JCCM: Java Card Certificate Management

Arturo García Ares, María Celeste Campo Vázquez, Andrés Marín López, Ignacio Díaz Asenjo, Carlos Delgado Kloos, Carlos García Rubio, Peter T. Breuer

Abstract— **El punto más débil del software criptográfico moderno está en el almacenamiento de claves. Las tarjetas inteligentes son una alternativa más segura al uso del almacenamiento tradicional en discos.**

El estándar [PKCS#11] define un interfaz a las aplicaciones que proporciona una visión lógica común de los dispositivos capaces de ofrecer servicios criptográficos. Los desarrollos que se han realizado hasta la actualidad de módulos PKCS #11 con tarjetas inteligentes emplean tarjetas no programables o bien no utilizan la facilidad de programación de las tarjetas que sí son programables. Estas implementaciones se limitan a adaptar la funcionalidad criptográfica predefinida en las tarjetas a la visión lógica que debe ofrecer un módulo PKCS #11, recortando la semántica definida por el estándar para adaptarla a la que ofrece el fabricante en cada tarjeta.

Este artículo describe nuestro sistema, Java Card Certificate Management (JCCM), en el que empleamos una tarjeta inteligente programable mediante tecnología Java Card como dispositivo criptográfico, trasladando parte de la implementación del estándar PKCS #11 a la tarjeta, resultando en una mayor flexibilidad e independencia del dispositivo.

JCCM está disponible para su uso con Netscape para navegar de forma segura, cifrar y firmar correos.

I. Introducción

El uso de aplicaciones criptográficas empieza a popularizarse a partir de la inclusión de capacidades criptográficas en los navegadores. Hoy en día la mayoría de los usuarios del web han accedido alguna vez a páginas *seguras* (utilizando el protocolo [HTTPS]), y poco a poco se empiezan a enviar correos electrónicos firmados o cifrados. La piedra angular de estas nuevas capacidades radica en la utilización de certificados digitales ([X.509]) que establecen un mecanismo para comprobar la identidad de un usuario o de un sitio, para dar fe de que un usuario ha firmado un mensaje y garantizar la integridad del mismo, o bien para proteger una clave de sesión mediante la cual cifrar un mensaje. Estos certificados encapsulan distintas claves, entre ellas la más importante es la clave privada. El punto más débil del sistema está en el almacenamiento de claves (en concreto de la clave privada). Las tarjetas inteligentes son una alternativa muy segura para el almacenamiento y gestión de dichas claves. Las tarjetas actuales son dispositivos inviolables (*tamper-proof*)[1] y además tienen una considerable capacidad de cómputo y almacenamiento.

Este trabajo ha sido desarrollado dentro del proyecto E-TICKET CYCYT N°2FD1997-1269-C02-01(TEL)

[1]La inviolabilidad de la tarjeta se entiende a efectos prácticos en la relación coste/beneficio del ataque. Si dispusieramos de un número ilimitado de medios físicos, podríamos comprometer la seguridad de una tarjeta moderna. En [USENIX 99] se describen una serie de sofisticados ataques químicos y físicos, y las sencillas, pero efectivas, contramedidas adoptadas por los fabricantes. El artículo concluye asertando la dificultad de que todavía hay ataques muy difíciles de parar (ataques con herramientas sofisticadas basadas en haces de gases ionizados) mientras que las tarjetas no dispongan de una fuente de alimentación propia.

Las aplicaciones que utilizamos, por ejemplo nuestro navegador favorito, incluye las capacidades necesarias para gestionar estos certificados digitales. Por ejemplo, el navegador de Netscape, tiene su propia biblioteca, la *Netscape Security Library* para estos fines.

El estándar [PKCS#11] define un interfaz que proporciona a las aplicaciones que desean utilizar servicios criptográficos una visión lógica común de los dispositivos capaces de ofrecer dichos servicios: dispositivos con capacidad de almacenamiento, proceso, y específicamente, capaces de proteger de manera efectiva los elementos sensibles (claves de cifrado) almacenados. Las aplicaciones que utilizamos suelen utilizar este API (por ejemplo, la NSL de Netscape sigue este estándar).

La alternativa más segura es utilizar módulos que implementen PKCS #11 para las aplicaciones que lo admitan, y utilizar tarjetas inteligentes para compartir la gestión y para almacenar de forma segura claves y certificados. Sin embargo, los desarrollos que se han realizado hasta la actualidad de módulos PKCS #11 ([SMARTSIGN], [GPKPKCS#11], [SLBCBPKCS#11]) con tarjetas inteligentes emplean tarjetas no programables o bien no utilizan la facilidad de programación de las tarjetas que sí son programables. Estas implementaciones se limitan a adaptar la funcionalidad criptográfica predefinida en las tarjetas a la visión lógica que debe ofrecer un módulo PKCS #11, recortando la semántica definida por el estándar para adaptarla a la que ofrece el fabricante en cada tarjeta.

Nuestro sistema, Java Card Certificate Management (JCCM), emplea una tarjeta inteligente programable mediante tecnología Java Card como dispositivo criptográfico, trasladando parte de la implementación del estándar PKCS #11 a la tarjeta, resultando en una mayor flexibilidad, e independencia del dispositivo.

En este artículo describimos la implementación del API definida por el estándar PKCS #11 versión 2.10, con el objetivo de permitir al usuario de Netscape beneficiarse del uso de tarjetas inteligentes como almacén seguro de certificados y claves de cifrado para su empleo en navegación segura, cifrado y firma de correo electrónico basada en el algoritmo RSA.

El firmado de mensajes de correo electrónico mediante algoritmos de clave pública requiere un complejo sistema en el que participan agentes de usuario habilitados para generar y verificar firmas digitales (Netscape), una infraestructura de clave pública (PKI) y dispositivos capaces de ejecutar algoritmos criptográficos y de almacenar datos sensibles de manera segura que llamaremos *tokens criptográficos* o simplemente *tokens*. Este artículo abarca únicamente el código ejecutado por el token y el fragmento del agente de usuario que constituye la implementación

del estándar PKCS #11. De acuerdo con esto, JCCM se compone de dos piezas complementarias:
• Una biblioteca de enlace dinámico (DLL) que sirve de interfaz entre la tarjeta inteligente y la biblioteca de seguridad (NSL) de Netscape –el agente de usuario.
• Un *cardlet*[2] que implementa las funciones requeridas.

Las secciones II, III y IV contienen una breve introducción a las tecnologías directamente involucradas: el estándar PKCS #11, la arquitectura de seguridad de Netscape y las tarjetas inteligentes respectivamente. La sección V describe nuestra implementación centrándose en la aplicación Java Card por ser ésta el rasgo característico, para terminar en la sección VII con las conclusiones.

II. El estándar PKCS #11

PKCS #11 es uno más del creciente grupo de estándares PKCS, acrónimo de Public-Key Cryptography Standards (estándares de criptografía de clave pública) y publicados por RSA Labs. Estos estándares abarcan aspectos del sistema que acabamos de esbozar tales como:
• Definición de los algoritmos criptográficos (tanto de clave pública como de clave simétrica) y algoritmos de apoyo.
• Procedimientos para aplicar los algoritmos criptográficos a flujos de bytes y para producir firmas digitales.
• Definición de tipos de datos (certificados, claves, ...) y sus correspondientes sintaxis de transferencia o formatos de representación binaria.
• Procedimiento para efectuar transacciones HTTP seguras
• Definición de formatos para transmisión de correo electrónico cifrado e inclusión de firma digital.
• Interfaz de programación para aplicaciones (API) que permite hacer uso de los algoritmos, procedimientos y tipos de datos.

