Securing Operating System Services Based on Smart Cards

Luigi Catuogno, Roberto Gassirà, Michele Masullo, and Ivan Visconti

Dipartimento di Informatica ed Applicazioni,
Università degli Studi di Salerno - Italy
{luicat, robgas, micmas, visconti}@dia.unisa.it

Abstract. The executions of operating system services based on smart cards allow one to personalize some functionalities of the operating system by using the secret information stored in a smart card and the basic computations that a smart card can perform. However, current solutions for integrating smart card features in operating system services require at least a partial execution of the operating system functionalities at "user level". Such executions decrease the security and the performance of the system as they are less robust compared to the kernel-level ones.

In this paper we present the design and implementation of SmartK, a kernel module that integrates directly in the Linux kernel the support of smart cards. The use of SmartK allows one to securely personalize an operating system service still maintaining its execution at kernel level.

1 Introduction

Cryptographic protocols allow the execution of many real world economic transactions (e.g., auctions, voting) in the digital world. Nevertheless, an important role in the digital world is played by the hardware and software architectures that run cryptographic protocols. Among the different hardware and software components, a central role is played by smart cards.

Smart card is one of the most interesting technologies that have been proposed in the past and are nowadays crucially used in many digital transactions (e.g., inside satellite decoders, ATM machines). Originally, development of card-*aware* applications was a non-trivial task since there was a lack of high-level card programming languages, standard devices and development tools. Currently, several smart-card manufacturers have joined into consortia in order to define common standards for each aspect of the interaction with smart cards (e.g., physical and electrical specification for cards and readers, specifications of the provided services, communication protocols among cards, readers and host computers, data representation). Moreover, many high-level tools that satisfy many requirements of software designers and developers have been recently introduced. Such tools are *application-oriented*, that is, their use is reasonable for user-level applications but it is not practical for kernel-level executions.

We focus on the use of smart cards in operating system services. Here, the smart card allows one to personalize some functionalities of the operating system.

S. Katsikas, J. López, G. Pernul (Eds.): TrustBus 2005, LNCS 3592, pp. 321–330, 2005.

Indeed, a smart card is a tamper-resistant miniature computer that performs some basic computations on input a secret information.

However, current solutions for supporting smart card features in operating system services require at least a partial execution of the operating system functionalities at "user level". Unfortunately the execution of system functionalities at user level decreases the security of the system as user-level executions are less robust compared to the kernel-level ones. Indeed, attacks to the kernel are generally harder compared to attacks to user level applications since kernel code is specifically protected to avoid tracking and replacing attacks. Furthermore, kernel-level applications offer a better performance since they are in general not affected by context switches or frequent copies of large memory buffers among user and kernel space. Erez Zadok, in [28,17] gives accurate and strong motivations in flavor of kernel-level implementations of system-relevant applications.

In this paper we present the design and implementation of SmartK, a kernel module that integrates directly in the Linux kernel the support of smart cards. The use of SmartK allows one to securely personalize an operating system service still maintaining the execution at kernel level. More generally, SmartK is a compact and easy-to-use tool for software development of kernel applications (e.g., device drivers, filesystems, kernel modules). Our design of SmartK focuses on modularity, therefore it is possible to plug in (transparently to the applications) different modules that allow the applications to work with different cards and different readers connected to different ports. Moreover, the size of SmartK is very tiny and does not significantly affects the performance of the kernel.

We stress that the aim of SmartK is not necessarily to replace the existing smart-card frameworks. Indeed, some of them are quite suitable for many user applications. Instead, the use of SmartK is crucial when card-based services must be supported by the kernel itself. In such cases, SmartK outperforms the existing available tools. We stress also that a kernel module that runs a large high-level framework has a large (and negative) impact on the performance of the system.

