Abstract

A smart card is a tamper-resistant miniature computer that performs some basic computations on input a secret information. So far, smart cards have been widely used for securing many digital transactions (e.g., pay television, ATM machines).

We focus on the implementation of operating system security services leveraging on smart cards. This very challenging feature allows one to personalize some functionalities of the operating system by simply changing a smart card. Current solutions for integrating smart card features in operating system services require at least a partial execution of some of the operating system functionalities at “user level”. Unfortunately, system functionalities built on top of components lying at both kernel and user levels may negatively affect the overall system security, due to the introduction of multiple points of failure.

In this work, we present the design and implementation of SmartK: a framework that integrates features of smart cards uniquely in the Linux kernel. In order to validate our approach, we propose a host of enhancements to the Linux operating system built on top of SmartK: 1) in-kernel clients’ authentication with Kerberos; 2) execution of trusted code; 3) key management in secure network filesystems.

In particular, we present an experimental Linux OS distribution (SalSA), which addresses the security issues related to downloading packages and to updating an operating system through the Internet.

1. Introduction

Cryptographic protocols allow the execution of many real-world economic transactions (e.g., auctions, voting) in the digital world. An important part of the worldwide economy is therefore moving to the digital world where transactions can be executed by enjoying many desired properties as efficiency, security and privacy.

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 that have been proposed in the past, a central role is played by smart cards.

Smart cards are nowadays crucially used in many digital transactions (e.g., pay television, ATM machines) and their use is still growing. 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. Indeed, a smart card is a tamper-resistant miniature computer that performs some basic computations on input a secret information. The use of such computations has been for instance proposed for remote logon (see Kerberos (Neuman and Ts’o, *Corresponding author.*

E-mail addresses: luicat@dia.unisa.it (L. Catuogno), robgas@dia.unisa.it (R. Gassirà), micmas@dia.unisa.it (M. Masullo), visconti@dia.unisa.it (I. Visconti).

1363-4127/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
and can be used to obtain ad-hoc operating system services.

However, current solutions for joining smart cards in operating system services require that the execution of card-based OS services takes partially place at “user level”, as the majority of existing smart card frameworks are implemented as user-space middleware and libraries.

Unfortunately the execution of OS functionalities composed of multiple components lying at both kernel and user level, may lead to reduced security and efficiency of the whole system. Instead the execution of kernel level-only services results in better performance and security. Indeed a considerable overhead due to the communication between components can be saved and potential multiple points of failure could be removed. Moreover, kernel level-only applications offer better performances since they are in general not affected by context switches or frequent copies of large memory buffers among user and kernel space (Fig. 1).

Finally, we observe that pushing smart card features completely inside the kernel decreases the “surface” available to potential threats, and consequently helps to reduce the size of the system’s trusted computing base for all those applications that rely on the OS security and integrity.

1.1. Our results

In this paper we present the design and implementation of SmartK, a framework that integrates directly in the Linux kernel the support for smart cards. The use of SmartK allows one to improve the security of any operating system service, leveraging on such technology, still maintaining the whole execution at kernel level.

We stress that SmartK is not a replacement for all current existing smart-card frameworks. Indeed, user-level smart card frameworks provides several features that allow the management of multiple IDs and application level protocols such as peer authentication, data encryption, digital signatures and secure channels establishment. Typical smart-card based applications mainly rely on such technologies in order to enforce user-grained security policies and often omit to implement platform authentication features, delegating them to the underlying OS (e.g., ATMs use smart cards to authenticate users).

However, SmartK is thought to fit those cases in which the OS needs to operate within a secured infrastructure “as a whole”, managing exclusively identification credentials and keys on a per-platform base. Thus, SmartK can be used to handle the credential needed, for example, to obtain an IP address by a DHCP server, as well as, to mount a remote filesystem. Hence, we may have a Linux platform that authenticates itself on a secure network (by handling its credentials through SmartK) and that runs a user-level application, such as an Internet browser. Multiple users can, through the browser, authenticate themselves to a private web server by means of their own smart cards, leveraging on a user-level smart card facility.

The design of SmartK focuses on modularity, therefore the system administrator can plug (transparently to the applications) different modules that allow the applications to work with different cards and different readers connected to different ports. Moreover, SmartK is very tiny and does not significantly affect the performance of the kernel.

A background on smart cards is given in Section 2. The design and implementation of SmartK, for the Linux operating system, is described in Section 3. In Section 4 we show three main applications of SmartK: 1) we show how to use SmartK for performing secure local and network log-on (see Section 4.1); 2) we show an application of SmartK to the run-time integrity verification of executables based on digital signatures, that is another typical example of card-aware application run by the kernel (see Section 4.2); 3) we show an application of SmartK for handling users’ encryption keys in a secure network file system (see Section 4.3). In Section 5 we present the prototype of a new Linux OS distribution: SalSA Linux, which is built on top of SmartK and addresses the security issues raised by the automatic download of packages and operating system updates over the Internet. SalSA Linux focuses on the provider–subscriber business model which is increasingly more frequent in the ICT industry.

