Breaking down architectural gaps in smart-card middleware design *

Tommaso Cucinotta, Marco Di Natale[1], and David Corcoran[2]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
[2] Identity Alliance, Austin, TX

Abstract. This paper presents an open and modular middleware for smart cards, interoperable across multiple card devices, and portable across various open platforms. The architectural design is centred around the definition of a smart card API that allows protected access to the storage and cryptographic facilities of a smart card. The proposed API allows partitioning of a smart card driver architecture into a lower *card-dependent* level, that formats and exchanges APDUs with the external device, and a higher *card-independent* level, that uses the API for implementing more sophisticated interfaces. The proposed architecture, along with a set of pilot applications such as secure remote shell, secure web services, local login and digital signature, has been developed and tested on various platforms, proving effectiveness of the new approach.

1 Introduction

Security of applications and services is becoming an increasingly important issue to be addressed since the early stages of the design and development of complex software systems. In order to achieve an adequate level of security while exchanging information or running transactions onto an open network, such as the Internet, cryptographic mechanisms need to be used. Cryptography enabled services guarantee that only authorised entities can access sensitive data or valuable services when cryptographic keys are properly managed by all parties.

Smart card (SC) technology is, among others, an enabler for guaranteeing the secure management of cryptographic keys. Card devices have a high degree of trustworthiness, for many reasons: a card owner always has physical control of the card; on-card architecture is very simple, hence on-board code and logic can be easily made functionally correct; SC hardware is designed to be tamper-proof, so that it is very hard and expensive to try to recover contained data by physical inspection. Smart cards are sufficiently powerful to perform cryptographic operations on-board, without any need to reveal cryptographic keys to the outside world. These operations are only allowed after a proper user identity verification, through the use of Personal Identification Numbers (PINs), or even biometrics information.

* This work has been partially supported by the European Commission within the IST project 2001-34820 ARTIST.

Even if smart cards have been widely adopted and supported on proprietary platforms, they are not being used on open platforms due to the lack of open solutions. On these systems, open source libraries and applications allow the use of cryptography for data protection, but the achieved security level is strongly limited because of the use of software-only cryptography. SC technology typically features a "proprietary" approach in which every manufacturer deliberately deviates from standards in order to give its products some added functionality and to link its customers to the company as long as possible. Standard APIs for interoperability do exist [1, 2], but only a few vendors provide their implementation on open platforms, and for only one or a limited set of devices. This situation discourages smart card integration and has the consequence of an overall reduction in the use of smart card devices, hindering the development of their potential in increasing security of computer applications and services.

As a result, computer systems based on open platform are especially subject to be overpopulated with cryptographic keys that are managed in software and stored onto hard-drives, possibly protected by weak passwords quickly chosen by careless users. The MUSCLE[3] Card middleware, which is being introduced in this paper, constitutes a step toward openness and simplification in smart card middleware design and implementation.

The paper is structured as follows. The next section introduces common concepts about SC middleware and makes an overview of other open architectures for smart cards. Section 3 introduces the proposed architecture and features an overview of the new API. Finally, we draw our conclusions in Section 4.

2 Background on smart card middleware

The world of smart cards is characterised by various card-reader (serial, PS/2, USB, wireless) and card device (storage only, crypto-enabled, GSM-enabled, programmable) types. In spite of the effort made by standard organisations [3, 4], card devices have many restrictions and non standard filesystem structures.

The simplest way of increasing an application or system security through the use of smart card technology is by delegating management and use of one or more cryptographic keys to the card device. For PKI applications, one or more public key certificates can be stored on the card for easing mobility of the card among various physical locations. This is usually achieved through common, high-level, application programming interfaces that support on-card operations in a manner that is independent of the card and reader devices.

Two APIs that have been defined for this purpose are PKCS#11 [1] by RSA Labs, and PCSC [2], Part 6, by the PCSC Workgroup. Such high level APIs are made available to applications through a smart card middleware that requires various drivers to be installed on the system, depending on the actual reader and card devices that are going to be used. A generic smart card middleware architecture is depicted in Figure 1(a).