2 Background

Specifications. Informally, a smart card is a plastic card (with the same size of a credit card) with either a magnetic strip or a micro chip. The physical properties of a smart card (e.g., the size, the position of contacts, their number), the electrical specifications (e.g., power, signals) and the communication protocols have been standardized, in order to allow cards, readers and applications (*off-card* applications) produced by different factories to be used together. The standard ISO-7816[9] provides a definition of these characteristics for a smart card. The card and the reader communicate by means of a master/slave half-duplex protocol. Once the card is inserted in the slot, the reader powers on it and sends to it the *reset* signal. The card sends back an important message called *Answer To Reset* (ATR). The ATR message contains all information needed to establish the connection between card and reader. The ISO-7816/3 document defines the format of the ATR message and two communication protocol: $T = 0$ and $T = 1$.

The $T = 0$ protocol is byte-oriented, and allows one to send just one command per time, the $T = 1$ is a block-oriented protocol and allows one to send sequences of commands.

ISO-7816/4 commands are sent to the card as a record called APDU (Application Protocol Data Unit) that contains the description of the invoked command and its arguments. The card also replies to the commands by means of another type of record: the Response APDU.

Development Frameworks. The known smart card frameworks are user-oriented, therefore they can be used by operating system services in the following two ways: 1) The frameworks are executed at user level. This is precisely what we want to avoid since for security reasons, operating system services should be run at kernel level. 2) The frameworks are compiled directly in the kernel. This *brute-force* approach hurts the performance of the kernel.

The "Application Independent Card Terminal Application Programming Interface for ICC applications" (CT-API)[7], is a simple package for the development of card-*aware* applications. CT-API is a library that manages the specific reader's device driver and provides a raw programming interface.

The "Interoperability Specification for ICCs and Personal Computer Systems" (PC/SC, for short)[18,19] is a standard definition of a complete framework for smart card deployment. PC/SC specifies the architecture and the components of a distributed "card environment", the services provided by each component and the protocols that components use to communicate. Moreover, PC/SC also defines a standard API for the development of off-card applications. PC/SC was initially used only on MS Windows platforms, but recently, it is also used in UNIX-like systems, with the support of the "Movement for the Use of Smart Cards in a Linux Environment (MUSCLE)"[15]. Actually, both CT-API and PC/SC implement a raw programming interface for the interaction with the smart card.

The Open Card Framework (OCF)[16] offers a powerful tool for developers of smart card-enabled software, based on the Java technology. OFC provides a high-level programming interface (composed of several Java classes) that implements the ISO-7816 protocol.

The RSA Laboratories produces and maintains the PKCS standard documents. This documents introduce a widely accepted set of specifications for cryptographic data structures, operations and procedures. Documents PKCS11 and PKCS15[21,22] concern interface and information format of Cryptographic Tokens (a set of cryptographic capable devices that includes smart cards). Moreover, they define an architecture and an API for the development of cryptographic applications based on these tokens.

The Smart Card File System (SCFS)[11] is a tool that allows the host machine to mount a smart card as a disk, and therefore to access the stored data by means of the standard UNIX system calls.

Webcard[20] implements a tiny web server on a Java card. Card-*ware* applications access to data stored on the card by using the HTTP protocol.

Trusted Computing Architectures. The Trusted Computing Group [26] consortium has been formed by some important hardware and software corporations (*e.g.* Microsoft, IBM and Intel), in order to define a standard technology for enhancing the security of computing environments that span over different platforms and devices.

According to the TCG specifications, a *Trusted Platform*, should feature a safe storage for sensitive data, the capacity of verifying the integrity of a platform component and the capability to prove to a challenger the integrity of the platform through an attestation.

The Trusted Platform Module (TPM) is a hardware device available for different platforms like PCs, PDAs and cellular phones that implement the features listed above. Applications, firmware and the other components that use the TPM features, are developed on top of a software layer defined by the Software Stack Specification (TSS). Moreover, the TPM provides cryptographic functions such as hashing, random number generation, asymmetric key generation and encryption/decryption.

Microsoft Next Generation Secure Computing Base (NGSCB)[14], is one of earliest technology based on the TCG specifications and will be integrated in the upcoming version of the Windows operating system. Microsoft stresses that NGSCB provides a lot of benefits to costumers (e.g., protection against viruses and unauthorized accesses, platform and data integrities, enhanced authentications) but many known researchers[1,25,24] are afraid by these benefits.