In Section 6 we give a brief survey of technologies related to the design of operating-system level access to smart cards and cryptographic devices. Section 7 concludes the paper.

2. Background

In this section we give a background on smart cards and the main development frameworks based on them. We also briefly discuss the use of a trusted computing architecture for securing operating system services.

2.1. Smart cards

Smart cards (Hendry, 2001; Jurgensen and Guthery, 2002) are one of the most interesting and successful technologies that...
have been proposed in the past. The research on smart cards has produced plenty of technological improvements that have had a significant impact on the everyday life. Nowadays, although the field has reached its maturity, further advances in terms of provided features, applications, protocols and development tools, still appear with a certain regularity.

In this brief survey, we focus on the developer’s point of view. 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, there is nowadays a large availability of high-level tools that meet requirements of software designers and developers.

We briefly survey in Section 2.2 some of the main smart card frameworks used to develop card-aware applications. Such tools provide libraries and middlewares that implement standard APIs and functionalities. In some cases, these tools have been so successful that they are nowadays considered (de facto) standard. However, we remark that such tools are essentially application-oriented, that is, they are easily usable by user-level applications but their use at kernel level is problematic.

2.1.1. Specifications

In this paper, by “smart card” we intend an Integrated Circuit Card (ICC), that is a card equipped with a programmable microprocessor (CPU), a ROM that contains on-card software provided by the factory (e.g., the Card Operating System), an I/O controller, a RAM for temporary data and a non-volatile data storage device (EEPROM). The card communicates by means of either electrical contacts (contact cards) or through a small antenna (contactless cards) or by means of both options (combi cards). The card reader (Interface Device - IFD) is a device that physically connects the host computer and the card. The IFD can be connected to the computer through different kind of ports (e.g., serial, USB, PCMCIA) or even embedded in it. The reader provides the power to the card and takes care of data transfer and communication synchronization.

The physical properties of the card (e.g., the size, the position of contacts), 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 companies to be jointly used. The standard ISO 7816 (see The International Organization for Standardization and The International Electrotechnical Commission, 1995) provides a definition of these specifications for a smart card. Each part of the standard concerns a specific aspect of smart cards. Part 1 defines physical characteristics, part 2 defines the dimension and location of the contacts, part 3 defines electronic signals and protocols, part 4 defines the “Interindustry Commands for Interchange”, i.e., a set of commands to provide data access facilities by means of a filesystem-like organization of the memory and some security features for filesystem objects.

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 does not immediately provide the power to the card, in order not to damage the card supplying power to a wrong component. Indeed, the reader first verifies that the position of the contacts is correct. Then the reader powers on the card and sends it the reset signal. The card sends back an important message called Answer To Reset (ATR). The ATR is a sequence of bytes (up to 33) that contains all information needed to establish the connection between card and reader, as the transfer rate, data representation and the communication protocol. The ISO 7816-3 document defines the format of ATR and the two communication protocols: $T = 0$ 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 referred to as the Response APDU.

The ISO 7816 is so widely adopted that virtually all smart cards support it. Moreover, several other standards have been proposed in order to manage some specific functionalities. An important specification for business purposes has been given by the EMV (EMVCo) standard, a joint standard introduced by Europay, Mastercard and VISA, for content, structure and programming of chip-based payment cards.

Modern Mobile Telecommunication facilities are based on the smart card technology. A special smart card, namely the Subscriber Identity Module (SIM) is a mandatory component of terminal equipments in the GSM (Rahmema, 1993) and UMTS networks (Richardson, 2000). The European Telecommunication Standard Institute (ETSI) (European Telecommunication Standard Institute), and the 3rd Generation Partnership Project (3GPP) (3rd Generation Partnership Project) issued several standard specifications, regarding the design and the behavior of such a module since early nineties (European Telecommunication Standard Institute, 1992; European Telecommunication Standard Institute, 2001; European Telecommunication Standard Institute, 2011).

Protocols such as ISO 7916 and EMV provide a powerful and complete interface to the smart card. However, such a low-level interface is not always the most productive option to develop applications that deal with structured data and stateful communication protocols (e.g., handling X509 certificates, computing/verifying digital signatures). Several specifications have been proposed in order to create an abstract layer between the low-level APDU-oriented layer and the high-level application level.