La biblioteca descrita por el estándar PKCS #11 recibe el nombre de Cryptoki, abreviatura de "cryptographic token interface". A lo largo de este artículo utilizaremos el término "el estándar" para referirnos al estándar PKCS #11, y Cryptoki para referirnos a la biblioteca definida por el estándar.

Cryptoki aisla a la aplicación de los detalles del dispositivo criptográfico. En terminología de Cryptoki, estos dispositivos se llaman *tokens*. Estos tokens pueden ser de carácter portátil (p.e. una tarjeta inteligente) o fijos (p.e. una tarjeta PCI insertada en la placa madre del ordenador). Los tokens portátiles se insertan en su correspondiente *slot*, nombre asignado por Cryptoki para la abstracción del terminal lector. Un token almacena *objetos* y es capaz de ejecutar *primitivas criptográficas* empleando esos objetos, posiblemente tras un paso inicial de autenticación del usuario/aplicación al token mediante la presentación de un PIN. Toda comunicación con el token se efectúa dentro del contexto de una *sesión*, que representa un canal de comunicación establecido con un token presente en un slot.

Todas estas abstracciones son soportadas mediante las funciones y estructuras de datos definidas por el estándar.

[2]Programa residente en la tarjeta inteligente (o token) escrito acorde con la especificación Java Card

Funciones de propósito general	Funciones de gestión de slots y tokens
`C_Initialize`	`C_GetSlotList`
`C_Finalize`	`C_GetSlotInfo`
`C_GetInfo`	`C_GetTokenInfo`
Funciones criptográficas	`C_GetMechanismList`
`C_EncryptInit`	`C_SetPIN`
`C_Encrypt`	Funciones de gestión de sesiones
`C_EncryptUpdate`	`C_OpenSession`
`C_EncryptFinal`	`C_CloseSession`
`C_DecryptInit`	`C_CloseAllSessions`
`C_Decrypt`	`C_GetSessionInfo`
`C_DecryptUpdate`	`C_Login`
`C_DecryptFinal`	`C_Logout`
`C_SignInit`	Funciones de gestión de objetos
`C_Sign`	`C_CreateObject`
`C_SignUpdate`	`C_CopyObject`
`C_SignFinal`	`C_DestroyObject`
`C_VerifyInit`	`C_GetAttributeValue`
`C_Verify`	`C_SetAttributeValue`
`C_VerifyUpdate`	`C_FindObjectsInit`
`C_VerifyFinal`	`C_FindObjects`
`C_GenerateKey`	`C_FindObjectsFinal`
`C_GenerateKeyPair`	
`C_WrapKey`	
`C_UnwrapKey`	

TABLE I
Muestra representativa de funciones de Cryptoki

Estas funciones (ver tabla I) pertenecen a varios grupos:
• Funciones de propósito general: Inicialización y obtención de información general acerca de la biblioteca.
• Funciones de gestión de slots y tokens: Permiten obtener información acerca de los slots presentes en el sistema, el estado de presencia o ausencia de un token en cada slot, capacidades de los tokens presentes y establecimiento de PINs.
• Funciones de gestión de sesiones: Apertura y cierre de sesiones, obtención de información acerca de la sesión y autenticación al token mediante la presentación de un PIN.
• Funciones de gestión de objetos: Permiten crear, buscar y obtener objetos.
• Funciones criptográficas: Permiten ejecutar primitivas criptográficas de las siguientes categorías: cifrado y descifrado de flujos de bytes, firmado y verificación de firmas y generación, importación y exportación de claves, entre otras.

Un objeto es una lista de parejas atributo/valor. Todos los posibles objetos están perfectamente definidos en el estándar mediante la lista de atributos que lo componen y el tipo de cada atributo. Se emplea una pseudo-orientación a objetos para definir los distintos tipos de objetos, de manera que se definen tipos básicos de los cuales derivan tipos especializados que incorporan todos los atributos de los tipos base y añaden atributos propios (ver fig. 1). Los

principales tipos de objetos son "Certificate" y "Key" pero también existen los tipos "HW Feature", empleado para describir capacidades especiales de los tokens, y "Data" que se utiliza para almacenar un objeto binario sin interpretar.

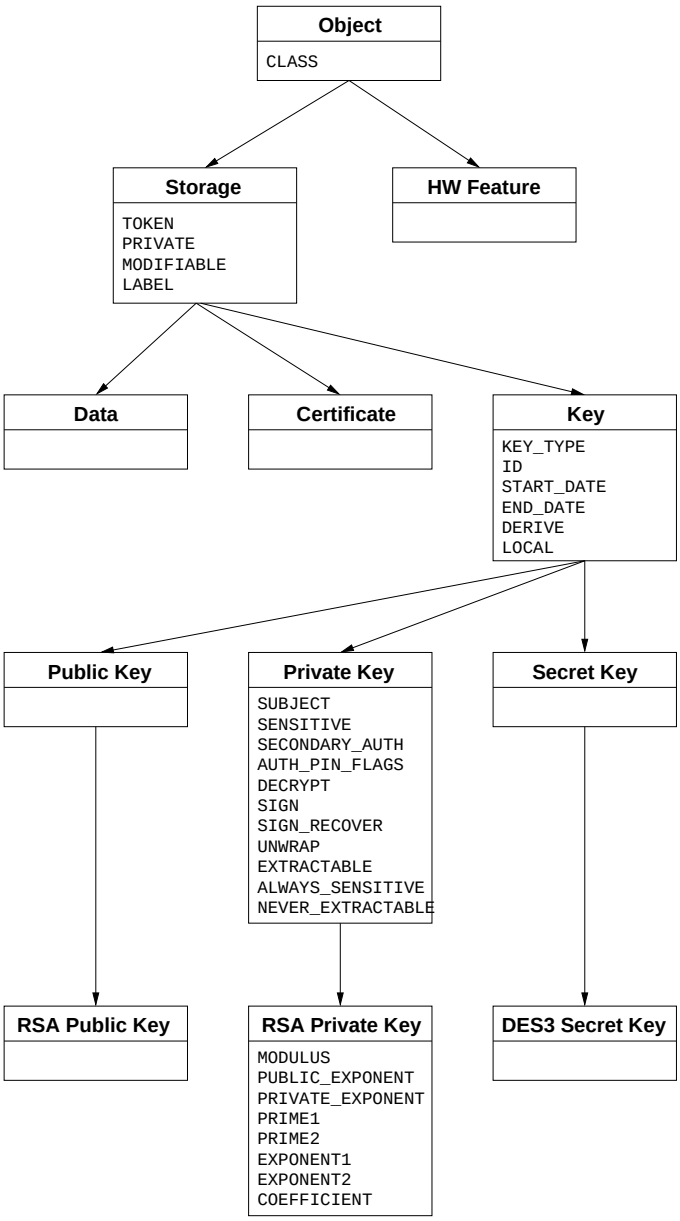

Fig. 1. Jerarquía parcial de objetos de Cryptoki y detalle de atributos de clases participantes en la jerarquía de RSA Private Key

Los objetos más importantes son los de tipo clave ("Key"). Todas las funciones criptográficas requieren o generan objetos de este tipo. Como queda reflejado en la fig. 1, el tipo clave se especializa en clave pública ("Public Key"), privada ("Private Key") y secreta ("Secret Key"). A su vez, cada uno de estos tipos se especializa en tipos concretos de claves dependiendo del algoritmo de cifrado con el que deban ser utilizadas. En la fig. 1 se han incluido todos los atributos de los objetos que forman parte del tipo concreto "RSA Private Key".

