conocida, se nos presenta la opción de navegar entre los archivos de la estructura del disco en cuestión. En este ejemplo supongamos que el archivo **mypolicy** se encuentre en el directorio **C:\Test** y pulsamos el botón **Open**.

Esto rellenará la ventana de Policy Tool con la información del archivo de política **mypolicy**, incluyendo el nombre del archivo de política y la parte **CodeBase** de la entrada de política creada en el documento de seguridad de SUN.



Conceder los Permisos Requeridos

Para conceder a la aplicación **GetProps** permiso para leer los valores de las propiedades **"user.home"** y **"java.home"**, debemos crear una entrada de política que conceda esos permisos. Elegimos el botón **Add Policy Entry** en la ventana principal de Policy Tool. Esto nos muestra la caja de diálogo Policy Entry, como se muestra en el sreen shot:



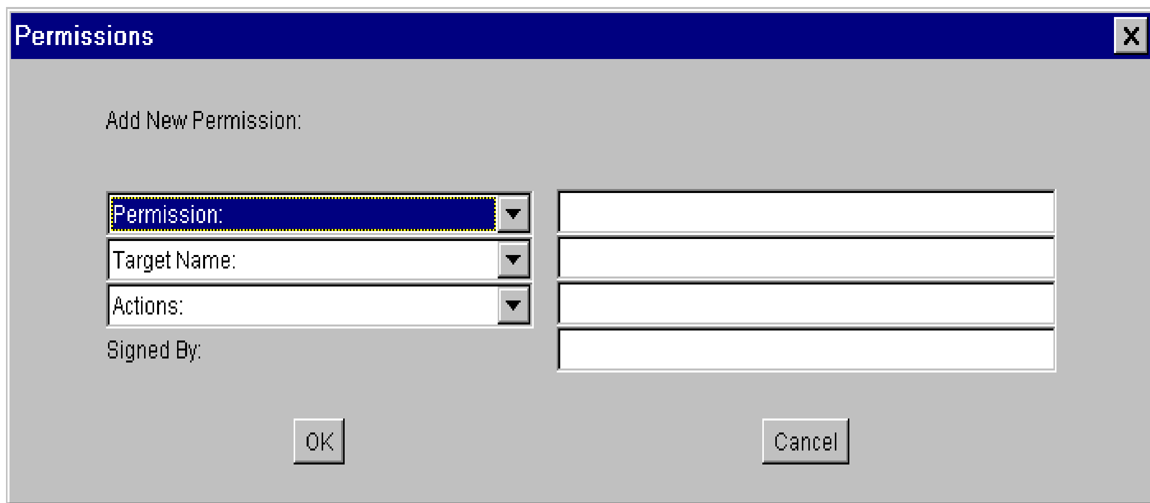
Tecleamos el siguiente URL de archivo dentro de la caja de texto **CodeBase** para indicar que vamos a conceder permiso al código que venga del directorio especificado, que es el directorio en el que se encuentra **GetProps.class**.

file:/C:/Test/

Las diagonales sencillas son debido a que lo que tecleamos es un URL, de esta manera no debemos poner diagonales invertidas como lo haríamos en MS-DOS por ejemplo para definir rutas absolutas o relativas. Dejamos en blanco la caja de texto **SignedBy**, ya que no necesitamos que el código esté firmado.

Para añadir el permiso para leer el valor de la propiedad **"user.home"**, elegimos el botón **Add Permission**.

En este momento aparece la pantalla de **Permissions** como se muestra en el screen shot:



Ahora procedemos de esta manera:

1. Elegimos **Property Permission** en la lista desplegable Permission. El nombre completo del tipo de permiso (**java.util.PropertyPermission**) aparecerá en la caja de texto que hay a la derecha de la lista desplegable.
2. Tecleamos el siguiente texto en la caja de texto que hay a la derecha de la lista etiquetada Target Name para especificar la propiedad **"user.home"**

   user.home

3. Especificamos el permiso para leer esta propiedad seleccionando la opción **read** en la lista desplegable Actions.

Ahora la caja de diálogo Permissions se parecerá a esta:

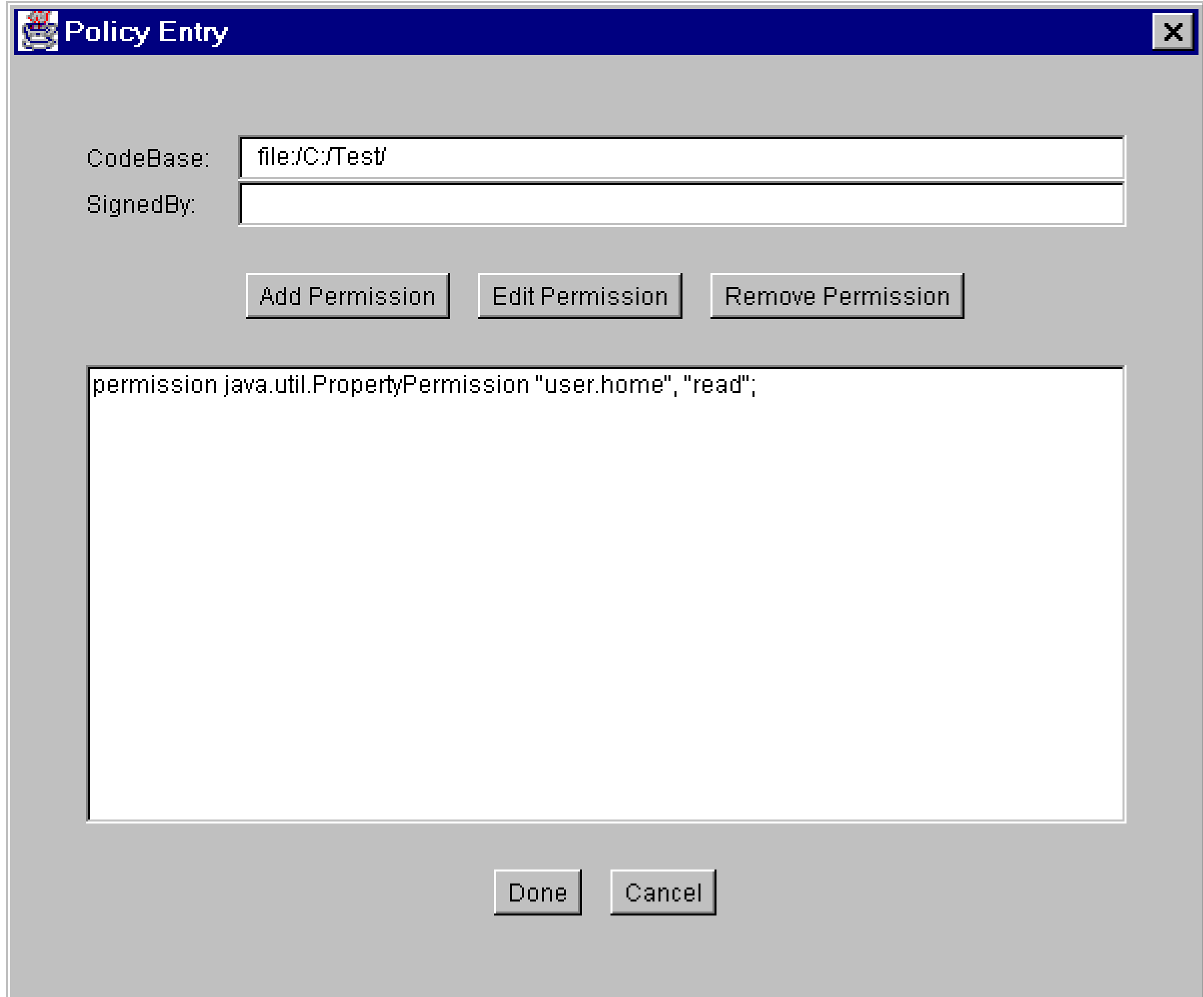
Permissions

Add New Permission:

Property Permission	java.util.PropertyPermission
Target Name:	user.home
read	read
Signed By:	

OK Cancel

Pulsamos el botón **OK**. El nuevo permiso aparecerá en una lista en la ventana de policy entry.



Para poder leer el valor de la propiedad **"java.home"**, elegimos **Add Permission** y en la caja de diálogo Permissions hacemos lo siguiente:

1. Elegimos **Property Permission** en el combo box de Permission. El nombre completo del tipo de permiso aparece a la derecha del combo box.
2. La segunda caja de texto (a la derecha del combo Target Name) es donde debemos teclear java.home para especificar la propiedad **"java.home"**.
3. Especificamos el permiso para leer esta propiedad seleccionando la opción **read** en el combo box de Actions.

Ahora Permissions deberá ser similar a:

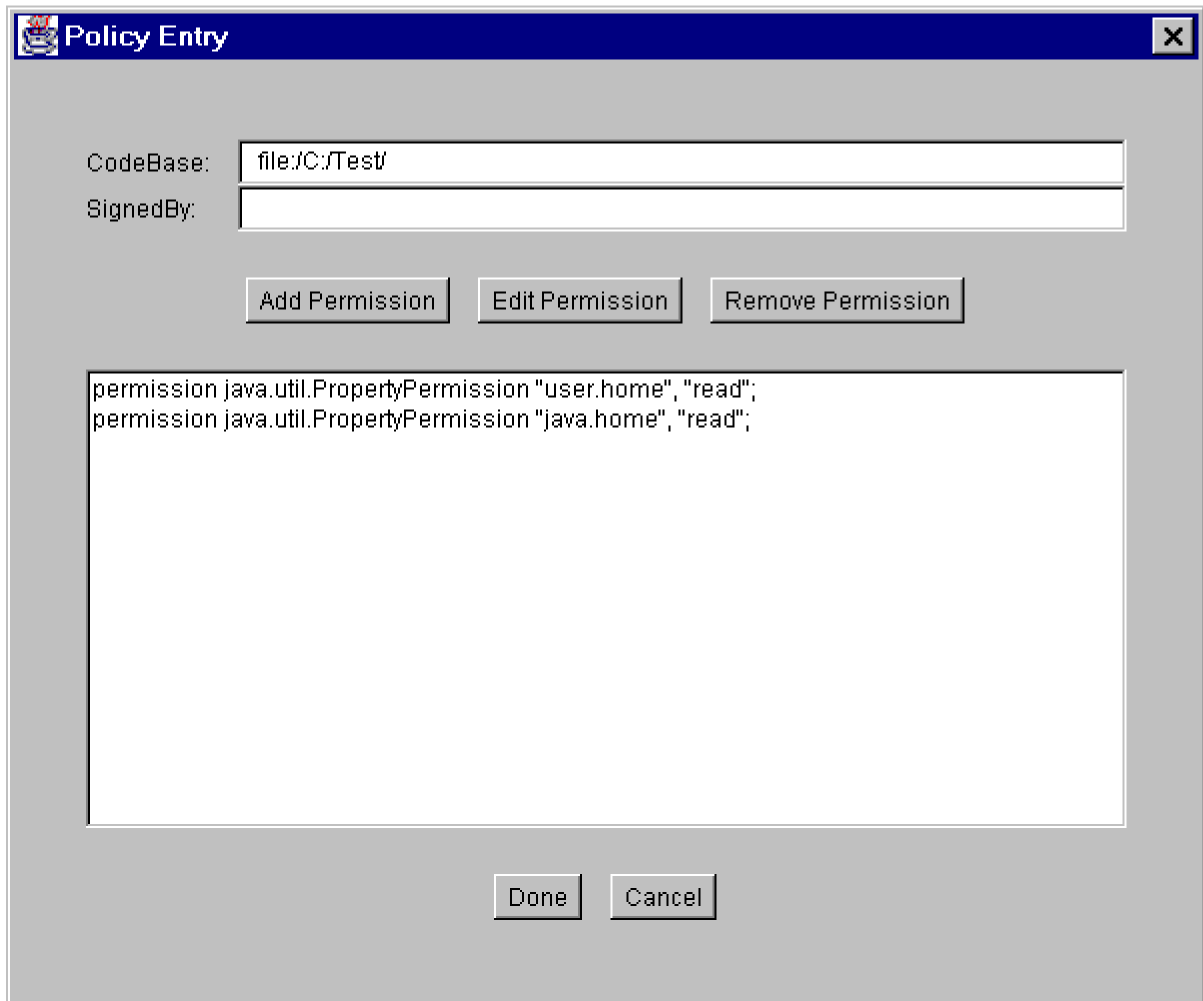
Permissions

Add New Permission:

