Seguridad en JAVA

Sergio Talens-Oliag <sto@iti.upv.es>

Instituto Tecnológico de Informática (ITI)

Diciembre 1999

Índice General

1 Introducción

Java es dos cosas: un lenguaje de programación orientado a objetos y una plataforma software en la que se ejecutan programas.

En este tema hablaremos de la seguridad en Java, centrándonos en los aspectos relacionados con la seguridad de la *plataforma Java*, aunque describiremos brevemente las características que hacen de Java un lenguaje de programación *seguro*.

Los componentes principales de la plataforma Java son:

- **Un entorno de ejecución proporcionado por la *Máquina Virtual Java* (Java VM o JVM)**. Este entorno de ejecución no es ni interpretado ni compilado; es un sistema híbrido. Los programas se compilan en un formato independiente de la máquina denominado *bytecode* y son interpretados por la Máquina Virtual, que es el único componente de Java que depende de la plataforma en la que trabajamos (y, por lo tanto, debe ser portada a cada arquitectura). La implementación de la JVM es la responsable de la seguridad incorporada en Java, por lo que es importante que su implementación sea adecuada.
- **Un conjunto de interfaces y arquitecturas** proporcionadas por la *Interfaz con el programador de aplicaciones Java* (API de Java). El API de Java es una colección de componentes software, agupados en bibliotecas o *paquetes* de componentes relacionados, que proporcionan multitud de capacidades útiles para el diseño de interfaces gáficos, acceso a redes, gestión de E/S, etc. En lo que respecta a la seguridad el API de Java nos proporciona paquetes que implementan o dan acceso a herramientas de seguridad de alto y bajo nivel como algoritmos de encriptación, firmas electrónicas, gestion de certificados, etc.

Estudiaremos la seguridad en el entorno Java desde varios puntos de vista:

- **Entorno de ejecución**. La seguridad de un sistema Java para los usuarios está en la máquina virtual, ya que esta es la que controla qué se puede ejecutar y de que modo, permitiendonos controlar qué pueden hacer los programas al ejecutarse en nuestro sistema.
- **Interfaces y arquitecturas de seguridad**. Para que las aplicaciones cliente/servidor sean seguras es necesario emplear técnicas criptográficas. Java proporciona una arquitectura en la que integrar estas técnicas y un conjunto de interfaces para simplificar su uso en el desarrollo de aplicaciones.

El tema se ha estructurado en cuatro bloques más o menos independientes.

En el primer bloque introduciremos la seguridad desde un punto de vista teórico, según los siguientes puntos:

- **Criptología**. En este apartado daremos una breve introducción a los campos de la criptografía y el criptoanálisis, presentando su terminología, las herramientas disponibles y sus aplicaciones.
- **Técnicas criptográficas**. En este punto estudiaremos con más detalle algunas de las herramientas presentadas anteriormente, indicando sus características principales: grado de seguridad, implementaciones, disponibilidad, etc.
- **Certificados digitales**. Descripción de qué son los certificados digitales, las entidades certificadoras y las infraestructuras de clave pública.

- **Protocolos de red seguros**. En este apartado describiremos los protocolo SSL y TLS.

En el segundo bloque nos centraremos en la seguridad del entorno de ejecución:

- **Seguridad en el entorno Java**. En este apartado hablaremos de la evolución de los mecanismos de seguridad en Java y explicaremos de que modo se gestionan en el JDK 1.2.

En el tercer bloque hablaremos de la arquitectura de seguridad Java y las APIs de seguridad:

- **Arquitectura Criptográfica**. Donde se describe la arquitectura de soporte de algoritmos criptográficos para el programador y las extensiones de seguridad.
- **Interfaces de seguridad**. En este apartado se describen en detalle las APIs de seguridad de Java y como emplearlas. Al igual que en el bloque anterior, este punto se ha separado del anterior para poder usarlo como referencia.

En el cuarto y último bloque introduciremos las últimas tecnologías de seguridad incorporadas a Java:

- **Extensión de Sockets Seguros de Java (JSSE)**. Implementación del SSL/TLS en Java.
- **Servicio de Autentificación y Autorización de Java (JAAS)**. Autentificación de usuarios en Java.

2 Criptología

Comenzaremos el estudio de la criptología dando algunas definiciones básicas, a continuación describiremos los sistemas criptográficos mencionando sus características más importantes, luego describiremos los distintos tipos de algoritmos criptográficos y terminaremos mencionando algunas de las aplicaciones de la criptografía.

Para aquellos interesados en profundizar en lo tratado en este punto se recomiendan las referencias 6 ([RSALab98]) y 7 ([SCHNEI96]).

2.1 Definiciones

Criptología

Es el estudio de la *criptografía* y el *criptoanálisis*.

Criptografía

Vista en términos sociales, es la ciencia de hacer que el coste de adquirir o alterar información de modo impropio sea mayor que el posible valor obtenido al hacerlo.

Vista en términos más formales, es la práctica y el estudio de técnicas de *encriptación* y *desencriptación* de información, es decir, de técnicas para codificar un mensaje haciéndolo ininteligible (*encriptación*) y recuperar el mensaje original a partir de esa versión ininteligible (*desencriptación*).

Algoritmo criptográfico

Es un método matemático que se emplea para *encriptar* y *desencriptar* un mensaje. Generalmente funciona empleando una o más *claves* (números o cadenas de caracteres) como parámetros del algoritmo, de modo que sean necesarias para recuperar el mensaje a partir de la versión cifrada.

El mensaje antes de encriptar se denomina *texto en claro* y una vez encriptado se denomina *texto cifrado*.

Sistema criptográfico

Es un sistema para *encriptar* y *desencriptar* información compuesto por un conjunto de *algoritmos criptográficos*, *claves* y, posiblemente, varios *textos en claro* con sus correspondientes versiones en *texto cifrado*.

Los sistemas criptográficos actuales se basan en tres tipos de algoritmos criptográficos: de clave secreta o simétricos, de clave pública o asimétricos y de resumen de mensajes (funciones de dispersión).

Algoritmos de resumen de mensajes

Transforman mensajes de tamaño variable a textos cifrados de tamaño fijo sin emplear claves. Se emplean para convertir mensajes grandes en representaciones más manejables.

Algoritmos de clave secreta o simétricos

Convierten un mensaje en un texto cifrado del mismo tamaño que el original. Emplean una sola clave para encriptar y desencriptar. Son los algoritmos empleados para transferir grandes cantidades de información de modo seguro.

Algoritmos de clave pública o asimétricos

Encriptan un mensaje generando un texto cifrado del mismo tamaño que el original. Usan una clave para encriptar el mensaje (clave privada) y otra para desencriptar (clave pública). Tienen un coste computacional alto y se suelen emplear para distribuir las claves de los algoritmos simétricos.

Criptoanálisis

Es el conjunto de procedimientos, procesos y métodos empleados para romper un *algoritmo criptográfico*, *desencriptar* un *texto cifrado* o descubrir las *claves* empleadas para generarlo.

2.2 Algoritmos de cifrado simétrico

Dentro de estos algoritmos distinguimos dos tipos de algoritmos en función de la cantidad de datos de entrada que manejan a la vez: algoritmos de cifrado por bloques y algoritmos de cifrado de flujo.

2.2.1 Cifrado por bloques

Los algoritmos de cifrado por bloques toman bloques de tamaño fijo del texto en claro y producen un bloque de tamaño fijo de texto cifrado, generalmente del mismo tamaño que la entrada. El tamaño del bloque debe ser lo suficientemente grande como para evitar *ataques de texto cifrado*. La asignación de bloques de entrada a bloques de salida debe ser uno a uno para hacer el proceso reversible y parecer aleatoria.

Para la asignación de bloques los algoritmos de cifrado simétrico realizan *sustituciones* y *permutaciones* en el texto en claro hasta obtener el texto cifrado.

La *sustitución* es el reemplazo de un valor de entrada por otro de los posibles valores de salida, en general, si usamos un tamaño de bloque k, el bloque de entrada puede ser sustituido por cualquiera de los 2^k bloques posibles.

La *permutación* es un tipo especial de sustitución en el que los bits de un bloque de entrada son reordenados para producir el bloque cifrado, de este modo se preservan las estadísticas del bloque de entrada (el número de unos y ceros).

Los algoritmos de cifrado por bloques *iterativos* funcionan aplicando en sucesivas rotaciones una transformación (*función de rotación*) a un bloque de texto en claro. La misma función es aplicada a los datos usando una subclave obtenida de la clave secreta proporcionada por el usuario. El número de rotaciones en un algoritmo de cifrado por bloques iterativo depende del nivel de seguridad deseado.

Un tipo especial de algoritmos de cifrado por bloques *iterativos* son los denominados *algoritmos de cifrado de Feistel*. En estos algoritmos el texto cifrado se obtiene del texto en claro aplicando repetidamente la misma transformación o función de rotación. El funcionamiento es como sigue: el texto a encriptar se divide en dos mitades, la función de rotación se aplica a una mitad usando una subclave y la salida de la función se emplea para hacer una o-exclusiva con la otra mitad, entonces se intercambian las mitades y se repite la misma operación hasta la última rotación, en la que no hay intercambio. Una característica interesante de estos algoritmos es que la encriptación y desencriptación son idénticas estructuralmente, aunque las subclaves empleadas en la encriptación se toman en orden inverso en la desencriptación.

Para aplicar un algoritmo por bloques es necesario descomponer el texto de entrada en bloques de tamaño fijo. Esto se puede hacer de varias maneras:

1. **ECB (Electronic Code Book)**. Se parte el mensaje en bloques de k bits, rellenando el ultimo si es necesario y se encripta cada bloque. Para desencriptar se trocea el texto cifrado en bloques de k bits y se desencripta cada bloque. Este sistema es vulnerable a ataques ya que dos bloques idénticos de la entrada generan el mismo bloque de salida. En la práctica no se utiliza.
2. **CBC (Cipher Block Chaining)**. Este método soluciona el problema del ECB haciendo una o-exclusiva de cada bloque de texto en claro con el bloque anterior cifrado antes de encriptar. Para el primer bloque se usa un *vector de inicialización*. Este es uno de los esquemas más empleados en la práctica.
3. **OFB (Output Feedback Mode)**. Este sistema emplea la *clave de la sesión* para crear un bloque pseudoaleatorio grande (*pad*) que se aplica en o-exclusiva al texto en claro para generar el texto cifrado. Este método tiene la ventaja de que el *pad* puede ser generado independientemente del texto en claro, lo que incrementa la velocidad de encriptación y desencriptación.
4. **CFB (Cipher Feedback Mode)**. Variante del método anterior para mensajes muy largos.

2.2.2 Cifrado de flujo de datos

Generalmente operan sobre 1 bit (o sobre bytes o palabras de 16 ó 32 bits) de los datos de entrada cada vez. El algoritmo genera una secuencia (*secuencia cifrante* o *keystream* en inglés) de bits que se emplea como clave. La encriptación se realiza combinando la secuencia cifrante con el texto en claro.

El paradigma de este tipo de algoritmos es el *One Time Pad*, que funciona aplicando una XOR (o-exclusiva) a cada bit de la entrada junto con otro generado aleatoriamente para obtener cada bit de la salida. La secuencia de bits aleatorios es la clave de la sesión, secuencia de cifrado o el *pad*, que es del mismo tamaño que la entrada y la salida. Para recuperar el texto original el texto cifrado debe pasar por el mismo proceso empleado para encriptar usando el mismo *pad*. Este algoritmo es conocido por ser el único incondicionalmente seguro, aunque, como las claves son del mismo tamaño que la entrada, es de poca utilidad práctica.

Los algoritmos de este tipo son intentos de conseguir algoritmos prácticos que se aproximen al funcionamiento del *one time pad*.

2.3 Algoritmos de clave pública o asimétricos

La criptografía de clave pública fue inventada en 1975 por *Whitfield Diffie* y *Matin Hellman*. Se basa en emplear un par de claves distintas, una *pública* y otra *privada*. La idea fundamental es que las claves están ligadas matemáticamente pero es computacionalmente imposible obtener una a partir de la otra.

2.3.1 Fundamentos matemáticos

Las *funciones de una sola dirección* son aquellas en las que obtener el resultado en una dirección es fácil, pero en la otra es casi imposible. Los algoritmos criptográficos de clave pública se basan en *funciones de una sola dirección con puerta trasera*, que son aquellos en los que el problema es resoluble en la dirección opuesta (la que antes era muy difícil) empleando una ayuda (la *puerta trasera*).

Los siguientes problemas matemáticos son considerados como *funciones de una sola dirección con puerta trasera* y son la base de la mayoría de algoritmos de clave pública actuales:

- **Factorización de enteros**. Un número entero siempre se puede representar como un producto de números primos denominados *factores primos*. La factorización de enteros consiste en encontrar los factores primos de un número. Los algoritmos criptográficos basados en este problema aprovechan el hecho de que la multiplicación de números primos grandes es computacionalmente sencilla pero la factorización un número grande en sus factores primos es muy cara computacionalmente.
- **Logaritmos discretos**. En *aritmética módulo n* dos enteros son equivalentes si tienen el mismo resto cuando son divididos por n. El *resto* de la división m/n es el menor entero no negativo que difiere de m por un múltiplo de n. La *exponenciación discreta* ($a^x \bmod n$) es la exponenciación en *aritmética módulo n*. Por ejemplo, $3^4 \bmod 10$ es $81 \bmod 10$, que es equivalente a $1 \bmod 10$. El *logaritmo discreto* es la operación inversa a la *exponenciación discreta*, el problema es encontrar la x tal que $a^x = b \bmod n$. Por ejemplo, si $11^x = 1 \bmod 10$, entonces $x = 2$. Los algoritmos que emplean este tipo de problema se basan en que la *exponenciación módulo n* es un problema fácil y hallar el *logaritmo discreto* es un problema difícil.
- **Logaritmos discretos de curva elíptica**. Es una variación del problema anterior más cara computacionalmente, lo que permite usar claves más pequeñas que mejoran las prestaciones de los algoritmos y reducen el tamaño de los textos cifrados.

2.3.2 Tipos de algoritmo

En función de su relación matemática distinguimos varios tipos de algoritmo:

- **Reversible**. Es aquel en el que un mensaje encriptado con la clave privada puede ser desencriptado usando la clave pública y viceversa (uno encriptado usando la clave pública puede ser desencriptado usando la privada).
- **Irreversible**. Es aquel en el que un mensaje encriptado usando la clave privada puede ser desencriptado con la clave pública pero la clave privada no desencripta los mensajes cifrados usando la clave pública.
- **De intercambio de claves**. Sólo permiten negociar de forma segura una clave secreta entre dos partes. Hay que indicar que los algoritmos *reversibles* también se pueden emplear para esta función, pero los *irreversibles* no.

2.3.3 Aplicaciones

Este tipo de algoritmos tienen dos aplicaciones fundamentales:

1. **Encriptación**. Si un usuario A quiere mandar un mensaje a otro usuario B, lo encripta usando la clave pública de B. Cuando B lo recibe lo desencripta usando su clave privada. Si alguien intercepta el mensaje no puede descifrarlo, ya que no conoce la clave privada de B (de hecho, ni tan siquiera A es capaz de desencriptar el mensaje).

2. **Firmas digitales**. Si B encripta un mensaje usando su clave privada cualquiera que tenga su clave pública podrá obtener el texto en claro correspondiente; si alguien quiere hacerse pasar por B tendrá que cifrar el mensaje usando la misma *clave privada* o no se descifrará correctamente con la *clave pública* de B. Lo que B ha hecho es **firmar digitalmente** el mensaje. El proceso de desencriptar con una clave pública un mensaje firmado se denomina **verificación de firma**.

Estos algoritmos son mucho más caros que los de clave secreta, por lo que no se usan para encriptar mucha información. Su principal aplicación está en la fase inicial de una comunicación, ya que permiten que los dos extremos se autentifiquen e intercambien claves secretas para encriptar con un algoritmo simétrico.

El problema fundamental de este tipo de algoritmos es la distribución de las claves; aunque la clave pública se puede distribuir libremente (A la puede enviar por correo o decírsela a B por teléfono), nos queda el problema de la suplantación (C le puede dar su clave pública a B haciéndose pasar por A). Para solventar estos problemas se emplean autoridades certificadoras y certificados digitales, que discutiremos más adelante.

