



Proyecto cofinanciado por el Ministerio de Industria, Turismo y Comercio dentro del Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica 2004-2007 en el Programa Nacional de Tecnologías de Servicios de la Sociedad de la Información y Convocatoria Software de Código Abierto y Publicaciones en Internet.



Proyecto cofinanciado por el Fondo Europeo de Desarrollo Regional (FEDER)



Proyecto cofinanciado por el Gobierno del Principado de Asturias a través del IDEPA en su Programa de Innovación Empresarial IN+NOVA del año 2006.

in+nova

PROGRAMA DE INNOVACIÓN EMPRESARIAL



Proyecto cofinanciado por GADD-Grupo Meana, S.A.



PROYECTO

Open CERTIAC

Software Abierto de Certificación Electrónica, Registro Telemático e Información y Atención Ciudadana

SOLUCIONES J2SE DE FIRMA DIGITAL PARA OPENCERTIAC

Versión 1.2

01-06-2007



CONTROL DE CAMBIOS

Versión	Fecha	Autor	Cambios realizados
1.1	27-04-2007	GADD	Versión inicial del documento



ESCENARIOS	4
1. Infraestructura JCA	4
1.1 Providers	4
1.2 Extensión JCE	6
1.3 Conceptos	7
1.4. Manejo de claves	10
1.5. Resumen de conceptos criptográficos	11
2. Firma digital con Mozilla/Firefox	11
2.1. Repositorio de certificados NSS	12
2.2. Mozilla JSS	13
3. Soluciones de firma para Mozilla/Firefox	15
3.1. Plugin de firma	15
3.2. Applet Java de firma	16
4. SmartCards	17
4.1. Acceso a una SmartCard	17
4.2. Acceso al repositorio de claves	20
4.3. Obtener certificados y claves privadas	20
4.4. Firmar con una Smart Card	21
SOLUCIONES JAVA PARA PKIX	22
5. Bouncy Castle	22
5.1. Provider Bouncy Castle	22
5.2. Certificados	22
5.3. Key Stores	23
5.4. PKCS#7	23
6.1. Manual de Usuario	24
6.2. Manual de programación (API)	24



ESCENARIOS

1. Infraestructura JCA

JCA es el nombre que recibe la infraestructura criptográfica de Java™. La JCA o *Java Cryptography Architecture* es un marco de trabajo que proporciona el acceso y desarrollo de aplicaciones criptográficas en la plataforma Java™. En las primeras versiones, sólo cubría funcionalidades de firma y huella (*message digest*). Actualmente, Java 2 SDK engloba además la infraestructura de manejo de certificados X.509 v3.

La organización de esta infraestructura es un proceso continuo desde las primeras versiones de Java. El enfoque ha ido cambiando hacia una arquitectura más configurable y flexible, basada en la idea de *provider*, que permite implementaciones múltiples e interoperables. Aún así, La situación actual puede ser bastante oscura para quien se inicia en la programación de servicios criptográficos.

La documentación *on-line* ¹ de Sun acerca de la JCA es tan profunda como poco clarificadora. En esta sección se muestran los aspectos más relevantes de dicha documentación, intentando ilustrar los puntos más importantes con ejemplos reales de implementación, detalle que se echa en falta en la documentación oficial.

1.1 Providers

La *Cryptography Security Provider* (que llamaremos simplemente *provider* en este documento) se refiere a un conjunto de paquetes que ofrecen una implementación concreta a los interfaces criptográficos de la API de seguridad de Java.

Por ejemplo, en la JDK 1.1 un *provider* podía contener una implementación de uno o más algoritmos de firma, algoritmos de *digest* y generación de claves. La versión Java 2 SDK añade cinco tipos de servicios más: factoría de claves (*key factories*), creación y manejo de repositorios de claves o *key stores*, manejo de parámetros de los algoritmos y factorías de certificados. Esto incluye también un *provider* para la generación de números aleatorios, cuyos algoritmos estaban “*hard-coded*” en las implementaciones previas de la JDK.

La versión del *runtime* de Java de Sun viene con un *provider* por defecto, llamado SUN. Los nuevos *providers* pueden ser añadidos estática o dinámicamente.

Por ejemplo, para añadir un *provider* de *IAIK*² de manera dinámica se puede utilizar el siguiente código:

```
[...] Security.addProvider(new IAIK());
```

¹<http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html>

²<http://jce.iaik.tugraz.at/>



En todo momento existe una lista de los providers de seguridad instalados en el entorno de la máquina virtual de Java. Por ejemplo, podríamos buscar si existe un provider IAIK instalado y borrarlo:



```
String providerName = (new IAIK()).getName();
// Recuperar la lista de Providers instalados Provider[] aprovider =
Security.getProviders();
for (int i = 0; i < aprovider.length; i++) Provider provider =
aprovider[i]; if (provider.getName().equals(providerName)) // Borramos el
provider Security.removeProvider(providerName);
Un provider PKCS#11, por ejemplo el de una tarjeta SmartCard (Ver
Sección 4), puede estar definido en un fichero de configuración. En ese caso
se podría añadir dinámicamente de la siguiente manera:
String configName = "C:
smartcards
config
pkcs11.cfg" Provider p; p = new sun.security.pkcs11.SunPKCS11(configName);
Security.addProvider(p);
```

Se hablará más sobre el fichero *pkcs11.cfg* en la sección 1; las líneas pertinentes de ese fichero tendrían el aspecto siguiente:

```
name = SmartCard library = c:32201n.dll
```

Para añadir un provider estáticamente hay que editar el fichero de propiedades del Java Security (`$JAVA_HOME/lib/security/java.security`). Por ejemplo:

```
configuration for security providers 1-6 omitted
security.provider.7=sun.security.pkcs11.SunPKCS11 C:11.cfg
```

1.2 Extensión JCE

La extensión criptográfica de Java™ (JCE, *Java Cryptography Extension*) provee un marco de trabajo e implementaciones para cifrado, generación y aceptación de claves, y soporte para algoritmos MAC (*Message Authentication Code*). Dicho soporte incluye algoritmos para cifrados simétricos, asimétricos, de bloque y de flujo. JCE ha sido integrada a la SDK desde la versión de Java™1.4.

La API de JCE engloba:

- Cifrado simétrico de datos, como DES, RC2, e IDEA.
- Cifrado simétrico en flujo, como RC4.
- Cifrado asimétrico, como RSA.
- Cifrado basado en palabra clave, PBA.
- Aceptación de claves.
- Códigos de aceptación de message, MAC.

J2SE 5 incluye un *provider* JCE estándar, llamado SUNJCE, que tiene los siguientes servicios criptográficos:

- Una implementación de DES, Triple DES, y Blowfish.