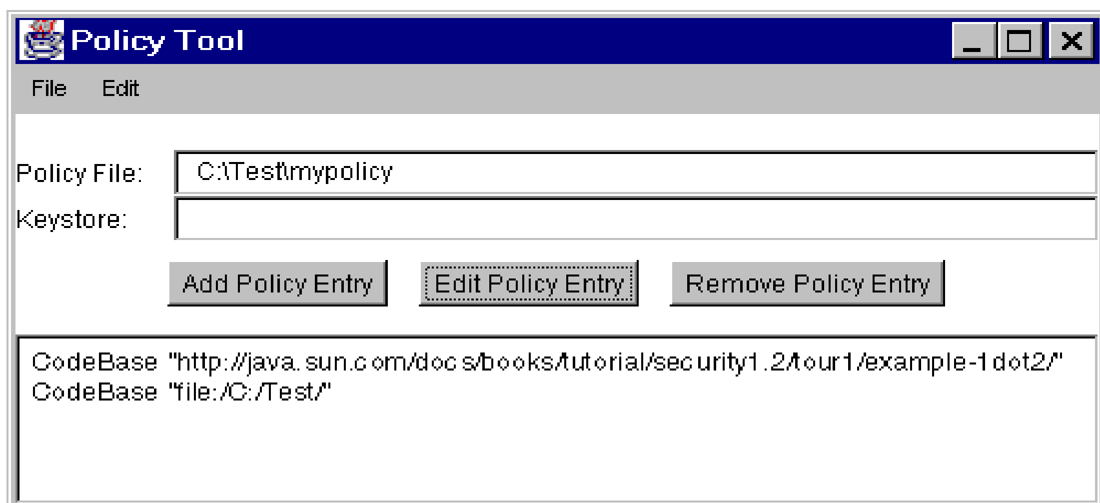
Property Permission	java.util.PropertyPermission
Target Name:	java.home
read	read
Signed By:	

OK Cancel

Al aceptar este nuevo permiso, éste aparecerá de nuevo en la lista de la ventana de policy entry, junto con el anterior permiso:



Cuando hayamos especificado las entradas de política, elegimos el botón **Done**. Ahora la ventana Policy Tool incluye una línea representando la nueva entrada de política, mostrando el valor **CodeBase**.

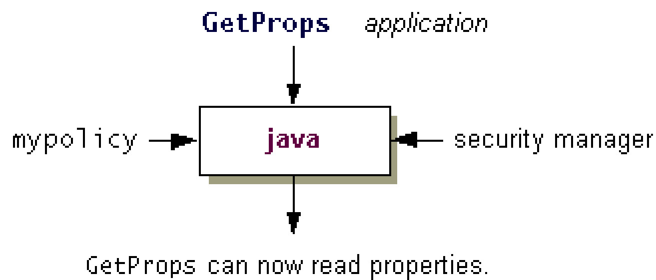


Grabar el Archivo de Política

Para grabar el archivo de política, simplemente elegimos el comando **Save** en el menú **File**. Para salir del Policy Tool con la opción **Exit**.

Efectos del Archivo de Política

Ahora que hemos creado el archivo de política **mypolicy** deberíamos poder ejecutar con éxito la aplicación **GetProps** para leer las propiedades especificadas con un controlador de seguridad, como se ve en la siguiente figura.



Para efectos específicos del IE, siempre que ejecutemos un servlet o una aplicación con un controlador se seguridad, los archivos de política que son cargados y usados por defecto son los especificados en el **"security properties file"**, que está localizado en uno de los siguientes directorios:

Windows
java.home\lib\security\java.security

UNIX
java.home/lib/security/java.security

Donde vale la pena indicar que java.home indica el directorio en el que se instaló el JRE.

Tenemos dos posibles formas de hacer que el archivo **mypolicy** sea considerado como parte de la política general, además de los archivos de política especificados en el archivo de propiedades de seguridad. Podemos especificar el archivo de política adicional en una propiedad pasada al sistema de ejecución, o añadir una nueva línea al archivo de propiedades de seguridad.

API y Herramientas para Código Seguro e Intercambio de Archivos en IE

Es importante incluir información general para justificar el uso de las firmas digitales, los certificados y los repositorios de llaves. Para el IE es importante hablar de toda la infraestructura que JAVA puede manejar en el ámbito de código seguro o bien para el intercambio de archivos. Recordemos que el front-end del IE está colgado en Internet y por lo tanto debe contar con toda la infraestructura necesaria para proveer seguridad a todos sus usuarios.

Seguridad de Código y Documentos

Si le enviamos electrónicamente a alguien un documento importante, o un applet o una aplicación para que los ejecute, el receptor necesita una forma de verificar que el documento o el código es efectivamente de nosotros y que no se ha modificado en el trayecto (por ejemplo, por un usuario malicioso que lo ha interceptado). Las firmas digitales, los certificados y los keystores nos ayudan a conseguir la seguridad de los archivos que enviamos.

Este módulo es trascendental para el sistema IE ya que se requiere que la información que se envía sea validada y esté íntegra en todo momento.

Firmas Digitales

La idea básica del uso de las firmas digitales es la siguiente.

1. Nosotros "firmamos" el documento o código usando una de nuestras **llaves privadas**, que podemos generar usando **keytool** o los métodos del API de seguridad. Esto es, generamos una firma digital para el documento o el código usando la herramienta **jarsigner** o los métodos del API.
2. Enviamos al destinatario el documento o código y la firma.
3. Al destinatario le anexamos también la **llave pública** correspondiente a la llave privada usada para generar la firma. Aunque en un ejemplo más real, es muy posible que el destinatario ya cuenta con dicha llave pública.
4. El destinatario usa la llave pública para verificar la autenticidad de la firma y la integridad del documento/código.

Esto nos lleva a otro problema: ¿Cómo saber que la propia llave pública emitida anteriormente es auténtica? Para que el destinatario se asegure de dicha integridad, es muy normal adjuntar un certificado que avale la autenticidad de la llave pública en lugar de sólo mandar dicha llave.

Certificados

Un certificado contiene:

- Una llave pública.
- La información sobre la entidad, en este caso, los propietarios del sistema IE, a quien pertenece el certificado. Esta entrada se puede tratar como el sujeto o el propietario del certificado. Dicha información incluye los siguientes atributos: el nombre de la entidad, organización, población, estado y código del país, en nuestro caso son los datos del tec de monterrey.
- Una firma digital. Un certificado está firmado por una entidad confiable que avala la autenticidad de que la llave pública que emite es la llave pública real de otra entidad, el propietario.
- La información de la instancia emisora de dicho certificado.

Una forma de que el destinatario pueda verificar si un certificado es válido, es verificando su firma digital, usando la llave pública del emisor. Esta llave también puede estar almacenada en otro certificado cuya firma puede ser verificada usando la llave pública del emisor del otro certificado. Podemos parar la comprobación cuando alcancemos una llave pública en la que creamos y la utilicemos para almacenar la firma del certificado correspondiente.

Si el destinatario no puede establecer una cadena verdadera (por ejemplo, porque el emisor del certificado no esta disponible), se pueden calcular la **huellas dactilares** del certificado, con el comando **-printcert** o **-import** de la herramienta keytool. Cada huella dactilar es un número relativamente corto que es único e identifica el certificado. El destinatario puede entonces llamar al propietario del certificado y comparar las huellas dactilares del certificado recibido con las enviadas por el certificado. Si las huellas coninciden, los certificados son iguales.

Así podemos asegurarnos de que un certificado no fue modificado durante el trayecto en el que fue mandado. Otra cosa incierta cuando se trabaja con certificados es la identidad del emisor. Algunas veces un certificado está **auto-firmado**, es decir, está firmado usando la llave privada correspondiente a la llave pública que hay en el certificado, el emisor es el mismo que el sujeto.

De otro modo el emisor necesita obtener un certificado de una tercera instancia, referida como Autoridades de Certificación (CA). Para hacer esto, enviamos una petición de certificado de firma auto-firmado (CSR) al CA. El CA verifica la firma del CSR y nuestra identidad, mimetizándola con otro comprobante fiable de nuestra identidad, como por ejemplo nuestra licencia de conducir u otra información. El CA garantiza que nosotros somos los propietarios de la llave pública enviando un certificado firmado con su propia llave privada (la del CA). Cualquiera que crea en la emisión de la llave pública del CA puede verificar la firma del certificado. En muchos casos el envío del propio CA puede tener un certificado de otro CA de mayor nivel en la jerarquías de CA, empezando una **cadena de certificados**.

Los certificados de entidades en las que creemos normalmente se importan en nuestro keystore (repositorio de llaves) como **"certificados confiables"**. La llave pública de dichos certificados puede ser utilizada para verificar las firmas generadas usando la correspondiente llave privada. Dichas verificaciones se pueden hacer mediante:

- La herramienta **jarsigner** (si el documento o el código firmado aparecen en un archivo JAR)
- Los métodos del API
- El sistema de ejecución, cuando se intenta el acceso a un recurso y el archivo de política específica que el acceso a ese recurso está permitido para le código que lo intenta, el acceso sólo se permite si la firma es auténtica. Los archivos .class y la firma deben estar en un archivo .JAR.

Si estamos enviando documentos o código firmados a otras personas, necesitamos suministrarlos con el certificado que contiene la llave pública correspondiente a la llave privada usada para firmarlo. El comando **-export** de **keytool** o los métodos del API pueden exportar nuestro certificado desde nuestro repositorio de llaves a un archivo, que puede ser enviado a cualquiera que lo necesite. Una persona que reciba el certificado puede importarlo dentro de su keystore como certificado confiable, usando, por ejemplo, los métodos del API o el comando **-import** de **keytool**.

Si usamos la herramienta **jarsigner** para generar la firma de un archivo JAR, la herramienta recupera nuestro certificado y su correspondiente cadena de certificados desde nuestro repositorio de llaves. Luego, la herramienta almacena cada certificado junto con la firma en el archivo JAR.

Keystores (Repositorios de Llaves)

Las llaves privadas y sus certificados de llaves públicas asociadas están almacenadas en unas bases de datos protegidas por passwords llamadas **keystores**. Un keystore puede contener dos tipos de entradas: las entradas de certificados confiables, y entradas llave/certificado. Cada una de ellas contiene una llave

privada y el correspondiente certificado de la llave pública. Cada entrada en el keystore está identificada por un **alias**.

Un propietario de un keystore puede tener múltiples llaves en él, accedidas mediante diferentes alias. Un alias se nombra típicamente según las reglas particulares en las que el propietario del keystore usa las llaves asociadas. Un alias también podría identificar el propósito de la llave. Por ejemplo, el alias **signPersonalEmail** podría usarse para identificar una entrada del keystore cuya llave privada es usada para firmar e-mail personales, y el alias **signJarFiles** podría usarse para identificar una entrada cuya llave privada se usa para firmar archivos JAR.