2.4 Algoritmos de resumen de mensajes

Un algoritmo de *resumen de mensajes* o *función de dispersión criptográfica* es aquel que toma como entrada un mensaje de longitud variable y produce un resumen de longitud fija. En inglés el resumen se llama *message digest*, *digest* o *hash* y el algoritmo *message digest algorithm* o *one way hash algorithm*.

Estos algoritmos deben tener tres propiedades para ser criptográficamente seguros:

1. No debe ser posible averiguar el mensaje de entrada basándose sólo en su resumen, es decir, el algoritmo es una función irreversible de una sola dirección.
2. Dado un resumen debe ser imposible encontrar un mensaje que lo genere.
3. Debe ser computacionalmente imposible encontrar dos mensajes que generen el mismo resumen.

Los algoritmos de este tipo se emplean en la generación de *códigos de autentificación de mensajes* y en las *firmas digitales*.

2.5 Códigos de autentificación de mensajes y firmas digitales

Un **código de autentificación de mensaje** (*message authentication code* o *MAC*) es un bloque de datos de tamaño fijo que se envía con un mensaje para averiguar su origen e integridad. Son muy útiles para proporcionar autentificación e integridad sin confidencialidad. Para generar MACs se pueden usar algoritmos de clave secreta, de clave pública y algoritmos de resumen de mensajes.

Un tipo de MAC muy empleado en la actualidad es el **código de autentificación de mensaje resumido** (*hashed message authentication code* o *HMAC*). Lo que hacemos es generar el MAC aplicando una función de dispersión criptográfica a un conjunto formado por un mensaje y un código secreto. Así, el que recibe el mensaje puede calcular su propio MAC con el mensaje y el código secreto (que comparte con el que ha generado el MAC). Si no coinciden sabemos que el mensaje ha sido manipulado. Este tipo de técnicas se emplean para proteger comunicaciones a nivel de la capa de red.

La **firma digital** es un item que responde del origen e integridad de un mensaje. El que escribe un mensaje lo firma usando una *clave de firmado* y manda el mensaje y la firma digital. El destinatario usa una *clave de verificación* para comprobar el origen del mensaje y que no ha sido modificado durante el tránsito.

Para firmar los mensajes se emplean algoritmos de clave pública y funciones de dispersión. El proceso es como sigue:

1. El emisor genera un resumen del mensaje, lo encripta con su clave privada (*clave de firmado*) y envía el mensaje y el texto cifrado que corresponde al resumen del mensaje.

2. El destinatario genera un resumen del mensaje que recibe y desencripta el resumen cifrado que lo acompañaba usando la clave pública del emisor (*clave de verificación*).>Si al comparar los resúmenes ambos son iguales el mensaje es válido y ha sido firmado por el emisor real, ya que de otro modo no se hubiera podido desencriptar correctamente con su clave pública.

Hay que indicar que los MAC y las firmas digitales se diferencian en un punto importante: aunque los MAC se pueden usar para verificar la autenticidad de los mensajes, no se pueden usar para firmar los mensajes, ya que sólo se usa una clave secreta que comparten el emisor y el receptor, lo que hace que ambos puedan generar la misma firma.

2.6 Seguridad de los sistemas criptográficos

La seguridad de un sistema criptográfico depende generalmente de que al menos una de las claves empleadas sea secreta, más que de que el algoritmo de encriptación sea secreto.

El publicar los algoritmos empleados por un sistema criptográfico para que sean revisados públicamente es una buena práctica que permite que se mejoren algoritmos no totalmente seguros o se considere que un algoritmo no tiene debilidades.

Los algoritmos criptográficos tienen distintos grados de seguridad:

1. **Seguro computacionalmente**. Con suficiente poder de cálculo y almacenamiento el sistema puede ser roto, pero a un coste tan elevado que no es práctico. De cualquier modo, el coste computacional para considerar que un algoritmo es seguro ha ido cambiando con el paso del tiempo; algoritmos antes considerados seguros, como el DES, han sido rotos en meses con sistemas distribuidos y en días con sistemas diseñados específicamente para la tarea, como se describe en `[EFF98]`.

2. **Seguro incondicionalmente**. Son aquellos en los que aun disponiendo de recursos y gran cantidad de texto cifrado no es posible romper el algoritmo. Los únicos sistemas incondicionalmente seguros son los *One Time Pads*.

Un sistema criptográfico puede ser roto en varios niveles:

1. **Deducción de información**. Se obtiene parte de información de la clave o del texto en claro.

2. **Deducción de una instancia**. Se obtiene el texto en claro a partir de un texto cifrado.

3. **Deducción global**. A partir de la deducción de una instancia se obtiene un algoritmo que obtiene los mismos resultados que el algoritmo original.

4. **Rotura total**. Se recupera la clave y se puede descifrar cualquier mensaje encriptado con la misma clave.

Para romper un algoritmo se pueden emplear distintos tipos de ataque criptoanalítico:

1. **Ataque de sólo texto cifrado**. El analista dispone de un texto cifrado y quiere obtener el texto en claro o la clave. Se pueden usar métodos de *fuerza bruta* (probando todas las claves posibles hasta que obtenemos un mensaje con sentido) o basados en *diccionario* (probando únicamente con un subconjunto de las claves posibles, por ejemplo si las claves son palabras). Es importante disponer de suficiente texto en clave para que sea fácil identificar cual es el texto en claro correcto.

2. **Ataque de texto en claro conocido**. El analista dispone de un texto en claro y su correspondiente texto cifrado, lo que permite reducir el espacio de búsqueda de claves u obtener estadísticas que pueden usarse para hacer deducciones en otros textos cifrados.

3. **Ataque de texto en claro conocido adaptativo**. Es igual que el anterior pero el analista puede elegir nuevos textos dinámicamente y alterar sus elecciones en función de los resultados que va obteniendo.

4. **Ataque de texto en claro elegido**. El analista puede elegir el texto en claro y obtener el texto cifrado correspondiente. Este tipo de ataque puede evitar duplicados y centrarse más en las debilidades del algoritmo.

5. **Ataque de texto en claro elegido adaptativo**. Es la versión adaptativa del ataque anterior.

Para que un sistema criptográfico sea considerado como fuerte debe tener las siguientes características:

- Debe disponer de un número muy elevado de claves posibles, de modo que sea poco razonable intentar descifrar un mensaje por el método de la fuerza bruta (probando todas las claves).
- Debe producir texto cifrado que parezca aleatorio a un test estadístico estándar.
- Debe resistir todos los métodos conocidos de romper los códigos, es decir, debe ser resistente al criptoanálisis.

2.7 Ataques más importantes sobre algoritmos criptográficos

En este apartado mencionaremos los ataques más importantes contra los algoritmos criptográficos clasificados por tipo de algoritmo.

2.7.1 Algoritmos de cifrado simétrico por bloques

Criptoanálisis diferencial. Se realizan sobre algoritmos de cifrado por bloques iterativos. Es un ataque de texto claro elegido que se basa en el análisis de la evolución de las diferencias de dos textos en claro relacionados cuando son encriptados con la misma clave. Mediante el análisis de los datos disponibles se pueden asignar probabilidades a cada una de las claves posibles . Eventualmente, la clave más probable puede ser identificada como la correcta.

Criptoanálisis lineal. Este es un ataque de texto en claro conocido que usa una aproximación lineal para describir el funcionamiento del algoritmo. Dados suficientes pares de texto en claro y cifrado se pueden obtener datos sobre la clave.

Explotación de claves débiles. Hay algoritmos para los que se pueden encontrar claves que se comportan de modo especial, por ejemplo dando origen a ciertas regularidades en la encriptación o un bajo nivel de encriptación. Si el número de claves débiles es pequeño no tiene importancia, pero si el algoritmo tiene muchas de estas claves es fácil que se vea comprometido.

Ataques algebraicos. Son una clase de técnicas que basan su éxito en que los algoritmos criptográficos muestren un alto grado de estructura matemática. Por ejemplo, si un algoritmo tiene estructura de grupo, al encriptar con una clave, y luego volver a encriptar con otra obtenemos un texto cifrado que podría haber sido generado con el mismo algoritmo y una sola clave, lo que hace al algoritmo bastante débil.

2.7.2 Algoritmos de cifrado simétrico de flujo de datos

Los principales ataques a este tipo de algoritmos buscan debilidades en la estructura del mismo que le permitan descubrir partes de la secuencia de cifrado. Una de las características fundamentales es el periodo de la clave de cifrado, ya que si es muy corto y se descubre una parte de la clave se puede emplear en sucesivos periodos del algoritmo.

Complejidad lineal. Una técnica empleada para atacar estos algoritmos es el uso de un *registro de desplazamiento lineal con realimentación* (*linear feedback shift register*) para replicar parte de una secuencia. A partir de esta técnica aparece la *complejidad lineal* de una secuencia, que será el tamaño del registro que necesitemos para replicarla.

Ataques de correlación. Otros ataques intentan recuperar parte de una secuencia de cifrado ya empleada. Dentro de estos ataques hay una clase que podemos denominar *divide y vencerás* que consiste en encontrar algún fragmento característico de la secuencia de cifrado y atacarla con un método de fuerza bruta y se comparar las secuencias generadas con la secuencia de cifrado real. Este método lleva a lo que se denomina *ataques de correlación* y *ataques de correlación rápidos*.

2.7.3 Algoritmos de resumen de mensajes

Las funciones de dispersión deben tener dos propiedades para ser útiles en criptografía: deben ser funciones de una sola dirección y no tener colisiones. El ataque por fuerza bruta consiste en seleccionar entradas del algoritmo aleatoriamente y buscar una que nos de el valor que buscamos (la función no es de una sola dirección) o un par de entradas que generen la misma salida (la función tiene colisiones).

Ataque del cumpleaños. Se trata de una clase de ataques por fuerza bruta. El nombre viene de la *paradoja del cumpleaños*: la probabilidad de que dos o más personas en un grupo de 23 personas cumplan años el mismo día es superior a 1/2.

Si una función retorna uno de k valores equiprobables cuando se le proporciona una entrada aleatoria, cuando le proporcionamos repetidamente valores de entrada distintos, obtendremos dos salidas iguales después de $1.2k^{1/2}$ ejecuciones. Si buscamos una colisión en una función de dispersión, por la paradoja del cumpleaños sabemos que después de probar $1.2 * 2^{pi/2}$ entradas tendremos alguna.

Pseudo-colisiones. Otro problema de estos algoritmos son las *pseudo-colisiones*, que son las colisiones producidas en la función de compresión empleada en el proceso iterativo de una función de dispersión. En principio que haya pseudo-colisiones no implica que el algoritmo no se seguro.

2.8 Aplicaciones de la criptografía

La criptografía es una disciplina con multitud de aplicaciones, muchas de las cuales están en uso hoy en día. Entre las más importantes destacamos las siguientes:

- **Seguridad de las comunicaciones**. Es la principal aplicación de la criptografía a las redes de computadores, ya que permiten establecer canales seguros sobre redes que no lo son. Además, con la potencia de cálculo actual y empleando algoritmos de cifrado simétrico (que se intercambian usando algoritmos de clave pública) se consigue la privacidad sin perder velocidad en la transferencia.
- **Identificación y autentificación**. Gracias al uso de firmas digitales y otras técnicas criptográficas es posible identificar a un individuo o validar el acceso a un recurso en un entorno de red con más garantías que con los sistemas de usuario y clave tradicionales.
- **Certificación**. La certificación es un esquema mediante el cual agentes fiables (como una entidad certificadora) validan la identidad de agentes desconocidos (como usuarios reales). El sistema de

certificación es la extensión lógica del uso de la criptografía para identificar y autentificar cuando se emplea a gran escala.

- **Comercio electrónico**. Gracias al empleo de canales seguros y a los mecanismos de identificación se posibilita el comercio electrónico, ya que tanto las empresas como los usuarios tienen garantías de que las operaciones no pueden ser espiadas, reduciéndose el riesgo de fraudes y robos.

3 Técnicas criptográficas

En este apartado presentaremos los principales algoritmos criptográficos empleados hoy en día. Una discusión ampliada de muchas de las técnicas mencionadas aquí se puede encontrar en las referencia `7 ([SCHNEI96])`.

3.1 Algoritmos de cifrado simétrico

3.1.1 DES

El DES (*Data Encription Standard* o *Estándar de Encriptación de Datos*) es el nombre del documento FIPS (Federal Information Processing Standard) 46-1 del Instituto Nacional de Estándares y Tecnología (NIST) del Departamento de Comercio de Estados Unidos. Fue publicado en 1977. En este documento se describe el DEA (*Data Encription Algorithm* o *Algoritmo de Encriptación de Datos*. Es el algoritmo de cifrado simétrico más estudiado, mejor conocido y más empleado del mundo.

El DEA (llamado con frecuencia DES) es un algoritmo de cifrado por bloques de 64 bits de tamaño. Emplea una clave de 56 bits durante la ejecución (se eliminan 8 bits de paridad del bloque de 64). El algoritmo fue diseñado para ser implementado en hardware. Cuando se utiliza en comunicaciones ambos participantes deben conocer la clave secreta (para intercambiarla se suelen emplear algoritmos de clave pública). El algoritmo se puede usar para encriptar y desencriptar mensajes, generar y verificar códigos de autentificación de mensajes (MAC) y para encriptación de un sólo usuario (p. ej para guardar un archivo en disco).

Aunque el DES era un algoritmo computacionalmente seguro, esto ha dejado de ser cierto, ya que con hardware específico es posible realizar ataques por fuerza bruta que descubran una clave en pocos días (ver referencia `1 ([EFF98])`). El problema principal es que el tamaño de la clave (56 bits) es demasiado pequeño para la potencia de cálculo actual. De hecho, el DES dejó de ser el algoritmo empleado por el gobierno norteamericano en Noviembre de 1998 y de momento (hasta que el AES sea elegido), emplean el Triple DES.

3.1.2 Triple-DES

Consiste en encriptar tres veces una clave DES. Esto se puede hacer de varias maneras:

- DES-EEE3: Tres encriptaciones DES con tres claves distintas.
- DES-EDE3: Tres operaciones DES con la secuencia encriptar-desencriptar-encriptar con tres claves diferentes.
- DES-EEE2 y DES-EDE2: Igual que los anteriores pero la primera y tercera operación emplean la misma clave.

Dependiendo del método elegido, el grado de seguridad varía; el método más seguro es el DES-EEE3.

3.1.3 AES

El AES (*Advanced Encription Standard* o *Estándar Criptográfico Avanzado*) es un algoritmo de cifrado por bloques destinado a reemplazar al DES como estándar.

En la actualidad se han aceptado 15 propuestas de estándar de las que saldrán 5 candidatos para una revisión más completa. El proceso no parece que vaya a terminar hasta pasado el año 2000.

Se puede encontrar más información en el URL `<http://csrc.nist.gov/encryption/aes/aes_home.htm>`

3.1.4 RC2

El RC2 es un algoritmo de cifrado por bloques de clave de tamaño variable diseñado por Ron Rivest de RSA Data Security (la RC quiere decir *Ron's Code* o *Rivest's Cipher*).

El algoritmo trabaja con bloques de 64 bits y entre dos y tres veces más rápido que el DES en software. Se puede hacer más o menos seguro que el DES contra algoritmos de fuerza bruta eligiendo el tamaño de clave apropiadamente.

El algoritmo está diseñado para reemplazar al DES.

3.1.5 RC4

El RC4 es un algoritmo de cifrado de flujo diseñado por Ron Rivest para RSA Data Security. Es un algoritmo de tamaño de clave variable con operaciones a nivel de byte. Se basa en el uso de una permutación aleatoria y tiene un periodo estimado de más de 10^{100}. Además, es un algoritmo de ejecución rápida en software.

El algoritmo se emplea para encriptación de ficheros y para encriptar la comunicación en protocolos como el SSL (TLS).

