SEIP: Simple and Efficient Integrity Protection for Open Mobile Platforms

Xinwen Zhang[1], Jean-Pierre Seifert[2], and Onur Acıiçmez[1]

[1] Samsung Information Systems America, San Jose, CA, USA
{xinwen.z,o.aciicmez}@samsung.com
[2] Deutsche Telekom Laboratories and Technical University of Berlin
jean-pierre.seifert@telekom.de

Abstract. SEIP is a simple and efficient but yet effective solution for the integrity protection of real-world cellular phone platforms, which is motivated by the disadvantages of applying traditional integrity models on these performance and user experience constrained devices. The major security objective of SEIP is to protect trusted services and resources (e.g., those belonging to cellular service providers and device manufacturers) from third party code. We propose a set of simple integrity protection rules based upon open mobile operating system environments and respective application behaviors. Our design leverages the unique features of mobile devices, such as service convergence and limited permissions of user installed applications, and easily identifies the borderline between trusted and untrusted domains on mobile platform. Our approach thus significantly simplifies policy specifications while still achieves a high assurance of platform integrity. SEIP is deployed within a commercially available Linux-based smartphone and demonstrates that it can effectively prevent certain malware. The security policy of our implementation is less than 20kB, and a performance study shows that it is lightweight.

1 Introduction

With the increasing computing capability and network connectivity of mobile devices such as cellular phones and smartphones, more applications and services are deployed on these platforms. Thus, their computing environments become more general-purpose and open than ever before. The security issue on these environments has gained considerable attention nowadays. According to McAfee's 2008 Mobile Security Report [7], nearly 14% of global mobile users have been directly infected or have known someone who was infected by a mobile virus. More than 86% of consumers worry about receiving inappropriate or unsolicited content, fraudulent bill increases, or information loss and theft. The number of infected mobile devices increases remarkably according to McAfee's 2009 report [8].

Existing research on mobile device security mainly focuses on porting PC counterpart technologies to mobile devices, such as signature- and anomaly-based analysis [17,18,23,26,30,33]. However, there are several reasons that make

M. Soriano, S. Qing, and J. López (Eds.): ICICS 2010, LNCS 6476, pp. 107–125, 2010.

these infeasible. First of all, mobile devices such as cellular phones are still limited in computing power. This mandates that any security solution must be very efficient and leave only a tiny footprint in the limited memory. Second, in order to save battery energy, an always concurrently running PC-like anti-virus solution is, of course, not acceptable. Third, security functionality should require minimum or zero interactions from a mobile user, e.g., the end user shouldn't be required to configure individual security policies. This demands then that the solution must be simple but general enough so that most users can rely on default configurations — even after new application installations.

On the other side, existing exploits in mobile phones have shown that user downloaded and installed applications are major threats to mobile services. According to F-secure [24], by the end of 2007, more than 370 different malware have been detected on various cell phones, including viruses, Trojans, and spyware. Most existing infections are due to user downloaded applications, such as Dampig[1], Fontal, Locknut, and Skulls. Other major infection mechanisms include Bluetooth and MMS (Multimedia Message Service), such as Cabir, CommWarrior, and Mabir. Many exploits compromise the integrity of a mobile platform by maliciously modifying data or code on the device (cf. Section 2.1 for integrity compromising behaviors on mobile devices). PandaLab reports the same trends [11] in 2008. Based on the observation that user downloaded applications are the major security threats, one objective of securing mobile terminal should be confining the influence of user installed applications. This objective requires restricting the permissions of applications to access sensitive resources and functions of mobile customers, device manufacturers, and remote service providers, thus maintaining high integrity of a mobile device, which usually indicates its expected behavior. Considering the increasing attacks through Bluetooth and MMS interfaces, an effective integrity protection should confine the interactions between any code received from these communication interfaces and system parts.

Towards *simple, efficient, and yet effective solution*, we propose SEIP, a mandatory access control (MAC) based integrity protection mechanism of mobile phone terminals. Our mechanism is based upon information flow control between *trusted* (e.g., customer, device manufacturer, and service providers) and *untrusted* (e.g., user downloaded or received through Bluetooth and MMS) domains. By confining untrusted applications' write operations to trusted domains, our solution effectively maintain runtime integrity status of a device. To achieve the design objectives of simplicity and efficiency, several challenges exist. First, we need to efficiently identifies the interfaces between trusted and untrusted domains and thus simplifies the required policy specification. For example, in many SELinux systems for desktop and servers, very fine-grained policy rules are defined to confine process permissions based on a least-privilege principle. However, there is no clear integrity model behind them, and it is difficult to have the assurance or to verify if a system is running in a good integrity state. Secondly, many trusted processes on a mobile