- Generadores de claves para claves válidas para algoritmos DES, Triple DES, Blowfish, HMAC-MD5 y HMAC-SHA1.
- Una implementación de MD5 con DES-CBC y codificación basada en *password* (PBE), definido en PKCS#5.
- Factorías de claves privadas, proporcionando conversiones bidireccionales entre DES, Triple DES y objetos PBE opacos y representaciones transparentes de las claves subyacentes.
- Una implementación del algoritmo de aceptación de clave Diffie-Hellman entre dos o más participantes.
- Generador de pares de claves pública/privada para Diffie-Hellman.
- Generador de parámetros para el algoritmo de Diffie-Hellman.
- Factoría de claves para Diffie-Hellman con conversiones bidireccionales entre objetos de clave opacos y representaciones transparentes de las claves subyacentes.
- Gestores de parámetros para los algoritmos de Diffie-Hellman, DES, Triple DES, Blowfish, y PBE.
- Una implementación del *hash* de claves para HMAC-MD5 y HMAC-SHA1, definidos en RFC 2104.
- Una implementación del algoritmo de relleno descrito en PKCS#5.
- Una implementación de un repositorio de claves, *keystore* para el tipo de repositorio propio llamado JCEKS.

La JCE incluida en la JDK contiene dos componentes software:

- El marco de trabajo que define y soporta los servicios que son implementados por los *providers*. Este *framework* incluye todo en el paquete `javax.crypto`.
- Un provider llamado SUNJCE.

1.3 Conceptos

A continuación, presentaremos una serie de conceptos básicos que se manejan en el API JCE.

1.3.1 Clases *Engine* y algoritmos

Una clase *engine* o generadora define un servicio criptográfico de un modo abstracto, es decir, sin una implementación concreta.

Un servicio criptográfico siempre va asociado a un algoritmo o tipo particular y, o bien proporciona operaciones criptográficas (por ejemplo para firmas digitales o *message digests*), genera o proporciona material criptográfico (claves o parámetros) requerido para operaciones criptográficas, o bien genera objetos de datos (repositorios de claves y certificados) que encapsulan claves criptográficas de manera segura.

Por ejemplo, dos clases *engine* son `Signature` y `KeyFactory`. La primera proporciona el acceso a las funcionalidades de un algoritmo específico de firma digital; su método



`getInstance` permite instanciar un objeto especificando qué algoritmo concreto implementa. Una `KeyFactory` DSA proporciona una clave DSA privada o pública en un formato usable para los métodos `initSign` o `initVerify` de un objeto `DSA Signature`.

A continuación se presenta un ejemplo de cómo utilizar una `KeyFactory` para instanciar una clave pública DSA a partir de su codificación X509. Suponemos dos individuos, María y Pedro, que quieren intercambiar información, y María ha recibido una firma digital de Pedro. Pedro envió también su clave pública para verificar su firma (escenario habitual cuando se envía un certificado de usuario). María realiza las siguientes acciones:

```
// Crear unas especificaciones de clave a partir // de la codificación que
ha enviado Pedro. X509EncodedKeySpec pedroPubKeySpec; pedroPubKeySpec = new
X509EncodedKeySpec(pedroEncodedPubKey);
// Instanciar una factoría de claves DSA. KeyFactory keyFactory =
KeyFactory.getInstance("DSA");
// Generamos el objeto clave pública con la factoría // y el codificado
enviado por Pedro. PublicKey pedroPubKey; pedroPubKey =
keyFactory.generatePublic(pedroPubKeySpec);
// Instanciamos el generador Signature para firmas DSA Signature sig =
Signature.getInstance("DSA");
// Verificamos los datos firmados por Pedro. sig.initVerify(pedroPubKey);
sig.update(data); sig.verify(signature);
```

La *Java Cryptography Architecture* engloba las clases del paquete `Security` de Java 2 SDK relacionadas con criptografía, incluyendo las clases *engine*. Como se muestra en el ejemplo de María y Pedro, los usuarios del API solicitan y utilizan instancias de las clases *engine* para manejar las correspondientes operaciones. A continuación, se listan las clases *engine* definidas en Java 2 SDK:

- `MessageDigest`: Utilizado para calcular el resumen del mensaje (*hash*) para los datos especificados.
- `Signature`: Usada para firmar datos y verificar firmas digitales.
- `KeyPairGenerator`: Utilizada para generar pares de claves (pública y privada) adecuadas para cada algoritmo asimétrico.
- `KeyFactory`: Utilizada para convertir claves criptográficas opacas de tipo `Key` a especificaciones de clave (representaciones transparentes del material subyacente) y viceversa.
- `CertificateFactory`: Usado para crear certificados de clave pública y listas de revocación (CRLs).
- `KeyStore`: Usado para manejar un repositorio de claves o *key store*. Un repositorio de claves es una base de datos de claves. Las claves privadas en un repositorio tienen asociadas una cadena de certificados, para autenticar la clave pública correspondiente. También contiene certificados de autoridades de confianza.
- `AlgorithmParameters`: Para manejar los parámetros de un algoritmo determinado, incluyendo codificación y decodificación de parámetros.



- `AlgorithmParameterGenerator`: Utilizado para generar un conjunto de parámetros utilizables con un algoritmo.
- `SecureRandom`: Usado para la generación de números pseudo-aleatorios.
- A partir de la versión Java 1.4 se incluyen además:
- `CertPathBuilder`: Utilizado para construir cadenas de certificados (también llamadas rutas de certificado).
- `CertPathValidator`: Utilizado para validar cadenas de certificados.
- `CertStore`: Usado para recuperar certificados y CRLs de un repositorio.

Hay que recordar que un **engine** o **generador** crea objetos con contenidos nuevos, mientras que una **factoría** crea objetos a partir de material existente (como por ejemplo, una codificación).

Una clase *engine* proporciona un interfaz a las funciones de un tipo específico de servicio criptográfico (independientemente de cualquier algoritmo criptográfico). Define los métodos del API que permiten a las aplicaciones acceder al servicio criptográfico que provee. Por ejemplo, la clase `Signature` permite el acceso a las funciones de firma digital. La implementación provista por la subclase `SignatureSpi` podría ser para un algoritmo de firma concreto, como SHA-1 con DSA, SHA-1 con RSA o MD5 con RSA.

El siguiente código ilustra la creación de una firma SHA-1 con DSA.

```
/* Generate a DSA signature */
try
/* Generar un par de claves */
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
KeyPair pair = keyGen.generateKeyPair(); PrivateKey priv =
pair.getPrivate(); PublicKey pub = pair.getPublic();
/* * Crear un objeto Signature e inicializarlo con * la clave privada. */
Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
dsa.initSign(priv);
/* Actualizar y firmar los datos */
FileInputStream fis = new FileInputStream(args[0]); BufferedInputStream
bufin = new BufferedInputStream(fis); byte[] buffer = new byte[1024]; int
len; while (bufin.available() != 0) len = bufin.read(buffer);
dsa.update(buffer, 0, len);
bufin.close();
/* * Todos los datos han sido leídos, generamos una * firma para ellos. */
byte[] realSig = dsa.sign();
/* Guardamos la firma en un fichero */ FileOutputStream sigfos = new
FileOutputStream("sig"); sigfos.write(realSig);
sigfos.close();
/* Guardamos la firma en fichero */ byte[] key = pub.getEncoded();
FileOutputStream keyfos = new FileOutputStream("suepk"); keyfos.write(key);
keyfos.close();
catch (Exception e) System.err.println( "Caught exception " +
e.toString());
;
```



NOTA: La firma generada en el ejemplo anterior no tiene ningún envoltorio concreto, como PKCS#7. Para generar ficheros de firma PKCS#7, se recomienda el uso de algún paquete que proporcione esta funcionalidad. Como veremos en la sección 5, la librería CMS de *BouncyCastle* proporciona métodos para crear firmas y escribirlas a disco en formato PKCS#7.