3.1.6 RC5

El RC5 es un algoritmo parametrizable con tamaño de bloque variable, tamaño de clave variable y número de rotaciones variable. Los valores más comunes de los parámetros son 64 o 128 bits para el tamaño de bloque, de 0 a 255 rotaciones y claves de 0 a 2048 bits. Fue diseñado en 1994 por Ron Rivest.

El RC5 tiene 3 rutinas: expansión de la clave, encriptación y desencriptación. En la primera rutina la clave proporcionada por el usuario se expande para llenar una tabla de claves cuyo tamaño depende del número de rotaciones. La tabla se emplea en la encriptación y desencriptación. Para la encriptación sólo se emplean tres operaciones: suma de enteros, o-exclusiva de bits y rotación de variables.

La mezcla de rotaciones dependientes de los datos y de distintas operaciones lo hace resistente al criptoanálisis lineal y diferencial. El algoritmo RC5 es fácil de implementar y analizar y, de momento, se considera que es seguro.

3.1.7 IDEA

El IDEA (*International Data Encription Algorithm*) es un algoritmo de cifrado por bloques de 64 bits iterativo. La clave es de 128 bits. La encriptación precisa 8 rotaciones complejas. El algoritmo funciona de la misma forma para encriptar que para desencriptar (excepto en el cálculo de las subclaves). El algoritmo es fácilmente implementable en hardware y software, aunque algunas de las operaciones que realiza no son eficientes en software, por lo que su eficiencia es similar a la del DES.

El algoritmo es considerado inmune al criptoanálisis diferencial y no se conocen ataques por criptoanálisis lineal ni debilidades algebraicas. La única debilidad conocida es un conjunto de 251 claves débiles, pero dado que el algoritmo tiene 2^{128} claves posibles no se considera un problema serio.

3.1.8 SAFER

El SAFER (*Secure And Fast Encription Routine*) es un algoritmo de cifrado por bloques no propietario. Está orientado a bytes y emplea un tamaño de bloque de 64 bits y claves de 64 (SAFER K-64) o 128 bits (SAFER K-128). Tiene un número variable de rotaciones, pero es recomendable emplear como mínimo 6.

El algoritmo original fue considerado inmune al criptoanálisis lineal y diferencial, pero Knudsen descubrió una debilidad en el generador de claves y el algoritmo fue modificado (SAFER SK-64 y SAFER SK-128).

3.1.9 Blowfish

Es un algoritmo de cifrado por bloques de 64 bits desarrollado por Scheiner. Es un algoritmo de tipo Feistel y cada rotación consiste en una permutación que depende de la clave y una sustitución que depende de la clave y los datos. Todas las operaciones se basan en o-exclusivas sobre palabras de 32 bits. La clave tiene tamaño variable (con un máximo de 448 bits) y se emplea para generar varios vectores de subclaves.

Este algoritmo se diseño para máquinas de 32 bits y es considerablemente más rápido que el DES.

El algoritmo es considerado seguro aunque se han descubierto algunas claves débiles, un ataque contra una versión del algoritmo con tres rotaciones y un ataque diferencial contra una variante del algoritmo.

3.2 Algoritmos de clave pública

3.2.1 RSA

El RSA, llamado así por las siglas de sus creadores (*Rivest, Shamir y Adelman*), es el algoritmo de clave pública más popular. El algoritmo se puede usar para encriptar comunicaciones, firmas digitales e intercambio de claves.

La clave es de tamaño variable, generalmente se usan claves entre 512 y 2048 bits. Las claves más grandes aumentan la seguridad del algoritmo pero disminuyen su eficiencia y generan más texto cifrado. Los bloques de texto en claro pueden ser de cualquier tamaño, siempre que sea menor que la longitud de la clave. Los bloques de texto cifrado generados son del tamaño de la clave.

La *clave pública* del algoritmo tiene la forma (e, n), donde e es el exponente y n el módulo. La longitud de la clave es igual al número de bits de n. El *módulo* se obtiene multiplicando dos números primos grandes, p y q. Los números se seleccionan aleatoriamente y se guardan en secreto. La *clave privada* tiene la forma (d, n), donde d es el producto inverso de $e \,\texttt{modulo}\, (p-1)(q-1)$ (es decir, $(ed-1)$ es divisible por $(p-1)(q-1)$).

El cálculo de d a partir de p y q es sencillo, pero es computacionalmente imposible calcular d sin conocer p y q para valores grandes de n, ya que obtener sus valores es equivalente a factorizar n, que es un problema intratable computacionalmente.

El funcionamiento del algoritmo es como sigue:

- **Encriptación**.Para encriptar un mensaje un usuario calcula $c = m^e \,\texttt{modulo}\, n$, donde m es el texto en claro, c es el texto cifrado y (e, n) es la clave pública del destinatario.
- **Desencriptación**. Para desencriptar el mensaje el destinatario calcula $c^d \,\texttt{modulo}\, n = (m^e)^d \,\texttt{modulo}\, n = m^{ed} \,\texttt{modulo}\, n = m$, donde (d, n) es la clave privada del destinatario. Hay que

indicar que la última sustitución es posible por el modo en que hemos escogido los números, ya que d es el producto inverso de $e \,\texttt{modulo}\, n$, por lo que $m^{ed} = m$.

- **Firmado**. Si el emisor desea enviar el mensaje firmado usa su clave privada para calcular $c = m^d \,\texttt{modulo}\, n$ y el destinatario lo valida calculando $c^e \,\texttt{modulo}\, n = (m^d)^e \,\texttt{modulo}\, n = m^{de} \,\texttt{modulo}\, n = m$, donde (e, n) es la clave pública del emisor.

El algoritmo es lento, ya que emplea operaciones matemáticas que tienen un coste elevado y trabaja con claves de gran tamaño. Parte del problema está en la elección del exponente e, ya que un exponente de 512 bits escogido aleatoriamente precisa 768 multiplicaciones en promedio. Para solucionarlo se suelen escoger los valores 3 ó 65537, que precisan 3 y 17 multiplicaciones respectivamente. La elección de un exponente fijo no disminuye la seguridad del algoritmo si se emplean esquemas de criptografía de clave pública adecuados, como por ejemplo el relleno de mensajes con bits aleatorios.

Adicionalmente, el uso de exponentes fijos hace que la encriptación sea más rápida que la desencriptación y la verificación más rápida que la firma. Esta última característica es incluso deseable, ya que un usuario firma una vez un mensaje pero es posible que la firma se valide muchas veces.

Comparado con los sistemas de cifrado simétrico como el DES, el algoritmo de RSA es 100 veces más lento en software y de 1000 a 10000 veces más lento en hardware.

3.2.2 Diffie-Hellman

El algoritmo de Diffie Hellman es un algoritmo de clave pública que permite el intercambio seguro de un secreto compartido. Generalmente se emplea junto con algoritmos de cifrado simétrico, como método para acordar una clave secreta. El algoritmo no se puede usar para encriptar conversaciones o firmas digitales.

El funcionamiento del algoritmo es como sigue:

- El emisor escoge un número primo grande p y un generador g ($g<p$) y se los envía al destinatario. A continuación escoge un número grande d_A como clave privada y calcula la clave pública correspondiente $e_A = g^{d_A} \,\texttt{modulo}\, p$.
- De modo similar, el destinatario escoge una clave privada d_B y una clave privada $e_B = g^{d_B} \,\texttt{modulo}\, p$.
- Ambos participantes intercambian sus claves públicas y calculan un secreto compartido.

  El del emisor será $s_A = e_B^{d_A} = (g^{d_B})^{d_A} = g^{d_B d_A} \,\texttt{modulo}\, p$.

  Y el del destinatario $s_B = e_A^{d_B} = (g^{d_A})^{d_B} = g^{d_A d_B} = g^{d_B d_A} \,\texttt{modulo}\, p$.

Con este sistema, aunque un tercero interceptara los números p y g y las claves públicas e_A y e_B, no podría calcular el secreto compartido sin tener una de las claves privadas, lo que equivale a calcular el logaritmo discreto de una de las claves públicas, que es un problema intratable computacionalmente.

El problema fundamental de este algoritmo es que es sensible a ataques activos del tipo *hombre en el medio*. Si la comunicación es interceptada por un tercero, este se puede hacer pasar por el emisor cara al destinatario y viceversa, ya que no disponemos de ningún mecanismo para validar la identidad de los participantes en la comunicación. Así, el *hombre en el medio* podría acordar una clave con cada participante y retransmitir los datos entre ellos, escuchando la conversación en ambos sentidos.

3.3 Funciones de dispersión

3.3.1 SHA y SHA-1

El SHA (*Secure Hash Algorithm*) es un algoritmo de resumen seguro desarrollado por el NIST. El SHA-1 es una versión corregida del algoritmo publicada en 1994. El algoritmo es un estándar ANSI.

El algoritmo toma un mensaje de menos de 2^{64} bits y genera un resumen de 160 bits. Es más lento que el MD5, pero la mayor longitud de clave lo hace más resistente a ataques de colisión por fuerza bruta y de inversión.

3.3.2 MD2, MD4 y MD5

Los tres son algoritmos de resumen de mensajes (el MD viene de *Message Digest*) desarrollados por Rivest.

Los tres toman un mensaje de longitud arbitraria y generan un resumen de 128 bits. El MD2 está optimizado para máquinas de 8 bits, mientras que el MD4 y MD5 son para arquitecturas de 32 bits. El código para los tres algoritmos se puede encontrar en los RFCs 1319, 1320 y 1321.

El MD2 funciona rellenando el mensaje para que tenga una longitud en bytes múltiplo de 16. Sobre ese mensaje se calcula un checksum de 16 bytes que se añade al mensaje y la función de dispersión se aplica al mensaje resultante. El único problema que se le conoce es que si se omite el checksum se pueden obtener colisiones.

El MD4 fue desarrollado en 1990 por Rivest. El mensaje se rellena para que su longitud en bits más 448 sea divisible por 512. Una representación de la longitud del mensaje de 64 bits se concatena entonces con el mensaje. El mensaje se procesa iterativamente en bloques de 512 bits y cada bloque es procesado en tres rotaciones distintas. El algoritmo ha sido criptoanalizado y se han encontrado debilidades, de hecho es posible encontrar colisiones en menos de un minuto en máquinas modernas, por lo que el algoritmo se considera a todos los efectos roto.

El MD5 fue desarrollado en 1991 por Rivest. Es básicamente el MD4 con mejoras en la seguridad, aunque es más lento que este. El tamaño del resumen y la necesidad del relleno son iguales que en el MD4. Consta de cuatro rotaciones que tienen un diseño ligeramente diferente a las del MD4. El algoritmo ha sido criptoanalizado con técnicas similares a las del MD4 y se han encontrado pseudo-colisiones en la función de compresión, pero no en el algoritmo completo. Adicionalmente, se ha estimado que es posible construir una máquina capaz de atacar el algoritmo por fuerza bruta y encontrar una colisión en 24 días, aunque el coste de la máquina era de 10 millones de dolares en 1994.

3.4 Firmas digitales

3.4.1 DSA y DSS

El DSA (*Digital Signature Algorithm* o *Algoritmo Estándar de Firmado*) es el algoritmo de firmado digital incluido en el DSS (*Digital Signature Standard* o *Estándar de Firmas Digitales*) del NIST Norteamericano. Se publicó en 1994.

El DSA está basado en el problema de los logaritmos discretos y sólo puede emplearse para las firmas digitales (a diferencia del RSA, que también puede emplearse para encriptar). La elección de este algoritmo como estándar de firmado generó multitud de críticas: se pierde flexibilidad respecto al RSA (que además, ya era un estándar *de hecho*), la verificación de firmas es lenta, el proceso de elección fue poco claro y la versión original empleaba claves que lo hacían poco seguro.

El algoritmo es más rápido para generar la firma que para validarla, al revés de lo que sucede con el RSA. Emplea claves de 1024 bits (originalmente eran 512 bits, pero se aumento por falta de seguridad). No se

conocen ataques eficientes contra este algoritmo, sólo existen problemas con un conjunto de números primos, pero son fácilmente evitables si se siguen los sistemas adecuados de generación de claves.

4 Certificados digitales

Los certificados digitales son el equivalente digital del DNI, en lo que a la autentificación de individuos se refiere, ya que permiten que un individuo demuestre que es quien dice ser, es decir, que está en posesión de la clave secreta asociada a su certificado.

Para los usuarios proporcionan un mecanismo para verificar la autenticidad de programas y documentos obtenidos a través de la red, el envío de correo encriptado y/o firmado digitalmente, el control de acceso a recursos, etc.

En este apartado explicaremos qué son los certificados digitales, cuales son los formatos estándar, como podemos controlar sus periodos de validez o anularlos si se ven comprometidos, quien los genera y las infraestructuras necesarias para soportarlos.

Un libro introductorio en el que se discuten la mayor parte de los temas tratados en este apartado es la referencia 2 (`[FEGHHI99]`).

4.1 Certificados Digitales

Un *certificado de clave pública* es un punto de unión entre la clave pública de una entidad y uno o más atributos referidos a su identidad. El certificado garantiza que la clave pública pertenece a la entidad identificada y que la entidad posee la correspondiente clave privada.

Los *certificados de clave pública* se denominan comúnmente *Certificado Digital*, *ID Digital* o simplemente *certificado*. La entidad identificada se denomina *sujeto del certificado* o *subscriptor* (si es una entidad legal como, por ejemplo, una persona).

Los certificados digitales sólo son útiles si existe alguna *Autoridad Certificadora* (*Certification Authority* o *CA*) que los valide, ya que si uno se certifica a sí mismo no hay ninguna garantía de que su identidad sea la que anuncia, y por lo tanto, no debe ser aceptada por un tercero que no lo conozca.

Es importante ser capaz de verificar que una autoridad certificadora ha emitido un certificado y detectar si un certificado no es válido. Para evitar la falsificación de certificados, la entidad certificadora después de autentificar la identidad de un sujeto, firma el certificado digitalmente.

Los *certificados digitales* proporcionan un mecanismo criptográfico para implementar la autentificación; también proporcionan un mecanismo seguro y escalable para distribuir claves públicas en comunidades grandes.

4.2 Certificados X.509

El formato de certificados X.509 es un estándar del ITU-T (*International Telecommunication Union-Telecommunication Standarization Sector*) y el ISO/IEC (*International Standards Organization / International Electrotechnical Commission*) que se publicó por primera vez en 1988. El formato de la versión 1 fue extendido en 1993 para incluir dos nuevos campos que permiten soportar el control de acceso a directorios. Después de emplear el X.509 v2 para intentar desarrollar un estándar de correo electrónico seguro, el formato fue revisado para permitir la extensión con campos adicionales, dando lugar al X.509 v3, publicado en 1996.

Los elementos del formato de un certificado X.509 v3 son:

- **Versión**. El campo de versión contiene el número de versión del certificado codificado. Los valores aceptables son 1, 2 y 3.
- **Número de serie del certificado**. Este campo es un entero asignado por la autoridad certificadora. Cada certificado emitido por una CA debe tener un número de serie único.
- **Identificador del algoritmo de firmado**. Este campo identifica el algoritmo empleado para firmar el certificado (como por ejemplo el RSA o el DSA).
- **Nombre del emisor**. Este campo identifica la CA que ha firmado y emitido el certificado.
- **Periodo de validez**. Este campo indica el periodo de tiempo durante el cual el certificado es válido y la CA está obligada a mantener información sobre el estado del mismo. El campo consiste en una fecha inicial, la fecha en la que el certificado empieza a ser válido y la fecha después de la cual el certificado deja de serlo.
- **Nombre del sujeto**. Este campo identifica la identidad cuya clave pública está certificada en el campo siguiente. El nombre debe ser único para cada entidad certificada por una CA dada, aunque puede emitir más de un certificado con el mismo nombre si es para la misma entidad.
- **Información de clave pública del sujeto**. Este campo contiene la clave pública, sus parámetros y el identificador del algoritmo con el que se emplea la clave.
- **Identificador único del emisor**. Este es un campo opcional que permite reutilizar nombres de emisor.
- **Identificador único del sujeto**. Este es un campo opcional que permite reutilizar nombres de sujeto.
- **Extensiones**.