[1] Description of all un-referred viruses and malware in this paper can be found at `http://www.f-secure.com/en_EMEA/security/security-threats/virus/`

device provides functions to both trusted and untrusted applications, mainly the framework services such as telephony server, message service, inter-process communications, and application configuration service. Therefore simply denying the communications between trusted and untrusted process decreases the openness of mobile devices – mobile users enjoy downloading and trying applications from different resources. We address these challenges by efficiently determining boundaries between trusted and untrusted domains from unique filesystem layout on mobile devices, and by classifying trusted subjects (processes) according to their variant interaction behaviors with other processes. We propose a set of integrity protection rules to control the inter-process communications between different types of subjects.

We have implemented and deployed SEIP on a real-world Linux-based mobile phone device, and we leverage SELinux to define a respective policy. Our policy size is less than 20kB in binary form and requires less than 10 domains and types. We demonstrate that our implementation can prevent major types of attacks through mobile malware. Our performance study shows that the incurred overhead is significantly smaller compared to PC counterpart technology.

Outline. In the next section we discuss threats to mobile platform integrity and the overview of SEIP. Details of our design and integrity rules are described in Section 3, and implementation and evaluation in Section 4. We present related work on integrity model and mobile platform security in Section 5, and conclude this paper in Section 6.

2 Overview

2.1 Integrity Threat Model

We focus our study on the integrity protection of mobile platforms. Particularly for this purpose, we study the adversary model of mobile malware from two aspects: infection mechanisms and compromising mechanisms from application level.

Infection Mechanisms. With the constraints of computing capability, network bandwidth, and I/O, the major usage of mobile devices is for consuming data and services from remote service providers, instead of providing data and services to others. Based on this feature, the system integrity objective for mobile devices is different from that in desktop and server environments. For typical server platforms, one of the major security objectives is to protect network-faced applications and services such as httpd, smtpd, ftpd, and samba, which accept unverified inputs from others [27]. For mobile phone platforms, on the other side, the major security objective is to protect system integrity threaten by user installed applications. According to mobile security reports from F-Secure [24] and PandaLab [11], most existing malware on cell phones are unintentionally downloaded and installed by user, and so far there are no worms that do not need user interaction for spreading.

Although most phones do not have Internet-faced services, many phones have local and low bandwidth communication services such as file-sharing via Bluetooth. Also, the usage of multimedia messaging service (MMS) has been increasing. Many viruses and Trojans have been found in Symbian phones which spread through Bluetooth and/or MMS such as Cabir, CommWarrior, and Mabir. Therefore, any code or data received via these services should be regarded as untrusted, unless a user explicitly prompt to trust it. The same consideration applies for any received data via web browsers.

Integrity Compromising Mechanisms. Many mobile malware compromise a platform's integrity by disabling legal phone or platform functions. For example, once installed, mobile viruses like Dampig, Fontal, Locknut, and Skulls maliciously modify system files and configurations thus disable application manager and other legal applications. Doomboot installs corrupted system binaries into c: drive of a Symbian phone, and when the phone boots these corrupted binaries are loaded instead of the correct ones, and the phone crashes at boot. Similarly Skulls can break phone services like messaging and camera.

As a mobile phone contains lots of sensitive data of its user and network service provider, they can be targets of attacks. For example, Cardblock sets a random password to a phone memory card thus makes it no longer accessible. It also deletes system directories and destroys information about installed applications, MMS and SMS messages, phone numbers stored on the phone, and other critical system data. Other malware such as Pbstealer and Flexispy do not compromise the integrity of a platform, but stealthily copy user contact information and call/message history and send to external hosts.

Monetary loss is an increasing threat on mobile phones. Many viruses and Trojans trick a device to make expensive calls or send messages. For example, Redbrowser infects mobile phones running Java (J2ME) and sends SMSs to a fixed premium rate number at a rate of $5 - $6 per message, which is charged to the user's account. Mquito, which is distributed with a cracked version of game Mosquitos in pirate channels, sends an SMS message to a premium rate number before the game starts normally.

2.2 SEIP Overview

To effectively achieve integrity protection goals while respecting the constraints of mobile computing environments, we propose the following tactics for our design.