[3] Movement for the Use of Smart Cards in a Linux Environment.

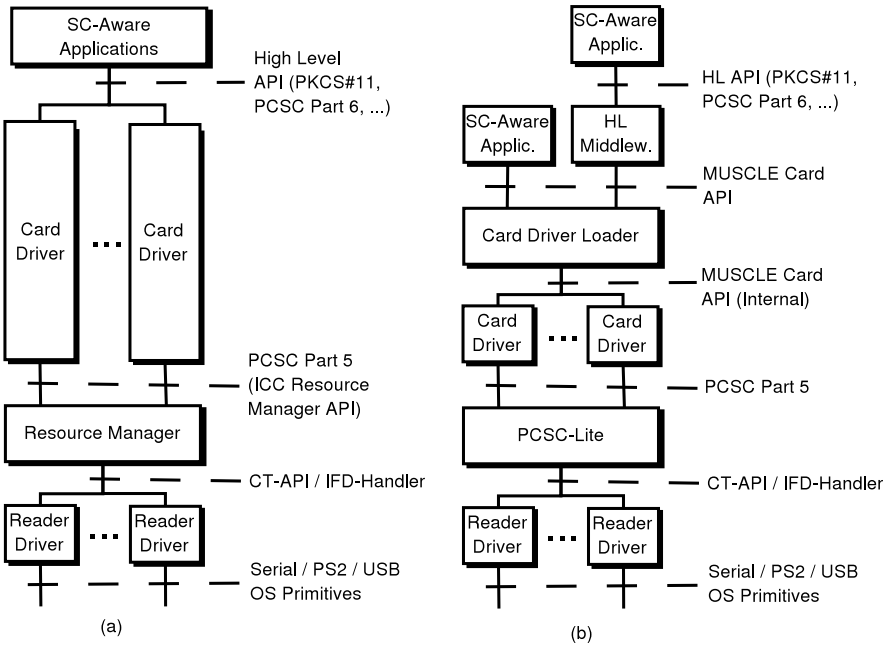

Fig. 1. (a) Architecture of a traditional smart card middleware. (b) Architecture of the proposed smart card middleware.

At the bottom layers, a resource manager component is required for managing the SC readers that are available on the system, and making their services available to higher level components, in a way that is independent of the hardware connected to the system. This is done through the PCSC ICC Resource Manager interface [2, Part 5], which provides function calls for listing the available readers, querying a reader about the inserted card(s), enabling or disabling the power to an inserted card, and establishing an exclusive or shared communication channel for data exchanges with a card. The reader driver takes care of translating the requests into the low-level Protocol Data Units (PDUs) to be transmitted to the reader through the low level serial OS primitives. Reader drivers implement the CT-API or the PCSC IFD-Handler API [2, Part 3, Appendix A], and the resource manager translates calls to the PCSC Part 5 interface to the appropriate lower level API calls. The higher software stack performs data exchanges through command APDUs compliant to the ISO T=0 or T=1 protocols [5].

The top level component of the middleware is traditionally a monolithic component, provided by card vendors, that implements the PKCS#11 or PCSC Part 6 interfaces. These APIs have calls that allow the application to locate, manage and use cryptographic keys and public key certificates that are available on the card. The card driver translates such requests into the appropriate lower level ISO T=0 or T=1 command APDUs to be exchanged with the card. Typically, it supports a range of similar card devices provided by the same vendor. Furthermore, it must comply with the higher level API, what requires additional tasks to be performed in the component, such as session management and transaction

handling. Such tasks are quite similar in the driver implementations provided by different vendors, where the only changes regard the specific way information is exchanged with the card by means of APDU exchanges. This is why we investigated on the possibility of introducing a further abstraction layer, breaking the traditional driver architecture through the use of a middle-level API.

In fact, our architecture has a lower level (LL) driver, which formats and exchanges command APDUs with the card device, and a higher level (HL) one, which performs the additional management tasks required for the compliance with the higher level interface. This is done through the introduction of a middle-level API, clearly identifying the boundary and commitments of the two sublayers. As it will be shown in Section 3, the main benefit of such an approach is that it is possible to write the HL-API-specific management code only once. Interoperability among card devices is achieved by writing, for each card, a different LL driver implementing the common middle-level API.