RSA Laboratories (http://www.rsa.com) presented a set of specifications, namely the Public-Key cryptography Standards (PKCS) documents that nowadays, are widely accepted as standards de facto. In particular, the PKCS# 11 (RSA Laboratories, 2004) defines the Cryptoki (Cryptographic token interface): a platform-independent API to cryptographic devices, such as smart cards and so called “Hardware Security Modules” (HSM). The PKCS# 11 API defines most commonly used cryptographic object types (e.g., RSA keys, X.509 Certificates, DES/Triple DES keys) and all functions
needed to use, create/generate, modify and delete such objects.

The PKCS# 15 (RSA Laboratories, 2000) establishes types and formats for data related to user authentication to be used by standards-aware security-oriented applications. In particular, the standard defines how data (e.g., certificates, biometric information, cryptographic keys, user IDs and credentials) are stored in security tokens, as well as how the token’s file system is arranged.

Java Cards (Chen, 2000) are smart cards equipped featuring an optimized embedded Java virtual machine, firstly introduced in 1996. On-card applications (cardlets) are uploaded on the card as bytecodes. This technology has the remarkable advantage of allowing the development of portable cardlets, so that applications providing different smart card services and features (i.e., those ones related to ATM and SIM cards) can be designed and implemented independently the actual device they will be deployed to. Java card specifications have been maintained by Sun Microsystems (later acquired by Oracle). The last release (3.0.4) dates to September, 2011 (Oracle, 1364).

2.2. Development frameworks

The “Application Independent Card Terminal Application Programming Interface for ICC applications” (CT-API) (Detusche Telekom et al., 1998), is a simple package for the development of card-aware applications. CT-API is a library that includes 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) (PC/SC workgroup, 1997; PC/SC workgroup, 1999) is a standard definition of a complete framework for smart cards deployment. PC/SC specifies the architecture and the components of a distributed “card environment”, the services provided by each component and the protocols that the components use to communicate with each other. 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 then it was also used in UNIX-like systems (Corcoran, 1999), with the support of the “Movement for the Use of Smart Cards in a Linux Environment (MUSCLE)” (Muscle, 2006).

Actually, both CT-API and PC/SC implement a raw programming interface for the interaction with smart cards. They provide some functions for session handling and some functions that take care of sending APDUs (that the developer has to build “by hand”) to the card.

The OpenSC project (ellinghaus opensc-pro) provides a library and a set of utilities for accessing ISO 7816 and PKCS# 15 compliant smart-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 the integration of ISO 7816-4 and PKCS# 15 compliant pre-formatted devices.

The Open Card Framework (OCF) (OpenCard Consortium, 1998), introduced by a consortium led by IBM and GemPlus, consisted of a rich set of high-level Java APIs and classes that wrapped every component of the ISO 7816 protocol, for the sake of portability of smart card based software. The project apparently ended in 2004.

3. Design and implementation of SmartK

In this section we present our design and implementation of a kernel module for Linux (SmartK) that outperforms previous proposals.

SmartK provides a simple framework for smart-card management at kernel level. That is, 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 obtaining a framework that serves both kernel modules and user applications.

3.1. The interface

SmartK is a framework that exposes a very simple interface that we describe below.

\texttt{smartk\_init\_card} starts the connection with 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 adopted data representation.

\texttt{smartk\_data} sends commands and receives responses. This function transparently wraps all steps needed by data transfer, according with information collected during the initialization.

\texttt{smartk\_cleanup\_card} closes the communication, cleans all memory buffers, and turns off the power to the card.

This interface implements any off-card application. A similar approach but 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, an application organizes data as specified by the protocol provided by the card (e.g., T = 0 protocol). Then the application sends data to the reader through the port. This is achieved by sending the proper signals and, if necessary, re-encoding data with the communication parameters that have been negotiated during the startup.

3.2. Object oriented style

The whole framework has been designed following an Object Oriented style (Fig. 2). 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).

3.3. Modules

SmartK is composed by the following four modules.\footnote{We now discuss the specific case of using a towitoko micro reader that uses 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.o is the core of the framework. It provides the interfaces to kernel-level applications and to other modules of SmartK.

pt_t0_smartk.o implements the API according to the T = 0 protocol.

ifd_towitoko_smartk.o is the Towitoko reader’s driver. The use of a different reader only needs another module that replaces this one.

io_serial_smartk.o is a simple interface for the communication with a serial port. The use of a different port only needs another module that replaces this one.

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. 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 to the core object. Thus, all objects can invoke each other’s methods by reaching them only through this object (i.e., methods of all modules take a pointer to core as argument and use it as the this pointer in C++). This approach maintains modules independent with each other and limits the number of symbols exported by each module.