En terminología de Cryptoki, los algoritmos de cifrado se denominan *mecanismos*. Evidentemente el estándar PKCS #11 no define los algoritmos en sí, pero sí identifica cuáles pueden ser soportados por una implementación de Cryptoki. Cada implementación es libre de soportar únicamente un cierto número de mecanismos. Cada mecanismo tiene asignado un identificador, opera sobre unos tipos concretos de objetos y puede o no requerir ciertos parámetros propios.

Aclaremos con un ejemplo la relación que existe entre objetos, mecanismos y funciones de cifrado. La fig. 1 muestra una relación de todos los atributos que forman un objeto de tipo clave privada RSA, incluyendo las clases base. Conviene señalar que una implementación de Cryptoki que acepte objetos de este tipo (toda implementación — como la que nos ocupa — que soporte mecanismos basados en el algoritmo RSA deberá soportar, entre otros, objetos "RSA Private Key") debe ofrecer a la aplicación esa visión lógica, con todos los atributos, aunque el hardware subyacente utilice una representación distinta. A continuación se describen los pasos que deberá realizar una aplicación que desee utilizar una clave privada RSA almacenada en un token para efectuar una operación de firma digital. La secuencia de llamadas a funciones de Cryptoki necesaria para generar una firma termina en:

```
C_SignInit(...);
C_Sign(...);
```

Los prototipos respectivos de estas funciones son:

```
CK_RV S_SignInit(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey);
CK_RV C_Sign(
  CK_SESSION_HANDLE hSession,
  CK_BYTE_PTR pData,
  CK_ULONG ulDataLen,
  CK_BYTE_PTR pSignature,
  CK_ULONG_PTR pulSignatureLen);
```

No nos detendremos a explicar en detalle los tipos de datos involucrados, pues su nombre es suficientemente descriptivo; los aspectos a destacar son:

- El primer parámetro de las dos funciones es un manejador (*handle*) de sesión. Este manejador habrá sido obtenido previamente por la aplicación mediante la correspondiente llamada a `C_OpenSession()`.
- La operación de firma se inicia con una llamada a `C_SignInit()`. Esta función recibe, además del manejador de sesión, una especificación del algoritmo a emplear (parámetro `pMechanism`) y el manejador del objeto de tipo clave que habrá de emplearse para efectuar la operación. Este manejador habrá sido obtenido previamente por la aplicación mediante la secuencia de llamadas

```
C_FindObjectsInit(...);
C_FindObjects(...);
C_FindObjectsFinal(...);
```

- Una vez establecidos los parámetros de la operación de firma, se realiza la operación en sí. La llamada a `C_Sign()` acepta un puntero a los datos a firmar (`pData`), la longitud

de los mismos (`ulDataLen`), el buffer donde se depositará el resultado de la operación (`pSignature`) y la longitud de este buffer (`pulSignatureLen`).

III. Netscape y PKCS #11

Netscape incorpora una arquitectura de seguridad que permite tráfico HTTP seguro y gestionar correo cifrado y firmado. Esta arquitectura de seguridad, conocida como "Netscape Security Library" (NSL), hace uso de las primitivas presentes en PKCS #11 para ofrecer la funcionalidad de alto nivel a la que nos referimos. Netscape se distribuye con un módulo PKCS #11 interno que constituye una implementación bastante completa del estándar: incluye tanto los mecanismos basados en RSA como algunos mecanismos de clave simétrica. La versión y la distribución (con o sin restricciones de exportación) de Netscape imponen los mecanismos que podemos utilizar y las limitaciones de los mismos en cuanto a longitud máxima de claves. Este módulo PKCS #11 interno ofrece una visión lógica del propio ordenador como dispositivo criptográfico: utiliza el sistema de archivos como almacenamiento persistente para almacenar los objetos de Cryptoki y la capacidad de proceso de la CPU para efectuar las operaciones de cifrado. Gracias a este módulo interno Netscape es plenamente capaz de ofrecer las capacidades mencionadas, pero adolece de un problema: la utilización del sistema de archivos del ordenador para almacenar certificados y claves rompe la premisa inicial de almacenamiento seguro de datos sensibles. La solución adoptada por Netscape es permitir la incorporación de módulos PKCS #11 externos que sirvan de interfaz a hardware criptográfico especializado, como es el caso de nuestro módulo. Otra peculiaridad de la NSL es que permite la presencia simultánea de varios módulos PKCS #11: nuestro módulo puede coexistir con el módulo interno, por lo que no es necesario implementar los mecanismos ya soportados por este módulo. Estas y otras características propias del uso que Netscape hace de Cryptoki han influido en nuestra implementación. En concreto:

- Existen funciones definidas en el estándar que nunca son llamadas por Netscape. Estas funciones no han sido implementadas.
- El estándar permite a las aplicaciones hacer un uso concurrente de Cryptoki, sin embargo Netscape utiliza un único hilo de ejecución para las operaciones de NSL. Nuestra implementación no soporta acceso concurrente.
- La NSL efectúa ciertas secuencias de llamadas a funciones de Cryptoki por triplicado. Estas secuencias son ciertas búsquedas y transferencias de objetos que, por involucrar proceso en la tarjeta, consumen una cantidad de tiempo que depende del número de objetos encontrados, pero que en una situación típica de una tarjeta que almacena un certificado y su clave privada asociada resulta del orden de decenas de segundos. Ello nos obligó a establecer una caché de búsquedas y de valores de objetos.