Simplified boundary of trusted and untrusted domains. Instead of considering fine-grained least privileges for individual applications, we focus on integrity of trusted domains. With this principle, we identify domain boundary along with relatively simpler filesystem layout in many Linux-based mobile phones than in PC and server environments. Specifically, based on our investigation, most phone-related services from device manufacture and network provider are deployed on dedicated filesystems, while user downloaded application can only be installed on another dedicated filesystem or directory, or flash memory card. Thus, for example, one policy can specify that by default all applications

belonging to the manufacturer or service provider are trusted for integrity purpose, while user installed applications are untrusted. An untrusted application can be upgraded to trusted one only through extra authentication mechanisms or explicit authorization from user.

MAC-based integrity model. Many mobile platforms use digital signature to verify whether a downloaded application can have certain permissions, such as Symbian [22], Qtopia [13], MOTOMAGX [9], and J2ME [3]. All the permissions specified by a signed application profile are high level APIs, e.g., making phone call or sending messages. However, first of all, these approaches cannot check the parameters of allowed API calls, thus a malicious application still may get sensitive access or functions with allowed APIs, such as call or send SMS messages to premium rate numbers. Secondly, API invocation control cannot restrict the behaviors of the target application when it makes low level system calls, e.g., invoking system process or changing code and data of trusted programs. This is especially true for platforms that allow installing native applications such as LiMo [4], Maemo [6], GPE [2], Qtopia [12], and JNI-enabled Android. Thirdly and most importantly, most of these "ad-hoc" approaches do not apply any kind of foundational security model. Logically, it is nearly impossible to specify and verify policies for fundamental security properties such as system integrity. In our design, we use MAC-based security model, thus different processes from the same user can be assigned with different permissions. Further, our approach enables deeper security checks than simply allowing/denying API calls. Finally, instead of considering extremely fine-grained permission control thus requiring a complete and formal verified policy, we focus on read- and write-like permissions that affect system integrity status, thus makes integrity verification feasible.

Trusted subjects handling both trusted and untrusted data. Traditional integrity models either prohibit information flow from low integrity sources to high integrity processes (e.g., BIBA [16]), or degrade the integrity level of high integrity processes once receiving low data (e.g., LOMAC [21]). However, both approaches are not flexible enough for the security and performance requirements of mobile platforms. Specifically, due to function convergence, one important feature of mobile phone devices is that resources are maintained by individual framework services and running as daemons which accepting requests from both trusted and untrusted processes. For example, in LiMo platform [4], a message framework controls *all* message channels between the platform and base stations, and implements all message-related functions. Any program that needs to receive/send SMS or MMS messages has to call the interfaces provided by this framework, instead of directly interacting with the wireless modem driver. Other typical frameworks include telephony service serving voice conversation and SIM card accesses, and network manager serving network access such as GPRS, WiFi, and Bluetooth. All these frameworks are implemented as individual daemons with shared libraries, and expose their functions through public interfaces (e.g., telephony APIs). Similar mechanisms are used for platform management functions such as application management (application installation and

launch), configuration management, and data storage. Many frameworks need to accept inputs from both trusted and untrusted applications during runtime — to provide, for e.g., telephony or message services. However, due to performance reasons they cannot frequently change their security levels. Thus, traditional integrity models such as BIBA and LOMAC are not flexible enough to support such security requirements. Similarly, "domain transitions" used in SELinux are also infeasible for mobile platforms.

Towards this issue, we propose some particular trusted processes can accept untrusted information while maintaining their integrity level. The critical requirement here is that accepted untrusted information does not affect the behavior of such trusted process and others. We achieve this goal via separating information received from subjects of different integrity levels. Note that an important feature that distinguishes our approach from traditional information flow–based integrity models is that, we do not sanitize low integrity information and increase its integrity level, as Clark-Wilson–like [19,25,32] models do. Instead, we separate the data from different integrity levels within a trusted subject thus receiving untrusted data does not affect its behavior.

3 Design of SEIP

This section presents design details and integrity rules of SEIP for mobile platforms based on discussed security threats and our strategies. Although we describe within the context of Linux-based mobile systems, our approach can be applied to other phone systems such as Symbian, as they have similar internal software architecture. One assumption is that we do not consider attacks in kernel and hardware, such as installing kernel rootkits or re-flashing unauthentic kernel and filesystem images to devices. That is, our goal is to prevent software-based attacks from application level.