### 3.4 Implementation details

The module smartk.o provides the methods register_protocol_smartk, register_ifd_smartk and register_io_smartk that are invoked to plug the components in the framework. These methods link the related objects to the object core. Once the smartk.o module has been loaded, it instantiates the 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 the protocol T = 0 is provided). It features a very simple interface composed by three methods: activate_card, data and deactivate_card.

An ifd_smartk object implements the functions required by the communication with the reader. Its methods allow one to enable and disable the reader and the card, to transmit/receive data and to 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 (e.g., baud rate, parity), 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. Interactions with the reader are performed by means of methods of the ifd object (through their pointers to 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. This module implements the mechanism used by the Linux kernel to manage data flow through the serial port. The module instantiates the object serial and initializes and registers it by means of register_io_smartk.

In order to set up the SmartK environment, we suggest to configure the kernel to load the module serial.o as a dynamic module. All modules should be loaded in this order: smartk.o, io_serial_smartk.o (that loads serial.o on its own), ifd_towitoko_smartk.o, and finally the module pt_t0_smartk.o.

As discussed above, all aspects of the interaction with the card are modular. For instance, 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.

### 3.5 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 a 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 service given by this module is to allow user applications the use of any reader, card and port by means of SmartK.

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 invokes 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 (deactivate_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. The read () operation destructively accesses the buffer.

### 3.6. Implementation

SmartK has been implemented as a set of kernel modules that implement each aspect of the communication between applications and cards, and that communicate by means of a general interface. The design of SmartK is intended to be modular in order to allow system administrators to choose freely among different readers and cards. SmartK supports the update of any "plug-in" (e.g., card communication protocol, reader, reader connection port) at run-time. Moreover, SmartK does not significantly affect kernel performances and does not significantly increase the size of the kernel memory image. In Fig. 3 we show the size in bytes of each module. The whole framework uses about 25 Kb (the PC/SC lite libraries spans over 800 Kb).

SmartK has been developed on a Linux operating system with kernel 2.4.20 (Bovet and Cesati second edition, 2002; Rubini and Corbet second edition, 2003). The only supported RS232 reader is the Towitoko micro and we partially implemented the card communication protocol T = 0. 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 1.1.110. Sources are available on the SmartK Home Page at the URL http://smartk.dia.unisa.it.

### 4. Deploying SmartK

Here we present three proof-of-concept SmartK applications that we developed in order to address issues within operating system security.

#### 4.1. Accessing system-level kerberized services

The setting in which Kerberos (Neuman and Ts’o, September 1994; Kohl et al., 1994) works is the following. There exists an open distributed computing environment (DCE) where the users of a workstation 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 to 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 one 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 (Gaskell and Looi, 1995; Itoi and Honeyman, 1999) where they proposed the use of smart cards for performing user authentication in Kerberos. Indeed, the use of smart cards in Kerberos solves three critical problems: the need for a secure encryption device, the need for a secure key storage and the existence of dictionary attacks on passwords.

There are several cases in which the user of a “kerberized” distributed system is the operating system as a whole. In these cases, it is the kernel that authenticates itself to the AS and issues a ticket request to the TGS. This is a scenario in which SmartK fits naturally. We addressed, within such a scenario, two study cases: authenticated Dynamic Host Configuration Protocols (DHCP) negotiations (Hornstein and Lemon, 1999) and mounting remote filesystems through secure NFS (Stern et al., 2001). We extended both client-side components of these services in order to use an authentication credential stored on a smart card through SmartK. In our early experiments, we set up a sample network in which, diskless clients could obtain an IP address and mount a remote filesystem only if the correct smart card was plugged into the reader at boot time.

#### 4.2. Run-time verification of executables

Run-time verification of the integrity of executables, is one of the approaches for preventing execution of malicious code on a networked computer. Informally, with this technique, the executables of the operating system are extended with a digital signature that the loader verifies each time the execution is invoked. The execution continues only if the verification process succeeds. In case an intrusion occurs, the malicious code possibly installed by the intruder (e.g., rootkits), will not be executed by the operating system.

We point out as implementations of this approach CryptoMark (Beattie et al., 2000), the ELF signing architecture by van Doorn et al. (2001), and the WLF Project (Catugno and Visconti, 2002; Catugno and Visconti, 2004), which we discuss later. The AEGIS (Arbaugh et al., 1997; Itoi et al., 2001) project provides an architecture for the verification of the integrity of the operating system at bootstrap and to verify the integrity of any file "on demand". Both Arbaugh’s system and WLF are designed to be built on top of it.

Run-time verification of executables constitutes a typical field of application for SmartK. Indeed, it is securely 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.

In the rest of this section, we briefly introduce WLF and describe the implementation of a smart-card based key management scheme that has been built on top of SmartK. Note that in this paper, we call WLF both the handler for signed ELF files and the whole project. In this work we do not discuss features related to any specific binary format and the WLF handler has to be intended both for signed ELF handler and for signed script handler.