La fig. 2 muestra una traza de todas las llamadas efectuadas por Netscape a nuestra biblioteca para emitir un correo firmado, incluyendo todas las llamadas desde el arranque de la aplicación. La traza ha sido simplificada eliminando todas las entradas salvo la correspondiente a la salida de cada función de Cryptoki. Las líneas han sido numeradas para poder referirnos a fragmentos en la explicación que sigue a continuación:

```
1   Mar  2 12:06:20: C_GetFunctionList Returns (CKR_OK) 0x0
2   Mar  2 12:06:20: C_Initialize Returns (CKR_OK) 0x0
3   Mar  2 12:06:20: C_GetInfo Returns (CKR_OK) 0x0
4   Mar  2 12:06:20: C_GetSlotList Returns (CKR_OK) 0x0
5   Mar  2 12:06:20: C_GetSlotList Returns (CKR_OK) 0x0
6   Mar  2 12:06:21: C_GetSlotInfo Returns (CKR_OK) 0x0
7   Mar  2 12:06:21: C_GetTokenInfo Returns (CKR_OK) 0x0
8   Mar  2 12:06:21: C_GetMechanismList Returns (CKR_OK) 0x0
9   Mar  2 12:06:21: C_GetMechanismList Returns (CKR_OK) 0x0
10  Mar  2 12:06:21: C_OpenSession Returns (CKR_OK) 0x0
11  Mar  2 12:06:53: C_GetSlotInfo Returns (CKR_OK) 0x0
12  Mar  2 12:06:53: C_GetSessionInfo Returns (CKR_OK) 0x0
13  Mar  2 12:06:53: C_GetSessionInfo Returns (CKR_OK) 0x0
14  Mar  2 12:06:57: C_Login Returns (CKR_OK) 0x0
15  Mar  2 12:06:59: C_FindObjectsInit Returns (CKR_OK) 0x0
16  Mar  2 12:06:59: C_FindObjects Returns (CKR_OK) 0x0
17  Mar  2 12:06:59: C_FindObjectsFinal Returns (CKR_OK) 0x0
18  Mar  2 12:07:00: C_FindObjectsInit Returns (CKR_OK) 0x0
19  Mar  2 12:07:00: C_FindObjects Returns (CKR_OK) 0x0
20  Mar  2 12:07:00: C_FindObjectsFinal Returns (CKR_OK) 0x0
21  Mar  2 12:07:07: C_GetAttributeValue Returns (CKR_OK) 0x0
22  Mar  2 12:07:07: C_GetAttributeValue Returns (CKR_OK) 0x0
23  Mar  2 12:07:08: C_GetSessionInfo Returns (CKR_OK) 0x0
24  Mar  2 12:07:08: C_GetAttributeValue Returns (CKR_OK) 0x0
25  Mar  2 12:07:08: C_GetAttributeValue Returns (CKR_OK) 0x0
26  Mar  2 12:07:12: C_FindObjectsInit Returns (CKR_OK) 0x0
26  Mar  2 12:07:12: C_FindObjects Returns (CKR_OK) 0x0
27  Mar  2 12:07:12: C_FindObjectsFinal Returns (CKR_OK) 0x0
28  Mar  2 12:07:12: C_GetMechanismList Returns (CKR_OK) 0x0
29  Mar  2 12:07:12: C_GetMechanismList Returns (CKR_OK) 0x0
30  Mar  2 12:07:16: C_FindObjectsInit Returns (CKR_OK) 0x0
31  Mar  2 12:07:16: C_FindObjects Returns (CKR_OK) 0x0
32  Mar  2 12:07:16: C_FindObjectsFinal Returns (CKR_OK) 0x0
33  Mar  2 12:07:17: C_GetAttributeValue Returns (CKR_OK) 0x0
34  Mar  2 12:07:17: C_GetAttributeValue Returns (CKR_OK) 0x0
35  Mar  2 12:07:17: C_GetSessionInfo Returns (CKR_OK) 0x0
36  Mar  2 12:07:17: C_GetAttributeValue Returns (CKR_OK) 0x0
37  Mar  2 12:07:17: C_GetAttributeValue Returns (CKR_OK) 0x0
38  Mar  2 12:07:18: C_FindObjectsInit Returns (CKR_OK) 0x0
39  Mar  2 12:07:18: C_FindObjects Returns (CKR_OK) 0x0
40  Mar  2 12:07:18: C_FindObjectsFinal Returns (CKR_OK) 0x0
41  Mar  2 12:07:18: C_GetSlotInfo Returns (CKR_OK) 0x0
42  Mar  2 12:07:18: C_GetSessionInfo Returns (CKR_OK) 0x0
43  Mar  2 12:07:18: C_GetSessionInfo Returns (CKR_OK) 0x0
44  Mar  2 12:07:18: C_FindObjectsInit Returns (CKR_OK) 0x0
45  Mar  2 12:07:18: C_FindObjects Returns (CKR_OK) 0x0
46  Mar  2 12:07:18: C_FindObjectsFinal Returns (CKR_OK) 0x0
47  Mar  2 12:07:18: C_GetAttributeValue Returns (CKR_OK) 0x0
48  Mar  2 12:07:19: C_GetAttributeValue Returns (CKR_OK) 0x0
49  Mar  2 12:07:19: C_GetSessionInfo Returns (CKR_OK) 0x0
50  Mar  2 12:07:19: C_GetAttributeValue Returns (CKR_OK) 0x0
51  Mar  2 12:07:19: C_GetAttributeValue Returns (CKR_OK) 0x0
52  Mar  2 12:07:19: C_FindObjectsInit Returns (CKR_OK) 0x0
53  Mar  2 12:07:19: C_FindObjects Returns (CKR_OK) 0x0
54  Mar  2 12:07:19: C_FindObjectsFinal Returns (CKR_OK) 0x0
55  Mar  2 12:07:19: C_GetSessionInfo Returns (CKR_OK) 0x0
56  Mar  2 12:07:20: C_GetAttributeValue Returns (CKR_OK) 0x0
57  Mar  2 12:07:20: C_GetAttributeValue Returns (CKR_OK) 0x0
58  Mar  2 12:07:20: C_FindObjectsInit Returns (CKR_OK) 0x0
59  Mar  2 12:07:20: C_FindObjects Returns (CKR_OK) 0x0
60  Mar  2 12:07:20: C_FindObjectsFinal Returns (CKR_OK) 0x0
61  Mar  2 12:07:22: C_GetAttributeValue Returns (CKR_OK) 0x0
62  Mar  2 12:07:23: C_GetAttributeValue Returns (CKR_OK) 0x0
63  Mar  2 12:07:23: C_GetAttributeValue Returns (CKR_OK) 0x0
64  Mar  2 12:07:23: C_GetSessionInfo Returns (CKR_OK) 0x0
65  Mar  2 12:07:23: C_SignInit Returns (CKR_OK) 0x0
66  Mar  2 12:07:25: C_Sign Returns (CKR_OK) 0x0
```

Fig. 2. Traza de llamadas efectuadas por Netscape a la biblioteca Cryptoki para enviar un mensaje de correo electrónico firmado

- *Líneas 1 a 10*: Funciones de inicialización de biblioteca y obtención de información básica. Son llamadas al arrancar Netscape antes de que iniciemos cualquier operación. El token estaba presente en el lector (es lo que Netscape determina al llamar a la función `C_GetSlotInfo()`), es por ello

que se obtiene información acerca del token presente y sus capacidades mediante las llamadas a `C_GetTokenInfo()` y `C_GetMechanismList()`. Por último, se inicia una sesión con el token.

• *Líneas 11 a 14*: Si nos fijamos en los instantes de traza asociados a cada línea, veremos que transcurren 32s entre la línea 10 y la 11; corresponden al tiempo transcurrido en escribir el mensaje de correo electrónico que dió lugar a esta traza y pulsar el botón de enviar. Estas líneas corresponden a la solicitud del PIN por parte de Netscape y la correspondiente llamada a `C_Login()`. El tiempo transcurrido entre las línas 13 y 14 corresponde a la introducción del PIN en el cuadro de diálogo.

• *Líneas 15 a 66*: El resto de la traza corresponde a las llamadas efectuadas por Netscape a nuestro módulo para realizar la operación de firma. Las dos últimas corresponden a la realización de la firma en sí. El tiempo empleado en efectuar esta operación (líneas 65 a 66) es de tan sólo 2s, mientras que el tiempo total empleado para la firma (líneas 15 a 66) es de 26s. Esos 24s restantes se emplean en búsquedas de certificados y claves en el token y en transferencias de los mismos al ordenador; éstas son, por tanto, las culpables de que la operación tarde tanto.

Hay que hacer constar que sólo se incurre en esta penalización la primera vez que se utiliza el certificado y la clave privada asociada; la generación de una firma digital para un nuevo mensaje consume únicamente 6s.

IV. tarjetas inteligentes y Java Card

Una tarjeta inteligente contiene un pequeño ordenador que consta de un microprocesador, memoria volátil (RAM) y memoria persistente (EEPROM). Las capacidades típicas de estos elementos son: microprocesador de 8 bits a 4MHz, RAM de 512 bytes, 32Kb de EEPROM y 16Kb de ROM. Los contactos presentes en la superficie de la tarjeta aportan la tensión de alimentación, la señal de reloj y un canal de comunicación serie (a 9600bps[3]). Estos contactos son gobernados por el terminal lector en el que esté insertada la tarjeta, por lo tanto el microprocesador sólo estará activo cuando la tarjeta esté insertada en un terminal. Extraer la tarjeta del terminal tiene el mismo efecto que apagar un ordenador: se interrumpe la actividad de la CPU y se pierde el contenido de la RAM, pero el almacenamiento persistente permanece; en un ordenador este almacenamiento persistente está constituido por dispositivos magnéticos (discos duros), mientras que en una tarjeta inteligente es la memoria EEPROM. Otra característica importante de este tipo de memoria es su relativamente corta esperanza de vida, unos 100000 ciclos de escritura, y su elevado tiempo de acceso para ciclos de escritura, del orden de 10ms.