La herramienta **keytool** puede usarse para:

- Crear llaves privadas y sus certificados de llaves públicas asociadas.
- Emitir peticiones de certificados, que enviamos a la autoridad de certificación apropiada.
- Importar respuestas de certificados, obtenidos desde la autoridad de certificación contactada.
- Importar certificados de llaves privadas pertenecientes a otras partes como certificados confiables.
- Manejar nuestro keystore

Los métodos del API también pueden usarse para acceder y modificar un keystore. Así pues el IE pudiese tener todo un llavero (keystore) con los certificados y las llaves asociadas que efectivamente respalda una CA. O del mismo modo, pero en sentido inverso, los usuarios del IE pueden estar seguros que su información viaje a través de un medio seguro, cifrada por un organismo emisor de certificados válidos.

Características Del API De Seguridad del JDK que utilizaremos en IE

Un programa ejecutado por la persona que tiene el documento original, podrá.

- Generar llaves para el manejo de IE
- Generar una firma digital para los datos que se envíen en el sistema usando la llave privada
- Exportar la llave pública y la firma a archivos

El destinatario de los datos, la firma y la llave pública podrá:

- Importar la llave pública
- Verificar la autenticidad de la firma del sistema IE

Uso De Herramientas Para Firmar Código en el sistema IE

Podemos firmar código y conceder permisos, o bien en otro alcance podemos también brindar seguridad con el uso de herramientas a la acción de intercambiar archivos. En ambos casos, los dos primeros pasos para el emisor del código o el documento son:

- Crear un archivo JAR que contenga el documento o archivo JAR, usando la herramienta **jar**.
- Generar llaves (si no existen ya), usando el comando **keytool -genkey**.

Los dos siguientes pasos son opcionales:

- Usar el comando **keytool -certreq** para enviar la petición de firma del certificado resultante a una autoridad de verificación (CA) como VeriSign.
- Usar el comando **keytool -import** para importar la respuesta del CA.

Los dos siguientes pasos son necesarios:

- Firmar el archivo JAR, usando la herramienta **jarsigner** y la llave privada generada en el paso 2.
- Exportar el certificado de la llave pública, usando el comando **keytool -export**. Luego suministrar el archivo JAR firmado y el certificado al destinatario.

En ambos casos, el receptor del archivo JAR firmado y el certificado debería importar el certificado como un certificado confiable, usando el comando **keytool -import**. El **keytool** intentará construir una cadena de certificados confiables para importarla como un certificado confiable en el keystore. Si esto falla, **keytool** mostrará la huella del certificado y nos pedirá que la verifiquemos.

Si lo que se envió fue código, el receptor también necesita modificar el archivo de política para permitir los accesos a los recursos necesarios para el código firmado por la llave privada correspondiente a al certificado de la llave pública importada. Se puede utilizar **Policy Tool** para hacer esto.

Si lo que se envió fue uno o más documentos, el destinatario necesita verificar la autenticidad de la firma del archivo JAR, usando la herramienta **jarsigner**.

Generar una Petición de Firma de Certificado (CSR)

Cuando se usa **keytool** para generar una pareja de llaves pública/privada en el sistema, se crea una entrada en un keystore que contiene la propia llave privada y un certificado auto-firmado para la llave pública, es decir, el certificado esta firmado utilizando la correspondiente llave privada. Esto podría ser adecuado si el destinatario que va a recibir nuestro archivo firmado ya conoce y cree en nuestra identidad.

Sin embargo, un certificado es algo más veraz si está firmado por una autoridad de certificación (CA). Para obtener un certificado firmado por una CA, primero generamos un petición de firma de certificado (CSR), mediante el comando:

```
keytool -certreq -alias alias -file csrFile
```

Aquí alias se usa para acceder a la entrada del keystore que contiene la llave privada y el certificado de la llave pública, y csrFile especifica el nombre a usar por el CSR creado por este comando.

Luego debemos enviar este archivo a una CA, como VeriSign. El CA nos autentificará, el "sujeto", normalmente off-line, y luego firmará y devolverá un certificado de autentificación para nuestra llave pública, es decir, el CA confirma que nosotros somos los propietarios de la llave pública firmando el certificado. (En algunos casos, el CA, devolverá una cadena de certificados, cada uno autentificando la llave pública del firmante del certificado anterior en la cadena.)

Importar la Respuesta del CA

Si hemos enviado una petición de firma de certificado (CSR) a una autoridad de autenticidad (CA), necesitamos reemplazar el certificado original auto-firmado en nuestro keystore con uno que contenga la cadena de certificados devuelto por el CA.

Pero primero necesitamos una entrada "certificado confiable" en nuestro keystore (o en el archivo **cacerts**) que autentifica la llave pública del CA. Con esto la entrada de la firma del CA puede ser verificada, es decir, la firma del CA en el certificado, o en el certificado final en la cadena que el CA envía en respuesta a nuestro CSR, pueda ser verificada.

Importar a IE un Certificado desde un CA como un "Certificado confiable"

Antes de que importemos al sistema IE el certificado devuelto por un CA, necesitaremos uno o más "certificados confiables" en nuestro keystore o en el archivo **cacerts**.

- Si el certificado de respuesta es una cadena de certificados, necesitamos el certificado superior de la cadena -- el CA "raíz" certifica la autenticidad de las llaves públicas del resto de los CAs.
- Si la respuesta es un sólo certificado, necesitamos un certificado para el CA emisor (uno firmado por ella). Si este certificado no está auto-firmado, necesitamos un certificado para su firmante, etc., hasta el certificado del CA "raíz" auto-firmado.

El archivo **cacerts** representa un keystore de amplio sistema con certificados de CAs. El archivo reside en el directorio de las propiedades de seguridad del JRE, **java.home/lib/security**, donde java.home es el directorio de instalación del JRE.

El archivo **cacerts** se recibe con cinco certificados raíces de VeriSign. Si enviamos una CSR a VeriSign, no necesitamos importar un certificado de VeriSign como certificado confiable en nuestro keystore. Un certificado desde una CA normalmente está auto-firmado o firmado por otra CA, en cuyo caso también necesitamos un certificado de autentificación de la llave pública de la CA.

Debemos tener cuidado de que el certificado sea válido antes de importarlo como "certificado confiable" Debemos mirarlo primero (usando el comando **keytool -printcert** o el comando **keytool -import** sin la opción **-noprompt**), y asegurarnos de que las huellas del certificados mostradas corresponden con las esperadas. Podemos llamar a la persona que nos envió el certificado y comparar la huella. Sólo si las huellas son iguales está garantizado que el certificado no ha sido reemplazado en el tRobecto de envío por el certificado de otra persona (el atacante, por ejemplo).

Si creemos que el certificado es válido, podemos añadirlo a nuestro keystore con un comando como este:

```
keytool -import -alias ie
    -file SYMBIO.cer -keystore storefile
```

Este comando crea un entrada de "certificado confiable" en el keystore cuyo nombre es el especificado en storefile. La entrada contiene los datos del archivo **SYMBIO.cer**, y está firmada por el alias especificado.

Importar el Certificado Devuelto por el CA

Una vez que hemos importado los certificados confiables requeridos, podemos importar el certificado de respuesta y por lo tanto reemplazar nuestro certificado auto-firmado con una cadena de certificados. Esta cadena puede ser devuelta por el CA en respuesta a nuestra petición (si la respuesta del CA es una cadena) o una construida (si el CA devolvió sólo un certificado) usando el certificado devuelto y los certificados confiables que tenemos disponibles en el keystore o en el archivo **cacerts**.

Como ejemplo, supongamos que hemos enviado una petición de firma de certificado a VeriSign. Podemos importar la respuesta mediante el siguiente comando que asume que el certificado devuelto está en el archivo especificado por certReplyFile.

```
keytool -import -trustcacerts -keystore storefile
  -alias ie
  -file certReplyFile
```

Debemos teclear el comando en una sola línea.

El certificado de respuesta es validado usando Certificados Confiables desde el keystore y opcionalmente usando los certificados configurados en el archivo **cacerts** (si se especifica la opción **-trustcacerts**). Cada certificado de la cadena está verificado, usando el certificado del siguiente nivel superior en la cadena. Sólo necesitamos creer en el certificado del CA superior en la cadena. Si realmente no creemos en él, **keytool** mostrará la huella dactilar del certificado y nos preguntará si creemos en ella o no.

La nueva cadena de certificados de la entrada especificada por el **alias** reemplaza el viejo certificado (o cadena) asociado con esa entrada. La vieja cadena sólo puede ser reemplazada si hay un **keypass**, se suministra el password usada para proteger la clave privada de la entrada. Si no se proporciona password o si el password de la entrada es diferente de la existente, se le pedirá al usuario.

Para información más detallada sobre la generación de CSR y la importación de las respuestas, se puede ver la documentación de **keytool** en la web site pública de **java.sun.com**.

- **keytool documentation with Windows examples**
- **keytool documentation with UNIX examples**

Fimar Código y Conceder Permisos

Esta lección ilustra el uso de las herramientas relacionadas con la seguridad (**keytool**, **jarsigner**, y **Policy Tool**). También muestra el uso de la herramienta **jar** para situar archivos en un archivo JAR, para la posterior firma con la herramienta **jarsigner**. En esta parte no utilizamos un ejemplo específico del sistema IE para poder demostrar la funcionalidad general de la firma de código y la manera que se conceden permisos, pero el mecanismo es igual para nuestro caso en el sistema IE.

En esta lección primero ejecutaremos los pasos para crear una aplicación, ponerla en un archivo JAR, firmar el archivo JAR, y exportar el certificado de la clave pública correspondiente a la clave privada usada para firmar el archivo JAR.

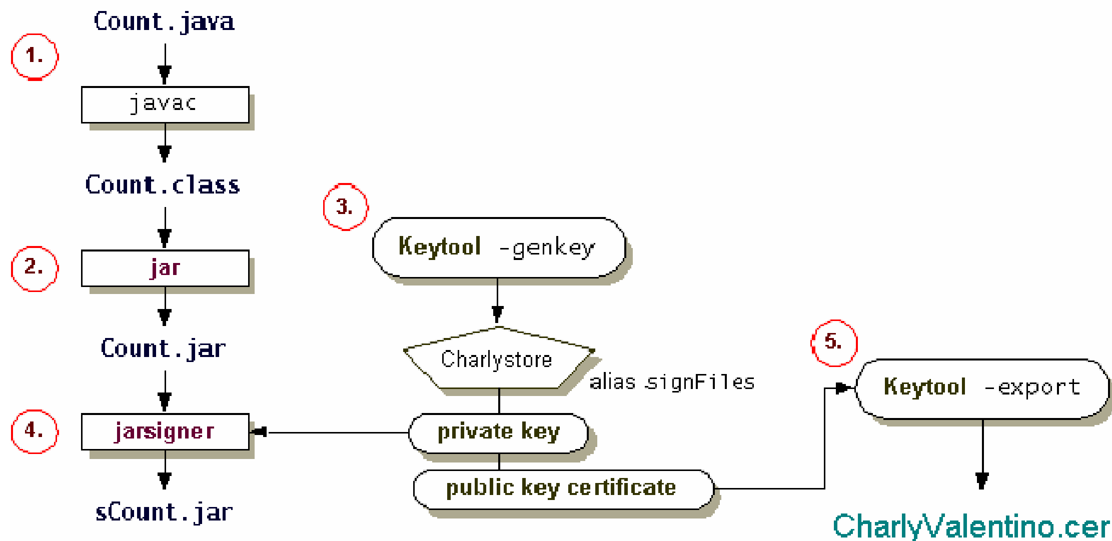
Luego actuaremos como el receptor del archivo JAR y del certificado. Veremos cómo la aplicación firmada, normalmente no puede leer un archivo cuando se ejecuta bajo un controlador de seguridad. Luego usaremos **keytool** para importar el certificado en nuestro keystore en una entrada con el alias **charly**, y Policy Tool para crear una entrada en el archivo de política de Rob para permitir que el código firmado por **charly** pueda leer el archivo especificado. Finalmente veremos como nuestra aplicación ejecutándose bajo un controlador se seguridad puede ahora leer el archivo, ya que se le ha concedido permiso para hacerlo.