Los interfaces proporcionados por una clase *engine* son implementados en términos del SPI (*Service Provider Interface*). Es decir, cada clase *engine* posee su correspondiente clase abstracta SPI, la cual define qué métodos y servicios han de ser implementados. El nombre de la clase SPI es el de la clase *engine* seguido de *Spi*.

Nótese que todas las clases SPI son abstractas, por lo que para proporcionar cada tipo de servicio, para un algoritmo específico, el *provider* tiene que derivar la clase SPI e implementar todos los métodos abstractos.

Para cada clase *engine* del API, se puede solicitar e instanciar una implementación concreta llamando a un método factoría de la clase. Éste método es estático y devuelve una instancia de la clase. Como hemos visto en los ejemplos, para conseguir un objeto del tipo *Signature* haremos lo siguiente: Llamar al método de factoría `getInstance` especificando el tipo de algoritmo (por ejemplo "SHA1WithDSA") y opcionalmente, el nombre de un *provider* o una clase *Provider*. Si no se especifica ningún *provider*, `getInstance` buscará entre los *providers* registrados, en orden de preferencia, una implementación del algoritmo indicado.

1.4. Manejo de claves

Para el manejo de claves y certificados se puede utilizar una base de datos llamada *keystore* o repositorio de claves. Las aplicaciones que necesitan firmar datos o autenticar, precisan de un tal *keystore*.

Estas aplicaciones pueden acceder al repositorio implementando la clase *KeyStore* definida en el paquete `java.security`. Sun provee una implementación por defecto de un repositorio de claves propietario, almacenado en un fichero y llamado JKS.

Las aplicaciones pueden elegir distintos tipos de repositorios a través de los distintos *providers*, utilizando el método `getInstance` de la clase *KeyStore*.

Por ejemplo, para instanciar un *KeyStore* JKS de SUN:

```
KeyStore ks = KeyStore.getInstance("JKS");
```

En este caso el sistema buscará *en el entorno* un *provider* que implemente el acceso al repositorio. Si hay más de una, cogerá la primera por orden de preferencia. También se puede especificar de forma explícita:



```
KeyStore ks = KeyStore.getInstance("JKS", "SUN");
```

En este caso el sistema buscará una implementación *en el paquete indicado*, elevando una excepción si no existe.

NOTA: Antes de acceder al repositorio, hay que cargarlo utilizando el método `load` de la clase `KeyStore`.

1.5. Resumen de conceptos criptográficos

Para acabar esta sección, definiremos a grandes rasgos los conceptos básicos de criptografía.

1.5.1 Cifrado y descifrado

Cifrado es el proceso de tomar unos datos (llamados “texto en claro”), una cadena de texto corta (clave) y producir datos (llamados “texto cifrado”) ilegible para quien no conozca la clave. Descifrado es el proceso inverso: tomar el texto cifrado y la clave para conseguir el texto en claro original.

1.5.2. Cifrado basado en password

La *Password Based Encryption* (PBE) deriva una clave de cifrado desde una palabra clave o *password*. Para hacer muy difícil recuperar el *password* para un supuesto atacante, la mayor parte de las implementaciones de PBE utilizan un número aleatorio, llamado *sal*, para crear la clave.

1.5.3. Cifrador

El cifrado y descifrado se realizan a través de un cifrador. Un cifrador es un objeto capaz de realizar cifrados y descifrados según un algoritmo.

1.5.4. Aceptación de claves

O *Key Agreement*. Es un protocolo por el cual dos o más participantes pueden establecer las mismas claves criptográficas, sin tener que intercambiar información secreta.

1.5.5. Código de autenticación de mensaje

O *Message Authentication Code*. Un MAC proporciona una manera, basada en clave privada, de verificar la integridad de la información transmitida o almacenada en un medio no confiable. Típicamente, estos códigos se utilizan entre dos partes que comparten una clave privada para validar la información transmitida entre ellas.

2. Firma digital con Mozilla/Firefox



La motivación principal de este proyecto es proveer a los usuarios del navegador Firefox o de la suite Mozilla herramientas de firma digital bajo Microsoft Windows, de manera equivalente a los proporcionados a través de la librería CAPICOM de Microsoft.

Por ello, será bueno introducir en esta sección algunos conceptos y detalles sobre la arquitectura de la plataforma Mozilla. Tras la introducción al sistema de certificados de esta arquitectura, se verán las posibles soluciones para cumplir los objetivos, con sus ventajas e inconvenientes.

2.1. Repositorio de certificados NSS

NSS (Network Security Services, Servicios de Seguridad de Red) es la librería de Mozilla, inicialmente escrita por ingenieros de Netscape, que implementa las herramientas de seguridad. Es una librería de código abierto, con las ventajas ampliamente conocidas de este tipo de licencias para código relacionado con la seguridad. Además, está avalada por la experiencia de Netscape, inventores del protocolo SSL (Secure Sockets Layer) y pioneros en la utilización de certificados X.509 en una aplicación comercial.

Su funcionalidad es amplia. De hecho, realmente se trata de un conjunto de librerías que implementan un gran número de estándares de seguridad, por ejemplo SSL v2 y v3, TLS, PKCS#5, PKCS#7, PKCS#11, PKCS#12, S/MIME y certificados X.509 v3³. La librería forma parte del núcleo de los clientes Mozilla, incluyendo la Suite Mozilla, Firefox y Thunderbird. Así mismo, otras muchas aplicaciones se apoyan en ella, por ejemplo OpenOffice, Evolution. Incluso la propia Sun la utiliza en sus soluciones de servidor.

La NSS tiene su propio repositorio de certificados y claves. Utiliza dos ficheros, `cert7.db` y `key3.db` como base de datos (los números denotan la versión del formato, por lo que podrían variar). A pesar de ser Mozilla un software de código abierto, el formato de estas bases de datos nunca ha sido liberado, aunque se conoce que emplean Berkeley DB v1.5, que es extremadamente obsoleto. Desde Mozilla se aducen varias razones:

- Poder cambiar el formato de la base de datos sin arrastrar la carga de soporte para aplicaciones externas.
- Las leyes de Estados Unidos sobre exportación de material de seguridad les impiden revelar cómo se realizan ciertas operaciones.
- El código actual es frágil respecto a corrupciones de la base de datos.
- Desaconsejan el uso de programas que utilicen bases de datos de certificados y claves, tal y como hacen ellos. El desarrollo se debería orientar a tokens criptográficos, lo que encaja mejor con el enfoque de las SmartCards.

Por esta última razón están migrando hacia un modelo de objetos PKCS#11.

³Un listado detallado de los estándares soportados se encuentra en <http://www.mozilla.org/projects/security/pki/nss/overview.html>



2.2. Mozilla JSS

JSS es un interfaz Java desarrollado por Mozilla para su librería NSS, soportando *casi* todos los estándares de seguridad y cifrado definidos en la NSS. También proporciona un interfaz para tipos ASN.1 y codificación BER/DER.