Las extensiones del X.509 v3 proporcionan una manera de asociar información adicional a sujetos, claves públicas, etc. Un campo de extensión tiene tres partes:

1. **Tipo de extensión**. Es un identificador de objeto que proporciona la semántica y el tipo de información (cadena de texto, fecha u otra estructura de datos) para un valor de extensión.
2. **Valor de la extensión**. Este subcampo contiene el valor actual del campo.
3. **Indicador de importancia**. Es un *flag* que indica a una aplicación si es seguro ignorar el campo de extensión si no reconoce el tipo. El indicador proporciona una manera de implementar aplicaciones que trabajan de modo seguro con certificados y evolucionan conforme se van añadiendo nuevas extensiones.

El ITU y el ISO/IEC han desarrollado y publicado un conjunto de extensiones estándar en un apéndice al X.509 v3:

- **Limitaciones básicas**. Este campo indica si el sujeto del certificado es una CA y el máximo nivel de profundidad de un camino de certificación a través de esa CA.
- **Política de certificación**. Este campo contiene las condiciones bajo las que la CA emitió el certificado y el propósito del certificado.
- **Uso de la clave**. Este campo restringe el propósito de la clave pública certificada, indicando, por ejemplo, que la clave sólo se debe usar para firmar, para la encriptación de claves, para la encriptación de datos, etc. Este campo suele marcarse como importante, ya que la clave sólo está certificada para un propósito y usarla para otro no estaría validado en el certificado.

El formato de certificados X.509 se especifica en un sistema de notación denominado *sintaxis abstracta uno* (*Abstract Sintax One* o ASN-1). Para la transmisión de los datos se aplica el DER (*Distinguished Encoding Rules* o *reglas de codificación distinguible*), que transforma el certificado en formato ASN-1 en una secuencia de octetos apropiada para la transmisión en redes reales.

4.3 Listas de Anulación de Certificados (CRLs)

Los certificados tienen un periodo de validez que va de unos meses a unos pocos años. Durante el tiempo que el certificado es válido la entidad certificadora que lo generó mantiene información sobre el estado de ese certificado.

La información más importante que guarda es el *estado de anulación*, que indica que el periodo de validez del certificado ha terminado antes de tiempo y el sistema que lo emplee no debe confiar en él. Las razones de anulación de un certificado son varias: la clave privada del sujeto se ha visto comprometida, la clave privada de la CA se ha visto comprometida o se ha producido un cambio en la afiliación del sujeto (por ejemplo cuando un empleado abandona una empresa).

Las *listas de anulación de certificados* (*Certification Revocation Lists* o CRL) son un mecanismo mediante el cual la CA publica y distribuye información a cerca de los certificados anulados a las aplicaciones que los emplean. Una CRL es una estructura de datos firmada por la CA que contiene su fecha y hora de publicación, el nombre de la entidad certificadora y los números de serie de los certificados anulados que aun no han expirado. Cuando una aplicación trabaja con certificados debe obtener la última CRL de la entidad que firma el certificado que está empleando y comprobar que su número de serie no está incluido en él.

Existen varios métodos para la actualización de CRLs:

1. **Muestreo de CRLs**. Las aplicaciones acceden a la CA o a almacenes de archivos y copian el último CRL a intervalos regulares. La pega de este esquema es que durante el periodo entre actualizaciones del CRL podemos aceptar un certificado ya anulado, por lo que el periodo debe ser corto.
2. **Anuncio de CRLs**. La entidad certificadora anuncia que ha habido un cambio en el CRL a las aplicaciones. El problema de este enfoque es el anuncio puede ser muy costoso y no sabemos que aplicaciones deben ser informadas.
3. **Verificación en línea**. Una aplicación hace una consulta en línea a la CA para determinar el estado de revocación de un certificado. Es el mejor método para las aplicaciones, pero es muy costoso para la CA.

4.4 Listas de Anulación de Certificados X.509

El formato de listas de anulación de certificados X.509 es un estándar del ITU-T y la ISO/IEC que se publicó por primera vez en 1988 como versión 1. El formato fue modificado para incluir campos de extensión, dando origen al al formato X.509 v2 CRL.

Los campos básicos de un formato X.509 CRL (válidos para las versiones 1 y 2) son:

- **Versión**. Debe especificar la versión 2 si hay algún campo de extensión.
- **Firma**. El campo contiene identificador del algoritmo empleado para firmar la CRL.
- **Nombre del generador**. Este campo contiene el nombre de la entidad que ha generado y firmado la CRL.
- **Esta actualización**. Fecha y hora de la generación de la CRL.

- **Próxima actualización**. Indica la fecha y hora de la próxima actualización. El siguiente CRL puede ser generado antes de la fecha indicada pero no después de ella.
- **Certificado del usuario**. Contiene el número de serie de un certificado anulado.
- **Fecha de anulación**. Indica la fecha efectiva de la anulación.

Existe también un conjunto de campos de entrada de extensión en las CRLs X.509 v2, como el **código de razón**, que identifica la causa de la anulación: sin especificar, compromiso de clave, compromiso de la CA, cambio de afiliación, superado (el certificado ha sido reemplazado), cese de operación (el certificado ya no sirve para su propósito original), certificado en espera (el certificado está suspendido temporalmente) y elimina de la CRL (un certificado que aparecía en una CRL previa debe ser eliminado).

Adicionalmente también se ha añadido un conjunto de extensiones para las X.509v2 CRL con los mismos subcampos que en los certificados X.509v3. Estas extensiones permiten que una comunidad o entidad se defina sus propios campos de extensión privados.

4.5 Autoridades Certificadoras

Una *autoridad certificadora* es una organización fiable que acepta solicitudes de certificados de entidades, las valida, genera certificados y mantiene la información de su estado.

Una CA debe proporcionar una *Declaración de Prácticas de Certificación* (*Certification Practice Statement* o *CPS*) que indique claramente sus políticas y prácticas relativas a la seguridad y mantenimiento de los certificados, la responsabilidades de la CA respecto a los sistemas que emplean sus certificados y las obligaciones de los subscriptores respecto de la misma.

Las labores de un CA son:

- **Admisión de solicitudes**. Un usuario rellena un formulario y lo envía a la CA solicitando un certificado. La generación de las claves pública y privada son responsabilidad del usuario o de un sistema asociado a la CA.
- **Autentificación del sujeto**. Antes de firmar la información proporcionada por el sujeto la CA debe verificar su identidad. Dependiendo del nivel de seguridad deseado y el tipo de certificado se deberán tomar las medidas oportunas para la validación.
- **Generación de certificados**. Después de recibir una solicitud y validar los datos la CA genera el certificado correspondiente y lo firma con su clave privada. Posteriormente lo manda al subscriptor y, opcionalmente, lo envía a un almacén de certificados para su distribución.
- **Distribución de certificados**. La entidad certificadora puede proporcionar un servicio de distribución de certificados para que las aplicaciones tengan acceso y puedan obtener los certificados de sus subscriptores. Los métodos de distribución pueden ser: correo electrónico, servicios de directorio como el X.500 o el LDAP, etc.
- **Anulación de certificados**. Al igual que sucede con las solicitudes de certificados, la CA debe validar el origen y autenticidad de una solicitud de anulación. La CA debe mantener información sobre una anulación durante todo el tiempo de validez del certificado original.
- **Almacenes de datos**. Hoy en día existe una noción formal de *almacén* donde se guardan los certificados y la información de las anulaciones. La designación oficial de una base de datos como almacén tiene por objeto señalar que el trabajo con los certificados es fiable y de confianza.

4.6 Infraestructuras de Clave Pública

La difusión de las técnicas de clave pública requiere una *infraestructura* que defina un conjunto de estándares, autoridades de certificación, estructuras entre múltiples CAs, métodos para descubrir y validar rutas de certificación, protocolos operacionales, protocolos de gestión, herramientas que pueden operar entre sí y un marco legislativo.

Los protocolos operacionales se dirigen al problema del envío de certificados y CRLs a los sistemas que emplean certificados. Los protocolos de gestión tratan de los requisitos para la interacción de dos componentes de la infraestructura: registro, inicialización, certificación, anulación y recuperación de claves.

Una estructura entre múltiples CAs proporciona una o más rutas de certificación entre un subscriptor y una aplicación. Una *ruta de certificación* (o cadena de certificación) es una secuencia de uno o más puntos conectados entre el subscriptor y una CA raíz. Una *CA raíz* es una autoridad en la que confía la aplicación, ya que tiene almacenada de forma segura su clave pública.

Un sistema que emplea certificados necesita obtener una ruta de certificación entre un subscriptor y un CA raíz antes de evaluar el nivel de confianza en el certificado del subscriptor. El problema de determinar una ruta de certificación entre dos subscriptores arbitrarios en una estructura de interconexiones entre diferentes CAs se denomina *descubrimiento de rutas de certificación*. El problema de verificar la asociación entre el nombre del subscriptor y su clave pública en una ruta de certificación se denomina *validación de la ruta de certificación*.

5 Los protocolos SSL y TLS

En este apartadodiscutiremos mecanismos para establecer canales seguros para aplicaciones de red a nivel de la capa de transporte. Trataremos los protocolos SSL y TSL, que son los más utilizados en la actualidad para proporcionar versiones seguras de protocolos de red como el `http` (`https`).

5.1 El protocolo SSL

Originalmente diseñado por Netscape para establecer comunicaciones seguras con protocolos como HTTP o FTP. Permite negociar qué algoritmos se van a emplear, intercambiar las claves de encriptación y la autentificación de clientes y servidores.

Existen tres versiones del protocolo, la cuarta es una mejora del SSLv3 y se conoce con el nombre de TLS, que es la especificación que vamos a estudiar.

5.2 El protocolo TLS

El protocolo TLS (*Transport Layer Security*) es una evolución del protocolo SSL (*Secure Sockets Layer*). La última propuesta de estándar está documentada en la referencia `5 ([RFC_2246])`.

Los objetivos del protocolo son varios:

1. **Seguridad criptográfica**. El protocolo se debe emplear para establecer una conexión segura entre dos partes.
2. **Interoperabilidad**. Aplicaciones distintas deben poder intercambiar parámetros criptográficos sin necesidad de que ninguna de las dos conozca el código de la otra.
3. **Extensibilidad**. El protocolo permite la incorporación de nuevos algoritmos criptográficos.

4. **Eficiencia**. Los algoritmos criptográficos son costosos computacionalmente, por lo que el protocolo incluye un esquema de *cache de sesiones* para reducir el número de sesiones que deben inicializarse desde cero (usando criptografía de clave pública).

El protocolo está dividido en dos niveles:

- **Protocolo de registro TLS** (*TLS Record Protocol*).
- **Protocolo de mutuo acuerdo TLS** (*TLS Handshake Protocol*).

El de más bajo nivel es el *Protocolo de Registro*, que se implementa sobre un protocolo de transporte fiable como el TCP. El protocolo proporciona seguridad en la conexión con dos propiedades fundamentales:

1. **La conexión es privada**. Para encriptar los datos se usan algoritmos de cifrado simétrico. Las claves se generan para cada conexión y se basan en un secreto negociado por otro protocolo (como el de mutuo acuerdo). El protocolo también se puede usar sin encriptación.
2. **La conexión es fiable**. El transporte de mensajes incluye una verificación de integridad.

El *protocolo de registro*se emplea para encapsular varios protocolos de más alto nivel, uno de ellos, el *protocolo de mutuo acuerdo*, permite al servidor y al cliente autentificarse mutuamente y negociar un algoritmo de encriptación y sus claves antes de que el protocolo de aplicación transmita o reciba datos.

El *protocolo de mutuo acuerdo* proporciona seguridad en la conexión con tres propiedades básicas:

1. La identidad del interlocutor puede ser autentificada usando criptografía de clave pública. Esta autentificación puede ser opcional, pero generalmente es necesaria al menos para uno de los interlocutores.
2. La negociación de un secreto compartido es segura.
3. La negociación es fiable, nadie puede modificar la negociación sin ser detectado por los interlocutores.

5.3 El protocolo de registro TLS

El protocolo de registro TLS es un protocolo por capas. En cada nivel los mensajes incluyen campos para el tamaño, descripción y contenido. El protocolo toma un mensaje para ser transmitido, lo divide en bloques, comprime los datos (opcionalmente), los encripta, genera un MAC y transmite el resultado.

En el lado del receptor se sigue un proceso inverso: descifrado, verificación, descompresión y reensamblaje.

El estándar describe cuatro clientes del protocolo:

1. El protocolo de mutuo acuerdo
2. El protocolo de alerta
3. El protocolo de cambio de especificaciones criptográficas
4. El protocolo de datos de aplicación

5.4 El protocolo de mutuo acuerdo TLS

El protocolo consta de tres subprotocolos que se emplean para permitir que los interlocutores lleguen a un acuerdo respecto a los parámetros de seguridad para el nivel de registro, se autentifiquen, instancien parámetros de seguridad negociados y se comuniquen condiciones de error.

El protocolo es responsable de negociar una sesión que consta de los siguientes items:

1. **Identificador de sesión**. Secuencia de bytes aleatoria elegida por el servidor para identificar el estado de una sesión activa o reanudable.
2. **Certificado del interlocutor**. Certificado X.509 v3 del interlocutor. Este elemento puede ser nulo.
3. **Método de compresión**. Algoritmo empleado para comprimir los datos antes de encriptarlos.
4. **Especificaciones del algoritmo de encriptación**. Especifica el algoritmo de encriptación (nulo, DES, etc.) y el algoritmo de firmado (MD5 o SHA). También define atributos criptográficos como el tamaño de la clave de la función de dispersión.
5. **Secreto principal**. Clave secreta de 48 bytes compartida entre el cliente y el servidor.
6. **Reutilizable**. Valor que indica si la sesión puede ser empleada para iniciar nuevas conexiones.

5.4.1 Protocolo de cambio de especificaciones criptográficas

Este protocolo marca las transiciones entre distintas estrategias de cifrado. Consta de un mensaje que se encripta y comprime con las especificaciones actuales de la conexión (no las pendientes).

Cuando el destinatario recibe este mensaje la capa de registro copia el estado de lectura pendiente al estado de lectura actual. De forma similar, el emisor cambia su estado de escritura al enviar este mensaje.

Este mensaje se envía durante el acuerdo, después de haber acordado los parámetros de seguridad pero antes de que se envíe el mensaje de verificación finalizada.

5.4.2 Protocolo de alerta

Uno de los tipos de mensaje que soporta la capa de registro es el de alerta. Estos mensajes incluyen la severidad de la alerta y una descripción de la misma. Los mensajes de alerta con nivel de `fatal` provocan la inmediata terminación de la comunicación.

Existen distintos tipos de alertas:

- **Alerta de cierre**. El cliente y el servidor deben saber que la conexión se está cerrando para evitar un ataque de truncado. Cualquiera de los dos puede iniciar el intercambio de mensajes de cierre. Cualquier información recibida después de la alerta de cierre es ignorada.
- **Alerta de error**. La gestión de errores el el protocolo de mutuo acuerdo es muy simple, cuando uno de los interlocutores detecta un error lo envía al otro y, si se trata de un error fatal, cierran la conexión.

5.4.3 Protocolo de mutuo acuerdo

Los parámetros del estado de la sesión son producidos por este protocolo, que opera sobre el protocolo de registro TLS. Cuando un cliente y un servidor empiezan a comunicarse, acuerdan la versión del protocolo, selección de algoritmos criptográficos, opcionalmente se autentifican mutuamente y emplean algoritmos de clave pública para generar secretos compartidos.

El protocolo de mutuo acuerdo consta de los siguientes pasos:

1. Intercambio de *mensajes de saludo* (*hello messages*) para acordar los algoritmos a emplear, intercambiar valores aleatorios y verificar si es una sesión reanudada.
2. Intercambiar los parámetros criptográficos necesarios para permitir que el cliente y el servidor acuerden un pre-secreto.
3. Intercambio de certificados e información criptográfica para permitir que cliente y servidor se autentifiquen.
4. Generar un secreto principal a partir del pre-secreto e intercambiar valores aleatorios.
5. Proporcionar los parámetros de seguridad a la capa de registro.
6. Permitir al cliente y al servidor verificar que su interlocutor a calculado los mismos parámetros de seguridad y que el acuerdo se produjo sin alteraciones por parte de un tercero.