4.2.1. WLF overview
The WLF project (Catuogno and Visconti, 2002) proposes a prototypical 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 verifies the corresponding files. In case the verification succeeds, then the execution is performed as usual, otherwise the execution fails.

In the Linux kernel, each executable is interpreted and executed by an appropriate handler. In a WLF system each handler includes a verify() function that accomplishes the signature verification task. Public keys are managed by a distinct module (that we call key agent), that takes care of retrieving keys from a given repository and of providing them to a WLF handler.

The signature of executable files is performed by running the following procedure:

1. The system is booted in secure mode (i.e., disconnected from the network and provided by the needed private keys).
2. The administrator installs all unsigned packages and then signs them with the signature tools (wlfsign and scriptsign).
3. The administrator removes private keys from the system and reboots it in normal mode updating the public-key repository with each new public key that is needed to verify the new signed executables.

Note that even in secure mode, the system needs to know all public keys because all its executables are signed.

4.2.2. 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 and provides them (on demand) to the WLF handlers. Currently, WLF is equipped with two key management schemes: the basic and the floppy key management scheme (respectively BKM and FKM). The BKM simply satisfies testing requirements, it loads public keys from a character device (/dev/wlf). Users push keys (contained in files) into the kernel by means of an ioctl call on the device. The FKM loads keys from a read-only floppy disk.

The SmartK Key Management scheme (SKM) is a kernel module that implements a key agent for WLF. When SKM is run, it loads in memory all public keys that are stored on 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. Therefore, secure verification of executables is performed by running all steps at kernel level.

A similar functionality can be implemented using a card-aware version of wlfsign. When new executables have to be installed in the system they can be signed by the smart card. In this case the secret key is stored in the smart-card that computes the signature, therefore the secret key is not revealed.

4.3. TCFS

The Transparent Cryptographic File System (TCFS) (Cattaneo et al., 2001) aims at providing strong security to network filesystems while remaining easy to use. A Linux kernel module named TCFS client provides all services of an NFS client (it is designed to replace NFS when filesystem security is required) and moreover, it features file and directory encryption per-file and per-user. In other words, on a TCFS filesystem, users can decide which files and/or directories have to be encrypted. TCFS does not need a devoted server, it works with a default NFS server, all cryptographic duties are performed on the client side.

Files are stored in encrypted form so that the remote server can not access to their contents, but at the same time, administration tasks like backup/restore operation and volume checks and repair, can be accomplished as usual. In TCFS each file is encrypted using its own key (file key). The file key is encrypted with the user’s key (master key) and stored together with other meta data into a file header. A mechanism to realize unauthorized changes of the file contents is also provided (for more details refer to 6). In order to access encrypted files, the user has to provide her master key to TCFS.

Key management is a critical issue in cryptographic systems. In the design of TCFS, key management has been kept separated from the actual cryptographic filesystem. TCFS simply provides a raw interface to obtain the master key from the user by means of an ad-hoc ioctl call. Hence, in TCFS, a key management scheme is implemented on top of this interface as a suite of utilities that handle authenticated sessions with the user and once obtained the master key, simple push it into the core of TCFS.

In TCFS, in order to access to her encrypted files, the user runs the tcfslogin utility, and simply types her login password. Master keys are stored in the /etc/tcfsdb database, encrypted with the user password. The tcfslogin utility simply extracts the user’s master key from the database, decrypts it with the user password and pushes it in the kernel by means of the ioctl system call. TCFS keeps in memory these keys and uses them to obtain file keys of all files the user access in her job session.

4.3.1. TCFSCards

The approach adopted by TCFS to get user’s master key is not secure, since it is based on the assumption that the TCFS client

---

3 We stress that smart cards are tamper-resistant devices, therefore should be preferred to other devices when a few critical data have to be stored.
is trusted. Actually allowing the kernel to keep in memory the user’s master key, exposes it to be discovered by raw access to the computer memory, or by adversaries that impersonate the legitimate user after she opened the session.

Here we propose a possible improvement of TCFS based on the SmartK framework that achieves a better protection of the master keys. In our renewed architecture, each user is provided of a card that we name TCFSCard and that stores the owner’s master key but never reveals it. In order to begin her TCFS session, the user has just to put the smart card inside the reader. We provide the TCFS client of a SmartK-aware module that authenticates the user when she begins her session. Any time an encrypted file is opened, it extracts the encrypted file key from the file header and sends it to TCFSCard. Eventually, TCFSCard sends back the decrypted file key.