Una clase fundamental en el paquete criptográfico JSS es el `CryptoManager`, que se utiliza para inicializar todo el subsistema y para manejar certificados, claves y *tokens*. Su inicialización se realiza a través de métodos estáticos, y debe de realizarse antes de crear ninguna instancia.

```
[...] private static final String dbdir;  
dbdir="/home/luis/.mozilla/firefox/ujdlv86z.default/";  
[...]  
CryptoManager.initialize(dbdir); cm = CryptoManager.getInstance();
```



Como se ve, el `CryptoManager` es inicializado *antes* de su instanciación con el directorio donde se encuentra el perfil de usuario de Firefox. Es uno de los primeros problemas que se encuentran al utilizar JSS, ya que dicha ruta (que contiene los ficheros `certX.db` y `keyX.db`) depende de cada cliente, usuario y plataforma.

2.2.1. Provider JSS para JCA

JSS 3.2 implementa varios algoritmos JCE. Todas las operaciones criptográficas en JSS se realizan con un token PKCS#11 particular, implementado en software o en hardware. No existe una forma clara de definir este token a través del API JCA. Por defecto, el *provider* JSS realiza todas las operaciones menos `MessageDigest` en el Token de Almacenamiento de Claves Interno, un token software incluido en JSS/NSS. Las operaciones `MessageDigest` se realizan por defecto en el Token Criptográfico Interno, otro token software incluido en JSS. No hay ninguna buena razón para ello, pero se realiza así debido a una particularidad de la NSS.

Si se quiere utilizar un token diferente, hay que establecerlo con `CryptoManager.setThreadToken()`. Haciendo esto, el *provider* JCA de JSS utilizará dicho token en el hilo actual. Al llamar al método `getInstance` en una clase JCA, el *provider* JSS tomará el token por defecto para este hilo (realizando una llamada a `CryptoManager.getThreadToken()`).

Para aclarar esto, se muestra el siguiente ejemplo de código, donde se crean dos tokens que son utilizados para varias operaciones JCA.

```
// Obtener tokens PKCS 11 CryptoManager manager =  
CryptoManager.getInstance(); CryptoToken tokenA =  
manager.getTokenByName("TokenA"); CryptoToken tokenB =  
manager.getTokenByName("TokenB");  
// Crear un KeyPairGenerator RSA con el TokenA  
manager.setThreadToken(tokenA); KeyPairGenerator rsaKpg =  
KeyPairGenerator.getInstance("Mozilla-JSS", "RSA");  
// Crear un KeyPairGenerator RSA con el TokenB  
manager.setThreadToken(tokenB); KeyPairGenerator dsaKpg =  
KeyPairGenerator.getInstance("Mozilla-JSS", "DSA");  
// Generar un KeyPair RSA. Esto sucede en // el TokenA, porque fue el token  
por defecto // para el hilo, cuando se creó rsaKpg. rsaKpg.initialize(1024);  
KeyPair rsaPair = rsaKpg.generateKeyPair();  
// Generar un KeyPair RSA. Esto sucede en // el TokenB, porque // fue el  
token por defecto // para el hilo, cuando se creó dsaKpg.  
dsaKpg.initialize(1024); KeyPair dsaPair = dsaKpg.generateKeyPair();
```

La JSS soporta las siguientes clases:

- Cipher
- DSAPrivateKey
- DSAPublicKey
- KeyFactory



- KeyGenerator
- KeyPairGenerator
- Mac
- MessageDigest
- RSAPrivateKey
- RSAPublicKey
- SecretKeyFactory
- SecretKey
- SecureRandom
- Signature

Pero **no** soporta la clase `KeyStore`.

3. Soluciones de firma para Mozilla/Firefox

A la hora de implementar una solución para la firma digital en Firefox, se plantean las dos soluciones expuestas a continuación.

Desde el punto de vista del programador, el uso de JSS es el descrito hasta el momento. Primero, hay que inicializar el objeto `CryptoManager`, instanciarlo y, a partir de ahí, utilizar las funciones que provee.

El mayor problema del trabajo con JSS es que su gran dependencia de la configuración del sistema de la aplicación cliente.

3.1. Plugin de firma

Una de las características más potentes del navegador Firefox es su capacidad de expandir la funcionalidad utilizando plug-ins o extensiones. Por lo tanto, se podría pensar en realizar una extensión que accediera a las funciones criptográficas de Mozilla a través de su librería NSS.

Esta solución traería más inconvenientes que ventajas, ya que sería totalmente dependiente de la versión del navegador instalado. Con las sucesivas actualizaciones del navegador habría que actualizar también la extensión, por lo que su mantenimiento se extendería en el tiempo indefinidamente. Además, el usuario debería aceptar la instalación de un programa externo en su sistema, lo que en ocasiones no es recomendable, sobre cuando se trata de manejar datos delicados.

Como ventaja se podría citar el total acceso a las funciones de la librería interna NSS, sin tener que utilizar envoltorios para otros lenguajes, como es el caso de JSS. Sin embargo, desde el punto de vista del desarrollador, esto es un proceso costoso ya que hay que conocer tanto el API para el desarrollo de extensiones como el API para la NSS, además de tener conocimientos de XUL (por ejemplo) para dotar de una interfaz a la extensión.



3.2. Applet Java de firma

La otra solución para realizar una firma digital desde un cliente, utilizando Firefox, es la creación de un applet que se ejecute en el navegador y que implemente las funcionalidades deseadas. Dentro de este enfoque se tiene la posibilidad de utilizar o no la librería JSS de Mozilla. En las siguientes secciones se llevará a cabo un análisis de ambas posibilidades.

3.2.1. Applet Java utilizando JSS

A pesar de las aparentes facilidades que parece proporcionar la librería JSS/NSS, un análisis más profundo pone de manifiesto las debilidades de esta solución, sobre todo en términos de estabilidad. La propia JSS en sí parece no tener claro el camino a seguir para ajustarse a las especificaciones de la arquitectura JCA, sin implementación de una clase *KeyStore* estándar y con un formato cerrado para los repositorios de claves.

A todas las dificultades referidas al desarrollo con la librería JSS, hay que añadir la alta dependencia de una correcta instalación por parte del cliente, tarea que no es para nada sencilla. La JSS depende de varias librerías (dll's en Windows y .so en Linux) que no se incluyen necesariamente con la distribución de la solución NSS, (ss3.dll, libnspr4.dll, libplc4.dll y libplds4.dll). Además, la versión de estas librerías y de la propia JSS depende de la versión de desarrollo y, en el caso de Linux, del propio kernel.

Para ilustrar la dificultad presentada al usuario con esta instalación, a continuación se presenta un ejemplo clásico de instrucciones de instalación para un applet basado en JSS sería:

- Proporcionar un archivo con las versiones adecuadas de las librerías, incluyendo la JSS.
- Copiar los ficheros ss3.dll, libnspr4.dll, libplc4.dll y libplds4.dll en el directorio de instalación, por ejemplo C:/Archivos de Programa/Firefox. Es posible que sea necesario sobrescribir alguno.
- Copiar el fichero jar (jss*.jar) en el directorio /lib/ext de todos los entornos de máquina virtual de Java instalados. Típicamente se hallan bajo C:/Archivos de Programa/Java.
- Reiniciar el navegador.