Para más información sobre las firmas digitales, los certificados y las herramientas, puedes ver la lección anterior.

Nota Importante: Necesitamos hacer todo el trabajo de esta lección en el directorio en el que hayamos almacenado la aplicación de ejemplo, pero deberíamos almacenar lo archivos de datos necesarios para la aplicación en un directorio diferente. Todos los ejemplos asumen que estamos trabajando en el directorio **C:\Test**, y que los archivos de datos están el directorio **C:\TestData**.

Pasos para el que Firma el Código

El firmante de código realiza los siguientes pasos que se muestran en la siguiente figura.



- **Descargar y Probar la Aplicación de Ejemplo**

En esta sección vamos a crear una aplicación de ejemplo para entender mejor el funcionamiento que hemos tratado anteriormente.

Crea un archivo llamado **Count.java** en tu computadora copiando o descargando el código fuente **Count.java**. Los ejemplos de esta lección asumen que lo has situado en el directorio **C:\Test**.

Ahora compilamos y ejecutamos la aplicación **Count** para ver qué hace. Necesitamos especificar (como argumento) el nombre y el path de un archivo a leer

Importante: Para esta lección, pon el archivo de datos en otro directorio distinto al que contiene la aplicación **Count**. Los ejemplos asumen que has puesto el archivo de datos en el directorio **C:\TestData**. Más adelante en esta lección verás como una aplicación que se ejecuta bajo un controlador de seguridad no puede leer archivos a menos que explícitamente se le de permiso para hacerlo. Sin embargo, una aplicación, **siempre** puede leer un archivo de su propio directorio (o un subdirectorio); sin necesitar permiso explícito.

Un ejemplo de ejecución podría ser

```
C:\Test>java Count C:\TestData\data
Counted 65 chars.
```

- **Crear un Archivo JAR que Contenga el Archivo Class**

Para crear in archivo JAR que contenga el archivo **Count.class**. Debemos teclear lo siguiente en nuestra línea de comandos.

jar cvf Count.jar Count.class

Esto crea el archivo JAR, **Count.jar**, y sitúa el archivo **Count.class** dentro de él.

- **Generar Claves**

Si un firmante de código no tiene todavía una clave privada para firmar el código, primero debe generar la clave junto con una correspondiente clave pública que puede ser usada por el sistema de ejecución del receptor para verificar la firma.

Como esta lección asume que uno no cuenta con dichas claves, crearemos un keystore llamado **charlystore** y se ha crear una entrada con una pareja de claves pública/privada recién creada (con la clave pública en un certificado).

Ahora imaginemos que somos Charly Valentino y que trabajamos en el departamento de ventas de la compañía SYMBIOSYS. Tecleamos el siguiente comando para crear el keystore llamado **charlystore** y para generar las claves para Charly Valentino.

keytool -genkey -alias signFiles -keypass kpi135
-keystore charlystore -storepass ab987c

(Nota: Debe teclearse como una sola línea.)

- **Subpartes del Comando keytool**

Veamos que significa cada una de las subpartes de **keytool**.

- El comando para generar claves es -genkey.
- La subparte -alias signFiles indica el alias que se va usar en el futuro para referirse a la entrada del keystore que contiene las claves que se van a generar.
- La subparte -keypass kpi135 especifica una password para la clave privada a generar. Esta password siempre en necesaria en caso de acceder a la entrada del keystore que contiene la clave. La entrada no tiene que tener su propia password ; si no incluimos una opción -keypass, se nos pedirá el password de la clave y nos ofrecerá la opción de permitir que sea la misma que la del keystore.
- La subparte -keystore charlystore indica el nombre (y opcionalmente el path) del keystore que estamos creando o usando.
- La subparte -storepass ab987c indica el password del keystore. Si no incluimos la opción **-storepass**, se nos pedirá el password para el keystore.

Por razones de seguridad no se recomienda ingresar los password en la línea de comandos, porque podrían ser interceptados fácilmente. En su lugar hay que desactivar las opciones **-keypass** y **-storepass** y teclear el password cuando se pida.

- **Información de Nombre-Distinguido**

Si usamos el comando anterior, nos preguntará por la información del nombre-distinguido. En las siguientes preguntas, la parte en negrita indica lo que deberíamos responder.

What is your first and last name?

```
  [Unknown]:  Charly Valentino
What is the name of your organizational unit?
  [Unknown]:  Systems Security
What is the name of your organization?
  [Unknown]:  SYMBIOSYS
What is the name of your City or Locality?
  [Unknown]:  SF
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Charly Valentino, OU=Systems Security, O=SYMBIOSYS,
    L=SF, ST=CA, C=US> correct?
  [no]:  y
```

- **Resultados del Comando**

El comando **keytool** crea el keystore llamado **charlystore** (si no existía anteriormente) en el mismo directorio en que se ejecuta el comando y le asigna el password **ab987c**. El comando genera una pareja de claves pública/privada para la entidad cuyo nombre distinguido tiene un nombre común de Charly Valentino y la unidad organizativa de Systems Security.

El comando crea un certificado auto-firmado que incluye la clave pública y la información del nombre distinguido. (El nombre distinguido que suministramos se usará en el campo "subject" en el certificado). Este certificado será válido durante 90 días, el periodo por defecto de validez si no especificamos una opción -validity. El certificado está asociado con la clave privada en una entrada del keystore referida por el alias **signFiles**. La clave privada tiene asignada el password **kpi135**.

Nota: El comando podría ser más corto si aceptamos las opciones por default y si deseamos que se nos pidan varios valores. Siempre que ejecutemos el comando **keytool**, se usan las opciones no especificadas que tengan valores por defecto, y se nos pedirá cualquier valor necesario. Para el comando **genkey**, las opciones con valores por defecto incluida el alias (cuyo valor por defecto es **mykey**), validity (90 días), y keystore (el archivo llamado **.keystore** en nuestro directorio home). Los valores requeridos son dname, storepass, y keypass.

- **Firmar el Archivo JAR**

Ahora estamos listos para firmar el archivo JAR. Tecleando lo siguiente en la línea de comandos firmaremos el archivo JAR **Count.jar**, usando la clave privada en la entrada del keystore con el alias **signFiles**, y el nombre del archivo JAR firmado resultante será **sCount.jar**.

```
jarsigner -keystore charlystore -signedjar sCount.jar Count.jar signFiles
```

Se nos pedirá el password del keystore (**ab987c**) y el password de la clave privada (**kpi135**).

Nota: La herramienta **jarsigner** extrae el certificado de la entrada del keystore cuyo alias es **signFiles** y lo adjunta a la firma generada para el archivo JAR firmado.

- **Exportar el Certificado de la Clave Pública**

Ahora tenemos un archivo JAR firmado **sCount.jar**. El sistema de ejecución del receptor del código necesitará autentificar la firma cuando la aplicación **Count** del archivo JAR firmado intente leer un archivo y un archivo de política le conceda el permiso para este código firmado.

Para permitir que el sistema de ejecución autentifique la firma, el keystore del receptor necesita tener la clave pública correspondiente a la clave privada usada para generar la firma. Podemos dársela al receptor enviando una copia del certificado de autentificación de la clave pública. Copiamos este certificado desde el keystore **charlystore** a un archivo llamado **CharlyValentino.cer** de esta forma.

```
keytool -export -keystore charlystore -alias signFiles -file CharlyValentino.cer
```

Se nos pedirá el password del keystore (**ab987c**).

Pasos para el Receptor del Código

Ahora actuaremos como Rob, que recibe el archivo JAR firmado y el archivo del certificado de Charly. Realizaremos los siguientes pasos.

- **Observar las Restricciones de la Aplicación**

La última parte de la lección para controlar aplicaciones muestra cómo una aplicación puede ser ejecutada bajo un control de seguridad llamando al intérprete con el nuevo argumento de la línea de comando **-Djava.security.manager**. ¿Pero qué pasa si la aplicación reside dentro de un archivo JAR?

Una de las opciones del intérprete es la opción **-cp** (class path), con la que se especifica un path de búsqueda para las clases de la aplicación y los recursos. Así, por ejemplo, para ejecutar la aplicación **Count** dentro del archivo JAR **sCount.jar**, especificando el archivo **C:\TestData\data** como su argumento, podríamos teclear lo siguiente desde el directorio que contiene **sCount.jar**.

```
java -cp sCount.jar Count C:\TestData\data
```

Para ejecutar la aplicación con un controlador de seguridad, simplemente añadimos **-Djava.security.manager**, como en:

```
java -Djava.security.manager -cp sCount.jar Count C:\TestData\data
```

Cuando ejecutemos este comando, debe de salir la siguiente excepción.

```
Exception in thread "main" java.security.AccessControlException.
access denied (java.io.FilePermission C:\TestData\data read)
   at java.security.AccessControlContext.checkPermission(Compiled Code)
   at java.security.AccessController.checkPermission(Compiled Code)
   at java.lang.SecurityManager.checkPermission(Compiled Code)
   at java.lang.SecurityManager.checkRead(Compiled Code)
   at java.io.FileInputStream.<init>(Compiled Code)
   at Count.main(Compiled Code)
```

Esta **AccessControlException** esta informando de que la aplicación no tiene permiso para leer el archivo **C:\TestData\data**. Esta excepción se crea porque una aplicación que se está ejecutando bajo un controlador de seguridad no puede leer o escribir archivos o acceder a otros recursos a menos que tenga permiso explícito para hacerlo.

- **Importar el Certificado como un Certificado Confiable**

Antes de poder conceder permiso al código firmado para que lea un archivo específico, necesitamos importar el certificado de Charly como un certificado confiable en nuestro keystore.

Supongamos que hemos recibido de Charly.

- El archivo JAR **sCount.jar**, que contiene el archivo **Count.class**, y
- el archivo **CharlyValentino.cer**, que contiene el certificado de clave pública para la clave pública correspondiente a la clave privada usada para firmar el archivo JAR.

Aunque acabemos de crear estos archivos y realmente no han sido transportados desde ningún lugar, podemos simular que somos otra persona distinta al creador y emisor, Charly. Imaginemos que ahora somos Rob, crearemos un keystore llamado **Robstore** y lo usaremos para importar dentro el certificado dentro de de una entrada con el alias de **charly**.

Un keystore se crea siempre que usemos un comando **keytool** especificando un keystore que no exista. Así podemos crear el **Robstore** e importar el certificado mediante un solo comando **keytool**. Haremos lo siguiente en nuestra línea de comandos.

1. Vamos al directorio que contiene el archivo del certificado de la clave pública **CharlyValentino.cer**. (Ya deberíamos estar en él, ya que esta lección asume que estamos trabajando en un sólo directorio).
2. Tecleamos el siguiente comando en una línea.
   3. **keytool -import -alias charly**
   4. **-file CharlyValentino.cer -keystore Robstore**

Como el keystore no existe todavía, se creará, y nos pedirá que introduzcamos el password para el keystore, tecleamos cualquier password que queramos.

El comando **keytool** imprimirá la información del certificado y nos pedirá que lo verifiquemos, por ejemplo, comparando la huella dactilar del certificado mostrado con aquellas obtenidas de otra fuente de información (contrastada). (Cada huella dactilar es un número relativamente corto y único que identifica un certificado). Por ejemplo, en el mundo real podríamos llamar a Charly y preguntarle cual debería ser su huella dactilar. Ella puede obtener la huella del archivo **CharlyValentino.cer**.

keytool -printcert -file CharlyValentino.cer

Si las huellas dactilares corresponden, el certificado no ha sido modificado durante el tránsito. En este caso podemos permitir que **keytool** proceda con la inclusión de la entrada de certificado confiable en el keystore. La entrada contiene los datos del certificado de la clave pública obtenidos del archivo **CharlyValentino.cer** y se le asigna el alias **charly**.