2.1 Related projects

The OpenSC [6] project provides a library and a set of utilities for accessing ISO 7816 [3] and PKCS#15 [7] compliant card devices. It provides a good set of middleware components, as well as modules for their integration within widely used secure applications, constituting an effective solution for integration of ISO 7816-4 and PKCS#15 compliant, pre-formatted devices. Though, various cards exist today with custom, proprietary APDUs for filesystem management, which adhere to ISO 7816-4 only in a read-only fashion, and/or do not respect the PKCS#15 standard for managing information about the on board cryptographic material. Such devices cannot be directly supported within this architecture, especially on the side of card-personalisation.

The SecTok [8] project provides a library for the management of files onto an ISO 7816-4 compliant device. It does not support cryptographic functionality of the devices, thus it cannot be used in the context of cryptographic smart cards.

The Open Card Framework (OCF) [9] is a Java based development platform for smart card development. It aims at reducing dependence among card terminal vendors, card operating system providers and card issuers, by the adoption of a consistent and expandable framework. The core architecture of OCF features two main parts: the CardTerminal layer, providing access to physical card terminals and inserted smart cards, and the CardService layer, providing support for the wide variety of card operating systems. OCF is a promising framework for smart card integration within Java applications. Despite the modular and expandable design, its main limitations are due to the lack of support of some readers due to the way I/O is managed at the lowest levels of the architecture, and the inherent difficulties and overhead needed in order to access such functionality from programs written in different programming languages than Java.

The GPKCS#11 project [10] aims at providing support functionality that ease the development of a PKCS#11 driver for cryptographic tokens. Unfortunately, the project lacked detailed documentation about its features, and it has not been maintained since year 2000.

The Common Data Security Architecture (CDSA) [11] is an open standard introducing an interoperable, multi-platform, extensible software infrastructure for providing high level security services to C and C++ applications. It features a common API for authentication, encryption, and security policy management. As far as smart card technology is concerned, the CDSA standard supports external cryptographic devices through the use of PKCS#11 modules, while the overall architecture is designed and focused around higher level security services, such as certificate and CRL management, verification of signatures, authentication, and others.

The architecture that is being introduced in this paper, at the authors' knowledge, is the only open architecture completely modular that allows multiple heterogeneous devices to be supported through the implementation of a common lower level API, which exposes sufficient functionality needed by most PKI applications since the time of issuing of the card by a CA, up to the final use of the device by applications. The efforts needed for the implementation of such drivers is limited, with respect to the full implementation of one of the well known standards for smart cards, such as PKCS#11 or PCSC level 6. Still connectivity with such standards is possible through the implementation of the higher level API through the MUSCLE Card API, what can be done in a separate module, and once and for all.

3 Proposed architecture

The middleware architecture of the MUSCLE Card project is shown in Figure 1(b). At the bottom layers, the PCSC-Lite project provides an open and stable daemon for managing the SC-related hardware resources of the PC (e.g. serial/USB ports, connected readers). Various readers are supported through reader drivers, most of which open source, implementing either the CT-API or the IFD-Handler interface. Devices connected to serial and PS2 ports need to be already connected when the daemon starts, while USB devices can be plugged at run-time, provided that the drivers are installed onto the system.

At the above layer, independence from the card is achieved by using a common API. Specifically, the Card Driver Loader, at the time the card is inserted, identifies the inserted device through the Answer To Reset (ATR) bytes, then loads dynamically the driver that can manage the card. Differently from traditional approaches, in which higher level APIs such as PKCS-11 or PCSC Level 6 are implemented by card drivers, in the proposed architecture a card driver implements a simpler API (see Section 3.1).