En resumen, el soporte para JSS acumula dificultades importantes para el despliegue si (como se ha tomado como hipótesis), el proceso de firma con OpenCERTIAC debe ser para el usuario sin formación completamente transparente y sin esfuerzo.

3.2.2. Applet Java sin JSS

El uso de un applet que no utilice JSS soluciona los problemas que dependen de dicha librería y proporciona alguna ventaja extra, como por ejemplo, que el applet puede funcionar en cualquier otro navegador con soporte de applets Java, como es el caso de Internet Explorer.

Como contrapartida, no es posible el acceso a ficheros .db de Firefox, es decir, a repositorios de claves de estos navegadores. Esta funcionalidad no es fundamental, ya que el proyecto está orientado a repositorios de token (tarjetas inteligentes, por ejemplo). En cualquier caso, si el



certificado de usuario se encontrase en un repositorio de Mozilla, éste podría ser exportado igualmente a un formato más estándar, como PKCS#12, a través de una simple operación con el navegador. Esta operación se habría de realizar una única vez y el certificado quedaría disponible para posteriores usos.

No obstante, se recalca la orientación de este proyecto hacia repositorios de certificados en tarjetas inteligentes. De este modo, el desarrollo del plugin no tendría más dependencias que el de una aplicación *stand alone*, y con el uso de librerías como las *Bouncy Castle* o la propia *jeFirma* se simplifica mucho el desarrollo. Con este enfoque, las dependencias del applet final van a ser ficheros *jar* que pueden ser descargados on-line por el navegador cliente, de manera transparente, en el momento de ejecutar el applet.

4. SmartCards

Las tarjetas inteligentes o *SmartCards* son tarjetas magnéticas o con chip que almacenan y protegen información delicada (claves privadas, certificados y otra) de una manera mucho más segura que un simple fichero.

La mayor parte de las *smart cards* modernas tienen un *criptoprocesador* y un área de datos protegida que no puede copiarse. De hecho sólo el criptoprocesador, que es el encargado de cifrar y firmar los datos, es quien tiene acceso al área protegida de la tarjeta. Los desarrolladores no pueden extraer la clave privada de la tarjeta, sino que sólo pueden utilizar los servicios de su criptoprocesador.

Es imposible extraer la información protegida de la tarjeta, pero aunque así fuera, se requiere la presencia física de la tarjeta para poder acceder a dicha información. Por ello, en caso de pérdida o robo, su propietario puede informar a la autoridad correspondiente para que revoque la validez de la tarjeta.

Si se utiliza un repositorio de certificados PKCS#12, almacenado en un fichero (normalmente con la extensión PFX), existe el riesgo de que alguien copie el fichero y obtenga el acceso a su password. Podría obtenerse, por ejemplo, utilizando un *keylogger*⁴, un troyano o cualquier otra herramienta al efecto. Sin embargo, al utilizar una tarjeta inteligente con un *criptoprocesador*, no es posible extraer y copiar las claves privadas debido a las propias limitaciones hardware. La única posibilidad es robar físicamente la propia tarjeta. Aún así, el riesgo de que eso ocurra es muy menor que en comparación con copiar un fichero.

4.1. Acceso a una SmartCard

Como se ha mencionado anteriormente, las tarjetas inteligentes tienen su propio criptoprocesador, su propio sistema operativo, memoria protegida y su propio sistema de ficheros. El acceso se produce a diferentes niveles y con distintos protocolos. El estándar ISO

⁴Un *keylogger* es un programa que almacena las pulsaciones de tecla sin el consentimiento del usuario



7816 especifica los componentes principales que ha de tener una tarjeta inteligente y define el acceso de bajo nivel a los mismos.

Para el acceso a más alto nivel se utiliza el estándar PKCS#11, que define el interfaz de la aplicación con el dispositivo criptográfico (llamado normalmente *token criptográfico*). El software que se distribuye con la tarjeta normalmente contiene una implementación de PKCS#11 específica para la tarjeta y su lector. Dicha implementación es normalmente una librería (.dll en Windows o un .so en Linux y UNIX) que se puede cargar dinámicamente desde la aplicación.

Por ejemplo, si se utiliza una tarjeta de la Fábrica Nacional de Moneda y Timbre, bajo Windows XP la librería que implementa PKCS#11 estará en el una librería dinámica de nombre `PkcsV2GK.dll`.

El estándar PKCS#11 no permite la extracción física de las claves privadas de la tarjeta, pero es posible utilizarlas para cifrar, descifrar o firmar datos. Cuando se realizan estas operaciones, el usuario debe introducir un código o PIN previamente, el cual protege el acceso a la tarjeta. PKCS#11 provee un interfaz para el acceso a las claves protegidas y a los certificados almacenados en la tarjeta. Por esa razón, las tarjetas inteligentes se manejan de una manera similar a los repositorios PKCS#12. Pero una tarjeta tiene muchas más funcionalidades que un repositorio PKCS#12 ya que puede, por ejemplo, cifrar, descifrar y firmar por hardware.

La plataforma Java 2 incluye soporte para tarjetas inteligentes. El acceso a las tarjetas se realiza a través del interfaz PKCS#11 y la interacción se lleva a cabo a través del *provider* `SunPKCS11`. Hay que recordar que ese provider ha de estar registrado en el entorno antes de ser usado. Como se ha dicho en la sección 1, el registro se puede realizar de forma dinámica o estática. Para el registro dinámico del *provider* `SunPKCS11`, se ha de instanciar la clase `sun.security.pkcs11.SunPKCS11` registrada en JCA. El nombre del fichero de configuración (véase nuevamente la sección 1) puede ser pasado como parámetro. Así sucede en el siguiente ejemplo:

```
String pkcs11ConfigFile = "c:\nsmartcards\nconfig\npkcs11.cfg"; Provider pkcs11Provider = new\nsun.security.pkcs11.SunPKCS11(pkcs11ConfigFile);\nSecurity.addProvider(pkcs11Provider);
```

El aspecto del fichero `pkcs11.cfg` es:

```
name = SmartCard library = c:32201n.dll
```

El fichero contiene dos configuraciones, para el nombre y para la librería. El valor de la propiedad `name` se utiliza como nombre para la instancia del *provider* PKCS#11 en la JCA. La propiedad `library` indica qué dll implementa el estándar PKCS#11 para esa tarjeta. Si se necesitara trabajar con varias tarjetas a la vez, el *provider* debería ser instanciado varias veces,



con un nombre diferente cada vez.

El siguiente ejemplo ilustra el registro dinámico de un *provider* PKCS#11 utilizando un *stream*:

```
String pkcs11config = "name = SmartCardn" + "library = c:32201n.dll"; byte[]  
pkcs11configBytes = pkcs11config.getBytes(); ByteArrayInputStream  
configStream = new ByteArrayInputStream(pkcs11configBytes); Provider  
pkcs11Provider = new sun.security.pkcs11.SunPKCS11(configStream);  
security.addProvider(pkcs11Provider);
```

En este ejemplo se ha utilizado un *stream* que lee las propiedades de la configuración desde una cadena de texto, en vez de utilizar un fichero.