The trusted computing architectures could be used to achieve the secure and efficient execution of operating system services. However, these architectures are not flexible since the cryptographic tasks are only based on the secret information encoded in a secure chip plugged in the motherboard. Moreover, the cost of such technologies and the trust and ethical issues that they generate slowdown their effective use.

3 Design and Implementation of SmartK

SmartK provides a simple framework for the management of smart cards at kernel level. Specifically, the end user of the SmartK API is a generic kernel module that features a service based on smart cards. This is crucial for our main contribution, i.e., securing operating system services based on smart cards. In the design of SmartK we therefore focus on achieving an efficient kernel module that serves both other kernel modules and user applications.

SmartK exposes a very simple interface that we describe below.

- `smartk_init_card` starts the connection to the card. This procedure supplies power to the card, receives the ATR message from the card, parses it and finally, collects and stores all communication parameters like the response time (and the timeout) of the card, the communication protocol and the data representation adopted.

- `smartk_data` sends commands and receives the corresponding responses. This function transparently wraps all steps needed by data transfer, according with the information collected during the initialization.
- `smark_cleanup_card` closes the communication, cleans all memory buffers, and turns off the power to the card.

This kind of interface implements any off-card application. A similar approach (at user level) can be found in the CT-API. The applications communicate by means of the I/O port, with the reader and the card. More precisely, the application organizes the data as specified by the protocol provided by the card (for example the $T = 0$ protocol). Then the application sends the data to the reader through the port. This is achieved by sending the proper signals and, if necessary, re-encoding the data with the communication parameters that have been negotiated during the startup. Therefore our framework has been designed following an object oriented style. For each part of the communication, SmartK features a specific class and each module of SmartK implements an object of a class.

SmartK is designed to be modular, it can support different readers, each one potentially connected to the host machine by means of a different port (e.g., serial, USB). Specifically, SmartK is composed by the following four modules[1].

- `smartk.o` is the core of the framework and provides the interfaces to the kernel-level applications and to the the other modules of SmartK;
- `pt_t0_smartk.o` implements the API of SmartK according to the $T = 0$ protocol;
- `ifd_towitoko_smartk.o` is the Towitoko reader's driver;
- `io_serial_smartk.o` is a simple interface for the communication with a serial port.

The module `smartk.o` is the skeleton of the whole framework. It provides an object-oriented infrastructure on top of which the other modules are plugged in. Moreover, it handles the object *core* of the class *smartk* that reports the status of the card (e.g., ATR, communication parameters) and provides a general interface to the objects implemented by the other modules.

In order to achieve the modularity of the architecture, all methods of the different objects are referenced by a pointer of the `core` object. Thus, each object can invoke the methods of each other object by reaching them only through this object. This approach maintains each module independent of each other module and limits the number of symbols exported by each module.

Implementation details. We call "registration" the assignment of pointers of the object *core*. The module *smartk.o* provides the methods `register_protocol_smartk`, `register_ifd_smartk` and `register_io_smartk` that are executed to plug in the components in the framework. These methods link the related objects to the object *core*. Once the `smartk.o` module has been loaded, it instances the

[1] We now discuss the specific case of using a towitoko micro reader that is connected to a serial port, since this is the solution that we have effectively implemented. The discussion however can be generalized to any reader and any port.

smartk object `core`. Then, during their initialization phase, the other modules instance their own objects and register them by means of the corresponding registration procedure.

A *pt_smartk* object implements the communication protocol with the smart card (in our prototype, only protocol $T = 0$ is provided). It features a very simple interface composed by the following three methods: `activate_card`, `data` and `disactivate_card`.

An *ifd_smartk* object implements the functions required for the communication with the reader. Its methods allow one to enable and disable the reader and the card, transmit/receive data and power on/off the card.

A *io_smartk* object takes care of maintaining the status of the communication with the I/O port. This object summarizes the status of the port (the serial port in our prototype) and provides a set of methods to init/free the port, set/get communication parameters (baud rate, parity etc.), send/receive data to/from the port.