The API exposes basic storage, cryptographic and access control functionality to the host machine, independently of the kind of card device the host is using. This interface is inspired by the protocol introduced in [12], in that most function calls are directly mapped into the APDUs of the protocol. This layer has been implemented in various card drivers for card devices that are different in architecture and nature. Examples are Schlumberger Cyberflex Access 32K and Gemplus 211/PK cards, two programmable cards based on the JavaCard

platform; the Schlumberger Cryptoflex 16K card, which exposes a set of ISO 7816-4 APDUs for filesystem management, and custom commands for cryptographic operations; the US Department of Defence (DoD) card, which exposes a custom data model. Details on the proposed API follow in the next subsections. On top of our API, further application and middleware layers have been developed. Specifically, an open source PKCS-11 module, mapping the PKCS-11 API calls into the appropriate sequences of MUSCLE Card API function calls, has been developed.

As an alternative, applications can directly use the proposed API in order to talk to smart card devices at a lower level, and take advantage of the exposed functionality, like access control mechanisms based on multiple PINs or other authentication means. The API has been directly used for embedding smartcard technology into a set of target applications, within the Smart Sign project (http://smartsign.sourceforge.net): a command line digital signature application (sign-mcard), a variant of the OpenSSH software (openssh-mcard). A PAM [13] module has been directly developed using this API, allowing smartcard based user authentication for applications using PAM on Unix like systems, like the Unix login. Finally, a CSP module for Windows platforms has also been developed, integrating functionality of the exposed architecture into applications like MS Outlook, Internet Explorer and Windows login.

3.1 MUSCLE Card API Overview

This section features a technical overview of the proposed API. The discussion is focused on the introduction of the API main features, and explanation of the main design choices behind its development. The complete API specification [14] is available for download at the URL: *http://www.musclecard.com*.

Objectives and design choices Main aim of the API design is to provide higher layer software components with an open, simple, card independent framework which exhibits sufficient generality to meet the requirements of a multitude of target applications, including digital signature, secure e-mail, secure login, secure remote terminal and secure on-line web services, both PKI based and not.

These requirements have been identified in having a means of generating, importing, exporting, and using cryptographic keys on the card. Another requirement is to have a means of creating, reading, and writing generic data on the card in generic "containers". This is useful, for example, to store on the card a public key certificate associated with a private key. Access to some of these resources needs to be granted only after host application and user authentication.

The result is a simple and light interface that has been proved to be effective in allowing integration of smart card technology into secure applications, as shown by our sample application cases. The API design allows future extensions, like the use of alternative key types or authentication mechanisms, as proved by the biometrics extensions that have recently been added [15].

The API does not address sophisticated card services that might be needed by specific applications. Multi-key digital signatures and authentication schemes

may need specific functionalities to be provided through the use of multiple cards. These applications can still benefit from the exposed middleware by extending it with the required functionality, given the open nature of the project.

The set of functions available in the proposed API is summarised in Table 1. API functionality has been divided into 5 general function sets, giving access to one or more of our middleware class of services, namely: session management, data storage, cryptographic key management, PIN management, access control, and a set of miscellaneous further functions. In the following, we provide detailed information on the intended use of the various API calls.

Session mgmt	ListTokens, EstablishConnection, ReleaseConnection WaitForTokenEvent, CancelEventWait, BeginTransaction, EndTransaction
Data storage	CreateObject, DeleteObject, ListObjects, WriteObject, ReadObject
Cryptography	GenerateKeyPair, ComputeCrypt, ImportKey, ExportKey, ListKeys
PIN mgmt	CreatePIN, ChangePIN, UnblockPIN, ListPINs
Access ctrl	VerifyPIN, GetChallenge, ExtAuthenticate, GetStatus, LogOutAll
Miscellaneous	WriteFramework, GetCapabilities, ExtendedFeature

Table 1. MUSCLE Card API function set

Session management The API has a minimal set of functions allowing the enumeration of connected readers and inserted smart cards, and management of the connections to the card devices. Establishment of a connection to a card device is a prerequisite for the use of any of the other functions of the API. Specifically, the `ListTokens` function is able to enumerate readers connected to the system, readers which have a card inserted, along with the type of inserted device, and the list of all supported card devices in the system. Furthermore, an application is able to block and wait until a card insertion or removal by using the `WaitForTokenEvent` function. Once a card is inserted into a reader, the `EstablishConnection` and `ReleaseConnection` functions allow to reset the device and prepare it for subsequent commands. When connecting to a card, it is possible to select either exclusive or shared access to the card. In the latter case, it is possible to acquire an exclusive lock on the device with a call to the `BeginTransaction` function, and release it with the `EndTransaction` function.