Cuando no se necesita más el *provider* PKCS#11, se puede eliminar del entorno, liberando así los recursos utilizados:



```
Provider pkcs11Provider = ...; String pkcs11ProviderName =  
pkcs11Provider.getName(); Security.removeProvider(pkcs11ProviderName);
```

Algunas implementaciones necesitan que el *provider* sea eliminado después del proceso de firma. De otro modo, la sesión con la tarjeta inteligente se mantiene abierta y podría fallar un intento posterior de firmar. Otra forma de eliminar el provider es:

```
Security.removeProvider("SunPKCS11-SmartCard");
```

El nombre del provider se construye con el prefijo “SunPKCS11-”, seguido del nombre indicado en el fichero de configuración.

4.2. Acceso al repositorio de claves

Tras registrar y configurar correctamente el *provider* PKCS#11 de Sun, la tarjeta está lista para acceder a los certificados y claves. Esto es posible utilizando la clase estándar para el acceso de repositorios de claves, `java.security.KeyStore`. A continuación, se presenta un ejemplo para establecer un acceso seguro con el repositorio de claves protegido en la tarjeta:

```
char [] pin = '1', '2', '3', '4'; KeyStore smartCardKeyStore =  
KeyStore.getInstance("PKCS11"); smartCardKeyStore.load(null, pin);
```

Evidentemente, hay que proporcionar el PIN correcto para poder leer el repositorio.

4.3. Obtener certificados y claves privadas

Una vez establecido el acceso al repositorio de claves, tal y como se ha visto en la sección anterior, el acceso a los certificados de una tarjeta inteligente no difiere del de cualquiera otro repositorio. Todas las claves, certificados y cadenas de certificados están almacenados bajo un *alias* en el repositorio de la tarjeta. Los nombres de esos alias pueden ser obtenidos mediante un iterador.

A continuación se muestra un ejemplo en el que se muestran todos los certificados e información de sus claves privadas, para un repositorio concreto:

```
KeyStore keyStore = ...;  
Enumeration aliasesEnum = keyStore.aliases(); while  
(aliasesEnum.hasMoreElements()) String alias =  
(String)aliasesEnum.nextElement(); System.out.println("Alias: " + alias);  
X509Certificate cert = (X509Certificate) keyStore.getCertificate(alias);  
System.out.println("Certificate: " + cert); PrivateKey privateKey =  
(PrivateKey) keyStore.getKey(alias, null); System.out.println("Private key:  
" + privateKey);
```

Nótese que este ejemplo funcionaría con un `KeyStore` cualquiera, esté almacenado o no en una tarjeta inteligente. No se necesita ningún password para acceder a las claves privadas de



una tarjeta. Esto es así porque el código o PIN ya ha sido introducido con anterioridad durante la creación del objeto `KeyStore`. Por eso el segundo parámetro del método `getKey` (el `password`) es `null`. En un primer vistazo al código anterior puede parecer que se accede a las claves privadas, pero sólo se accede *indirectamente* para firmar, verificar firmas, cifrar o descifrar. En el ejemplo anterior, la clave privada **no se extrae**, sino que se obtiene un interfaz para acceder a ella.

4.4. Firmar con una Smart Card

Una vez que se obtiene el interfaz para una clave privada, podemos utilizarlo para firmar datos, como si se tratara de cualquier otro tipo de clave privada. La firma real se realiza calculando el valor de resumen o *hash* del documento a firmar, y enviando ese *hash* a la tarjeta, quien lo firmará con su criptoprocesador. Si la operación es correcta, la tarjeta devuelve la firma calculada. De esta forma, la clave privada está protegida de todo riesgo, ya que se mantiene en secreto, oculta en alguna parte de la tarjeta. A continuación se presenta un ejemplo que firma datos con un interfaz dado y una clave privada extraída de una tarjeta:

```
private static byte[] signDocument(byte[] aDocument, PrivateKey aPrivateKey)
throws GeneralSecurityException {
    Signature signatureAlgorithm =
    Signature.getInstance("SHA1withRSA");
    signatureAlgorithm.initSign(aPrivateKey);
    signatureAlgorithm.update(aDocument);
    byte[] digitalSignature =
    signatureAlgorithm.sign();
    return digitalSignature;
}
```



SOLUCIONES JAVA PARA PKIX

5. Bouncy Castle

El paquete *Bouncy Castle* (BC) es una implementación Java de algoritmos criptográficos. Está organizado de forma que su API sea adaptable para su uso bajo cualquier entorno (incluyendo el recientemente publicado J2ME) con la infraestructura adicional para ajustarse a los algoritmos del *framework* JCE. El paquete se distribuye bajo licencia MIT X Consortium.⁵

El paquete completo incluye seis ficheros *jar* de los cuales destacan por su interés para este proyecto *bcprov*.jar*, que contiene el *provider* BC y *bcmail*.jar*, que además de la API para correo electrónico incluye los envoltorios para PKCS#7.

La API de BC es ligera, esto significa que está pensada para aquellas circunstancias donde no se requiera una extensa API con integración JCE.

5.1. Provider Bouncy Castle

El *provider* BC cumple la especificación JCE escrito como un envoltorio sobre la API ligera de BC. Al igual que cualquier otro *provider*, el de BC está pensado para ser seleccionado en tiempo de ejecución. El siguiente ejemplo registra el BC *provider*, crea una clave de sesión aleatoriamente y cifra los datos, que en este caso son la simple cadena de texto "plaintext":

```
Key key; KeyGenerator keyGen; Cipher encrypt;
Security.addProvider(new BouncyCastleProvider());
// "BC" es el nombre del provider BC
keyGen = KeyGenerator.getInstance("DES", "BC"); keyGen.init(new SecureRandom());
key = keyGen.generateKey();
encrypt = Cipher.getInstance("DES/CBC/PKCS5Padding", "BC");
encrypt.init(Cipher.ENCRYPT_MODE, key);
bOut = new ByteArrayOutputStream(); cOut = new CipherOutputStream(bOut,
encrypt);
cOut.write("plaintext".getBytes()); cOut.close();
```

5.2. Certificados

El *provider* BC puede leer certificados X.509 (v2 o v3), como por ejemplo los creados con la clase *CertificateFactory* de *java.security.cert*. Puede manejar certificados en formato PEM (codificación base 64) o binarios DER. La factoría de certificados puede leer también cadenas de certificados (v2), con ambas codificaciones: PEM o DER.

⁵Se puede consultar una copia de esta licencia on-line en:
<http://www.bouncycastle.org/licence.html>



5.3. Key Stores

El paquete BC proporciona tres implementaciones distintas de repositorios de claves.

La primera, “BKS”, es un repositorio que puede funcionar con la herramienta *keytool* de la misma forma que un repositorio “JKS” de Sun.

La segunda, *KeyStore.BouncyCastle* o *KeyStore.UBER* que sólo funciona con *keytool* si el password es proporcionado desde la línea de comandos, ya que todo el repositorio está cifrado con un algoritmo PBE basado en SHA1 y Twofish (PBEWithSHAAndTwofish-CBC). Esto lo hace resistente a la inspección, cosa que no sucede con JKS ni BKS.

La tercera, es la de un repositorio compatible con PKCS#12. Este tipo de repositorios producen una situación ligeramente diferente a un repositorio de claves normal, ya que el password del repositorio es el único utilizado para guardar claves.