This solution has two advantages with respect to the original mechanism. First, the master key is never exposed to the client machine. Moreover, communication with the card does not occur in user-space but is directly accomplished by the operating system out of the user responsibility, hence, a malicious client can obtain a certain number of file keys but not the master key. Second, adversaries who impersonate a legitimate user, can not obtain any file key if the user’s TCFSCard is not present in the reader.

5. SalSA: A signed Linux OS distribution

The distribution of software over the Internet generates several security issues. Downloading software (including operating system upgrades, patches and other critical packages) exposes the system to some serious threats. Indeed, the download repository could have been tampered and the provided packages could have been replaced by malicious code. Moreover the legitimate server could have been replaced by an adversary machine that is impersonating it, by “spoofing” its IP address or exploiting any other security weaknesses of network segments along the path to the destination system. Moreover, even after the download, the package could be corrupted by an adversary agent (e.g., rootkit, computer virus) that has infected the client.

In this section we present the design of SalSA: a signed Linux distribution (i.e., a distribution of the Linux operating system where all executables are digitally signed and whose integrity is verified at run time). We outline a scenario where the software vendors can distribute their products over the network and the customers can securely download and deploy them.

SalSA is a Linux OS distribution in which, in short:

- all interpreters, libraries and executables are natively signed;
- only signed code can be run by the OS; the kernel verifies the code-attached signature at run-time, and allows the execution of the executable only if the verification succeeded;
- a package distribution service allows the system administrator (subscriber) to securely download and install signed updates and software packages from a pre-configured set of trusted providers;
- the system administrator is enabled to locally self-sign software she builds from third-party sources.

The SalSA Linux Distribution is based on a vanilla Linux Kernel which natively features WLF and SmartK modules. A set of specific WLFCards is provided in order to accomplish the administrative tasks within a signed operating system. Here follows a brief overview of the main OS operations in a SalSA Linux-box.

System installation. SalSA has been developed starting from the Slax (http://www.slax.org) live CD, a bootable CD ROM able to run the Linux operating system without affecting whatever is present on the computer’s hard disk. Moreover, Slax offers a host of applications that help to select and configure a customized new distribution to be installed either on a Live CD or on the hard disk.

SalSA itself is built as a Slax live CD with signed executables and scripts and a new installation management tool, SalSit, which helps to carry out the installation along with the configuration of all aspects concerning the WLF key and card management. When running in Secure mode, SalSit also allows to deal with signature management (i.e., replacing/ updating the signature of certain executables, as well as signing applications newly compiled from sources).

SalSA packages are signed archives that are handled as in Slackware or BSD. We stress that pre-installation and post-installation scripts are (obviously) signed as well as the usual integrity checksum is replaced by a cryptographic signature.

Booting SalSA. During the boot process and before accessing files, the kernel loads the SmartK modules and gets the public keys from the smart cards. Next the kernel verifies the signature of the first script to load (i.e., init) and executes it only if the verification was successful. The kernel continues the boot of the system following the above strategy.

As already mentioned for WLF, an important issue raised by this architecture consists in ensuring that the kernel (and the SmartK modules) have not been updated. The early solution we considered consisted in placing the kernel and the initrd images on a tamper resistant device such as a live CD or a non volatile storage device. Recently we are considering to leverage on the coreboot project (Minnich, 2012), which makes possible to boot an optimized and static Linux Kernel image stored along with the motherboard’s BIOS.

Software installation. The SalSA package distribution service is designed to operate within a third-party PKI where both providers and subscribers have their own set of certificates and public/secret key pairs. Although such a service can be implemented according to plenty of different business models, here, for the sake of simplicity, we assume that:

- A provider $F_i$ is simply the owner of a remote server who compiles and signs (with her secret key) the package she distributes over the Internet through a secure channel (e.g., protected by the TLS/SSL protocol suite). We denote the secret and the private keys of $F_i$ respectively by $S(F_i)$ and $P(F_i)$.
- A subscriber $C_i$ is the system administrator of a SalSA Linux-box.

Suppose the provider $F_i$ distributes a package $Pkg_i$ for the SalSA distribution. Executables in $Pkg_i$ have been signed by $F_i$.
with its private key \( S(F_i) \). \( F_i \) distributes the package along with her digital certificate and her public \( P(F_i) \). The subscriber \( C_j \) downloads the package, installs it and adds \( P(F_i) \) to its trusted public keys. To do this, \( C_j \) saves the new package and the attached key in a temporary storage area, then reboots the system in secure mode and, by means of the SalSit utility, adds \( P(F_i) \) to the current WLFCard and installs the package. Eventually, \( C_j \) reboots the system in normal mode.

More frequently, the subscriber’s platform downloads several packages (e.g., system or application updates) from the same provider. In this case, once the provider’s public key is added to the key repository, the installation procedure can be fully automated, as in the main operating systems (e.g., Microsoft Windows, Ubuntu-Linux).