- **Configurar un Archivo de Política para Conceder los Permisos Requeridos**

Luego usaremos un la herramienta Policy Tool para crear un archivo de política llamando **Robpolicy** y conceder un permiso para el código del archivo JAR firmado.

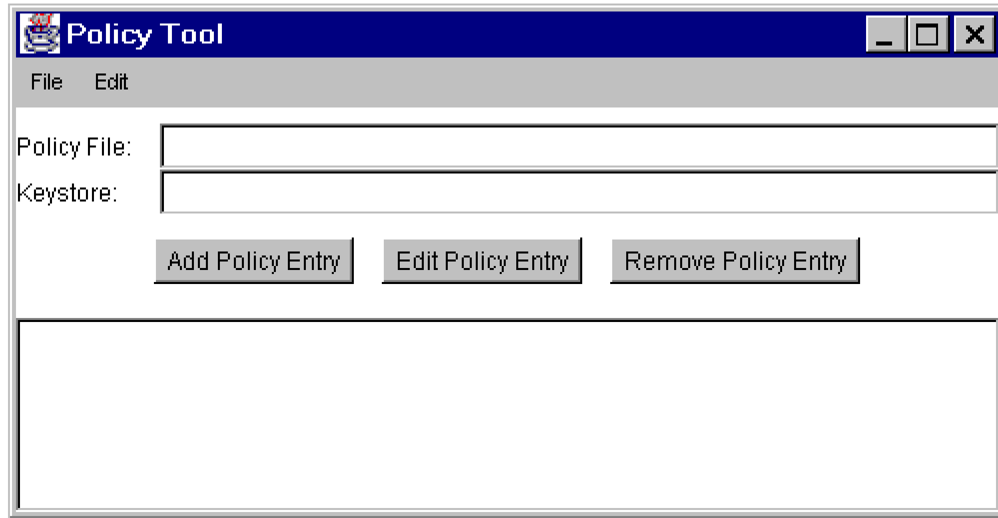
El archivo JAR debe haber sido firmado usando la clave privada correspondiente a la clave pública importada en el keystore de Rob (**Robstore**) en los pasos anteriores. El certificado que contiene la clave pública tiene el alias de **charly** en el keystore. Concederemos a dicho código permiso para leer cualquier archivo del directorio **C:\TestData**.

- **Arrancar Policy Tool**

Para arrancar Policy Tool, simplemente tecleamos esto en la línea de comandos.

policytool

Esto nos trae la ventana de Policy Tool. Siempre que se arranca, Policy Tool intenta rellenar su ventana con información de algo que algunas veces es referido como "archivo de política de usuario", que, por defecto, es un archivo llamado **.java.policy** que está en el directorio home. Si Policy Tool no puede encontrar ese archivo, informa de la situación y muestra una ventana Policy Tool en blanco (es decir, una ventana con cabeceras y botones pero sin datos, como se muestra en la figura.



Crearemos y trabajaremos sobre un archivo de política distinto del archivo de política de usuario, ya que las lecciones de esta sección no necesitan que se hagan modificaciones en el archivo de política de usuario oficial.

Asumiendo que estamos viendo una ventana de Policy Tool en blanco (si no es así, seleccionamos **New** en el menú **File**), podemos proceder inmediatamente a crear un nuevo archivo con las políticas.

- **Especificar el Keystore**

Para esta lección concederemos permiso a todo el código en archivos JAR firmados por el alias charly acceso de lectura a todos los archivos en el directorio **C:\TestData**. Necesitamos:

1. Especificar el keystore que contiene la información del certificado con el alias charly.
2. Crear la entrada de política que concede los permisos.

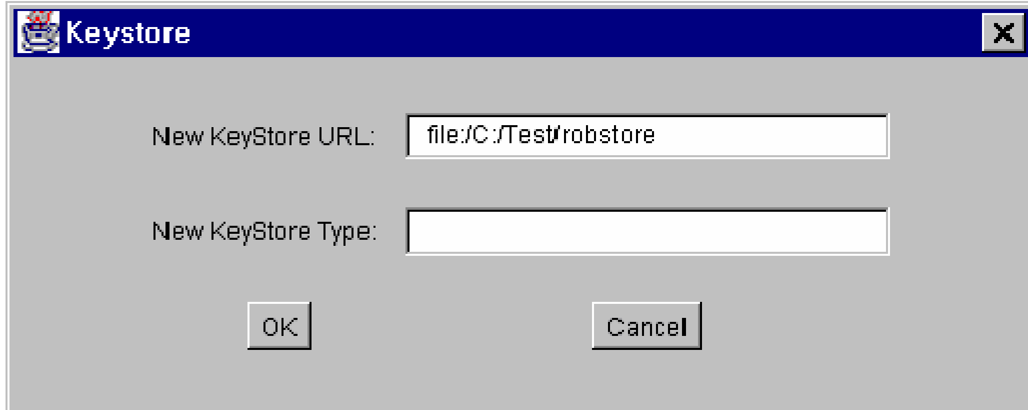
El keystore es uno llamado **Robstore** creado en el paso "Importar el Certificado como un Certificado Confiable".

Para especificar el keystore, elegimos el comando **Change Keystore** en el menú **Edit** en la ventana principal de Policy Tool. Esto nos muestra la caja de diálogo en que podemos especificar la URL del keystore especificado y el tipo de keystore.

Para especificar el keystore llamado **Robstore** en el directorio **Test** del disco **C:**, tecleamos la siguiente URL en la caja de texto etiquetada New KeyStore URL.

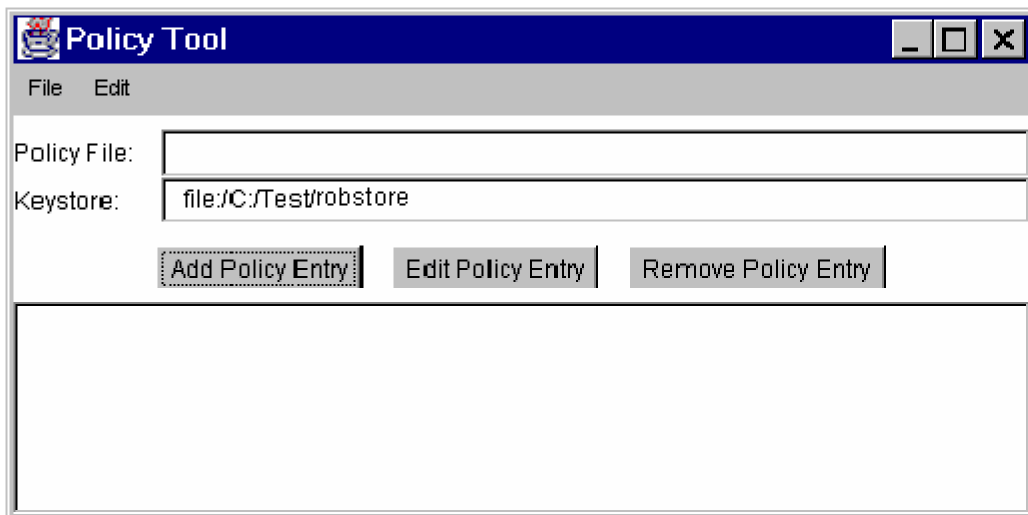
```
file:/C:/Test/Robstore
```

Podemos dejar en blanco la caja de texto etiquetada New KeyStore Type si el tipo de keystore es uno por defecto, como se especifica en archivo de propiedades de seguridad. Nuestro keystore será del tipo por defecto, por eso dejamos la caja de texto en blando. El resultado lo muestra la siguiente figura.



Nota: el valor de New KeyStore URL es una URL y por eso siempre se deben usar diagonales (nunca diagonales inversas), como separadores de directorio.

Cuando hayamos especificado la URL, seleccionamos **OK**. La caja de texto etiquetada Keystore se rellena con la URL.

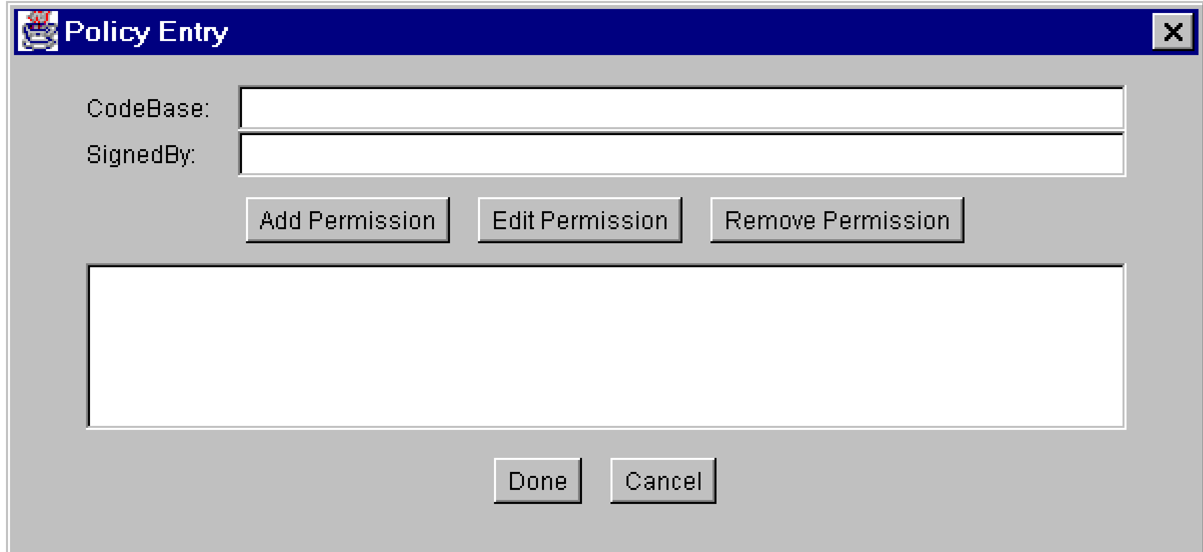


Luego, necesitamos especificar la nueva entrada de política.

- **Añadir una Entrada de Política con un SignedBy Alias**

Para conceder permiso al código firmado por **charly** para leer cualquier archivo en el directorio **C:\TestData**, necesitamos crear una entrada de política concediendo este permiso. Observa que "Code signed by **charly**" es una abreviatura de decir "Código en un archivo class contenido en un archivo JAR, donde JAR fue firmado usando la clave privada correspondiente a la clave pública que aparece en un certificado del keystore en una entrada con el alias **charly**."

Elegimos el botón **Add Policy Entry** en la ventana principal de Policy Tool. Esto nos trae la caja de diálogo Policy Entry.



Usando esta caja de diálogo, tecleamos el siguiente alias en la caja de texto **SignedBy**.

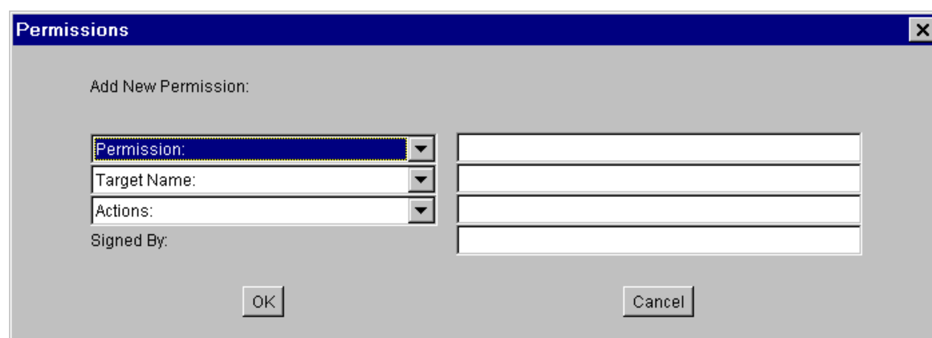
charly

Dejamos la caja de texto **CodeBase** en blanco, para conceder permiso a **todo** el código firmado por **charly**, el permiso, sin importar de donde venga.