La tarjeta inteligente es un dispositivo pasivo: recibe un comando proveniente del ordenador a través del terminal en el que esté insertada, lo procesa, genera una respuesta que es devuelta al ordenador y espera el siguiente comando. Estos comandos se denominan APDUs (Application Protocol Data Unit) y su estructura está definida en [ISO/IEC 7816-4]. Existen dos tipos de APDU, las que codifican un comando enviado por el ordenador y las que codifican la respuesta generada por la tarjeta; ambas estructuras quedan reflejadas en la fig. 3. Básicamente, las APDUs de comando constan de una cabecera de 4 bytes[4] de los cuales los dos primeros (CLA, INS) indican la operación a realizar y los dos siguientes (P1, P2) son parámetros cuya interpretación depende del código de operación. Esta cabecera puede ir seguida de una cantidad variable de datos que aporten información necesaria para la ejecución del comando que no pueda codificarse en los dos bytes de parámetros de la cabecera. La estructura de una APDU de respuesta es más sencilla: consiste en un código de resultado de dos bytes (SW1, SW2) precedido de una parte opcional de longitud variable. Los posibles valores de los dos bytes de código de respuesta están definidos en el estándar [ISO/IEC 7816-4], mientras que la interpretación de la parte variable corresponde a la aplicación y depende de la operación ejecutada por la tarjeta.

Formato de una APDU de comando:

CLA	INS	P1	P2

Formato de una APDU de respuesta:

Data	SW1	SW2

Fig. 3. Estructura de las APDUs de comando y de respuesta

Existen tarjetas no programables, que aceptan un juego de comandos predefinido, y tarjetas programables capaces de incorporar dinámicamente a su memoria persistente programas provenientes del exterior y enriquecer así el conjunto de comandos que son capaces de ejecutar. Una de las tecnologías de tarjetas programables es Java Card. Una Java Card es una tarjeta inteligente capaz de incorporar y ejecutar programas escritos en un subconjunto del lenguaje de programación Java. Las principales ventajas que aporta esta tecnología son:

• Independencia de la plataforma: las aplicaciones Java Card pueden ejecutarse sobre cualquier tarjeta acorde a la especificación, independientemente del fabricante de la misma.

• Multiaplicación: varias aplicaciones pueden ejecutarse en la misma tarjeta. Tanto la descarga como ejecución de estas aplicaciones se realiza con restricciones de seguridad.

• Descarga de aplicaciones despues de su uso: los usuarios finales podrán actualizar las aplicaciones que tienen en sus tarjetas, para adaptarlas a sus necesidades.

• Utilización del lenguaje Java, lo que permite desarrollar aplicaciones con un lenguaje de alto nivel orientado a objetos.

• Compatible con estándares sobre tarjetas inteligentes: Es compatible con otros estándares desarrollados para tarjetas inteligentes como el ISO 7816.

[3]El estándar [ISO/IEC 7816-3] permite una velocidad de hasta 115Kbps

[4]La visión aquí ofrecida de las APDUs es una simplificación. Los formatos exactos de APDUs están definidos en [ISO/IEC 7816-4]

El lenguaje Java Card es una versión reducida del lenguaje Java. Las principales limitaciones son:

- Los paquetes accesibles a una aplicación Java Card no son los mismos que los accesibles a una aplicación Java. En concreto, el paquete java.lang carece las clases `Thread` y `String` y la clase `Object` está muy simplificada.
- Limitaciones en la máquina virtual: No existe recolector de basura; todo objeto es instanciado en memoria persistente y permanece en ella hasta que se elimine el cardlet de la tarjeta. Esta es, como veremos más adelante, la diferencia más chocante respecto al lenguaje Java y la que más influencia el diseño de una aplicación Java Card. Los tipos primitivos soportados se limitan a los permitidos por la aritmética entera de 16 bits de que son capaces los procesadores de las tarjetas inteligentes, lo que reduce el conjunto de tipos a `byte`, `short` y `boolean`.

V. Arquitectura de JCCM

Los desarrollos que se han realizado hasta la actualidad de módulos PKCS #11 con tarjetas inteligentes emplean tarjetas no programables o bien no utilizan la facilidad de programación de las tarjetas que sí son programables. Estas implementaciones se limitan a adaptar la funcionalidad criptográfica predefinida en las tarjetas a la visión lógica que debe ofrecer un módulo PKCS #11. La semántica definida por el estándar se recorta para adaptarla a la que ofrece la tarjeta para la que se realiza la implementación, que además queda ligada a una tarjeta de un determinado fabricante.

La característica distintiva de nuestro desarrollo es la implementación por parte del cardlet presente en la tarjeta de parte de la funcionalidad definida en el estándar PKCS #11. Los objetos se almacenan en la tarjeta con los mismos atributos definidos en el estándar, y es el propio cardlet quien procesa las operaciones de gestión de objetos de Cryptoki, soportando así el almacenamiento de cualquier tipo de objeto e implementando la semántica completa de búsqueda, copia, creación y borrado. Las categorías de funciones de Cryptoki (ver tabla I) con contrapartida en el cardlet son:

- Funciones de gestión de slots y tokens.
- Funciones de gestión de sesiones.
- Funciones de gestión de objetos.
- Funciones criptográficas.

En los siguientes apartados se explican las diferentes partes de que consta la aplicación Java Card desarrollada en este proyecto.

A. Gestión de memoria dinámica en tarjetas Java Card

La memoria en una tarjeta inteligente es un recurso limitado y es una de las restricciones más importantes a tener en cuenta cuando se realizan aplicaciones para este tipo de sistemas. Para almacenar datos de forma persistente en una tarjeta inteligente, es necesario que residan en memoria EEPROM. El estándar [ISO/IEC 7816-4] definió para ello una estructura de sistema de ficheros similar a la que tenemos en un ordenador y estandarizó APDUs para poder gestionarla. Las primeras versiones de Java Card, hasta la 2.0, imitaban este modelo de gestión de la memoria persistente: existían clases que replicaban la funcionalidad de las APDUs de acceso a ficheros definidas por ISO y que trataban de imitar el modelo de *streams* de Java. A partir de la versión 2.1 se abandonó este modelo de acceso a memoria persistente para sustituirlo por el método nativo en Java de instanciación de objetos: para disponer de un bloque de bytes en el almacenamiento persistente se instancia un objeto de la clase deseada; el acceso estructurado a la EEPROM, a través de los miembros del objeto instanciado, es así más directo que el efectuado a través de streams. Cuando en Java Card se habla de almacenamiento persistente no se bromea; los objetos instanciados no serán destruidos ni tampoco será liberado el espacio que ocupen por ningún recolector de basura, pues el estándar Java Card no contempla esta facilidad. Por lo tanto, todo objeto instanciado permanecerá durante la vida del cardlet: hasta que éste sea eliminado de la tarjeta.