We stress that SalSA is suitable to be installed on user workstations or stand-alone servers that provide some consolidated services and software (e.g., web servers) and that simply need to be kept up-to-date, without significant changes in the configuration, during their life cycle.

In some cases, downloaded software and libraries are not already pre-compiled binaries. For several important reasons the system administrator has to build executables by re-compiling the sources. This may happen, for example, when a security patch should be applied to a vulnerable application, as well as when a new tool has been developed “in house”. In such cases, the administrator herself is enabled to self-sign the newly created code with her own secret key.

For example, suppose that subscriber \( C_j \) wishes to install some scripts he wrote, to accomplish certain tasks. \( C_j \) boots the system in secure mode (i.e., disconnected from the network) and provides the needed private keys. Hence, the administrator installs all unsigned packages and signs them with the WLFCard signature tools (wlsign and scriptsign) and updates (if needed) the public key repository. Eventually, \( C_j \) reboots the system in normal mode.

SalSA uses two kinds of WLFCards: the user card and the manager card for the installation task. The user card is plugged in the system when it runs in normal mode and contains the public keys of all accepted providers \( F_1, \ldots, F_i \) (i.e., the providers whose software can be executed on the system), plus the public key \( P(C_j) \), used to verify the executable locally signed by \( C_j \). The manager card is plugged in the system when it runs in secure mode. It provides the same public keys of the corresponding user card plus the private key \( S(C_j) \) that the local system administrator uses to sign any natively unsigned software she installs.

6. Related work

In this section we briefly describe some architectures which embeds smart card/security devices management in the operating system.

6.1. The Windows smart card subsystem

The Microsoft Windows smart card subsystem architecture \( \text{(Nirmalananthan, 2002)} \) consists of: reader driver library, device drivers, resource manager, and service providers. A device driver for a specific reader maps the reader’s functions to the native services provided by the Windows OS and the smart card infrastructure.

The resource manager is responsible for managing and controlling application accesses to the smart card. It identifies and tracks resources, it controls the allocation of readers and resources across multiple applications and it supports transaction primitives for accessing available services on a specific card.

All cryptographic operations are performed by independent modules known as cryptographic service providers (CSPs) that provide different cryptographic primitives implementation to the overlying applications through a generalized interface. An OS instance can host different CSP modules which wrap different types of smart cards rather than any different crypto library.

There are two device-independent APIs, namely: CryptoAPI and Microsoft Win32 SCard APIs. CryptoAPI contains functions that allow applications to encrypt or digitally sign data in a flexible manner, while providing protection for the user’s sensitive data and private key. The Microsoft Win32 SCard APIs are low-level APIs for accessing smart cards with the most flexibility for the application to control readers, cards and other components.

We stress that the Windows Smart Card Resource Manager supports only PC/SC compatible readers. A non-PC/SC reader can be used with Microsoft Windows but this would require additional ingredients for accessing the reader, and cards in the reader. The issues raised by the integration of unsupported readers (in particular a wireless reader) into a Windows Mobile powered hand-held device are addressed in Cattaneo et al. (2007).

6.2. Trusted computing

Trusted Computing (TC) aims at providing a framework to establish trust among the components of a heterogeneous computing environment. It is based on parameters and attributes of these elements and best practices associated with their production and deployment. The most important industrial initiative for the realization of the TC functionality has been the Trusted Computing Group (TCG) (Trusted Computing Group Consortium), an alliance of a large number of IT enterprises that published a set of specifications for extending conventional computer architectures with a variety of security-related features and cryptographic mechanisms (Trusted Computing Group, 2004).

The core component specified by the TCG is the Trusted Platform Module (TPM) (Trusted Computing Group, 2009), a chip added to the computer’s motherboard which constitutes the basis for other security and trust functionalities. The TPM provides a set of basic cryptographic facilities such as secure random number generators, secure non-volatile storage (that can not be manipulated by the platform that embeds the TPM), cryptographic functions, key generation algorithms, and hash functions.

Moreover, a Trusted Platform features the capability of verifying the integrity of a platform component and to prove to a challenger the integrity of the platform through an attestation (a signed result of a measurement.)
TCG specified a platform-independent software interface for accessing TPM functions, called Trusted Software Stack (TSS) (Trusted Software Stack Specifications, 2009). The TSS is compatible with existing cryptographic APIs (e.g., MS-CAPI or PKCS# 11) to allow current and future applications without explicit TCG support to use the cryptographic functions provided by the TPM. However, in order to take full advantage of the TPM functionalities, applications and operating systems must support the TSS directly.