5.4.4 Protocolo de datos de aplicación

Los mensajes de datos de la aplicación son transportados por la capa de registro y son fragmentados, comprimidos y encriptados basándose en el estado actual de la conexión. Los mensajes se tratan como datos transparentes para la capa de registro.

5.5 Aplicaciones e implementaciones

El protocolo SSL/TLS tiene multitud de aplicaciones en uso actualmente. La mayoría de ellas son versiones seguras de programas que emplean protocolos que no lo son. Hay versiones seguras de servidores y clientes de protocolos como el http, nntp, ldap, imap, pop3, etc.

Existen multitud de implementaciones del protocolo, tanto comerciales como de libre distribución. Una de las más populares es la biblioteca `openssl`, escrita en C y disponible bajo licencia GNU. Incluye todas las versiones del SSL y el TLS y un gran número de algoritmos criptográficos, algunos de los cuales ni tan sólo son empleados en el estándar TLS. La biblioteca está disponible en el URL `<http://www.openssl.org>`. En esa misma dirección se puede encontrar una lista de referencias a otras implementaciones gratuitas y comerciales de los protocolos SSL y TLS y aplicaciones que los emplean.

Java incluye soporte para el protocolo con la *Extensión de Sockets Seguros de Java* (JSSE), que discutiremos más adelante.

6 Seguridad en el entorno Java

En este apartado hablaremos sobre la seguridad del entorno de ejecución Java. Centraremos la discusión en los mecanísmos que se emplean para verificar que el código que se ejecuta es seguro y permitir al usuario controlar como accede a los recursos externos e incluso a los de la máquina virtual.

De todos modos, antes de entrar en la seguridad del entorno de ejecución, comentaremos brevemente las características que hacen de Java un lenguaje programación seguro, principalmente como recordatorio para el lector.

Dos buenas referencias sobre los temas tratados en este apartado son la sección sobre seguridad del Tutorial de Sun (referencia `3 ([DAGEF099])`) y la especificación de la Arquitectura de Seguridad de Java (referencia `4 ([GONGLI99])`).

6.1 El lenguaje de programación Java

En muchos lugares se dice que Java es un lenguaje de programación seguro, pero, ¿qué quiere decir esto en realidad?. Para contestar esta pregunta, es necesario entender las maneras en que un lenguaje de programación puede causar vulnerabilidades en las aplicaciones, las comunicaciones o en la integridad de datos. Prácticamente todos estos problemas están relacionados con el acceso a memoria. Para solucionar estos problemas el lenguaje Java se diseño eliminando algunas características presentes en otros lenguajes de programación como C++ (que es del que se partió para el diseño inicial) y añadiendo otras.

Algunos de los métodos incorporados al lenguaje Java para corregir los problemas relacionados con los accesos a memoria ilegales son:

- **Eliminación de la aritmética con punteros**. Java la elimina por completo la aritmética con punteros, que es una de las mayores fuentes de accesos ilegales a memoria en otros lenguajes. De cualquier forma, esto no quiere decir que Java no disponga de la noción de puntero, la incluye dándole el nombre de *referencia*. Usando referencias podemos definir estructuras dinámicas complejas como listas, colas, árboles, etc. igual que en otros lenguajes.
- **Comprobación de rangos en el acceso a vectores**. En Java se controlan todos los accesos a vector y se lanza una excepción cuando intentamos un acceso fuera de rango. En los lenguajes que esto no se hace, el programador puede acceder a memoria que está más allá de la reservada para el vector y modificar valores de otras variables de forma no controlada. De hecho, el intentar provocar estas salidas de rango en los *buffers* empleados en las aplicaciones es uno de los sistemas empleados para romper la seguridad de los programas.
- **Definición del comportamiento de las variables sin inicializar**. En otros lenguajes los valores de las variables no inicializadas están indeterminados, por lo que si accedemos a ellas antes de darles un valor podemos obtener resultados impredecibles. Esto en Java no es así, toda la memoria del *heap* se inicializa automáticamente, y la memoria de la *pila*, que es la que emplean las variables locales, debe ser inicializada por el programador (si en un programa se intenta usar una variable local antes de asignarle un valor el compilador genera un error).
- **Eliminación de la liberación de memoria controlada directamente por el programador**. En otros lenguajes, cuando un objeto creado dinámicamente deja de ser necesaro el responsable de liberar la memoria que tiene asignada es el programador. Es evidente que, de algún modo, es obligarorio el liberar la memoria, ya que si no lo hacemos, nuestra aplicación consumiría mucha más de la necesaria en un momento dado. El problema de esta liberación es que nos podemos equivocar y liberar objetos que aun necesitamos (con lo que, al acceder a ellos, ya no están y leemos datos de otros objetos o simplemente basura) e incluso intentar liberar más de una vez la misma memoria (según el lenguaje, compilador y sistema esto puede liberar memoria empleada por otros objetos, causar la muerte del programa, etc.). Java elimina este problema no proporcionando funciones de liberación de memoria; cuando las variables u objetos ya no son referenciadas la memoria que ocupan es liberada por un sistema de recolección de basura automático (*garbage collection*), que es parte del sistema de ejecución.

Además de lo anterior tambien se deben mencionar otras características del lenguaje que contribuyen a escribir código seguro como:

- El sistema de verificación de tipos en tiempo de compilación, que garantiza que las variables son de los tipos correctos.
- Los niveles de acceso a los miembros de las clases, que permite controlar la visiblidad de atributos y métodos.
- El modificador `final`, que permite impedir que se definan subclases cuando se aplica a una clase o que se puedan redefinir atributos cuando se les aplica a ellos.

6.2 El entorno de ejecución Java

La ejecución de un programa Java consite en interpretar los *bytecodes*, es decir, en transformarlos en código del sistema y ejecutar ese código conforme se va interpretando. La JVM es la encargada de realizar esta tarea. Por suerte, el código a interpretar no es un lenguaje de alto nivel, los bytecodes son en realidad código máquina escrito para el juego de instrucciones de la JVM, por lo que el proceso de interpretación es más rápido que el de otros lenguajes. Además, la máquina virtual puede emplear los denominados JIT (*Just In Time Compiler*), que traducen los bytecodes a código nativo optimizado una sola vez de modo que cada vez que la JVM vuelve a invocarlo se ejecuta directamente la versión ya interpretada.

Antes de que la JVM comience este proceso de interpretación debe realizar una serie de tareas para preparar el entorno en el que el programa se ejecutará. Este es el punto en el que se implementa la seguridad interna de Java.

Hay tres componentes en el proceso:

1. **Cargador de clases**. Este elemento se encarga de separar las clases que carga para evitar ataques. En Java 2 se definen tres grupos de clases asociados a distintas rutas de búsqueda: clases del sistema (asociadas a la ruta de arranque o *boot class path*), clases de extensión del sistema (asociadas a la ruta de extensión o *extension class path*) y clases del usuario o la aplicación (asociadas a la ruta de clases del usuario o la aplicación *user class path*). El orden de búsqueda de las clases es: sistema, extensión y usuario. Cuando una clase se encuentra se detiene la búsqueda (una aplicación sólo podría reemplazar una clase del sistema modificando la ruta de arranque, y eso no es posible sin tener acceso total al sistema). El cargador de clases es una clase Java y puede ser extendida para definir cargadores de clases especiales, pero sólo por las aplicaciones; si un applet pudiera definir su propio cargador podría modificar el cargador del sistema y potencialmente *tomar* la máquina en la que se ejecuta el navegador.

2. **Verificador de archivos de clases**. La función de este componente es validar los bytecodes. Aunque esta tarea podría parecer absurda, no lo es, ya que no todos los bytecodes tienen por qué ser correctos, pues se pueden generar a mano o empleando compiladores modificados. El sistema distingue entre código en el que se confía (generalmente las clases del sistema y las validadas por el usuario) y código en el que no se confía. Las clases consideradas seguras no se validan, pero el resto si. El verificador de archivos de clases forma parte de la JVM y, por lo tanto, sólo puede ser reemplazado cambiando la máquina virtual.

3. **Gestor de seguridad**. Se encarga de comprobar el acceso en tiempo de ejecución. Es una clase del sistema y puede ser extendida por las aplicaciones. La implementación por defecto de Java 2 define un sistema basado en políticas de acceso.

En la figura 1 (Proceso de Ejecución) se resume el proceso.

6.3 Modelos de Seguridad en Java

Desde su creación el entorno Java ha tenido presentes los problemas de seguridad y ha definido un modelo para controlar y limitar el acceso a los recursos desde los programas y aplicaciones. El modelo de seguridad ha ido evolucionando con las distintas versiones del *Entorno de Desarrollo Java* (de aquí en adelante denominado JDK, por sus siglas en inglés), pasando de un modelo muy sencillo y restrictivo, el del JDK 1.0, a uno más complejo y flexible desde la aparición del JDK 1.2.

En este apartado describiremos brevemente los mecanísmos de seguridad incorporados en Java y luego trataremos los modelos de seguridad definidos en las sucesivas versiones del JDK. Terminaremos dando una tabla comparativa de los distintos modelos de seguridad.

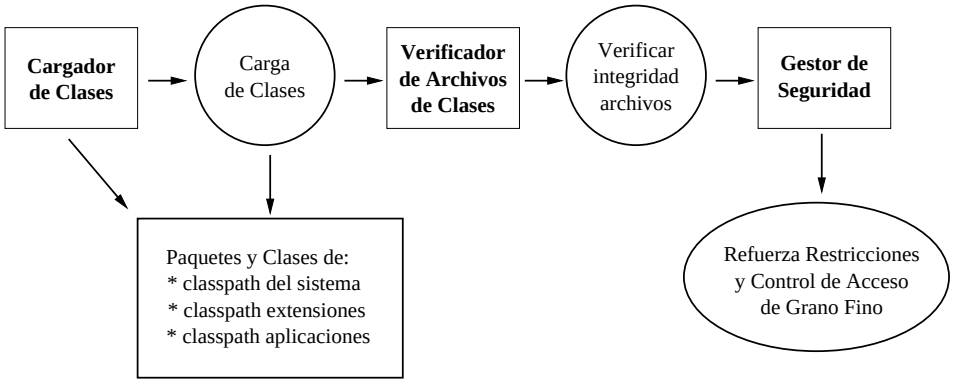

Figura 1: Proceso de Ejecución

En siguientes apartados nos centraremos en la seguridad en Java 2 (JDK 1.2), que es la que tiene un modelo más completo y, en definitiva, incluye todos los anteriores.

6.3.1 Mecanísmos de seguridad

Java fue diseñado para ofrecer las siguientes medidas de seguridad básicas:

- **Uso de un lenguaje de programación seguro**. El lenguaje de programación Java está diseñado para ser seguro, como hemos comentado en el apartado anterior.
- **Integración de un sistema de control de permisos para los programas**. Java define un mecanismo (denominado *mecanismo del cajón de arena*) que permite controlar que se le permite hacer a un programa y controlar como accede a los recursos.
- **Encriptación y uso de certificados**. Se definen mecanísmos para que los programadores puedan firmar el código, de manera que los usuarios puedan verificar quien es el propietario del código y que este no ha sido modificado despues de ser firmado.

La seguridad se basa en tres componentes fundamentales del entorno de ejecución:

1. **El cargador de clases** (*Class Loader*), que determina como y cuando pueden cargar código los programas y garantiza que los componentes del sistema no han sido reemplazados.
2. **El verificador de archivos de clases** (*Class file verifier*), que garantiza que el código tiene el formato correcto, que el bytecode no viola las restriciones de seguridad de tipos de la JVM, que las pilas internas no puedan desbordarse ni por arriba ni por abajo y que las instucciones en bytecode tengan parámetos de tipos correctos.
3. **El gestor de seguridad** (*Security Manager*), que controla el acceso a los recursos en tiempo de ejecución. Los recursos sobre los que tiene control son multiples: E/S de red y ficheros, creación de cargadores de clases, manipulación de hilos de ejecución, ejecución de programas externos (del SO), detener la JVM, cargar código nativo en la máquina virtual, realizar determinadas operaciones en el entorno de ventanas o cargar ciertos tipos de clases.

En apartados posteriores se describirá que son y como funcionan estos componentes.

6.3.2 Seguridad en el JDK 1.0

El modelo de seguridad original de la plataforma Java es el conocido como el *modelo del cajón de arena* (*sandbox model*), que proporcionaba un entorno muy restringido en el que ejecutar código no fiable obtenido de la red.

En este modelo trabajamos con dos niveles de acceso a los recursos: total, para programas locales, o muy restringido, para programas remotos. La seguridad se consigue gracias al empleo del cargador de clases, el verificador de clases y el gestor de seguridad.

La pega fundamental de este modelo es que es demasiado restrictivo, ya que no permite que los programas remotos hagan nada útil, por estar restringidos al modelo del cajón de arena.

6.3.3 Seguridad en el JDK 1.1

Como el modelo del JDK 1.0 era demasido restrictivo se introdujo el concepto de *código remoto firmado*, que sigue garantizando la seguridad de los clientes, pero permite que código obtenido remotamente salga del cajón y tenga acceso a todos los recursos, siempre y cuando esté firmado por una entidad en la el cliente confía.

Aunque esto mejora un poco la situación, sigue siendo un control de dos nivels: total para código local o remoto firmado y restringido para código remoto sin firma o con firmas no validadas por el cliente.

Además del código firmado, el JDK 1.1 introdujo otras mejoras de seguridad:

1. Un par de herramientas de seguridad.
2. Un API para programación segura.

Las herramientas son los programas `jar` y `javakey`. El primero de ellos no es más que un programa archivador con un formato compatible con el `zip`, que permite reunir un conjunto de clases y otros fecheros (como por ejemplo imágenes o texto) en un solo archivo, almacenado normalmente con la extensión `.jar`. Esto hace posible la transferencia de aplicaciones de un modo compacto, en una sola conexión, y el firmado de los programas de modo conjunto. El programa `javakey` es el que permite el firmado de clases en los ficheros `jar`.

El API de seguridad introdujo paquetes de clases que proporcionan funciones criptográficas a los programadores, permitiendo el desarrollo de aplicaciones que usen estas técnicas de modo estándar. El API estaba diseñada para ser extensible e incluía herramientas para: generar firmas digitales, resumenes de mensajes, gestión de claves y el uso de listas de control de accesos (ACLs).

Los algoritmos criptográficos se proporcionaban separados en la Extensión Criptográfica de Java (JCE), como un paquete adicional al JDK estándar, principalmente por los problemas de exportación de los EEUU.

6.3.4 Seguridad en Java 2

En el JDK 1.2 se han introducido nuevas características que mejoran el soporte y el control de la seguridad:

- **Control de acceso de grano fino**. Uno de los problemas fundamentales de la seguridad en el JDK 1.1 es que el modelo sólo contempla dos niveles de permisos: acceso total o cajón de arena. Para solventar el problema el JDK 1.2 introduce un sistema de control de permisos de *grano fino* que peremite dar *permisos específicos* a *trozos de código específicos* para acceder a *recursos específicos* en el cliente, dependiendo de la firma del código y/o el URL del que este se obtuvo.

- **Control de acceso aplicado a todo el código**. El concepto de *código firmado* es ahora aplicable a todo el codigo, independientemente de su procedencia (local o remoto).
- **Facilidad de configuración de políticas de seguridad**. La nueva arquitectura de seguridad permite el ajuste sencillo de los permisos de acceso usando un *fichero de políticas* (*policy file*) en el que se definen los permisos para acceder a los recursos del sistema para todo el código (local o remoto, firmado o sin firmar). Gracias a ello, el usuario puede bajar aplicaciones de la red, instalarlas y ejecutarlas asignándoles sólo los permisos que necesiten.
- **Estructura de control de acceso extensible**. En versiones anteriores del JDK cuando se deseaba crear un nuevo tipo de permiso de acceso era necesario añadir un nuevo método `check` a la clase del gestor de seguridad. Esta versión permite definir `permisos tipo` que representan un acceso a recursos del sistema y el control automático de todos los permisos de un tipo correcto, lo que repercute en que, en la mayoría de casos, se hace innecesario añadir métodos al gestor de seguridad.