The communication protocol is implemented by the object *t0* of the class *pt_smartk* (module `pt_t0_smartk.o`). This object implements the $T = 0$ protocol as defined by the ISO-7816/3 document. Once the module `pt_t0_smartk.o` has been loaded, it registers the object `t0`. The interactions with the reader are performed by means of the methods of the `ifd` object (through their pointers in the core object).

The object *towitoko* of the class *ifd_smartk* (module `ifd_towitoko_smartk.o`) implements the driver of the reader. The module startup procedure initializes the reader through the method `init_reader` and registers the object by means of the `register_ifd_smartk` function. This function verifies that the serial port control module has been loaded and subsequently configures the port according to the reader properties. The object `towitoko` interacts with the serial port through methods of the object `serial`.

The object *serial* of the class *io_smartk* (module `io_serial_smartk.o`) performs the communication with the serial port. This module implements new *line discipline*[23]: the mechanism through with the linux kernel manages the data flow through the serial port. Once the line discipline has been enabled, the module instantiates the object `serial`, initializes and registers it by means of `register_io_smartk`.

As discussed above, all aspects of the interaction with the card are modular, thus, for example, in order to use a different reader one has to implement a different module `ifd-`*something*`.o` that has to be loaded instead of our IFD handler. Obviously, the new module has to provide a new implementation of the `ifd` object.

The test module. The module `test_mod` is a practical example of a SmartK end-user module. It was initially developed for debugging purposes, but it is an useful tool for the development of simple user-level card-*aware* applications. More precisely, this module is an example of how to write a kernel service that uses SmartK. Specifically, the services given by this module allow user applications the use of any reader, card and port by means of SmartK.

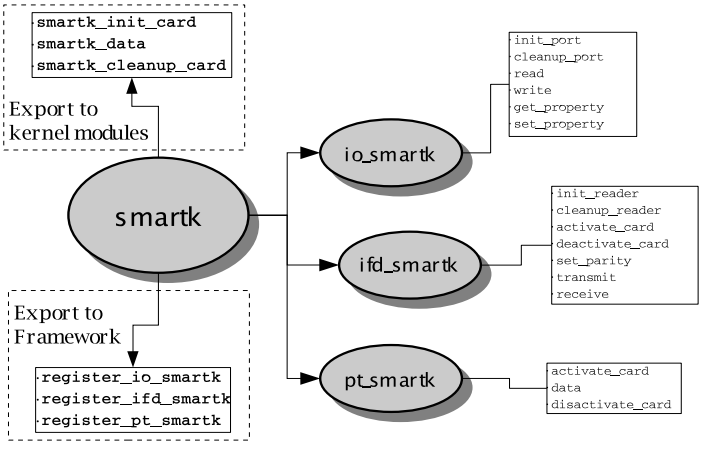

Fig. 1. SmartK data structures

Technically, `test_mod` allows user applications to communicate with smart cards by means of the usual I/O system calls on a character device (i.e., `/dev/smartk`). When the user application (*user*, for short) opens the device, the module executes the `activate_card` method of SmartK that initializes the communication and locks the device. When *user* closes the device, the module closes the communication, unlocks the device and cleans all buffers (`disactivate_card`). The `write()` operation uses the SmartK's `smartk_data` method to send APDUs to the card and to get the responses. The module keeps a private buffer where the responses returned by the `smartk_data` call are stored.

4 Securing Operating System Services

Here we discuss as a proof of concept two cases in which SmartK can be reliably used for securing operating system services based on smart cards.

Kerberos. The setting in which Kerberos [12,13] works is the following. There exists an open distributed computing environment (DCE) where the users of the workstations cannot be trusted. The setting is hostile since an intruder could pretend to be someone else. Therefore, an authentication system must be used.

Kerberos is an authentication system based on the existence of a trusted third-party that authenticate users of a DCE. More specifically, in case a user needs a service, he asks for a credential from the Kerberos authentication server (AS). The credential can be later sent to the ticket granting server (TGS) to obtain a service ticket. Finally, the service ticket allows the user to get the service from the corresponding server. The security problem of Kerberos is that an attacker can use a password guessing approach (by means of an off-line attack) to obtain the credential of another user. This problem was considered by [8]

where they proposed the use of smart cards for performing user authentication in Kerberos.