Data storage services Our middleware allows the definition of simple containers for applications' data called *objects*, identified by means of a string identifier (OID). Access control is enforced on a per-object and per-operation basis, distinguishing among create, read, write and delete operations (more details will be given later). The data storage services suffice for the target applications cited in the beginning of Section 3.1, by allowing them to store, retrieve and manage data onto a card in a secure and controlled way.

The `CreateObject` function allows creation of an empty object on the card, providing the object name, size and access control list (see forward for details

about this). The same information may be visioned by applications for all existing on card objects through subsequent calls to the `ListObject` function. Reading and writing of data to and from objects is performed, respectively, through the `ReadObject` and `WriteObject` functions. Execution of these functions may be restricted on a per-object and per-operation basis. The API specification does not define specific object contents, leaving the applications total freedom on what to store onto a card, like user private information, application specific data or public key certificates. As far as the card storage capacity is concerned, the interface specification gives only a view of the total available memory on the device, through the `GetStatus` function.

Cryptographic services The API allows up to 16 keys to be managed on the card, identified by means of a numeric key identifier. A full key pair can be stored by using two key identifiers. Key types are those provided by the Java Card 2.1.1 API: RSA, DSA, DES, Triple DES, Triple DES with 3 keys. The interface is designed to allow further key types in the future. Operations provided on cryptographic keys are import/export from/to the host, computation of cryptograms, and listing of keys. All key operations except key listing are allowed only after proper host application or user authentication. The API allows key pairs to be directly generated on board guaranteeing the private key is not exposed outside the card. In this case the public key can be obtained with a call to `ExportKey`, right after the key pair generation. When a key pair is created on-board, the host application specifies under what conditions subsequent reading, overwriting and use operations are allowed for each of the keys in the pair. The same rules can be specified when importing a new key from the outside world.

Security model and access control enforcement A simple Access Control List (ACL) based model is defined to protect on-board objects, allowing operations to be performed only after proper host application and user authentication. This may be performed by means of a PIN code verification, a *challenge-response* cryptographic authentication protocol, or a combination of both methods. Furthermore, the API has been designed to allow future support for other identification schemes, like fingerprint verification. Access rules for on-card resources are specified by using the concept of *identity*. This term refers to one of several authentication mechanisms that host applications and users can use to be authenticated to a smart card. Identities, PINs, and cryptographic keys are referred to by means of numeric identifiers. Different types of identity are defined: identities n.0-7 are labelled as *PIN-based* and are associated, respectively, with PIN codes n.0-7; identities n.8-13 are said *strong* and are associated, respectively, with cryptographic keys n.0-5 for the purpose of running challenge-response cryptographic authentication protocols; identities n.14-15 are *reserved* [4].

A successful run of any of the authentication mechanisms causes the *log in* of the associated identity, in addition to identities already logged in. The use of multiple identities allows a host application to switch to a higher security

[4] The fingerprint verification mechanism recently developed uses identity n.14.

level that grants access to more of the card's capabilities, as it runs additional authentication mechanisms. Furthermore the *LogOutAll* command allows a host application to return back to the unauthenticated security status.

An ACL specifies which identities are required to grant access to each operation of each object or key. Object operations are *read*, *write* and *delete*. Key operations are *overwrite* (either by means of regeneration or by means of import), *export*, and *use*. An ACL associated with an object or key is specified by means of three Access Control Words (ACW), each one relating to an operation. An ACW consists of 16 bits. Each bit corresponds to one of the 16 identities that can be logged in. An all-zero ACW means that the operation is publicly available, that is a host application can perform it without any prior authentication. An ACW with one or more bits set means that all of the corresponding identities must be logged in at the time the operation is performed. An all-one ACW has the special meaning of completely disabling the operation, independently of the connection security status. This is useful to disable reading of private keys.