En la figura 2 (Seguridad en el JDK 1.2) se presenta el modelo de seguridad de Java 2.

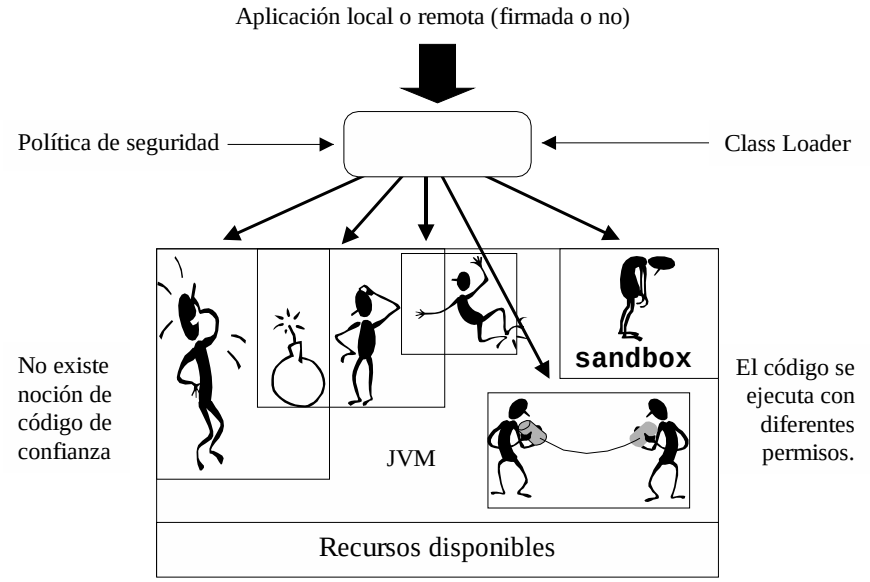

Figura 2: Seguridad en el JDK 1.2

Al igual que sucedió con el JDK 1.1 en el Java 2 han aparecido o se han mejorado las herramientas y el API de seguridad.

En Java 2 se incluyen cuatro herramientas de seguridad:

1. `jar`. Esta es básicamente la misma herramienta que apareció en el JDK 1.1 y se emplea para generar ficheros JAR.
2. `keytool`. Esta es una herramienta para la generación de pares de claves (públicas y privadas), importar y exportar certificados X.509 (versiones 1, 2 y 3), generar certificados X.509 V1 autofirmados y gestionar *almacenes de claves* (*keystores*).
3. `jarsigner`. Este programa se emplea para firmar ficheros JAR y verificar las firmas de ficheros JAR ya firmados. La herramienta emplea los almacenes de claves para buscar los datos que necesita para firmar ficheros (una clave privada), verificar firmas (clave pública) o verificar claves públicas (certificados). Esta herramienta junto a a la anterior reemplazan al antiguo `javakey`.

4. `policytool`. Esta utilidad se emplea para crear y modificar los ficheros de configuración de políticas de seguridad del cliente de modo sencillo.

El API de seguridad se ha incrementado con nuevos subpaquetes que mejoran el soporte de certificados X.509 y permiten crear *Listas de Revocación de Certificados* (CRLs) y *Peticiones de Firmado de Certificados* (CSRs). Los nuevos subpaquetes son el `java.security.cert` y `java.security.spec` y los discutiremos en puntos posteriores.

6.3.5 Evolución del modelo de seguridad

En la tabla 1 (comparación modelos de seguridad) se comparan los modelos de seguridad de Java en función de varias características.

Característica	**JDK 1.0**	**JDK 1.1**	**Java 2 SDK**
Acceso a los recursos del código local sin firma	Sin restricciones	Sin restricciones	Basado en política
Acceso a los recursos del código local firmado	No disponible	Sin restricciones si fiable o restringido por el *sandbox*	Basado en política
Acceso a los recursos del código remoto sin firma	Restringido por el *sandbox*	Restringido por el *sandbox*	Basado en política
Acceso a los recursos del código remoto firmado	No disponible	Sin restricciones si fiable o restringido por el *sandbox*	Basado en política
Servicios de firmado digital de código	No disponibles	JCA (DSA)	JCA (DSA)
Servicios criptográficos	No disponibles	JCE 1.1	JCE 1.2

Tabla 1: Comparación modelos de seguridad

6.4 Dominios protegidos, modelo de permisos y políticas de seguridad

En este punto explicaremos que son los dominios protegidos, como funciona el modelo de permisos de Java 2 y como se definen las políticas de seguridad.

6.4.1 Dominios protegidos

El concepto de *dominio protegido* es fundamental para la seguridad de los sistemas. El alcance de un dominio está definido por el conjunto de objetos que estan directamente accesibles para un *principal*, donde *principal* es una entidad en el sistema informático a la que se le han asignado *permisos*. El *cajón de arena* del JDK 1.0 es un ejemplo de dominio de protección con límites fijos.

El concepto de *dominio protegido* proporciona un mecanismo adecuado para agrupar y aislar unidades de protección. Por ejemplo, se pueden separar dos dominios de forma que la interacción entre ambos únicamente sea posible a través de código del sistema o de un protocolo explícito para la comunicación entre ambos.

Los dominos protegidos se dividen generalmente en dos categorias:

1. **Dominos del sistema**, que controlan el acceso a los recursos del sistema (sistema de archivos, acceso a la red, E/S).

2. **Dominios de aplicación**, que controlan el acceso a los recursos de una aplicación.

Conceptualmente un dominio incluye un conjunto de clases cuyas instancias tienen asignados los mismos *permisos*. Los dominios de protección se determinan mediante la *política de seguridad* activa en cada momento.

El entorno Java mantiene una asociación entre el código (clases e instancias) y sus dominios de protección. Esta relación es de varios a uno, es decir, disitintas clases pueden pertenecer a un mismo dominio, pero cada clase sólo está asociada un dominio. Para los dominios de protección se emplea otra tabla que relaciona cada dominio con sus permisos correspondientes. Evidentemente esta asociación es de muchos a muchos, varios dominios pueden tener el mismo permiso y cada dominio puede tener multiples permisos.

Un hilo de ejecución puede trabajar únicamente en un dominio protegido o puede ir pasando del dominio de la aplicación al del sistema y viceversa. Esto tiene implicaciones importantes en la seguridad, se plantean dos escenarios:

1. Si una aplicación accede al sistema y solicita una operación que requiere permisos del dominio del sistema, se debe garantizar que la aplicación no tendrá acceso directo al dominio del sistema, es decir, que su domino de protección no obtendrá ningún permiso del dominio del sistema.
2. Si una clase del sistema invoca un método de la aplicación, es esencial que los permisos de ese método cuando se ejecute sean los del domino de la aplicación, no los del sistema.

En definitiva, el modelo de dominios protegidos debe garantizar que un dominio menos *poderoso* no pueda obtener permisos adicionales al invocar o ser invocado por otro dominio más poderoso.

Ocasionalmente será necesario que código fiable de acceso temporal a más recursos de los que normalmente están disponibles para la aplicación que lo invocó. Por ejemplo, una aplicación no tiene porque tener acceso directo a los ficheros que contienen los tipos de letra, pero la útilidad del sistema que muestra un documento debe conseguir esos tipos. Para solucionarlo se emplea el método `doPrivileged`, que está disponible en todos los dominios.

Cuando un fragmento de código llama al método `doPrivileged`, se considera que el conjunto de permisos activo incluye un permiso si lo permite el domino protegido del código invocante y todos los dominos en los que se entra a continuación.

Durante la ejecución, cuando se solicita acceso a un recurso del sistema, el código de gestión de recursos invoca directa o indirectamente a un método especial de la clase `AccessControler` que evalua la petición y decide si debe concederse o no el acceso. Si se concede el permiso la ejecución continua y si no se lanza una excepción de seguridad.

6.4.2 Modelo de permisos

Los permisos en Java son clases que representan accesos a recursos del sistema. La clase fundamental es `java.security.Permission`, que es una clase abstracta de la que se deben definir subclases para representar accesos específicos.

Un permiso consta de un *objetivo* y una *acción*, aunque puede omitirse cualquiera de los dos. Por ejemplo, para acceder al sistema de ficheros local el *objetivo* puede ser un directorio o un fichero y las *acciones* pueden ser: leer, escribir, ejecutar y borrar.

Generalmente, una clase de permiso pertenece al paquete en el cual será usada. Por ejemplo, el permiso que representa el acceso al sistema de ficheros local es `java.io.FilePermission`.

Como ejemplo de permiso, el siguiente fragmento de código se puede emplear para generar un permiso de lectura del archivo `"abc"` en el directorio `/tmp`:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

Si queremos definir nuevos permisos es crucial implementar el método abstracto `implies`. Básicamente, la afirmación *a implica b* tiene el significado intuitivo, si una clase tiene el permiso *a* tiene tambien el permiso *b*. Esto es muy importante a la hora de tomar decisiones de control de acceso.

La clase abstracta `java.security.Permission` tiene asociadas dos clases importantes:

1. La clase abstracta `java.security.PermissionCollection`, que representa una colección de objetos de tipo `Permission` de una misma categoría (como por ejemplo de acceso a ficheros), para simplificar el agrupamiento.
2. La clase final `java.security.Permissions`, que representa una colección de colecciones de objetos `Permission`.

Antes del acceder a un recurso se toma la decisión de control de acceso al recurso, basada en los permisos que el código en ejecución tiene. Aunque cualquier código puede crear sus propios objetos de permisos, esto no implica que tales objetos obtengan los correspondientes permisos de acceso. Sólo los objetos de permisos que gestiona el sistema en tiempo de ejecución de Java obtienen los permisos que representan.

Los permisos definidos por el JDK 1.2 son:

AllPermission

Es un permiso que asigna todos los demás permisos.

AWTPermission

Permisos para el AWT (acceso al portapapeles, a los eventos, etc.).

FilePermission

Un permiso `java.io.FilePermission` representa permisos de acceso a archivos o directorios y consta de un nombre de ruta y un conjunto de acciones validas para esa ruta.

Si la ruta tiene el valor `*` se considera que el permiso se asigna a todos los ficheros del directorio acutal y si tiene el valor `-`, se refiere a los archivos del directorio actual y, recursivamente, a todos lo ficheros del directorio actual.

Las acciones posibles son: `read`, `write`, `execute` y `delete`.

NetPermission

Un permiso `java.net.NetPermission` es para varios permisos de red. Un permiso de red contiene un nombre pero no tiene lista de acciones, o se tiene el permiso o no se tiene.

PropertyPermission

Permisos para manipular `properties`.

ReflectPermission

Permisos para efectuar operaciones reflectivas. Son permisos sin acciones.

RuntimePermission

Permisos para tiempo de ejecución (acceso a ClassLoaders, SecurityManager, Threads, etc). Son permisos sin acciones.

SecurityPermission

Permisos relacionados con la seguridad. Son permisos sin acciones.

SerializablePermission

Permisos relacionados con la serialización de objetos. Son permisos sin acciones.

SocketPermission

Un permiso `java.net.SocketPermission` representa un acceso a la red a través de sockets. Un `SocketPermission` consta de una dirección de `host` y un conjunto de acciones especificando formas de conectar.

La especificación de la dirección del `host` se hace del siguiente modo:

```
host = (nombreHost | dirIP)[:rangoPuertos]
rangoPuertos = numPuerto | -numPuerto | numPuerto-[numPuerto]
```

El nombre del host puede ser según el DNS o en formato IP. El rango de puertos es opcional.

Las formas de conectar posibles son: `accept`, `connect`, `listen` y `resolve`.

En `<http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html>` se pueden encontrar tablas con los permisos definidos en el jdk1.2 y las implicaciones de seguridad que tiene asignarlos.

6.4.3 Políticas de seguridad

En el JDK 1.2 las políticas de seguridad se especifican en uno o más ficheros de configuración de políticas. Estos ficheros especifican que permisos están habilitados para el código obtenido de los *origenes de código* especificados.

Un archivo de políticas de seguridad se puede escribir directamente con un editor de texto ascii o usando la herramienta `policytool` del JDK.

Por defecto hay un archivo de políticas del sistema y, opcionalmente, otro archivo de políticas del usuario.

El objeto `Policy` por defecto se inicaliza la primera vez que se invoca su método `getPermissions()` o cuando se invoca su método `refresh()`. La inicialización supone el análisis (*parsing*) de los archivos de configuración de políticas y la configuración del objeto `Policy`.

Se puede encontrar una discusión sobre las *políticas de seguridad* en la documentación sobre seguridad del JDK 1.2: `<http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html>`.

6.5 El verificador de archivos de clases

Cualquier máquina virtual Java contiene un verificador de clases que asegura que las clases cargadas tienen la estructura interna correcta. Si el verificador de clases descubre un problema dentro de una clase genera una excepción. Debido a que una clase es una secuencia de bytes, la máquina virtual no puede saber si una clase en particular es realmente un bytecode correcto o no. Como consecuencia de esto, todas las implementaciones de la máquina virtual disponen de un verificador de clases que puede ser invocado sobre clases no seguras, asegurando de esta forma su corrección.

Uno de los objetivos del verificador de clases es ayudar a obtener aplicaciones robustas. Si un programador malintencionado generara una clase que contuviera un método cuyo código en bytes incluyera una instrucción de salto al final del método podría causar que la máquina virtual no funcionara si el método es invocado.

La especificación recomienda que la verificación del bytecode de las aplicaciones se realice justo después de que la clase halla sido cargada. El verificador de clases lleva a cabo su tarea de comprobación en dos fases:

1. La primera fase tiene lugar justo después de cargar la clase. En ella el verficador de clases comprueba la estructura interna de la clase, incluyendo la comprobación de la integridad de los bytecodes.
2. La segunda fase se lleva a cabo mientras se ejecuta el bytecode de la clase. En esta el verficador de clases confirma la existencia de las referencias simbólicas a clases, campos y métodos.

6.6 El cargador de clases

El cargador de clases es el responsable de encontrar y cargar los bytecodes que definen las clase. Una vez que se cargan, los bytecodes son verificados antes de que se puedan crear las clases reales.

Los cargadores de clases son a su vez clases Java, pero, ¿como se carga el primero?. En principio una máquina virtual Java debe incluir un cargador de clases *primario*, que es el encargado de arrancar el sistema de carga de clases. Este cargador estará escrito en un lenguaje como el C y no aparece en el contexto Java. El cargador primario carga las clases del sistema de archivos local de modo dependiente del sistema.

El cargador de clases cumple también varias tareas relacionadas con la seguridad:

- El responsable de cargar las clases del paquete `java.*` es el *cargador primario*, de hecho, todas las clases de este paquete tienen un cargador de clases `null`. Este hecho es importante para la seguridad por dos razones: en primer lugar nos garantiza que se cargaran correctamente, algo muy importante, ya que son básicas para el correcto funcionamiento del sistema, y en segundo tambien nos asegura que se cargarán del sistema de archivos local, lo que evita que una aplicación remota pueda reemplazarlas.
- El cargador de clases proporciona espacios de nombres diferentes para clases cargadas de orígenes diferentes, lo que evita que haya colisiones de nombres entre clases cargadas desde orígenes distintos.
- Clases cargadas de fuentes diferentes no pueden comunicarse dentro del espacio de la máquina virtual, lo que evita que programas no fiables obtengan información de otros que si lo son.

La implementación por defecto del `ClassLoader` del JDK busca las clases según los siguientes pasos:

1. Comprueba que la clase no está cargada.
2. Si la clase no está cargada y el cargador actual tiene un cargador padre, se la pide a este y si no al cargador principal.
3. Llama a un método personalizable para intentar encontrar la clase de otra forma.

Si después de estos pasos la clase no se ha encontrado se lanza una excepción `ClassNotFound`.

El sistema determina el tipo de cargador a emplear del siguiente modo:

- Cuando se carga una aplicación, se emplea una nueva instancia de la clase `URLClassLoader`.
- Cuando se carga un applet, se emplea una nueva instancia de la clase `AppletClassLoader`.
- Cuando se llama directamente al método `java.lang.Class.ForName`, se emplea el cargador principal.
- Si la solicitud de carga la realiza una clase existente, se emplea el cargador de esa clase.