Las funciones de Cryptoki permiten la creación dinámica de objetos y nosotros deseábamos que nuestra implementación no limitase esa flexibilidad. Podíamos haber diseñado una implementación capaz de almacenar un número predeterminado de objetos Cryptoki, p.e. un certificado X.509 y una clave privada RSA, pero eso limitaba la funcionalidad de nuestra implementación y por lo tanto las aplicaciones potenciales que podrían beneficiarse de ella. Topamos con el problema de desarrollar un cardlet capaz de crear, destruir y modificar dinámicamente un número indeterminado de objetos en un esquema de asignación de memoria que nunca libera los bloques asignados. La solución a este problema consiste en desarrollar un módulo de asignación de memoria dinámica que ofrezca funcionalidad similar a las tradicionales `malloc()` y `free()` de la biblioteca estándar del lenguaje C. La asignación de bloques se hace sobre un array de bytes Java cuyo tamaño se especifica en tiempo de compilación y que es reservado en memoria persistente en el instante de instalación del cardlet en la tarjeta.

La clase que implementa el esquema de memoria dinámica se llama `Malloc`. Su constructor recibe como único parámetro el tamaño del array que será gestionado, y presenta los correspondientes métodos `short alloc(short size)` y `void free(short addr)`. Se ha adoptado un sencillo esquema para gestionar los bloques libres y ocupados: todo bloque tiene la estructura reflejada en la fig. 4, una cabecera de dos bytes utilizado para la gestión de la memoria y el resto disponible para almacenar datos. La cabecera puede considerarse como una palabra de dos bytes, primero el byte de mayor peso, en la que el bit más a la izquierda es un indicador de si el bloque está libre (0) u ocupado (1) y los 15 bits restantes indican el tamaño del bloque en bytes excluyendo la cabecera. Por tanto, el tamaño máximo de bloque y de memoria dinámica que podemos manejar con este esquema es de 2^{15} bytes (32Kb), suficiente para abarcar toda la memoria persistente disponible en tarjetas inteligentes de tecnología actual. Otra implicación es que las direcciones son de dos bytes. El tamaño del array empleado en la implementación actual es de 4Kb, suficiente para albergar dos certificados propios, es

decir, para los que se almacena también una clave privada asociada, (cada pareja certificado,clave privada ocupa algo menos de 2Kb en nuestra estructura de almacenamiento persistente) y soportar además búsquedas, que requieren para su ejecución disponer de almacenamiento temporal.

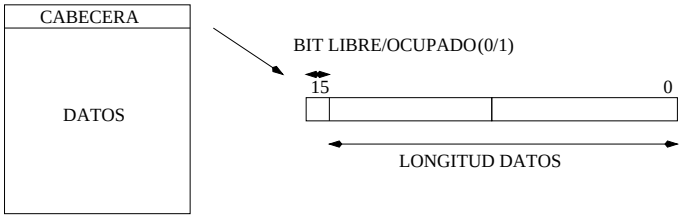

Fig. 4. Estructura de un bloque en memoria dinámica

En cualquier instante el array estará compuesto por una serie de bloques contiguos libres y ocupados mezclados, dependiendo de la secuencia de asignaciones y liberaciones previa. Inicialmente tenemos un bloque libre que abarca el array completo. Cada vez que se solicita un bloque se recorren los bloques en secuencia desde el primero (que siempre está en la dirección 0, índice `[0]` en el array) buscando un bloque libre de tamaño suficiente. Cuando ésto ocurre, si el bloque encontrado es de mayor tamaño que el bloque solicitado y suficientemente grande como para poder hacer un nuevo bloque de la parte sobrante, se divide en dos: un fragmento ocupado de tamaño exacto para albergar el tamaño solicitado y el fragmento restante que se marca como libre. Durante este recorrido de búsqueda se fusionan todos los bloques libres contiguos que se vayan encontrando. Si alcanzamos el final del array sin encontrar un bloque suficientemente grande se devuelve la dirección 0, que es una dirección no válida pues por definición apunta a la cabecera del primer bloque. Las clases que solicitan memoria reciben la dirección del campo de datos del bloque, no de la cabecera.

En la figura 5 representamos una posible imagen de la memoria en la que tendríamos cuatro bloques, dos de ellos ocupados (bloque 1 y 3) y dos libres (bloque 2 y 4), el bloque 4 es el que contiene la memoria libre del array sobre la que todavía no se ha realizado ninguna reserva.

La operación de liberación de un bloque ocupado consiste exclusivamente en el cambio del bit de estado en la cabecera. La fusión de bloques libres contiguos se realiza de manera perezosa, pues es la operación de asignación la encargada de fusionar estos bloques durante el proceso de búsqueda de memoria libre.

B. Almacenamiento y gestión de objetos de Cryptoki

En PKCS #11 los objetos están definidos como un array de atributos, donde cada atributo es una estructura de tamaño fijo con tres campos: un tipo de atributo, un puntero al valor y la longitud del valor. Las estructuras empleadas en el cardlet siguen un esquema muy parecido. La estructura empleada para los objetos, figura 6, es de longitud variable y consta de los siguientes campos:

- Enlace al siguiente objeto, 2 bytes. Todos los objetos

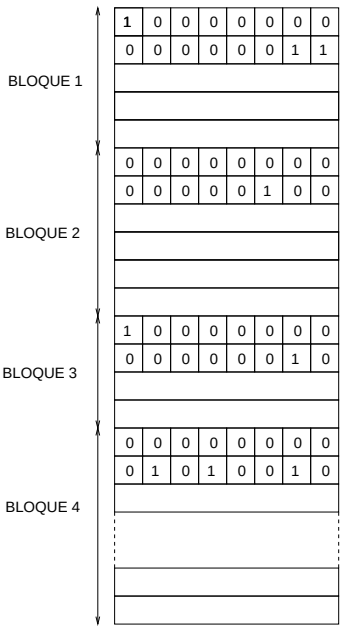

Fig. 5. Imagen de memoria

PKCS #11 que se crean en la tarjeta se mantienen en una lista enlazada.

- Número de atributos del objeto, 1 byte. Este campo puede deducirse en función del tamaño del bloque en memoria dinámica donde se aloja la estructura y el tamaño de la parte fija, pero se ha optado por incluir el número de atributos en la estructura del objeto porque el ahorro en consumo de memoria que hubiese supuesto su no inclusión es despreciable: típicamente la tarjeta almacenará únicamente dos objetos, un certificado y una clave privada asociada por lo que el ahorro total en este caso sería de 2 bytes.
- Un número variable de estructuras de atributos, tantas como número de atributos tenga el objeto.

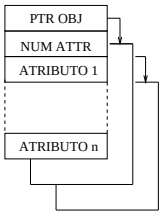

Fig. 6. Estructura de objetos

La estructura de un atributo, figura 7, consta de los siguientes campos:

- Tipo de atributo, 4 bytes. Es el valor binario definido en el estándar.
- Puntero al valor del atributo, 2 bytes. El campo valor, al ser de tamaño inherentemente variable, se almacena en su propio bloque de memoria dinámica. No almacenamos la longitud del valor en la estructura del atributo para ahorrar memoria. La longitud de este campo se obtiene de la cabecera del bloque de memoria dinámica en el que se aloja.