Consider now the case of an operating system that needs services from another system. In this case SmartK has a crucial role for securely run this transaction. Indeed, the functionality offered by the smart card for system authentication is run completely at kernel level.

Run-time verification of executables. Run-time verification of executables constitutes a typical field of application for SmartK. Indeed, this is a service that is implemented at kernel level, since the kernel parses and runs executables. We stress that the integration of a kernel-level architecture and a user-level smart card interface is unsafe and impractical. The WLF project[4] provides kernel modules for this verification process. It is build on top of AEGIS[2,10] that provides an architecture for the secure loading of the operating system during the bootstrap. We now briefly introduce WLF and describe the implementation of a smart card-based key management scheme that has been built on top of SmartK.

WLF Overview. The WLF project [4] proposes a prototypal implementation of an architecture for integrity checking of executables (both ELF binaries and script files) at run time for the Linux operating system. In a system equipped with WLF, all executables have been signed before their installation. The kernel (that is assumed to be safe) is provided with the public keys of the trusted software providers. Each time an execution is invoked, the kernel verifiers the corresponding files. If the verification succeeds, the execution is performed as usual, otherwise, the execution fails. In the Linux kernel, each executable is interpreted and executed by its proper handler. In a WLF system each handler includes a `verify()` function that executes the signature verification task. Public keys are managed by a distinct module (that we refer to as *key agent*), that takes care of loading keys from a given repository and providing them to a WLF handler.

The SmartK Key Management Scheme for WLF. The key agent in WLF is a kernel module that takes care of loading in memory the public keys from the storage device and provides them (on demand) to the WLF handlers. Currently, WLF is equipped with two key management schemes that were developed as proofs of concept: the *basic* and the *floppy* key management scheme (respectively BKM and FKM). The BKM simply satisfies testing requirements and loads public keys from a character device (`/dev/wlf`). Users push keys (contained in a file) into the kernel by means of an `ioctl` call on the device. The FKM loads keys from a read-only floppy disk. It comes out trivially that both systems are not suitable to be used in a real-world context.

The SmartK Key Management scheme (SKM) is a kernel module that implements a key agent for WLF. Since it is loaded, the SKM loads in memory all public keys that are stored on a a given smart card, that we refer to as *WLFCard*, by means of the APIs of SmartK, and then, it provides to the WLF handlers all required public keys.

5 Concluding Remarks

In this paper, we have discussed security issues for operating system services based on smart cards. First we have introduced the importance of using of smart cards for digital transactions. Then we have given the rationale for the need of a kernel-level framework for integrating smart-card features in the kernel of the operating system. Then we discussed the design and implementation of SmartK: a kernel-level framework for development of smart card-based services and applications for the Linux operating system. The integration in the kernel of such a tool, achieves a more compact and robust implementation of any intrinsically kernel-level security service based on smart card features. We have finally discussed the use of SmartK for operating system authentications (Kerberos) and we presented the implementation of a Key Agent for WLF, an operating system service for the verification of the integrity of Linux executables at run time. As we have discussed, such applications represent a typical example of off-card applications that should be run at kernel level and hence, are suitable "end-user" for SmartK. SmartK does not significantly affect the performance of the kernel and does not significantly increase the size of the kernel memory image as the total size of the modules is less than 20 kbytes. SmartK has been developed on a Linux operating system with kernel 2.4.20[5,23], the only reader currently supported is the Towitoko micro (serial port). Only the card communication protocol $T = 0$ has been partially implemented. We also implemented a simple management application that provides the usual administrative functionalities (e.g., format card, create and store keys) built on top of the PC/SC lite framework version 1.1.1[6]. Sources are available on the SmartK Home Page at the URL `http://smartk.dia.unisa.it`.

Acknowledgements. We would like to thank Pino Persiano and the anonymous reviewers for many useful suggestions and comments. The work presented in this paper has been supported in part by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT.