Para crear un `ClassLoader` personalizado basta implementar el método `loadClass` en una subclase:

```
protected abstract Class loadClass(String name, boolean resolve) throws ClassNotFoundException
```

En el método hay que realizar cinco operaciones:

1. Comprobar si la clase está cargada
2. Si no lo está, cargamos los datos de la clase según el método que queramos (por ejemplo mediante una consulta a una base de datos).
3. Llamamos al método `defineClass()` para convertir los bytes en una clase.

4. Resolver la clase invocando el método `resolveClass()`.

5. Retornamos la nueva clase creada.

A continuación se presenta un esquema del método loadClass comentado antes:

```
// método loadClass()
protected Class loadClass(String nom, boolean res) throws ClassNotFoundException {
  // -- Paso 1 --
  Class c = findLoadedClass (nom);
  if (c == null) {
    try {
      c = findSystemClass (nom);
    } catch (Exception e) {
      // Ignoramos excepciones
    }
  }
  if (c == null) {
    // -- Paso 2 --
    byte datos[] = cargarClase(nom);
    // -- Paso 3 --
    c = defineClass (nom, datos, 0, datos.length);
    if (c == null)
      throw new ClassNotFoundException (nom);
    // -- Paso 4 --
    if (res)
      resolveClass (c);
  }
  // -- Paso 5 --
  return c;
}
```

Para usar el cargador se escribiría un fragmento de código similar al siguiente:

```
ClassLoader cargador = new MiClassLoader(parametros);
Class c = cargador.loadClass ("MiClase", true);
MiClase mc = (MiClase)c.newInstance();
```

Para evitar tener que escribir nuesto propio cargador de clases el JDK 1.2 introduce el `URLClassLoader`, que es una subclase de `SecureClassLoader`. Con esta clase podemos cargar cualquier clase que pueda ser localizada mendiante un URL (`file:`, `http:`, `jar:`, etc). Si lo que necesita el programador es hacer una operación como encriptar u obtener la clase de una BDA, puede hacerlo con una subclase de la clase `URLClassLoader`.

Para usar un `URLClassLoader` sólo es necesario decirle al cargador donde están las clases, no hace falta hacer una subclase a menos que se tengan requisitos muy especiales. Los URLs que terminan con / se consideran directorios y cualquier otra cosa se intenta cargar como archivo JAR.

A continuación se presenta un ejemplo de uso:

```
try {
  URL listaURLsList[] = {
```

```
      new URL ("http://www.iti.upv.es/clases/"),
      new URL ("http://case.iti.upv.es/Monkey.zip"),
      new URL ("http://torpedo.upv.es/luis/norte/"),
      new File ("misClases.jar").toURL()
    };
    ClassLoader cargador = new URLClassLoader (listaURLs);
    Class c = cargador.loadClass("MiClase");
    MiClase mc = (MiClase) c.newInstance();
  } catch (MalformedURLException e) {
    // cargar la clase de otra manera o error
  }
```

6.7 El gestor de seguridad

Actualmente, todo el código del sistema del JDK invoca métodos del Gestor de seguridad para determinar la politica de seguridad activa y realizar verificaciones de control de acceso.

Por defecto, cuando se ejecutan *applets* siempre se carga una implementación del gestor de seguridad, pero cuando se lanzan aplicaciones esto no es así. Si el usuario desea que se instale uno debe invocar la máquina virtual definiendo la propieda `java.security.manager`:

```
java -Djava.security.manager NomApp
```

o la aplicación debe invocar el método `setSecurityManager` de la clase `java.lang.System` para instalar un gestor.

Si se le asigna valor a la propiedad `java.security.manager` se instalará el gestor de seguridad especificado:

```
java -Djava.security.manager=iti.GestorSeguridad NomApp
```

Un aspecto muy importante del gestor de seguridad es que una vez cargado no se puede reemplazar, de modo que ni las applets ni las aplicaciones pueden instalar el suyo cuando el usuario (applicaciones) o el sistema (applets) ya han cargado uno.

Por último hay que señalar que el gestor de seguridad está muy relacionado con la clase `AccessControler`: la clase `SecurityManager` representa el concepto de un punto central de control de acceso, mientras que la clase `AccessControler` implementa un algoritmo concreto de control de acceso con características especiales como el método `doPrivileged()`.

En versiones del JDK anteriores a la 1.2, los programadores redefinian el gestor de seguridad cuando necesitaban métodos de control de acceso específicos, pero ahora el método adecuado para hacer esto es emplear el `AccessController`.

6.8 Ficheros de configuración

La política de seguridad se configura a través de al menos dos ficheros del sistema:

```
$JAVA_HOME/lib/security/java.policy
$JAVA_HOME/lib/security/java.security
```

donde `$JAVA_HOME` es el directorio raíz del jdk1.2.

A continuación describiremos el formato de los dos ficheros, comenzando por sus características comunes. Si se desea una discusión ampliada se puede consultar la sección de seguridad de la documentación del jdk1.2 en el URL `<http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html>` o directamente consultando los ficheros instalados en su sistema.

6.8.1 Variables en los ficheros de configuración

Tanto el fichero `java.policy` como el fichero `java.security` pueden incorporar variables que les permiten hacer más dinámica la configuración de la política de seguridad. Las variables de extensión son similares a las variables del shell de Unix, por ejemplo la línea

```
permission java.io.FilePermission "${user.home}", "read"
```

permite que cualquier usuario disponga de permiso de lectura sobre su directorio `home`. La propiedad `${user.home}` tomará el valor del directorio del usuario.

Otra variable muy útil es `${/}`, cuyo valor será el carácter de separación de directorios propio del sistema operativo, es decir, en Unix equivale a / y en DOS a \.

Nota: No se pueden anidar propiedades, por ejemplo `${user.${foo}}`, no funcionaría bien incluso si `foo` tuviera el valor `home`.

6.8.2 El fichero `java.policy`

La política de seguridad del sistema se configura a partir de uno o más ficheros de configuración. Estos ficheros indican que permisos se conceden a que código y para que recursos específicos.

El fichero de configuración de la política de seguridad esencialmente contiene una lista de entradas. Puede contener una entrada *keystore* y una o más entradas *grant*.

El keystore es la base de datos de claves privadas y sus certificados asociados. La entrada `keystore` sirve para indicar dónde consultar las claves públicas de las entidades firmantes especificadas en las entradas `grant` del fichero. La entrada `keystore` debe aparecer en el fichero si cualquier entrada `grant` especifica el alias de una entidad que firma.

El fichero sólo puede contener una entrada `keystore` en el fichero (cualquier otra después de la primera será ignorada), y de ubicarse independientemente de cualquier otra entrada. Su sintaxis es:

```
keystore "url_fichero_keystore"
```

donde `url_fichero_keystore` especifica la localización URL de la base de datos. El URL es relativo a la localización del fichero de configuración, es decir, si el fichero de configuración se encuentra en el URL `<http://case.iti.upv.es/seguridad/iti.policy>` y la entrada del keystore es `keystore ".keystore"` el keystore se encontrará en el URL `<http://case.iti.upv.es/seguridad/.keystore>`. También puede especificarse un URL absoluto.

Cada entrada `grant` consiste básicamente en un la especificación del origen de una clase y sus permisos. Cada entrada `grant` del fichero sigue el siguiente formato, donde la palabra reservada grant indica el comienzo de una nueva entrada. Dentro de cada entrada, la palabra reservada permission marca el comienzo de un nuevo permiso en la entrada.

```
grant [SignedBy "lista_alias_entidades_firmantes"] [, CodeBase "URL"] {
permission clase_permiso ["recurso"][, "acción"][, SignedBy "lista_alias"];
permission ...
...
};
```

Se permiten espacios en blanco inmediatamente antes o después de las comas. La `clase_permiso` debe ser el nombre de una clase de permisos, por ejemplo `java.io.FilePermission` y no puede abreviarse escribiendo únicamente `FilePermission`.

El campo `acción` es opcional y puede omitirse si la clase del permiso no lo necesita. El campo `CodeBase` también es opcional si se omite significa *cualquier código*.

6.8.3 El fichero `java.security`

En este fichero se almacena las propiedades necesarias para configurar la seguridad del sistema. En el se almacenan propiedades tan necesarias como los proveedores de seguridad instalados en el sistema, la ubicación de los ficheros de configuración, nombre de la clase que implementa la política de seguridad, etc.

Los proveedores instalados se registran añadiendo al fichero una línea con el formato:

```
security.provider.n=nombre_clase_provider
```

donde `n` indica el número de proveedor. Veremos como se usa en el apartado dedicado a la arquitectura criptográfica de Java.

La localización de los ficheros de configuración se especifica como valores de propiedades cuyos nombres son de la forma `policy.url.n` donde `n` es un número. Por defecto el sistema tiene definidas las siguientes ubicaciones:

```
policy.url.1=${java.home}${/}lib${/}security${/}java.policy
policy.url.2=${user.home}${/}.java.policy
```

El sistema establece la política de seguridad primero leyendo `policy.url.1`, e incrementa el número hasta que no encuentra otra entrada. La evaluación es incremental, es decir, el sistema configura su política en base a los tres ficheros de configuración.

Nota: Si se dispone de `policy.url.1` y `policy.url.3`, el fichero 3 no se leerá, los números de ficheros han de ser correlativos.

6.9 Herramientas de seguridad

Java 2 incluye una serie de herramientas de seguridad que simplifican la gestión de la misma por parte de los usuarios.

A continuación vamos a presentar brevemente estas herramientas, para una explicación detallada del sistema remitimos al lector a la documentación del JDK: `<java.sun.com/products/jdk/1.2/docs/guide/security/SecurityToolsSummary.html>`.

6.9.1 keytool

La herramienta `keytool` se emplea para gestionar un almacén de claves (*keystore*), entre otras cosas permite:

1. Crear pares de claves (pública y privada).
2. Emitir solicitudes de certificados (que se envían al CA apropiado).
3. Importar replicas de certificados (obtenidos del CA contactado).
4. Designar claves públicas de otros como de confianza.

El almacen de claves es una base de datos protegida que contienen claves y certificados para un usuario o grupo de usuarios. El acceso al mismo está protegido por un sistema de contraseñas y además, cada clave privada puede protegerse con una contraseña individual.

6.9.2 jar

La herramienta `jar` se emplea para crear archivos JAR.

El formato JAR permite reunir multiples ficheros (clases, imágenes, textos) en un único archivo. Cuando se desea firmar una aplicación o applet se usa el programa `jar` para empaquetarla y luego se firma el resultado usando el programa `jarsigner`.

6.9.3 jarsigner

La herramienta `jarsigner` se emplea para firmar y validar la firma de archivos JAR.

La herramienta `jarsigner` emplea el almacen de claves que crea y gestiona el programa `keytool` para buscar las claves privadas y certificados que necesita para firmar archivos JAR.

Como los accesos al almacen de claves y a las claves privadas están protegidos con contraseña, sólo los usuarios que los conozcan podrán acceder a las claves para firmar archivos JAR.

6.9.4 policytool

La herramienta `policytool` se emplea para crear y modificar los archivos de configuración de políticas. La herramienta tiene una interfaz gráfica cómoda para trabajar con estos ficheros, lo que la hace preferible a la edición manual.

7 Arquitectura criptográfica

Desde el punto de vista de la seguridad, el conjunto de clases de seguridad distribuidas con el Java 2 SDK pueden dividirse en dos subconjuntos:

1. Clases relacionadas con el control de acceso y la gestión de permisos.
2. Clases relacionadas con la Criptografía.

Desde el JDK 1.1 Java incluye APIs de acceso a funciones criptográficas de propósito general, conocidas colectivamente como la *Arquitectura Criptográfica de Java (JCA)* y la *Extension Criptográfica de Java (JCE)*.

El JCA está formado por las clases básicas relacionadas con criptografía distribuidas con el JDK y el soporte para la encriptación lo proporciona el paquete de extensión JCE.

El JDK 1.1 introdujo la *Arquitectura de Proveedores*, que permite que coexistan multiples implementaciones de algoritmos criptográficos (es decir multiples implementaciones del JCE). La plataforma Java 2 extiende substancialmente la JCA, entre otras cosas se ha mejorado la infraestructura de gestión de certificados para soportar los certificados X.509 V3.

En este apartado discutiremos brevemente que son, para que sirven y como se utilizan estas APIs.

Si se desea completar la descripción realizada en este punto se puede consultar la guía sobre seguridad del JDK 1.2 disponble en la dirección `<http://java.sun.com/products/jdk/1.2/docs/guide/security/>`. Para obtener alguna implementación del JCA o el JCE se puede consultar una lista disponible en el URL `<http://java.sun.com/products/jce/jce12_providers.html>`.

7.1 Arquitectura Criptográfica de Java (JCA)

La JCA es un marco de trabajo para acceder y desarrollar funciones criptográficas en la plataforma Java. Se diseñó alrededor de dos principios básicos:

1. **Idependencia e interoperabilidad de las implementaciones**:
   - *La independencia de la implementación* se consige empleando una arquitectura basada en proveedores. El término *proveedor de servicios criptográficos* se refiere a un paquete o conjunto de paquetes que proporcionan una implementación concreta de los aspectos criptográficos de la API de seguridad de Java (como algoritmos de firmado, de funciones de dispersión o conversión de claves). Los proveedores deben poder cambiarse de modo transparente para las aplicaciones.
   - *La interoperabilidad de las implementaciones* significa que cada una de ellas puede trabajar con las demas, usar claves generadas por otra implementación o verificar sus claves.
2. **Independencia y extensibilidad de los algoritmos**:
   - *La independencia de los algoritmos* se consigue definiendo tipos de servicios criptográficos y definiendo clases que proporcionan la funcionalidad de estos servicios. Estas clases se denominan *clases motor* y ejemplos de ellas son las clases `MessageDigest`, `Signature` y `KeyFactory`.
   - *La extensibilidad de los algoritmos* significa que nuevos algoritmos que entren dentro de alguno de los tipos soportados (es decir, sean compatibles con las clases motor) puede ser añadido fácilmente.

La separación entre el JCA y el JCE está motivada por las reglas de exportación de los EEUU para la encriptación. Las clases distribuidas con el JDK estándar sólo proporcionan herramientas para el resumen de mensajes y las firmas digitales, lo que permite tener un sistema de autentificación fiable sobre el que implementar un sistema de control de acceso más flexible que el modelo del cajón de arena. El problema es que estas herramientas no son suficientes para el envío seguro de datos, que necesitan algoritmos de encriptación.

La solución a este último problema es el JCE, que emplea la misma estructura que el JCA y proporciona clases motor para implementar criptografía de clave simétrica y para la generación y manipulación de las claves que estos emplean.

7.1.1 Motor, algoritmo y proveedor

Antes de comenzar con la discusión del funcionamiento del JCA tenemos que definir algunos términos básicos:

Motor

En el contexto del JCA utilizamos el término *motor* (*engine*) para referirnos a una representación abstracta de un servicio criptográfico que no tiene una implementación concreta. Un servicio criptográfico siempre está asociado con un algoritmo o tipo de algoritmo y puede tener alguna de las siguientes funciones:

- Proporcionar operaciones criptográficas (como las empleadas en el firmado y el resumen de mensajes).
- Generar o proporcionar el material criptográfico (claves o parámetros) necesario para realizar las operaciones.
- Generar objetos (almacenes de claves o certificados) que agrupen claves criptográficas de modo seguro.

Algoritmo

Un *algoritmo* es una implementación de un motor. P. ej. el algoritmo MD5 es una implementación del motor de algoritmos de resumen de mensajes. La implementación interna puede variar dependiendo del código que proporcione la clase MD5.

Proveedor

Un *proveedor* es el encargado de proporcionar la implementación de uno o varios algoritmos al programador (es decir, darle acceso a una implementación interna concreta de los algoritmos).

7.1.2 El concepto de proveedor

La JCA define el concepto de proveedor mediante la clase `Provider` del paquete `java.security`. Se trata de una clase abstracta que debe ser redefinida por clases proveedor específicas.