Nota: si queremos restringir el permiso sólo para el código firmado por **charly** que venga del directorio **C:\Test**, deberíamos teclear la siguiente URL en la caja de texto **CodeBase**.

file:/C:/Test/*

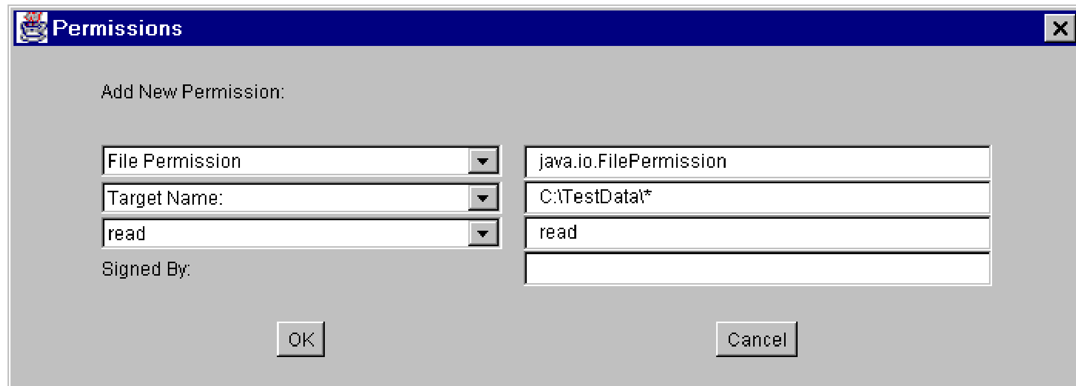
Para añadir el permiso, elegimos el botón **Add Permission**. Esto nos trae la caja de diálogo Permissions.



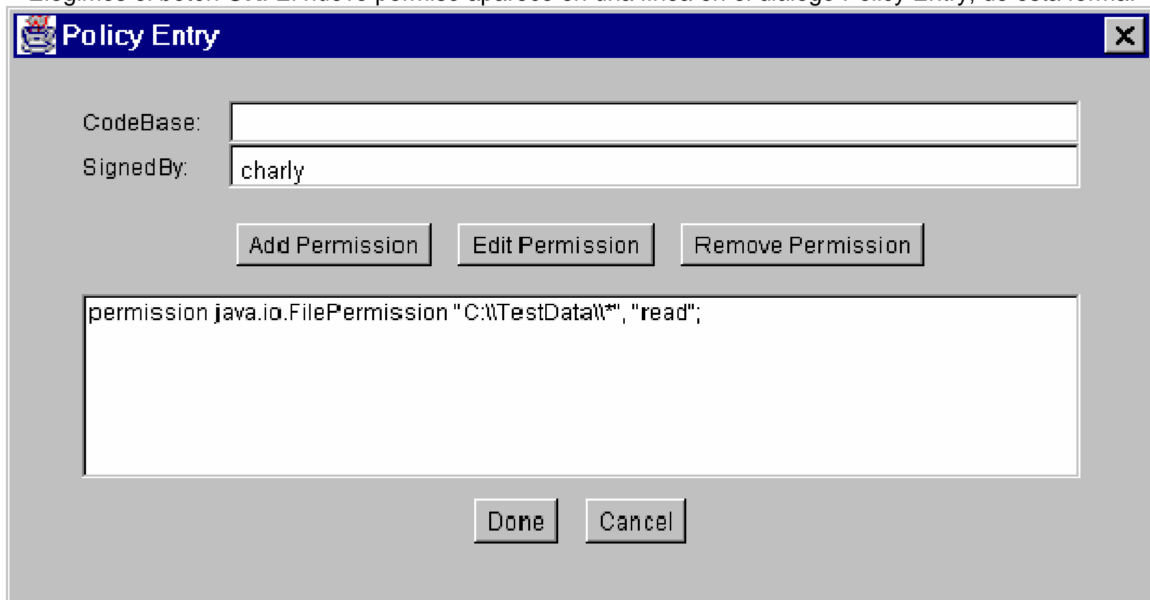
Ahora hacemos lo siguiente:

1. Elegimos **File Permission** desde la lista de permisos. El nombre del permiso completo (**java.io.FilePermission**) aparece en la caja de texto que hay a la derecha de la lista desplegable.
2. Tecleamos lo siguiente en la caja de texto que hay a la derecha de la lista etiquetada Target Name para especificar todos los archivos del directorio **C:\TestData**.
3. **C:\TestData***
4. Especificamos el acceso de lectura eligiendo la opción **read** en la lista de Actions.

Ahora la caja de Diálogo Permissions se parecerá lo que mostramos a continuación:

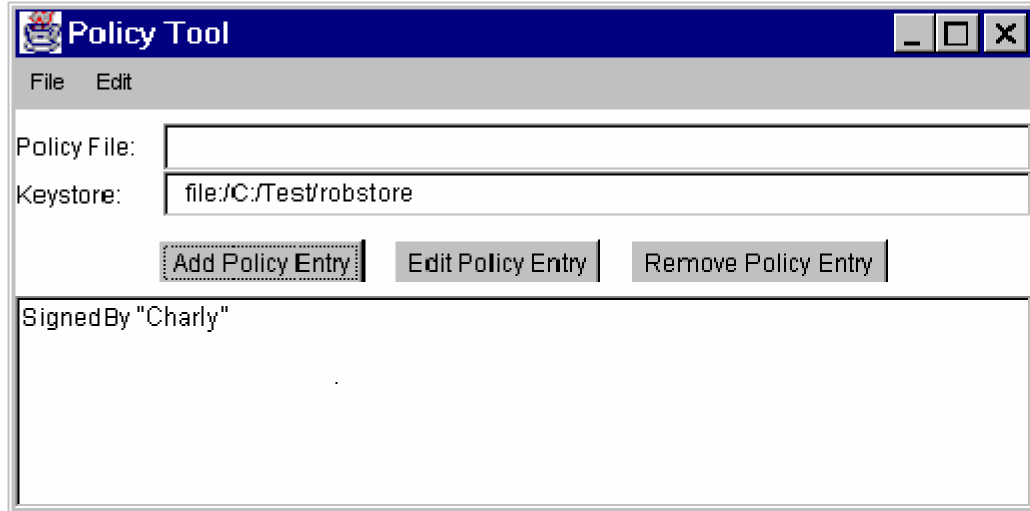


Elegimos el botón **OK**. El nuevo permiso aparece en una línea en el diálogo Policy Entry, de esta forma:



Nota: Cada diagonal inversa que hayamos tecleado en el path del archivo ha sido reemplazada por dos diagonales inversas, por conveniencia. Las cadenas en un archivo de política son procesadas por un tokenizer que permite usar "\" como carácter de escape (por ejemplo **\n** para indicar nueva línea), por eso el archivo de política requiere **dos** diagonales inversas para indicar una sola diagonal inversa. Si usamos una sola diagonal inversa como separadores de directorios, Policy Tool las convierte automáticamente en dobles diagonales inversas.

Ahora ya hemos especificado la entrada de política, elegimos el botón **Done** en el diálogo Policy Entry. La ventana Policy Tool contiene una línea que representa la entrada de política, mostrando el valor **SignedBy**, como se ve en la figura.



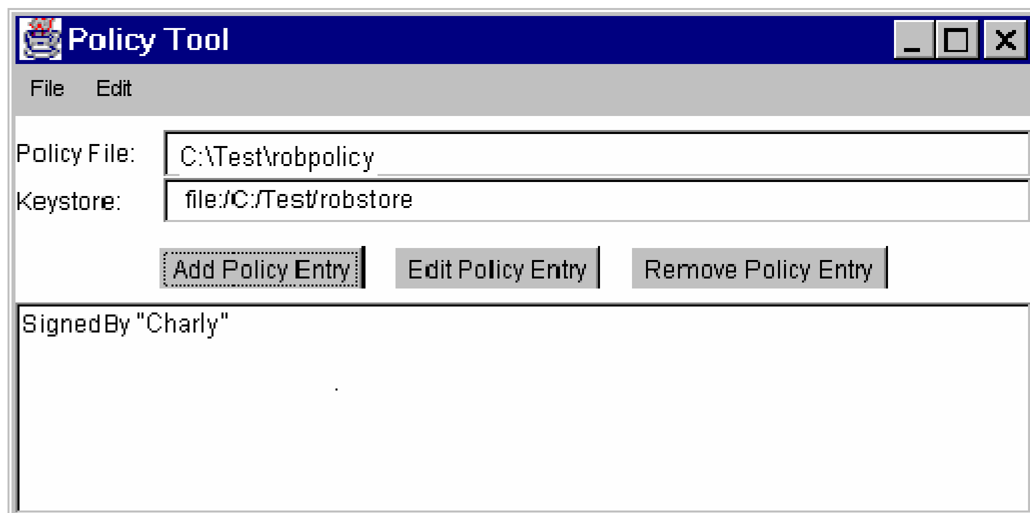
- **Grabar el Archivo de Política**

Para grabar el nuevo archivo de política que hemos creado, elegimos el comando **Save As** en el menú **File**. Esto nos trae la caja de diálogo **Save As**.

Navegamos por la estructura de directorio para obtener el directorio en el que salvar el archivo de política: el directorio **Test** del disco **C:**. Tecleamos el nombre de archivo.

Robpolicy

Luego elegimos el botón **Save**. Entonces se graba el archivo de política, y su nombre y path se muestran en la caja de texto etiquetada Policy File, como se muestra en la siguiente figura.



Luego salimos de Policy Tool seleccionando el comando **Exit** del menú **File**.

- **Ver los efectos del Archivo de Política**

En los pasos anteriores hemos creado una entrada en el archivo de política **Robpolicy** concediendo permiso al código firmado por **charly** para leer archivos en el directorio **C:\TestData** (o el directorio **testdata** en tu directorio home si estamos trabajando en UNIX). Ahora deberíamos poder ejecutar con éxito el programa **Count** para leer y contar los caracteres de un archivo del directorio especificado, incluso aunque estemos ejecutando la aplicación con un controlador de seguridad.

Como se describe al anteriormente, hay dos formas posibles en la podemos considerar el archivo de política **Robpolicy** como parte de la política general, además de los archivos de política especificados en el archivo de propiedades de seguridad. La primera opción es especificar el archivo de política adicional en una propiedad pasada el sistema de ejecución. La segunda opción es añadir una línea al archivo de propiedades de seguridad especificando el archivo de política adicional.

Glosario de Términos de Seguridad en Java 2

A continuación mostramos los términos utilizados en este documento que pueden no ser familiares al usuario no especializado en seguridad informática y son esenciales para el entendimiento del funcionamiento de la seguridad en el sistema IE.

Términos sobre Seguridad

Certificado

Un certificado es una sentencia firmada digitalmente de una entidad (persona, compañía, etc.), diciendo que la clave pública de alguna otra entidad tiene un valor particular. Si creemos la firma del certificado, creeremos que la asociación incluida en el certificado entre la clave pública especificada y la otra entidad es auténtica.

Algoritmo de Criptografía

Un algoritmo criptográfico es un algoritmo usado para asegurar uno de los puntos siguientes.

1. la confidencialidad de los datos
2. autentificación del emisor de los datos
3. integridad de los datos enviados
4. para que un emisor no puede negar que ha enviado un mensaje particular.

Un algoritmo de firma digital proporciona algunas de estas características.

Una versión separada (Java Cryptography Extensions) proporciona APIs y algoritmos relacionados con la encriptación y desencriptación.