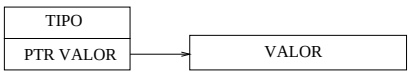

Fig. 7. Estructura de atributos

El almacenamiento de estas dos estructuras se hace sobre la memoria dinámica descrita en el apartado anterior. Existe una clase Java para ayudar en el manejo de cada una de estas estructuras, son la clase `ObjPatr` para los objetos y la clase `AttrPatr` para los atributos. Estas clases no pueden instanciarse debido al problema de no liberación de memoria mencionado en el apartado V-A, por lo que únicamente poseen métodos y miembros estáticos. Se utilizan para mapear porciones del array de memoria dinámica sobre la estructura correspondiente y para acceder cómodamente a los distintos campos de cada estructura: poseen métodos para establecer/obtener el valor de cada campo y para liberar toda la memoria dinámica asociada a cada una de estas estructuras. Estas dos clases extienden la clase `Patr`, que aporta los métodos básicos para acceder a campos de tipo multibyte. Antes de emplear una de estas clases para acceder a una estructura alojada en memoria dinámica es necesario establecer primero la dirección inicial de la estructura mediante una llamada a `setAddr(short addr)`, lo que establece la dirección de referencia empleada por todos los métodos que acceden a miembros de una estructura (`get...()`, `set...()`).

C. APDUs definidas por el cardlet

Toda la comunicación entre la aplicación Cryptoki del ordenador y el cardlet de la tarjeta se realiza a través de APDUs, que siguen el formato estándar definido en [ISO/IEC 7816-4]. Java Card nos permite definir nuestros códigos de instrucción y valores de los campos de cada una de ellas adaptados a nuestra aplicación, en la tabla II se resumen las APDUs[5] implementadas para cada cada una de las categorías de funciones definidas en el estándar PKCS #11.

D. Simulador del cardlet

Para facilitar tareas de depuración se ha desarrollado en Java un simulador del cardlet que emula su comportamiento en el ordenador, empleando sockets para simular el canal de comunicación tarjeta/ordenador. Para reutilizar el máximo código posible en ambas implementaciones, se realiza un tratamiento separado del procesamiento de las APDUs, es decir, el chequeo de la cabecera y la extracción de los campos necesarios para realizar el comando solicitado, del procesamiento en si del comando. Así existe una clase `Cryptoki` que se encarga del procesamiento de los comandos y que es común tanto para el simulador como para el cardlet, y dos clases extendidas de la misma, que son:

- Clase `CryptokiApplet`, es también una clase extendida de la clase `Applet` de Java Card, y que implementa los métodos "install", "select", "process"y "deselect", que debe poseer toda aplicación Java Card [JCADG 2.1], y que realiza el procesamiento de APDUs empleando las facilidades proporcionadas por el API. Esta clase está desarrollada para la tarjeta.
- Clase `CryptokiTest`, que trata el procesamiento de APDUs pero transmitidas a través de un socket. Esta clase está desarrollada para el propio simulador.

[5]Existen más APDUs de utilidades comunes a todas las categorías.

Categoría	APDU
Funciones de gestión de ''slot'' y ''token''	IDENT (Identidad del "token")
	DOWNLOAD_MECHANISM (Descarga de mecanismo)
Funciones de gestión de sesión	LOGIN (Login en el "token")
	LOGOUT (Logout en el "token")
Funciones de gestión de objetos	DOWNLOADOBJ_INIT (Inicio de transferencia de un objeto)
	DOWNLOADOBJ_ATTR (Transferencia de atributos de objetos)
	DOWNLOADOBJ_CREATE (Creación de objetos)
	DOWNLOADOBJ_SETATTR (Modificación de atributos de objetos)
	DOWNLOADOBJ_FIND (Búsqueda de objetos)
	GETATTR (Obtención de atributos)
	DELETEOBJ (Borrado objetos)
Funciones criptográficas	SIGN (Firma de datos)
	VERIFY (Verificación de firma)
	UNWRAPKEY (Descrifrado clave)

TABLE II
RELACIÓN ENTRE APDUS Y CATEGORÍAS DE FUNCIONES CRYPTOKI

VI. Implementación del cardlet en tarjetas comerciales

A la hora de implementar un cardlet en tarjetas comerciales, es necesario tener en cuenta las principales diferencias que existen entre las tarjetas y kit's de desarrollo Java Card existentes en la actualidad. En esta sección analizaremos cuales son estas diferencias y comentaremos nuestra experiencia con los dos kit's que hemos empleados hasta el momento en nuestro desarrollo.

A. Capacidades criptográficas en tarjetas Java Card

Las capacidades criptográficas en tarjetas inteligentes están limitadas a los algoritmos criptográficos que implementen los fabricantes, y por normativas de exportación en algunos paises, existen restricciones en el nivel de seguridad proporcionado, es decir, en la longitud de las claves.

En el caso de tarjetas Java Card existen también otras limitaciones, debidas en su mayor parte a que es un estándar reciente y que el API de seguridad ha sido estandarizado en su última versión (Java Card 2.1). Así en las tarjetas compatibles con versiones anteriores, los fabricantes proporcionan extensiones Java Card propietarias para acceder desde aplicaciones Java Card a los algoritmos criptográficos disponibles en la tarjeta, lo que obliga a tener desarrollos específicos para cada una de ellas.

B. Descarga de aplicaciones

El estándar Java Card no estandariza los procedimientos de descarga de aplicaciones en la tarjeta, por lo que la mayoría de fabricantes implementan sus propios mecanismos, por lo tanto para cada una de las tarjetas comerciales empleadas se deben de desarrollar módulos de descarga específicos. En la actualidad, existe una propuesta de VISA denominada Open Platform, [OP Card Specification], que estandariza el procedimiento de descarga de cardlet en la tarjeta y que está siendo aceptada e implementada por algunos de los principales fabricantes de tarjetas Java Card.

C. Implementación en tarjetas Java Card de Gemplus y Schlumberger

Para el desarrollo e implementación del cardlet se emplearon dos de los kits de desarrollo Java Card para Linux, que existían en el mercado[6]. El objetivo principal que se planteó al realizar la implementación en estas tarjetas fue restringirnos en la medida de lo posible a la especificación del estándar Java Card, sin emplear extensiones al API proporcionadas por los fabricantes.

C.1 GemXpresso RAD 211is de Gemplus

Este fue el primer kit que empleamos en nuestro desarrollo, [GemXpresso RAD 211 UG], [GemXpresso RAD 211 CRM], las características de las tarjetas Java Card, que contiene este kit son las siguientes:

- Java Card 2.1
- Visa Open Platform 2.0
- DES y 3-DES con restricciones de exportación.
- 20K bytes de EEPROM para la descarga de cardlets.

Algunas de las ventajas que proporciona el entorno de desarrollo que ofrece este kit es que sigue el estándar Visa Open Platform [OP Developer's Guide] para la descarga de aplicaciones, y además contiene un simulador, lo que permite depurar en gran parte el código del cardlet antes de ser descargado en la tarjeta.

La principal limitación que teniamos con estas tarjetas es que sólo implementan algoritmos de clave simétrica por lo que el cardlet que desarrollamos en este caso, se limita al almacenamiento de claves y certificados, que se transfieren al ordenador (incluida la clave privada) para realizar las operaciones criptográficas soportadas por nuestra implementación.

[6]En la actualidad ambos kits de desarrollo no se encuentran en venta en su versión para Linux

C.2 Cyberflex for Linux Starter's Kit 2.1

La necesidad de implementar operaciones criptográficas basadas en clave pública dentro de la tarjeta, nos llevo a la adquisición de un nuevo kit de desarrollo, el Cyberflex for Linux Starter's Kit 2.1 [Cyberflex SDK], las características de las tarjetas Java Card Cyberflex Access que proporciona el kit, son las siguientes:

- Java Card 2.0
- RSA, DES y 3-DES
- Sistema de ficheros ISO 7816-4.
- 15K bytes de EEPROM para la descarga de cardlets.