El siguiente ejemplo accede a un KeyStore de tipo PKCS#12 utilizando BC:

```
[...]
instream = new FileInputStream(p12file); p12keystore =
KeyStore.getInstance("PKCS12", "BC"); p12keystore.load(instream,
p12storepass.toCharArray()); instream.close();
```

5.4. PKCS#7

El paquete *org.bouncycastle.cms* provee el acceso a objetos RFC 3852 Cryptographic Message Syntax (CMS), también denominados PKCS#7. Este paquete proporciona envoltorios PKCS#7 para las firmas generadas y es el utilizado por la librería jeFirma con ese propósito.

Hasta el momento se han mostrado ejemplos de firma de datos, pero tan sólo se ha generado la firma en sí, a partir de los datos o de su *hash*. Para la interacción con otros programas las firmas se suelen presentar en formato PKCS#7, tanto en formato binario DER como en codificación PEM. En el siguiente ejemplo se crea un objeto de firma PKCS#7:

```
CMSSignedDataGenerator sdatagen = new CMSSignedDataGenerator();
// Añadir firmante, clave privada, certificado y algoritmo
sdatagen.addSigner(key, (X509Certificate) cert,
CMSSignedDataGenerator.DIGEST[EQUATION])
// Añadir la cadena de certificados y la lista // de revocación del
repositorio. sdatagen.addCertificatesAndCRLs(certStore);
CMSSignedData sData=null;
CMSProcessableFile pFile= new CMSProcessableFile(new File(filePath);
// Firma sData = sdatagen.generate(pFile, false, "BC");
Llegados a este punto, tenemos un objeto PKCS#7 en memoria, pero no es
posible escribir directamente la firma a disco. Para escribirla en el
formato adecuado, se pueden utilizar las herramientas provistas por BC
(PemWriter):
```



```
// Convertir a Stream ASN1 ASN1InputStream asn1Stream= new
ASN1InputStream(sdataArray);
ASN1Sequence ans1Seq= ASN1Sequence.getInstance(asn1Stream.readObject());
ContentInfo ci=new ContentInfo(ans1Seq);
// Crear el objeto "escritor" en PEM PEMWriter pw = new PEMWriter(new
OutputStreamWriter(fos));
// Escribir pw.writeObject(ci);
// Cerrar Streams pw.close(); fos.close();
```

6. Librería jeFirma

jeFirma es un conjunto de utilidades que pretenden replicar en Java 1.5 las funciones de firma digital que la empresa Gadd ha desarrollado en .NET. Básicamente, se trata de una librería que permita generar firmas para documentos digitales en formato PKCS#7, en modo *detached*, así como su verificación posterior.

6.1. Manual de Usuario

jeFirma se empaqueta en un único fichero *jar*. Las dependencias de la librería son las siguientes:

- Java Run Time System Library 1.5
- Librería Bouncy Castle. Son necesarios dos ficheros *jar*:
 1. *bcprov-jdk15-333.jar*. Implementación del provider BC.
 2. *bcmail-jdk15-333.jar*. Manejo de datos PKCS#7.

Para su uso en un *applet* hay que tener en cuenta que necesita trabajar conjuntamente con los *jar* de Bouncy Castle indicados anteriormente y que el cliente ha de tener una máquina virtual Java 1.5 o superior instalada en su sistema.

Aunque no se trata de una dependencia de la librería, los ejemplos que se han entregado con la misma requieren de un *jar* extra, el *forms-1.0.7.jar* que forma parte del proyecto JGoodies⁶.

6.2. Manual de programación (API)

La librería *jeFirma* se divide en tres paquetes: *jeFirma*, *jeFirma.Util* y *jeFirma.jeExample*. Los dos primeros son los únicos necesarios para la distribución de la librería.

6.2.1. Paquete jeFirma

Incluye dos ficheros:

- **JECryptography.java**. Es el fichero principal de la librería. Contiene la clase *JECryptography*, que implementa la funcionalidad de firma y verificación de documentos

⁶<http://www.jgoodies.com/>



digitales.

- **SignatureType**. Este fichero contiene la implementación de un tipo enumerado para describir los formatos de la firma, PEM o DER.

6.2.2. jeFirma :: Clase JECryptography

- **Constructor JECryptography (String ksPath, String ksType, Provider p)**

Constructor con *provider* explícito.

- *ksPath*. Es la ruta al repositorio de claves (key store). Este parámetro puede ser null, en cuyo caso el repositorio se supone en un dispositivo criptográfico, como una SmartCard.
- *ksType*. Es una cadena de texto que representa el tipo de repositorio de claves. Por ejemplo, se utilizará la cadena "JKS" para el formato de Sun, "BKS" para el formato de BouncyCastle o "PKCS11" para una SmartCard. El tipo de repositorio ha de estar soportado por el *provider* indicado.

Además, este constructor crea y añade un *provider* BouncyCastle, utilizado para los procesos de verificación y firma, por lo que si el tipo de repositorio de claves está soportado por BouncyCastle, se debería utilizar el constructor con *provider* implícito.

Si se produce algún error al acceder al repositorio de claves, se eleva una excepción *KeyStoreException*.

- **Constructor JECryptography (String ksPath, String ksType)**

Constructor con *provider* implícito.

- *ksPath*. Es la ruta al repositorio de claves (key store). Este parámetro puede ser null, en cuyo caso el repositorio se supone en un dispositivo criptográfico, como una SmartCard.
- *ksType*. Es una cadena de texto que representa el tipo de repositorio de claves. Por ejemplo, se utilizará la cadena "JKS" para el formato de Sun, "BKS" para el formato de BouncyCastle o "PKCS11" para una SmartCard. El tipo de repositorio tiene que estar soportado por algún *provider* registrado en la *Java™ Cryptography Architecture*.

Si se produce algún error al acceder al repositorio de claves, se eleva una excepción *KeyStoreException*.

- **Método void signDocument(String filePath, String signFilePath, String alias, char [] passCh, SignatureType type)**

Método de firma.



- *filePath*. Es la ruta absoluta al fichero que se quiere firmar.
- *signFilePath*. Es la ruta absoluta al fichero de firma que se va a crear.
- *alias*. Es el alias del certificado cuyo par de claves se va a utilizar para firmar.
- *passCh*. Es el password de acceso al par de claves (password del certificado).
- *type*. Es el tipo de firma. Puede ser *SignatureType.DER* para firma binaria o *SignatureType.PEM* para firma PEM (firma en base 64 con encabezados PKCS#7).

Si se produce algún error durante la firma, se eleva una excepción del tipo *SignException*.

- **Método void *verifyDocument(String filePath, String signFilePath, SignatureType type)***

Verificación de un fichero firmado.

- *filePath*. Es la ruta absoluta al fichero de datos (fichero que ha sido firmado).
- *signFilePath*. Es la ruta absoluta al fichero de firma en formato PKCS#7.
- *type*. Es el tipo de firma contenida en *signFilePath*. Puede ser *SignatureType.DER* si es una firma binaria o *SignatureType.PEM* si es una firma PEM (firma en base 64 con encabezados PKCS#7).

Si se produce algún error durante la firma, se eleva una excepción del tipo *VerifyException*.