Desencriptación

Desencriptación es lo contrario de encriptación; el proceso de tomar los textos ecriptados y la clave criptográfica, y producir un texto claro (los datos originales sin encriptar)

Firma Digital

Una firma digital es una cadena de bits que se ha calculado desde algunos datos (los datos a ser "firmados") y la clave privada de la entidad.

La firma puede ser usada para verificar que los datos vienen de la entidad y que no han sido modificadas durante el tránsito.

Como una firma escrita, una firma digital tiene muchas características útiles.

- Se puede verificar su autenticidad, mediante un cálculo que usa la clave pública correspondiente a la clave privada usada para generar la firma.
- No se puede privar de ella, asumiendo que la clave privada mantiene el secreto
- Es una función de los datos firmados y no puede ser reclamada como la firma de otros datos.
- No se pueden modificar los datos firmados; si lo son, la firma no se verificará como auténtica.

Dominio o Protección de Dominio

Una protección de domino ("dominio" para acortar) encierra un conjunto de clases cuyos ejemplares tienen concedidos los mismos permisos.

Además de un conjunto de permisos, un dominio comprende un **CodeSource**, que es un conjunto de **PublicKey** junto con un codebase (en el formado de una URL). Así, las clases firmadas por la misma clave y desde la misma URL se sitúan en el mismo dominio. Las clases que tienen los mismos permisos pero tienen diferentes códigos fuente pertenecen a diferentes dominios.

Actualmente en el JDK 1.2, los dominios se crean "bajo demanda" como resultado de la carga de las clases.

Hoy todo el código enviado como parte del JDK se considera código del sistema y ejecuta dentro del único dominio del sistema. Cada applet o aplicación se ejecutan en su propio dominio, determinado por su código fuente.

Encriptación

Encriptación es el proceso de tomar los datos (llamado texto claro) y una clave criptográfica y producir un texto encriptado, que son unos datos sin significado para cualquiera que no conozca la clave.

Clase Motor

Una "clase motor" define un servicio criptográfico de una forma abstracta (sin una implementación concreta).

Un servicio criptográfico siempre está asociado con un algoritmo o tipo particular, y proporciona operaciones de criptografía (como aquellas para firmas digitales o message digests), genera o suministra el material criptográfico (claves o parámetros) requeridos para las operaciones de criptografía, o genera objetos de datos (keystores o certificados) que encapsulan claves criptográficas (que pueden ser usadas en una operación criptográfica) de una forma segura. Por ejemplo, dos de las clases motor son **Signature** y **KeyFactory**. La clase **Signature** proporciona acceso a la funcionalidad de un algoritmo de firma digital. Un **KeyFactory** DSA suministra un clave privada DSA o una clave pública (desde sus especificaciones codificadas o transparentes) en un formato utilizable por los métodos **initSign** o **initVerify**, respectivamente, de un objeto **Signature** DSA.

Los clientes API piden y utilizan ejemplares de clases motor para llevar a cabo las operaciones correspondientes. Las siguientes clases motor están definidas en el JDK 1.2.

- **MessageDigest** - usada para calcular el mensaje message digest (hash) de los datos especificados.
- **Signature** - usada para firmar los datos y verificar las firmas digitales.
- **KeyPairGenerator** - usada para generar una pareja de claves pública y privada disponibles para una algoritmo especificados.
- **KeyFactory** - usada para convertir claves criptográficas del tipo Key en especificaciones de claves (representaciones de material clave), y viceversa.
- **CertificateFactory** - usado para crear certificados de clave pública y Certificate Revocation Lists (CRLs).
- **KeyStore** - usado para crear y manejar un keystore. Un keystore es una base de datos de claves. Las claves privadas de un keystore tienen una cadena de certificados asociadas con ellas, que autentifican la correspondiente clave pública. Un keystore también contiene certificados de entidades confiables.

- **AlgorithmParameters** - usado para manejar los parámetros de un algoritmo particular, incluyendo los parámetros de codificación y decodificación.
- **AlgorithmParameterGenerator** - usada para generar un conjunto de parámetros disponibles para un algoritmo especificado.
- **SecureRandom** - usada para generar números pseudo-aleatorios.

Una clase motor proporciona la interfase para la funcionalidad de un tipo específico de servicio criptográfico (independiente de un algoritmo criptográfico particular). Define métodos de "Application Programming Interface" (API) que permiten a las aplicaciones acceder a tipos específicos de servicios criptográficos que proporciona. Las implementaciones actuales (para uno o más proveedores) son aquellas para algoritmos específicos. La clase motor **Signature**, por ejemplo, proporciona acceso a la funcionalidad de un algoritmo de firma digital. La implementación actual suministrada en una subclase **SignatureSpi** será aquella para una clase específica de algoritmo de firma, como SHA1 con DSA, SHA1 con RSA, MD5 con RSA, e incluso algunos algoritmos de firmas propietarios.

Los interfaces de aplicación suministrados por una clase motor están implementados en términos de un **"Service Provider Interface" (SPI)**. Es decir, para cada clase motor, hay una correspondiente clase abstracta SPI. que define los métodos del Service Provider Interface que le proveedor de servicios criptográficos debe implementar.

Un ejemplar de una clase motor se crea llamando al método factory **getInstance** de la clase motor, que encapsula la implementación del SPI y lo devuelve al llamador. Cada método del API del ejemplar de la clase motor generada invoca al método del SPI correspondiente del objeto SPI encapsulado.

El nombre de cada clase SPI es el mismo que el de la correspondiente clase motor, seguido por "SPI". Por ejemplo, la clase SPI correspondiente a la clase motor **Signature** es la clase **SignatureSpi**.

Algoritmo Resumen de Mensaje

Un Resumen de mensaje es una función que toma una entrada de datos de tamaño arbitrario (referido como un mensaje) y genera una salida de tamaño fijo, llamada resumen (o picadillo). Un resumen tiene las siguientes propiedades

- Debería ser computacionalmente imposible para encontrar otra cadena de entrada que genere el mismo resumen.
- El resumen no revela nada sobre la entrada usada para generarlo.

Los algoritmos de resumen de mensaje se usan para producir un identificador de datos único y fiable . Los resúmenes algunas veces se llaman "huellas dactilares digitales".

Algunos algoritmos de firma digital usan un algoritmo de resumen de mensaje para calcular el conjunto de los datos que están siendo firmados, y luego firma digitalmente el valor del conjunto en vez de firmar los datos originales, ya que firmar los datos originales podría ser muy costoso.

Representación de Clave Oscura

Una representación de clave oscura es aquella en la que no tenemos acceso directo al material que constituye la clave. En otras palabras: "opaco" ofrece un acceso limita a la clave - tres métodos definidos por la interfase "Key": **getAlgorithm**, **getFormat**, y **getEncoded**.

Esto está en contraste con una representación transparente, en la que podemos acceder a cada valor del material de la clave individualmente, a través de uno de los métodos "get" definidos en la especificación de la clase correspondiente.

Representación Oscura de Parámetros

Una representación de parámetros oscura es aquella en la que no tenemos acceso directo a las campos de parámetros; sólo podemos el nombre del algoritmo asociado con el conjunto de parámetros y alguna clase de codificación para el conjunto de parámetros.

Esto contrasta con una representación transparente, en la que podemos acceder a cada valor del material de la clave individualmente, a través de uno de los métodos "get" definidos en la especificación de la clase correspondiente.

Permiso

Un permiso representa acceso a un recurso del sistema. Para poder conceder a un applet el acceso a un recurso (o a una aplicación ejecutándose con un controlador de seguridad), el permiso correspondiente debe concederse explícitamente.

La política efectiva para un entorno de aplicación Java específica aquellos permisos que están disponibles para varios fuentes.

Un permiso tiene típicamente un nombre (frecuentemente definido como "target name") y, en algunos casos, una lista separada por comas de una o más acciones.

El JDK tiene un número de tipos de permisos internos (clases), y nuevos tipos que pueden ser añadidos por los clientes.

Política

La política en efecto para un entorno de aplicación Java específica qué permisos están disponibles para el código de varias fuentes.

La localización de la fuente de la información de política se configura con la implementación de Política. El JDK contiene una implementación de política por defecto que obtiene su información desde archivos de configuración de política estáticos.

Archivo de Política

La política para un entorno de aplicación Java (especificación de los permisos disponibles para código de varias fuentes) está representada por un objeto Policy.

La localización de la fuente de la información de política se configura con la implementación de **Política**. El JDK 1.2 contiene una implementación de **Policy** por defecto que obtiene su información desde archivos de configuración de política estáticos, también conocidos como "archivos de política".

Clave Privada

Una clave privada es un número que se supone conocido sólo por una entidad particular. Es decir, las claves privadas siempre deben mantenerse en secreto. Podemos usarlas para generar firmas digitales. Una clave privada siempre está asociada con una sola clave pública.

Privilegiado

Siempre que se intenta un acceso a recursos, todo el código del thread hasta ese punto debe tener permiso para acceder a los recursos, a menos que el código del thread haya sido marcado como "privilegiado". Es decir, supongamos el control de acceso que checa las ocurrencias en la ejecución del thread que tiene una cadena de múltiples llamadas. Cuando se llama al método **checkPermission** de **AccessController** el llamador más reciente, el algoritmo básico para decidir si se permite o deniega el acceso requerido de esta forma.

Si el dominio de cualquier llamante de la cadena de llamadas no tiene el permiso requerido, se lanza una **AccessControlException**, a menos que lo siguiente sea cierto: un llamador cuyo dominio tenga concedido el permiso ha sido marcado como "privilegiado" y todas las llamadas subsecuentes (directas o indirectas) tienen el mismo permiso.

(Nota: El código de sistema automáticamente tiene todos los permisos.)

Marcar un código como "privilegiado" permite a un trozo de código confiable un permiso temporal para acceder a más recursos de los que tenía disponible el código que lo llamó. Esto es necesario en algunas situaciones. Por ejemplo, una aplicación podría no tener permitido el acceso directo a archivos que contienen fuentes, pero la utilidad del sistema para mostrar un documento debe obtener dichas fuentes, en beneficio del usuario. Para poder hacer esto, la utiliza del sistema se convierte en privilegiada mientras obtiene las fuentes.

Proveedor

Las implementaciones para varios algoritmos criptográficos están proporcionadas por Cryptographic Service Providers. Los proveedores son esencialmente paquetes que implementan una o más clases motor para algoritmos específicos. Por ejemplo, el proveedor del JDK, por defecto, llamado "SUN" suministra implementaciones para el algoritmo de firma DSA y los algoritmos de resumen de mensaje MD5 y SHA-1. Otros proveedores podrían definir sus propias implementaciones de estos algoritmos o de otros, como una implementación de una RSA-basada en algoritmo de firma o del algoritmo de resumen de mensaje MD2.

Clave Pública

Una clave pública es un número asociado con una entidad particular (por ejemplo, una persona o una organización). Una clave pública está diseñada para ser conocida por todos aquellos que necesiten interacciones verdaderas con dicha entidad. Una clave pública siempre está asociada con una única clave privada, y puede ser usada para verificar la firma digital generada usando la clave privada.

Controlador de Seguridad

Actualmente, todo el código de sistema del JDK llama a los métodos del controlador de seguridad para comprobar la política actualmente en efectivo para realizar controles de acceso. Hay un controlador de seguridad (implementación de **SecurityManager**) instalado siempre que se ejecuta un applet; el **appletviewer** y la mayoría de los navegadores, instalan un controlador de seguridad. El controlador de seguridad previene de que el código del applet pueda acceder a recursos a menos que tenga permiso explícito para hacer esto en una entrada en un archivo de política.