Gracias a las capacidades criptográficas de estas tarjetas, además del almacenamiento de claves y certificados, el cardlet que hemos desarrollado realiza funciones de firma RSA y de descifrado de claves RSA (`Unwrapkey`) según el algoritmo 3DES-EDE3/CBC.

Se adquirió este kit porque las tarjetas implementan RSA y así podiamos realizar la operación de firma internamente. La principal limitación con la que nos enfrentamos es que las tarjetas son Java Card 2.0 y en esta especificación todavía no se incluía un paquete de seguridad (`javacard.security` en Java Card 2.1). Así, para el acceso desde Java Card a las capacidades criptográficas, es necesario utilizar la extensión proporcionada por Schlumberger `javacardx.crypto`, por lo que la solución desarrollada sólo es válida para este tipo de tarjetas.

La clase desarrollada por Schlumberger presenta algunas limitaciones, que enumeramos a continuación y que están documentadas en su [FAQ Schlumberger] y en un anexo a los manuales del kit [Cyberflex PG]:

- Limitaciones de RSA desde Java Card:
  - Sólo se puede emplear RSA de 1024 bits.
  - Es necesario especificar la clave privada RSA en formato CRT(Chinese Remainder Theorem)
- Para descifrar la clave RSA internamente, era necesario implementar 3DES-EDE3/CBC, las tarjetas Cyberflex implementan 3-DES-EDE2/CBC, por lo que el algoritmo ha sido necesario implementarlo por software.

El mecanismo de descarga de aplicaciones que ofrece este kit de desarrollo es propietario de Schlumberger y se basa en la firma del cardlet según el algoritmo de clave simétrica DES.

VII. Conclusiones

El punto más débil del software criptográfico moderno está en el almacenamiento de claves. Las tarjetas inteligentes son una alternativa más segura al uso del almacenamiento tradicional en discos. En este artículo hemos presentado nuestro sistema de gestión de certificados basado en Java Card (JCCM), que presenta una alternativa flexible a la gestión de certificados de cualquier tipo en tarjetas inteligentes, frente a otros trabajos que emplean tarjetas no programables o bien no utilizan la facilidad de programación. Nuestro enfoque no se limita a adaptar la funcionalidad criptográfica predefinida en las tarjetas a la visión lógica que debe ofrecer un módulo PKCS #11, ni recorta la semántica definida por el estándar para adaptarla a la que ofrece el fabricante en cada tarjeta.

En la actualidad JCCM soporta dos tipos de tarjetas inteligentes: las GemXpresso RAD 211, y las Cyberflex Access de Schlumberger, y está disponible para su uso con Netscape para navegar de forma segura, cifrar y firmar correos.

La inclusión de tarjetas inteligentes en la cadena de seguridad cierra una de las áreas de riesgo al proteger de manera efectiva las claves privadas almacenadas en ella: estas claves no necesitan salir de la tarjeta al ser ésta capaz de efectuar operaciones criptográficas ella misma y la fabricación del chip incorpora medidas de protección de la información contenida en la EEPROM y de ocultación de la actividad del procesador. Sin embargo, hemos de ser conscientes de que su aportación a la cadena de seguridad de un sistema se limita a la ocultación de claves; existen otras áreas que presentan problemas potenciales de seguridad. Una de estas áreas es la confianza en el software que ejecuta el usuario; un programa malintencionado nunca podrá extraer una clave privada de la tarjeta, pero sí podrá emitir firmas digitales en nombre del usuario sin que éste sea consciente. La solución de estos problemas hasta la fecha pasa por dos alternativas: la firma completa del hardware y el software de una máquina y comprobación de la firma en cada arranque de la máquina, o bien por la definición de un nivel de seguridad en el que se encuentra la máquina y la certificación de todo el hardware y el software de la misma.

El trabajo futuro que estamos abordando es la integración de JCCM en un módulo PAM (Pluggable Authentication Module), para reforzar la seguridad de todas las aplicaciones que utilizamos normalmente y que utilizan PAM, desde entrar en la máquina hasta transferir ficheros o ejecuciones remotas.

References

[ISO/IEC 7816-4] "ISO/IEC 7816-4: Integrated circuit(s) cards with contacts. Part 4: Interindustry commands for interchange", ISO/IEC, 1995.

[ISO/IEC 7816-3] "ISO/IEC 7816-3: Integrated circuit(s) cards with contacts. Part 3: Electronic signals and transmission protocols", ISO/IEC, 1997.

[JCADG 2.1] "Java Card Applet Developer's Guide. Java Card Version 2.0", SUN Microsystems, Agosto de 1998.

[JCADG 2.0] 'Java Card Applet Developer's Guide. Java Card Version 2.1", SUN Microsystems, Agosto de 1999.

[OP Card Specification] "Open Platform Card Specification. Version 2.0.1", VISA, Abril de 2000.

[OP Developer's Guide] "Card Applet Developer's Guide", VISA, Marzo de 2000.

[GemXpresso RAD 211 UG] "GemXpresso RAD 211 User Guide Version 1.0", Gemplus, Octubre 1999

[GemXpresso RAD 211 CRM] "GemXpresso RAD 211 Card Reference Manual Version 1.0", Gemplus, Octubre 1999

[Cyberflex PG] "Cyberflex Access Developer's Series. Programmer's Guide", Schlumberger, Septiembre 1999.

[Cyberflex SDK] "Cyberflex Access Software Developer's Kit 2 - Release Notes", Schlumberger, Noviembre 1999.

[FAQ Schlumberger] Schlumberger, http://www.cyberflex.com/Support/support.html

[HTTPS] "HTTP Over TLS", Rescorla, E., IETF RFC 2818, Mayo 2000.

[X.509] "Internet X.509 Public Key Infrastructure Operational Protocols: FTP and HTTP". R. Housley, P. Hoffman. IETF RFC 2585, Mayo 1999.

[USENIX 99] "Design Principles for Tamper-Resistant Smartcard Processors" by Oliver Kömmerling, Markus Kuhn, Workshop on Smartcard Technology Proceedings, Chicago, Illinois, USA, Mayo 10-11, 1999

[SC SDK] 'Smart Card Developer's Kit", Scott B. Guthery, Timothy M. Jurgensen. Macmillan Technical Publishg. 1998.ISBN 1-57870-027-2.

[SC APP. DEV. JAVA] 'Smart Card. Application Developement Using Java", Uwe Hansmann, Martin S. Nicklous, Thomas Schack y Frank Seliger, Springer, 2000. ISBN 3-540-65829-7.

[PKCS#11] "PKCS #11 v2.10: Cryptographic Token Interface Standard", RSA Laboratories Inc., Diciembre 1999 (003-903052-210-000-000).

[PKCS#1] "PKCS #1 v2.1: RSA Cryptography Standard", RSA Laboratories Inc.

[PKCS#5] "PKCS #5 v2.0: Password-Based Cryptography Standard", RSA Laboratories Inc.

[PKCS#8] "PKCS #8 v1.2: Private-Key Information Syntax Standard", RSA Laboratories Inc.

[STALL99] 'Cryptography and Network Security: Principles and Practices", Stallings, W., 2ed, Prentice-Hall Inc., 1999

[SMARTSIGN] "Smart Sign", Tommaso Cucinotta, http://sourceforge.net/projects/smartsign

[GPKPKCS#11] "GemSAFE Products", Gemplus, http://www.gemplus.com/products/software/gemsafe/index.html

[SLBCBPKCS#11] "Cyberflex Access SDK", Schlumberger, http://www.cyberflex.com/Products