PIN management services Functions have been defined for PIN management, allowing to create, verify, change and unblock PINs. Specifically, the *CreatePIN* function allows to create a new PIN on the card, provided that the transport PIN has already been verified, and the *ListPIN* function allows listing of the existing PIN codes. Up to eight PIN codes are allowed in principle to be created onto a single card, though the actual maximum number depends on the underlying device, and may be queried by using the *GetCapabilities* function. The *VerifyPIN* function allows verification of a PIN code, and, if successful, logs in the corresponding identity. Finally, the *ChangePIN* function may be used to change the current PIN value, and the *UnblockPIN* function to unblock it after it blocked due to several verification tries with the wrong code.

Extensibility Our middleware allows connectivity to smart card devices at a lower level than the one that is usually required for the implementation of standard PKCS#11 or PCSC interfaces. The set of functionality that is exposed to applications has been voluntarily kept small, in order to achieve a simple API. Particular attention has been paid to extensions that could be needed in the future. In order to allow such extensions to be performed without compromising the previously developed software, the middleware has versioning built into it. The version information is available through the *GetStatus* command, by means of *minor* and *major* version numbers.

Card specific behaviour The API which has been just introduced provides a unified means, for higher level middleware components as well as applications, to access the described smart card services in a unified, card-independent way. However, it must be noted that only a JavaCard device with the MUSCLE Card Applet on-board is able to support the full set of functionality available through this API. Each specific card generally supports only a subset of such functionality. The API provides, through the *GetCapabilities* function, a means for query-

ing what features are supported by the particular device that is connected to the system.

4 Conclusions

In this paper we described an open middleware for smart cards, which is highly modular due to the adoption of a new interface layer that abstracts from the specifics of a card. Such interface has been designed to support minimal functionality needed by applications that use smart card devices to manage cryptographic keys and other kind of data, e.g. public key certificates. In the proposed middleware architecture, a traditional smart card driver is split into two sublayers: the lower level one focuses on abstracting the specifics of each single device; the higher level one implements a standard interface, such as PKCS#11, still leaving the applications freedom to use the lower level interface, if needed. For example, a smart card aware, biometrics enhanced, application can directly use the middle level interface for using added functionality. Development of target PKI enabled applications proved effectiveness of the new approach.

References

1. RSA Laboratories: PKCS-11 version 2.1.1 Final Draft: Cryptographic Token Interface Standard. (2001)
2. PCSC Workgroup: Interoperability Specification for ICCs and Personal Computer Systems. (1997)
3. International Standard Organization: ISO/IEC 7816-4/7/8/9: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Parts 4, 7, 8, 9. (1995)
4. GSA: Government Smart Card Interoperability Specification: Contract Modification. (2000)
5. International Standard Organization: ISO/IEC 7816-3: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 3. (1989)
6. Kirch, O.: OpenSC - smart cards on linux. In: Proc. of the 10^{th} International Linux System Technology Conference, Saarbruecken, Germany (2003)
7. RSA Laboratories: PKCS-15: A Cryptographic Token Information Format Standard. (1999)
8. Center for Information Technology Integration (CITI), University of Michigan: Sectok library and applications. (2001)
9. OpenCard Consortium: OpenCard Framework General Information Web Document. second edn. (1998)
10. TrustCenter: gpkcs11 - GNU PKCS#11 implementation. (2000)
11. The Open Group: Common Security: CDSA and CSSM, Version 2.3. (2000)
12. Cucinotta, T., Natale, M.D., Corcoran, D.: A protocol for programmable smart cards. In: Proc. of DEXA 2003, Prague, Czech Republic (2003)
13. Samar, V., Schemers, R.: Request for comments 86.0: Unified login with pluggable authentication modules (PAM) (1995)
14. Corcoran, D., Cucinotta, T.: MUSCLE Card API, version 1.3.0. (2001)
15. Brigo, R.: Protecting smart card access by on-board biometrics verification. Computer Engineering Thesis. University of Pisa (2002)