Un controlador de seguridad **no** se instala automáticamente cuando se ejecuta una aplicación, y así la aplicación tiene total acceso a los recursos (como fue siempre en el caso del JDK 1.1). Para aplicar la misma política de seguridad para una aplicación encontrada en el sistema local de archivos como si fuera un applet descargado, ejecutando la aplicación en la máquina virtual jaca con el nuevo argumento de la línea de comandos "-Djava.security.manager" o la propia aplicación debe llamar al método **setSecurityManager** en la clase **java.lang.System** para instalar un controlador de seguridad.

Certificado Auto-Firmado

Un certificado auto-firmado es aquél en que el emisor-firmante es el mismo que el sujeto la entidad cuya clave pública está siendo autentificada por el certificado).

Código Firmado

Una forma abreviada de decir "código en un archivo class que aparece en un archivo JAR que fue firmado".

Representación de Clave Transparente

Una representación de clave transparente significa que puedes acceder a cada valor material de la clave de forma individual, a través de uno de los métodos "get" definidos en la especificación de la clase correspondiente. Por ejemplo, DSAPrivateKeySpec define los métodos **getX**, **getP**, **getQ**, y **getG**, para acceder a la clave privada **x**, y los parámetros del algoritmo DSA usada para calcular la clave: la principal **p**, la sub-principal **q**, y la base **g**.

Esto está en contraste con una representación oscura, está definido por la interfase Key, en el que no tenemos acceso a los campos materiales de la clave.

Representación de Parámetros Transparente

Una representación transparente de un conjunto de parámetros significa que podemos acceder al valor de parámetro individualmente, a través de uno de los métodos "get" definidos en la especificación de la clase correspondiente. Por ejemplo, DSAParameterSpec define los métodos **getP**, **getQ**, y **getG**, para acceder a los parámetros comunitarios DSA "p", "q", y "g", respectivamente.

Esto está en contraste con una representación oscura, como la suministrada por la clase AlgorithmParameters, en la que no tenemos acceso directo a los campos de parámetros; sólo podemos obtener el nombre del algoritmo asociado con el juego de parámetros (mediante **getAlgorithm**) y algún tipo de codificación para el conjunto de parámetros (mediante **getEncoded**).

Resumen del API de Seguridad en Java 2

¿Qué Proporciona el API de Seguridad del JDK en beneficio al sistema IE?

El API de Seguridad del JDK es un API corazón de Java, construido alrededor del paquete **java.security** (y sus subpaquetes). A continuación mostramos una pequeña reseña de los paquetes de seguridad que se detallan a lo largo de este documento, de los cuales algunos fueron piezas fundamentales para desarrollar un plan de seguridad en el sistema IE.

Métodos del API

Los clientes pueden llamar a los métodos del API para incorporar funcionalidades de seguridad en sus aplicaciones, incluyendo.

- uso de servicios de criptografía implementada por el proveedor internet "SUN" y/o otros paquetes proveedores. Estos servicios incluyen firma digital, resumen de mensajes, generación de claves y algoritmos de generación de números aleatorios; creación de keystores y servicios de manejo; algoritmos de generación de parámetros y servicios de manejo; y "fábricas" de claves y certificados para la creación de claves y certificados desde material existente (por ejemplo, codificados).
- añadir chequeos de seguridad para asegurarnos de que el llamador tiene concedido un permiso personalizado (puedes ver Personalizar las Características de Seguridad).
- marcar código como "privilegiado" para que pueda ejercitar más permisos que los del llamador.
- obtener o seleccionar valores para las propiedades de seguridad del sistema.

La sección Generar y Verificar Firmas muestra cómo escribir programas que usan los aspectos criptográficos del API de Seguridad del JDK para generar (o importar) claves, generar una firma digital para datos usando la clave privada, y verificar la autenticidad de una firma. Este aspecto es crítico dentro del sistema IE para comprobar que los datos que se reciben son efectivamente de la entidad que dice enviarlos.

La sección Implementar Nuestros Propios Permisos ilustra la definición de nuestros propios permisos, añadiendo chequeos de seguridad a nuestro código para asegurarnos de que el llamado tiene el permiso especificado, y hacer código "privilegiado".

La sección Rápida Visión para Controlar Aplicaciones incluye un programa **GetProps** que obtiene los valores de las propiedades **"user.home"** y **"java.home"**.

Personalización de las Características de Seguridad

El API permite a los clientes del sistema IE definir e integrar sus propios:

- nuevos permisos para controlar el acceso y uso del sistema.
- implementaciones de servicios de criptografía (en uno o más paquetes proveedores) para evitar que se robe información.
- implementaciones de **SecurityManager** (para reemplazar al usado por defecto y que es cargado por los applets, y por las aplicaciones cuya ejecución requiere tener un controlador de seguridad.
- implementaciones de **Policy** (para reemplazar la implementación interna por defecto).

La sección Implementar Nuestros Propios Permisos ilustra la definición de nuestros propios permisos y añadir chequeos de seguridad a nuestro código para asegurar que el que solicita la información tiene el permiso especificado.

Añadir una implementación de un servicio de criptografía es algo que un gran número de desarrolladores espera hacer. Se puede ver **How to Implement a Provider for the Java Cryptography Architecture** en la web pública de **java.sun.com** para mayor información en este ámbito.

¿Qué pasa con la Encriptación y la Desencriptación en IE?

Los APIs para encriptación y desencriptación de datos, junto con algunas implementaciones de algoritmos por defecto, se liberan de forma separada en un "Java Cryptography Extension" (JCE) como una paquete añadido al JDK, en concordancia con las regulaciones de control de exportaciones de los EE.UU.

Dentro del sistema IE es muy importante la encripción de datos tanto por la parte de Web, como en el back-end en la base de datos para asegurar que no se pueda robar información y en el caso de que alguien logre acceder a las partes prohibidas del sistema, no pueda descifrar la información tan fácilmente. Los datos son enviados encriptados y desde que se ingresan a la base de datos se vuelven a encriptar.

Sumario de Archivos de Seguridad en Java 2

Los archivos relacionados con la seguridad que se encuentran dentro del JDK 1.2 son.

- El archivo de Propiedades de Seguridad **java.security**.
- El archivo de política del Sistema **java.policy**.
- El Keystore de Certificados **cacerts**.

Estos archivos internos residen en el directorio de propiedades de seguridad del JRE,

java.home/lib/security/ (Solaris)
java.home\lib\security\ (Windows)

(Nota: java.home indica el directorio en el que se instaló el JRE.)

Los archivos relacionados con la seguridad que opcionalmente **nosotros** podríamos crear son.

- El archivo de política del Usuario **.java.policy**.
- Keystores

Cada uno de estos archivos se describe más abajo.

El archivo de Propiedades de Seguridad java.security

En este archivo se configuran varias propiedades de seguridad para usarlas con las clases del paquete **java.security**.

Este archivo especifica

- nombres de paquetes, localizaciones y orden de precedencia.
- La clase a ejemplificar como el objeto **Policy** cuyo permiso estará disponible para el código de varias fuentes.
- URLs para los archivos de política a ser cargados y utilizados cuando se tomen decisiones de seguridad (si el objeto Policy ejemplarizado es uno que utiliza archivos de política).
- si se debe permitir o no la expansión de propiedades en el archivo de política, por ejemplo expandir **${java.home}** al valor de la propiedad **"java.home"**.
- si se puede especificar o no un archivo de política adicional en la línea de comandos con **-Djava.security.policy=somefile**.
- el tipo de keystore por defecto (inicialmente llamado "jks", el tipo de keystore propietario de Sun Microsystems)

Para ver más detalles, el archivo se encuentra en.

java.home/lib/security/java.security (Solaris)
java.home\lib\security\java.security (Windows)

(java.home indica el directorio en el que se instaló el JRE.)

El archivo de Política del Sistema java.policy

Un archivo de política especifica qué permisos están disponibles para el código de varias fuentes.

A este archivo nos referimos como archivo de política del "sistema" porque se utiliza para conceder grandes permisos del sistema. El archivo **java.policy** instalado con el JDK concede todos los permisos a las extensiones estándares, permite a cualquiera escuchar en un puerto no-privilegiado, y permite a cualquier código leer ciertas propiedades "estándar" como las propiedades "os.name" y "file.separator".

Si es necesario, el archivo de política del sistema puede ser modificado, con un editor de texto, o con la herramienta **policytool**. Este último no requiere que conozcamos el formato del archivo de política, usándolo nos ahorramos teclear y evita errores.

El archivo **java.policy** por defecto está localizado en.

java.home/lib/security/java.policy (Solaris)
java.home\lib\security\java.policy (Windows)

(java.home indica el directorio en el que instalamos el JRE.)

Las localizaciones de los archivos de política realmente se especifican en el archivo de propiedades de seguridad como los valores cuyos nombres tienen la forma.

policy.url.n=URL

donde "n" es un número. El archivo de política del sistema está definido en el archivo de propiedades de seguridad como.

policy.url.1=file:${java.home}/lib/security/java.policy

El KeyStore de Certificados cacerts

El archivo **cacerts** representa un keystore de certificados de Autoridades de Certificación (CA).

Los administradores de sistemas pueden configurar y manejar el archivo **cacerts** usando **keytool**, especificando "JKS" como el tipo de keystore (un tipo propietario definido por Sun Microsystems).

En este momento, el archivo **cacerts** viene con cinco certificados de CA de VeriSign.

El archivo **cacerts** se encuentra en.

java.home/lib/security/cacerts (Solaris)
java.home\lib\security\cacerts (Windows)

(java.home indica el directorio donde instalamos el JRE.)

El archivo de Política de Usuario.java.policy

Si queremos crear uno o más archivos de política para nuestro propio uso, podríamos hacerlo usando un editor de texto, o la herramienta **policytool**.

El archivo propiedades de seguridad incluido con el JDK contiene una línea "libre" que especifica un nombre por defecto y localización para un archivo de política de usuario.

```
policy.url.2=file:${user.home}/.java.policy
```

Donde **${user.home}** será reemplazado por el directorio "home" del usuario en el momento de la ejecución, determinado por el valor de la propiedad **"user.home"**.

El archivo especificado **no** tiene porqué existir. Pero si creamos un archivo con ese nombre, en esa localización, el sistema lo cargará cuando tome decisiones de seguridad.

Si queremos crear un archivo de política de usuario pero con otro nombre o localización, simplemente editamos esa línea en el archivo de propiedades de seguridad de la forma apropiada.

Si también queremos añadir archivos de política adiciones, añadimos para cada uno una línea con la forma.

```
policy.url.n=URL
```

donde n sea 3, 4, 5, etc., y URL es la dirección del archivo.

Keystores

Un keystore es una base de datos de claves. Las Claves Privadas de un keystore tienen asociada una cadena de certificados, que autentifican la correspondiente Clave Pública. Un keystore también contiene certificados de autoridades verdaderas. Necesitamos un keystore si.

- queremos generar claves públicas y privadas nosotros mismos.
- queremos usar nuestra clave privada para firmar archivos digitalmente.
- queremos exportar nuestro certificado de clave pública para que otros puedan verificar firmas digitales creadas con nuestra correspondiente clave privada.
- queremos generar una Petición de Certificado (CSR) para enviarlo a una Autoridad de Certificación (CA).
- queremos importar claves de otros (por ejemplo, podemos verificar sus firmas) o una respuesta de certificado desde un CA.

Usamos la herramienta **keytool** para crear y manejar nuestro keystore.

Referencias

- Java 2 : manual de usuario y tutorial / Agustín Froufe Quintas.
  Froufe Quintas, Agustín
- Java 2 : the complete reference / Patrick Naughton, Herbert Schildt.
  Naughton, Patrick.
- Professional Java Server programming : J2EE edition / Subrahmanyam Allamaraju ... [et al.]. Allamaraju, Subrahmanyam, coaut.
- Java Servlets Second Edition / Karl Moss
- Advanced Java 2 : development for enterprise applications / Clifford J. Berg.
  Berg, Clifford J.