The trusted computing architectures can be used with different performances 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 current TSS specification is highly complex, which makes its usage very difficult and error-prone for application developers, so that, the cost of such technologies and the trust and ethical issues (Anderson, 2003) that they generate slowdown their effective use.

6.3. Recent applications

Research on smart card applications has probably passed its peak though new applications and refinements are proposed on a regular basis. We found that this remarkable production still motivates our work and validates our approach.

Sorber et al. (2011) described a system that enables a smartphone operating system to assign computations on sensible data to a miniSD smart card.

Smartphone operating systems natively feature a framework which handles the SIM cards in order to provide the required security services to the hosted functionalities (e.g., accessing the phone network) and applications (e.g., home banking). Nevertheless, these services are usually not accessible to third party application developers due to restrictive manufacturers’ disclosure policies. This issue has relevant impacts on many forthcoming business models in the “mobile arena” and was addressed in Vasudevan et al. (2011), where authors argue that enabling Smartphone operating systems to disclose SIM-based security features to all smartphone applications does not affect the overall security properties.

The L4Andoid system (Lange et al., 2011), an operating system based on the microkernel technology, runs the phone’s native OS in a separated virtual machine side-by-side with third party trusted applications. The microkernel provides a security interface to the main SIM card (if allowed) or to other present cards either physically plugged into additional slots/readers or software-emulated.

Nepal et al. (2011), a USB Trusted Personal Device, powered by the Trusted Computing’s TPM is presented. Such a device independently runs a minimal operating system and several web based applications accessible by the host computer through a virtual network link. The on-board operating system performs the main TPM functionalities (including attestation) independently of the host platform. On board applications are developed using mainstream technologies such as Java, Tomcat and so on.

Cattaneo et al. (2010), a proxy signature and encryption system is proposed by using smart cards on a remotely accessible server.

7. Conclusion

In this paper, we presented SmartK: a kernel-level framework for development of smart card applications for the Linux operating system. We argued that 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.

In order to validate our approach, we proposed to apply SmartK to three OS-level security services: 1) in-kernel security services authentication through Kerberos; 2) handling the key repository for a run-time code-signing infrastructure (WLF); 3) providing a secure keystore for a distributed cryptographic file system. We stress that all such applications are kernel-level oriented and thus they represent the natural “end-users” for SmartK.

In addition, we presented the prototype of a new Linux OS distribution: SalSA Linux, which addresses the problem of securely deploying software over the Internet. SalSA Linux is built on top of SmartK and WLF. We stress that SalSA Linux is suitable for the security of critical network appliances and for embedded Linux-boxes as it provides a secure mechanism to manage system updates.

A very preliminary part of this work appeared in Catuogno et al. (2005).

REFERENCES

Catuogno L, Gassirià R, Masullo M, Visconti I. Securing operating system services based on smart cards. In: Trust,


Detusch Telekom, TUV Informationstechnik GmbH, GMD-Forschungszentrum Informationstechnik GmbH, TELETRUST Deutschland e.V. Application independent card terminal application programming interface for ICC applications; 1998.


European Telecommunication Standard Institute. Universal Mobile Telecommunications System (UMTS); LTE; Characteristics of the Universal Subscriber Identity Module (USIM) application, document no. TS 31.102; 2011.


Luigi Catuogno is a network and system administrator at the Department of Computer science of the University of Salerno, where he got his Laurea degree (M.Sc. equivalent) in 1998, his PhD in Computer Science in 2004 and a post-doc fellowship in 2009. He has been visiting student at New York University in 2002 and research assistant at Ruhr University of Bochum (Germany) in 2009. His research interests focus on operating systems security, trusted computing and privacy enhancing technologies.

Roberto Gassirà got his Laurea Degree in Computer Science in 2004 from the University of Salerno. He has been Security Analyst at Telecom Italia Mobile from 2004 to 2006. He is currently
“security researcher” at the Mobile Security Lab of Retis Consulting. His interests mainly focus on vulnerability assessment, penetration testing and reverse engineering techniques.

**Michele Masullo** got his Laurea in Computer Science from University of Salerno in 2004. He currently works at CRIAI: a research center for ICT and Industry Automation. His main interests focus in network security and management.

**Ivan Visconti** is an assistant professor at the Dipartimento di Informatica ed Applicazioni of the University of Salerno. He got his Laurea (M.Sc. equivalent) cum laude in Computer Science in 1998 and his PhD in Computer Science in 2003 from the University of Salerno. He has been a Post-Doctoral fellow in Paris at the Département d’Informatique of the Ecole Normale Supérieure. Recently he visited several research groups as Microsoft Research Redmond, Ruhr University of Bochum and University of California at Los Angeles. His research topics includes foundations of cryptography and network and system security. He taught several classes on operating systems and on network security for the bachelor and master in computer science at the University of Salerno.