El constructor de una clase proveedor ajusta los valores de varias propiedades que necesita el API de seguridad de Java para localizar los algoritmos u otras facilidades implementadas por el proveedor.

La clase `Provider` tiene métodos para acceder al nombre del proveedor, el número de versión y otras informaciones sobre las implementaciones de los algorirmos para la generación, conversión y gestión de claves y la generación de firmas y resumenes.

Para entender como funcionan los proveedores daremos un ejemplo. Supongamos que un programa necesita una implementación del algoritmo MD5. Para obtenerla el programador necesita crear una instancia del mismo y lo hará escribiendo la siguiente línea de código:

```
MessageDigest m = MessageDigest.getInstance("MD5");
```

Internamente, el método `getInstance()` solicita a la clase `java.security.Security` que le proporcione el objeto solicitado. Como no se ha especificado proveedor la clase `Security` consulta a todos los proveedores disponibles, solicitando una implementación del algoritmo "MD5", hasta que encuentra una o se queda sin proveedores. La consulta se realiza según la lista de proveedores del archivo `java.security`, que por defecto sólo contiene la entrada:

```
Security.provider.1=sun.security.provider.Sun
```

Si se encuentra una implementación se retorna una instancia de la clase retornada por el proveedor y si no se genera la excepción `NoSuchAlgorithmException`. En caso de que se haya obtenido una implementación, para generar el resumen de un vector de bytes (p. ej. `buf`) con el MD5 bastará invocar el método `update()` del algoritmo. Para obtener el vector resumen invocaremos el método `digest()`:

```
m.update(buf);
byte[] resumen = m.digest();
```

Por último sólo nos queda saber como se gestionan los proveedores, es decir, como se instalan o eliminan.

Existen dos modos de hacerlo:

1. *Estáticamente*, editando las entradas del fichero `java.security`
2. *Dinámicamente*, invocando desde el programa los métodos `addProvider()` o `insertProvider()` de la clase `java.security.Security` para añadirlos o al método `removeProvider()`

Si un programador desea saber los proveedores disponibles puede emplear los métodos `getProvider("nombre")` (para saber si un proveedor concreto está instalado) o `getProviders()` (que retorna un vector de cadenas con los nombres de los proveedores).

7.1.3 Las clases Motor

En el JDK 1.2 el JCA define las clases motor presentadas en la tabla 2 (Clases motor de la JCA 1.2) y el JCE las de la tabla 3 (Clases motor de la JCE 1.2).

Clase JCA 1.2	**Función**
`java.security.MessageDigest`	Calculo de resumen de mensajes (*hash*).
`java.security.Signature`	Firmado de datos y verificación firmas.
`java.security.KeyPairGenerator`	Generar pares de claves (pública y privada) para un algoritmo.
`java.security.KeyFactory`	Convertir claves de formato criptográfico a especificaciones de claves y vice versa.
`java.security.certificate.CertificateFactory`	Crear certificados de clave pública y listas de revocación (CRLs).
`java.security.KeyStore`	Crear y gestionar un almacen de claves (*keystore*).
`java.security.AlgorithmParameters`	Gestionar los parámetros de un algoritmo, incluyendo codificación y descodificación.
`java.security.AlgorithmParameterGenerator`	Generar un conjunto de parámetros para un algoritmo.
`java.security.SecureRandom`	Generar números aleatrorios o pseudo aleatrorios.

Tabla 2: Clases motor de la JCA 1.2

Clase JCE 1.2	**Función**
`java.crypto.Cipher`	Proporciona encriptación y desencriptación.
`java.crypto.KeyAgreement`	Proporciona un protocolo de intercambio de claves.
`java.crypto.KeyGenerator`	Proporciona un generador de claves simétricas.
`java.crypto.Mac`	Proporciona un algoritmo de autentificación de mensajes.
`java.crypto.SecretKeyFactory`	Representa una factoría de claves secretas.

Tabla 3: Clases motor de la JCE 1.2

Para instanciar una clase motor se debe invocar el método estático `getInstance()`, si se le pasa un nombre de algoritmo se intentará obtener una implementación de algún proveedor (ver ejemplo en 42) y si además se le pasa el nombre de proveedor se buscará la implementación solicitada.

Cada clase motor proporciona métodos para permitir a las aplicaciones el acceso a los servicios criptográficos específicos que proporciona, independientemente del algoritmo en particular. Las APIs de las clases motor se implementan en términos de *interfaces de proveedor de servicios* (*Service Provider Interface* o SPI), esto es, para cada clase motor se define una clase SPI abstracta que define los métodos que un proveedor debe implementar.

7.1.4 Algoritmos

El siguiente programa nos permite saber que proveedores y algoritmos tenemos instalados en nuestro sistema. Además, si lo invocamos con la opción `-l` nos dirá que algoritmos implementan (leyendo las propiedades del proveedor):

```
// InfoProveedores.java
import java.security.*;
```

```
import java.util.*;

class InfoProveedores {
  public static void main(String[] args) {
    boolean listarProps = false;
    if ( args.length > 0 && args[0].equals("-l") )
      listarProps=true;
    System.out.println("-----------------------------------");
    System.out.println("Proveedores instalados en su sistema");
    System.out.println("-----------------------------------");
    Provider[] listaProv = Security.getProviders();
    for (int i = 0; i < listaProv.length; i++) {
      System.out.println("Núm. proveedor : "    + (i + 1));
      System.out.println("Nombre         : "    + listaProv[i].getName());
      System.out.println("Versión        : "    + listaProv[i].getVersion());
      System.out.println("Información    :\n  " + listaProv[i].getInfo());
      System.out.println("Propiedades    :");
      if (listarProps) {
        Enumeration propiedades = listaProv[i].propertyNames();
        while (propiedades.hasMoreElements()) {
          String clave = (String) propiedades.nextElement();
          String valor = listaProv[i].getProperty(clave);
          System.out.println("  " + clave + " = " + valor);
        }
      }
      System.out.println("-----------------------------------");
    }
  }
}
```

Por defecto el JDK 1.2 incluye el proveedor de JCA SUN, para saber que algoritmos implementa ejecutar el programa con la opción `-f`.

7.2 Extensión Criptográfica de Java (JCE)

La JCE se proporciona como una extensión de la plataforma Java y proporciona implementaciones de algoritmos que permiten encriptar, generar claves, intercambiar claves y autentificar mensajes. La extensión complementa las interfaces e implementaciones de resumen y firmado de mensajes del JDK 1.2.

Al igual que el JCA, el JCE emplea un modelo basado en el uso de *proveedores*. El paquete consta de un paquete pricipal denominado `javax.crypto` y dos subpaquetes `javax.crypto.spec` y `javax.crypto.interfaces`.

El paquete principal consta de clases que representan los conceptos de cifrado, acuerdos de claves y codigos de autentificación de mensajes y sus clases de interfaz de proveedor (SPI).

El paquete `javax.crypto.spec` consta de varias clases de especificación de claves y de parámetros de algoritmos.

El paquete `javax.crypto.interfaces` presenta las interfaces de las claves empleadas en los algoritmos de tipo Diffie-Hellman (clases `DHKey`, `DHPrivateKey` y `DHPublicKey`).

La extensión puede instalarse en el sistema colocando el `jar` que la contenga en el directorio `jre/lib/ext` y añadiendo una entrada para el proveedor en el fichero `java.security`.

8 Interfaces de seguridad

En este apartado describiremos brevemente los paquetes de seguridad del JCA y el JCE. Necesariamente la presentación será breve, si se desea más información se puede consultar directamente la especificación de la Arquitecura Criptográfica de Java en `<http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html>` y el API del JDK 1.2 en `<http://java.sun.com/products/jdk/1.2/docs/api/>` y el API de la JCE en `<http://java.sun.com/security/JCE1.2/spec/apidoc/>`.

8.1 El paquete `java.security`

El paquete `java.security` consiste básicamente en clases abstractas e interfaces que encapsulan conceptos de seguridad como certificados, claves, resumenes de mensajes y firmas digitales.

En el JCA 1.1 los proveedores pueden implementar tres clases:

- `KeyPairGenerator`. Se emplea para crear claves públicas y privadas.
- `MessageDigest`. Prorciona la funcionalidad de algoritmos de resumen de mensajes como el MD5 y el SHA.
- `Signature`. Se emplea para el firmado digital de mensajes.

Una aplicación puede solicitar una implementación con el método `getInstance()`, como por ejemplo:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance ("DSA");
```

Y el sistema busca un proveedor que nos devuelva una implementación del mismo, como ya hemos explicado al comentar la arquitectura de seguridad de Java 2.

Si se desea una implementación específica se puede obtener llamando a `getInstance()` con más parámetros, por ejemplo si deseamos el algoritmo DSA del proveedor ITI haremos la siguiente llamada:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance ("DSA", "ITI");
```

El JCA 1.2 amplía considerablemente el número de clases que pueden implementar los proveedores:

- `AlgorithmParameterGenerator`. Se emplea para generar un conjunto de parametros adecuado para usar con algun algoritmo.
- `AlgorithmParameters`. Se emplea como representación opaca de los parametros de un algoritmo.
- `KeyFactory`. Las *factorias de claves* se emplean para convertir claves (claves criptográficas opacas de tipo `Key`) en especificaciones de claves (representaciones transparentes de las claves) y viceversa.
- `KeyStore`. Esta clase representa una colección de certificados y claves en memoria.
- `SecureRandom`. Esta clase proporciona un generador de números pseudo aleatorios criptográficamente seguro.

8.2 El paquete `java.security.spec`

El paquete `java.security.spec`, introducido en la versión 1.2 del JCA, añade interfaces y clases que describen formatos de especificación de claves. Estas clases permiten crear claves de seguridad en Java basadas en parámetros generados por herramientas externas al JDK.

Entre otras, existen clases para representar parámetros y claves de los algoritmos DSA y RSA y claves codificadas según la especificación del X.509.

8.3 El paquete `java.security.cert`

El paquete `java.security.cert` añade soporte para generar y usar certificados, así como leerlos de lugares como archivos JAR a través de las clases `JarURLConnection` y `JarEntry`. Ademas incluye clases e interfaces específicas para soportar certificados X.509.

Las clases más importantes incluidas en paquete son:

- `CertificateFactory`. Se emplea para generar certificados y listas de revocación (CRL).
- `Certificate`. Clase abstracta para gestionar certificados. Es una clase para agrupar certificados de diferentes formatos pero usos comunes importantes, como por ejemplo funciones de codificación y verificación o datos como claves públicas.
- `CRL`. Clase abstracta para gestionar distitos tipos de listas de revocación de certificados.
- `X509Certificate`. Clase abstracta para representar certificados X.509. Proporciona un método estandar para acceder a los atributos de un certificado X.509.
- `X509Extension`. Es una interfaz para las extensiones del formato X.509.
- `X509CRL`. Clase abstracta para una lista de revocación de certificados X.509.
- `X509CRLEntry`. Es una clase abstracta para las entradas de las listas de revocación.

8.4 El paquete `java.security.interfaces`

La versón 1.1 del JCA incluía en este paquete interfaces para usar el algoritmo DSA. El JCA 1.2 amplió la clase para incluir interfaces para el algoritmo RSA.

Las interfaces incluidas en paquete son:

- `DSAKey`. Interfaz para una clave DSA pública o privada.
- `DSAKeyGenerator`. Interfaz para un objeto capaz de generar pares de claves DSA.
- `DSAKey`. Interfaz para una clave DSA pública o privada.
- `DSAParams`. Interfaz para un conjunto de parámetros específicos del claves de tipo DSA.
- `DSAPrivateKey`. Interfaz para una clave privada DSA.
- `DSAPublicKey`. Interfaz para una clave pública DSA.
- `RSAPrivateCrtKey`. Interfaz para una clave privada RSA, como se define en el estándar PKCS#1, usando valores de información del Teorema Chino del Resto (CRT).
- `RSAPrivateKey`. Interfaz para una clave privada RSA.
- `RSAPublicKey`. Interfaz para una clave pública RSA.

8.5 El paquete `java.security.acl`

El paquete `java.security.acl` define soporte para listas de control de acceso que se pueden emplear para restringir el acceso a recursos de cualquier manera deseada. El paquete consta de interfaces y excepciones. Las implementaciones actuales en el JDK 1.1 de SUN están en el paquete `sun.security.acl`.

En el JDK 1.2 las clases de este paquete han sido reemplazadas por clases del paquete `java.security` como la clase `Permision`.

8.6 El paquete `javax.crypto`

Proporciona las clases e interfaces para realizar operaciones criptográficas. Las operaciones criptográficas definidas en este paquete incluyen encriptación, generación de claves y acuerdo de claves y generación de códigos de autentificación de mensajes (MAC).

El soporte para encriptación incluye algoritmos de cifrado simétricos, asimétricos, por bloques y de flujo.

La mayoría de clases incluidas en este paquete se basan en proveedores, las clases sólo definen el API.

Las clases más importantes incluidas en paquete son:

- `Cipher`. Repesenta un algoritmo de cifrado.
- `KeyAgreement`. Proporciona la funcionalidad de un protocolo de acuerdo o intercambio de claves.
- `KeyGenerator`. Propociona las funciones de un generador de claves simétricas.
- `Mac`. Proporciona las funciones de un generador de códigos de autentificación de mensajes.
- `SecretKeyFactory`. Representa un factoría de claves secretas.

8.7 El paquete `javax.crypto.spec`

Proporciona clases e interfaces para especificaciones de claves y parámetros de algoritmos.

Las especificaciones de claves son representaciones transparentes de las claves. Las claves se pueden especificar de un modo específico del algoritmo o en un formato de codificación independiente del algoritmo, como el ASN.1.

Este paquete contiene especificaciones de claves para claves Diffie-Hellman publicas y privadas, claves DES, Triple DES y claves secretas PBE.

La especificación de parámetros de un algoritmo son representaciones transparentes del conjunto de parámetros empleados con el algoritmo. Este paqute contiene especificaciones de parámetros para los algoritmos Diffie-Hellman, DES, Triple DES, PBE, RC2 y RC5.

8.8 El paquete `javax.crypto.interfaces`

Proporciona interfaces para claves Diffie-Hellman como se definen en el PKCS #3 de los laboratorios RSA.

Las interfaces incluidas en este paquete son:

- `DHKey`. Interfaz con una clave Diffie-Hellman.
- `DHPrivateKey`. Interfaz con una clave privada Diffie-Hellman.
- `DHPublicKey`. Interfaz con una clave pública Diffie-Hellman.

9 Extensión de Sockets Seguros de Java (JSSE)

En este apartado presentaremos la *extensión de sockets seguros de Java.*

La documentación del JSSE de SUN se puede encontrar en <http://java.sun.com/products/jsse/>

.

10 Servicio de Autentificación y Autorización de Java (JAAS)

En este apartado presentaremos el *servicio de autentificación y autorización de Java.*

La documentación del JAAS se puede encontrar en <http://java.sun.com/products/jaas/>.

11 Referencias

1. [EFF98]

Electronic Frontier Foundation.
Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design .
O'Reilly & Associates, 1998.

2. [FEGHHI99]

Feghhi, Jalal; Feghhi, Jalil; Williams, Peter.
Digital Certificates: applied Internet Security.
Addison Wesley, 1999.

3. [DAGEFO99]

Dageforde, Mary.
The Java Tutorial: Security in JDK 1.2.
24 Noviembre, 1999, <http://java.sun.com/docs/books/tutorial/security1.2/>.

4. [GONGLI99]

Gong, Li.
Java^{TM} Security Architecture (JDK 1.2).
2 Octubre, 1998, <ftp://ftp.javasoft.com/docs/jdk1.2/security-spec.pdf>.

5. [RFC_2246]

Dierks, T.; Allen, C..
RFC 2246: The TSL Protocol Version 1.0.
Enero 1999, <http://sunsite.org.uk/rfc/rfc2246.txt>.

6. [RSALab98]

RSA Data Security, Inc.
RSA Laboratories' Frequently Asked Questions About Today's Cryptography, v 4.0.
1998, <http://www.rsa.com/rsalabs/faq/>.

7. [SCHNEI96]

Schneier, Bruce.
Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd. Edition.
John Wiley & Sons, 1996.