- **Método Enumeration[EQUATION]String[EQUATION] *aliases()***

Obtener todos los alias del repositorio de claves. Este método accede al *Key Store* y recupera todos los alias que existan, devolviendo enumeración con las cadenas de texto que los definen.

- **Método void *load(char[] passCh)***

Acceso al repositorio de claves. Este método permite autenticarse para el acceso al *Key Store*. Tiene que ser llamado antes de utilizar cualquier otro método de acceso al *Key Store*.

- *passCH*. Es el password de acceso al repositorio.
- Dependiendo del tipo de error, este método eleva las siguientes excepciones:
- *IOException*. Si hay algún tipo de problema de Entrada/Salida con el fichero del *Key Store* o si el formato no puede ser reconocido.
- *NoSuchAlgorithmException*. Si no se puede encontrar el algoritmo utilizado para comprobar la integridad del *Key Store*.
- *CertificateException*. Si no se puede leer alguno de los certificados contenidos en el *Key Store*.
-

- **Método void *VerifyCertificate(String alias)***

-



Verificación del certificado, dentro del repositorio, seleccionándolo por su alias. Si se produce algún error en la verificación, se eleva una excepción. Pueden ser:

- *CertificateExpiredException*. Si el certificado ha expirado.
- *CertificateNotYetValidException* Si el certificado no es válido por alguna otra circunstancia.
- *KeyStoreException*. Si no se puede acceder al *Key Store*.

6.2.3. jeFirma :: Clase *SignatureType*

Esta clase implementa un enumerado para los tipos de firma, que pueden ser PEM o DER.

6.2.4. Paquete *jeFirma.Util*

Incluye una serie de utilidades para la clase *JECryptography* que bien por tener dependencias de paquetes gráficos o bien para su reutilización independiente del proceso de firma, se ha optado por aislar en un paquete propio.

El paquete incluye un único fichero, **JEUtil.java** con dos clases, *JEUtil* y *PasswordDialog*.

6.2.5. jeFirma.Util :: Clase *JEUtil*

Incluye los siguientes métodos:

- **Método *char[] getPasswordDialog(String _msgRequestPIN)***

Muestra un diálogo definido por la clase *PasswordDialog*. Se trata de un campo de texto, con la entrada de texto enmascarada con asteriscos y dos botones, “Aceptar” y “Cancelar”. Se opta por una implementación completa en vez de utilizar un *JOptionPane* estándar para tener el control total del foco y eventos de teclado.

- *_msgRequestPIN*. Es el mensaje que se muestra en el título del diálogo.
- Retorna un array de caracteres con la cadena de texto introducida.

- **Método *String chooseAlias(Enumeration[EQUATION]String[EQUATION] aliases)***

Muestra un diálogo para la selección de un alias.

- *aliases*. Es una enumeración de cadenas de texto que representan los diferentes alias de los certificados del repositorio de claves.

Los alias se muestran en un *Combo box* seleccionable. Devuelve la cadena de texto correspondiente al elemento del *Combo box* seleccionado en el momento de pulsar el botón de aceptación. Si sólo existe un alias en el repositorio, este método devuelve la cadena del alias



sin mostrar ningún diálogo.

- **Método `byte[] getBytesFromFile(File file)`**

Guarda los datos de un fichero en un array de bytes. Este método se utiliza para almacenar en memoria los ficheros de firma, que son ficheros de pocos bytes.

- *file*. Es el fichero a leer.
- Retorna un array de bytes con el contenido del fichero pasado como parámetro.

Si se produce algún problema en el acceso o lectura del fichero se eleva una excepción *IOException*.

6.2.6. jeFirma.Util :: Clase PasswordDialog

Es una extensión de un *JDialog* de *Swing*

6.2.7. Ejemplos de uso

Para crear un objeto *JECryptography* debemos saber el tipo de repositorio de claves, o *Key Store* que se va a manejar. Un ejemplo típico de uso sería el siguiente:

```
[...]
JECryptography jeCrypt;
// El provider que voy a utilizar // para acceder al KeyStore Provider p =
new org.bouncycastle.jce.provider.BouncyCastleProvider();
jeCrypt=new JECryptography(BKSPath,"BKS",p);
Donde BKSPath es la ruta al fichero del repositorio de claves.
Como queda dicho con anterioridad, la clase JECryptography crea de oficio un
provider de BouncyCastle, por lo que el código anterior es equivalente a:
[...]
JECryptography jeCrypt;
jeCrypt=new JECryptography(BKSPath,"BKS");
Cuando trabajamos con dispositivos SmartCard, se requiere o bien una cadena
de inicialización o bien un fichero de configuración para el provider
SunPKCS11.
El código en este caso tendría este aspecto:
JECryptography jeCrypt;
// Ruta de la DLL que implementa las funciones // del provider PKCS11 String
rutaDll="c:
windows
system32
PkcsV2GK.dll";
String pkcs11config = "name = SmartCardn" + " library = " + rutaDll;
// Nuevo provider para la SmartCard byte[] pkcs11configBytes =
pkcs11config.getBytes(); ByteArrayInputStream configStream = new
ByteArrayInputStream(pkcs11configBytes); Provider p = new
sun.security.pkcs11.SunPKCS11(configStream);
jeCrypt=new JECryptography(null,"SunPKCS11",p);
```



Una vez inicializado el objeto, antes de realizar cualquier operación, debemos “registrarnos” en el *Key Store*, es decir, realizar una llamada al método *load()*.

```
// Obtenemos el password del repositorio. char [] pin
=JEUtil.getPasswordDialog(msg);
if (pin==null) return false;
//Cargamos el KeyStore jeCrypt.load(pin);
```

A partir del momento de la carga del repositorio de claves, podremos acceder a los certificados que se encuentran en él. Por ejemplo, podríamos querer recuperar los alias de los certificados, para que el usuario elija uno para la firma. Un código que resolviera esto podría ser:

```
JECryptography jeCrypt;
// Inicialización [...]
// Recuperamos los alias Enumeration<String> alias = jeCrypt.alias();
// Mostramos al usuario un diálogo de selección String
alias=jeFirma.jeUtil.chooseAlias(alias);
```

En este momento, el *String* “*alias*” contiene la cadena de texto que identifica al certificado seleccionado. Hay que recordar la peculiaridad del método de utilidad *chooseAlias*, si sólo existe un certificado en el repositorio devuelve su alias sin mostrar diálogo alguno.

Vamos a firmar un fichero. Suponemos que tenemos algún tipo de interfaz que permite al usuario elegir el fichero a firmar y la ruta del fichero de firma.

```
char []pass=JEUtil.getPasswordDialog( "Introduzca su password");
// El password es null si pulsa "Cancelar" en el diálogo del password.
if (pass==null) JOptionPane.showMessageDialog(null, "Firma Cancelada");
return;
crypt.signDocument( filePath, p7sFilePath, alias, pass, sType);
```

El password introducido es el del par de claves que indica el alias, es decir, el del certificado seleccionado anteriormente. *sType* puede ser *SignatureType.PEM* o *SignatureType.DER*.

La verificación es aún más simple:

```
jeCrypt.verifyDocument( filePath, p7sFilePath, sType );
```

Podríamos querer verificar la validez de un certificado concreto:

```
jeCrypt.VerifyCertificate(alias);
```