References

1. Ross Anderson (2003) TCPA Frequently Asked Questions. `http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html`.
2. W. Arbaugh, D. Farber, J. Smith (1997) A Secure and Reliable Bootstrap Architecture. Proc. of IEEE Symposium on Security and Privacy '97, pp. 65–71.
3. S. M. Beattie, A. P. Black, C. Cowan, C. Pu, L. P. Yang (2000) CryptoMark: Locking the Stable door ahead of the Trojan Horse. White Paper, WireX Communications Inc.
4. L. Catuogno, I. Visconti (2004) An Architecture for Kernel-Level Verification of Executables at Run Time. The Computer Journal, Vol. 47, Num. 5, Pages 511-526.
5. D. P. Bovet, M. Cesati (2002) Understanding the Linux Kernel (second edition). O'Reilly Associates, Inc.
6. David Corcoran (1999) PC/SC lite API version 1.1.1. `http://www.linuxnet.com`.

7. Detusche Telekom *et al.* (1998) Application Independent Card Terminal Application Programming Interface for ICC Applications.
8. G. Gaskell, M. Looi (1995) Integrating Smart Cards Into Authentication Systems. Cryptography: Policy and Algorithms, pp. 270-281.
9. The International Organization for Standardization and The International Electrotechnical Commission (1995) ISO/IEC 7816 parts 1-4: Information technology - Identification cards - Integrated circuit(s) cards with contacts.
10. N. Itoi, W. A. Arbaugh, S. J. Pollak, D. M. Reeves (2001), Personal Secure Booting. LNCS vol. 2119, pp. 130–144.
11. N. Itoi, P. Honeyman, J. Rees (1999) SCFS: A UNIX Filesystem for Smartcards. *Proc. of the First USENIX Workshop on Smartcard Technology*, pp. 107-118.
12. B. Clifford Neuman and Theodore Ts'o (1994) Kerberos: An Authentication Service for Computer Networks, IEEE Communications, 32(9):33-38.
13. John T. Kohl, B. Clifford Neuman, and Theodore Y. T'so (1994) The Evolution of the Kerberos Authentication System. In Distributed Open Systems, pages 78-94. IEEE Computer Society Press.
14. Microsoft Corporation (2003), Security Model for the Next-Generation Secure Computing Base. `http://www.microsoft.com`.
15. MUSCLE (Movement for the use of smart cards in a Linux Environment). `http://www.linuxnet.com`.
16. Opencard Consortium (1998) OpenCard Framework, General Information Web Document. `http://www.opencard.org`.
17. S. Patil, A. Kashyap, G. Sivathanu, E. Zadok (2004) I3FS an In-Kernel Integrity Checker and Intrusion Detection File System Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA '04).
18. PC/SC workgroup, (1997) Presentation of the Interoperability specification for ICCs and Personal Computer System (PC/SC) Revision 1.0, parts 1-8. `http://www.pcscworkgroup.com/`.
19. PC/SC workgroup, (1999) Presentation of the Interoperability specification for ICCs and Personal Computer System (PC/SC), Revision 2.0. White Paper, `http://www.pcscworkgroup.com/`.
20. J. Rees, P. Honeyman (2000) Webcard: a Java Card Web Server, Proc. of CARDIS 2000, pp. 197-208.
21. RSA Security Inc. (2004) PKCS11: Cryptographic Token Interface Standard v.2.20 `http://www.rsasecurity.com/`.
22. RSA Security Inc. (2000) PKCS15: Cryptographic Token Information Format Standard v.1.1 `http://www.rsasecurity.com/`.
23. A. Rubini, J. Corbet, (2001) Linux Device Drivers, second edition. O'Reilly Associates, Inc.
24. Seth Schoen (2003) Trusted Computing: Promise and Risk, Report of Electronic Frontier Foundation. `http://www.eff.org`.
25. Richard Stallman (2002) Can you trust your computer. `http://www.gnu.org/philosophy/can-you-trust.html`.
26. Trusted Computing Group (2004), TCG Specification Architecture Overview.
27. L. van Doorn, G. Ballintijn, W. A. Arbaugh (2001) Signed Executables for Linux. University of Maryland Technical Report CS-TR-4259.
28. Erez Zadok (1999) Stackable File System as a Security Tool. CS dept. Columbia University Technical Report CUCS-036-99.