3.1 Trusted and Untrusted Domains

To preserve the integrity of a mobile device, we need to identify the integrity level of applications and resources. In mobile platforms, typically trusted applications such as those from device manufacture and wireless network provider are more carefully designed and tested as they provide system and network services to other applications. Thus, in our design we regard them as high integrity applications or subjects. Note that completely verifying the trustworthiness of a high integrity subject, e.g., via static code analysis, is out of the scope of SEIP. As aforementioned, our major objective is to prevent platform integrity compromising from user installed applications. Therefore by default all user installed applications later on the platform are regarded as low integrity. In some cases, a user installed application should be regarded as high integrity, e.g., if it is provided by the network carrier or trusted service provider and requires sensitive operations such as accessing SIM or user data, e.g., for mobile bank and payment applications. For applications belonging to 3rd party service providers,

it is in high or low level integrity based the trust agreement between the service provider and user or manufacturer/network provider. For example, an anti-virus agent on a smartphone from a trusted service provider needs to access many files and data of the user and network provider and should be protected from modification of low integrity software, therefore it is regarded as high integrity. Other high integrity applications can be trusted platform management agents such as device lock, certificate management, and embedded firewall.

Usually, extra authentication mechanism is usually desired when a user installed application is considered as high integrity or trusted, such as application source authentication via digital signature verification, or explicitly authorized via UI actions from the user.

Based on the integrity objective of SEIP–to protect system and service components from user installed applications, we specify the boundary in the filesystem of mobile devices. We lay out all Linux system binaries, shared libraries, privileged scripts, and non-mutable configuration files into dedicated file system or system partition. Similar layout can be used all phone related application binaries, configurations, and framework libraries. All user applications can only be installed in a writable filesystem or a directory and the `/mnt/mmc`, which is mounted when a flash memory card is inserted. However, many phone related files are mutable, including logs, tmp files, database files, application configuration files, and user-customizable configuration files, thus have to be located in writable filesystems. Policies should be carefully designed to distinguish the writing scope of a user application. We have observed similar approaches have been used on Motorola EZX series [10] and Android [1].

We note that filesystem layout is determined by the manufacturer of a device and such a separation can be enforced quite easily. This approach simplifies policy definitions and reduces runtime overhead compared to traditional approaches such as SELinux on PCs, which sets the trust boundaries based on individual files. Note that we assume there is secure re-flashing of the firmware of a device. If arbitrary re-flashing is enabled, an attacker can install untrusted code into trusted side of the filesystem offline, and re-flash the filesystem image to the device. When the devices boots, the untrusted code is loaded into trusted domain during runtime and can ruin the system integrity.

3.2 Subjects and Objects

Like traditional security models, our design distinguishes subjects and objects in OS. Basically, subjects are *active entities* that can access to objects, which are *passive entities* in a system such as files and sockets. Subjects are mainly active processes and daemons, and objects include all possible entities that can be accessed by processes, such as files, directories, filesystems, network objects, program and data files. Note that a subject can also be an object as it can be access by another process, e.g., being launched or killed. In an OS environment, there are many different types of access operations. For example, SELinux predefines a set of object classes and their operations. For integrity purposes, we focus on three access operations: create, read, and write. From information flow

perspective, all access operations between two existing entities can be mapped to read-like and write-like operations [14].

3.3 Trusted Subjects

We distinguish three types of trusted subjects[2] on mobile platforms, according to their functionalities and behaviors. Different integrity rules (cf. Section 3.4) are applied to them for integrity protection purpose.

Type I trusted subjects. This type includes high integrity system processes and services such as `init` and `busybox`, which are basically the trusted computing base (TCB) of the system. Another set of Type I trusted subjects includes some service daemons which do not need input from untrusted subjects, such as device status manager (cf. Section 4.3). Only high integrity subjects can have information flow to those subjects, while untrusted subjects can only read from them. Type I trusted subjects also include pre-installed applications from device manufacture or service provider, such as dialer, calendar, clock, calculator, contact manager, etc. As they usually only interact with other high integrity subjects and objects, their integrity level is constant.

Type II trusted subjects. These are applications provided by trusted resources, but usually read low integrity data only, such as browser, MMS agent, and media player. They are usually pre-deployed in many smartphones by default, which can be considered as trusted subjects in our strategy. However, they mostly read untrusted Internet content or play downloaded media files in flash memory card. These subjects usually do not communicate with other high integrity subjects in most current smartphone systems, and they do not write to objects which should be read by other trusted subjects. Therefore, in our design we downgrade their integrity level during runtime without affecting their functions and system performance[3].

Type III trusted subjects. These are mainly service daemons such as telephony, message, network manager, inter-process communication (IPC), application and platform configuration services. These subjects need to communicate with both low and high integrity subjects, and read and write both low and high integrity objects. For integrity purpose, we need to prevent any information flowing from low integrity entities to high integrity entities by using these daemons' functions. Careful investigation to the internal architecture of a daemon is needed to isolate information from trusted and untrusted entities. In general we adopt the following design principle towards this problem.

[2] The concept of trusted subjects in SEIP is different from that in traditional trusted operating system and database [29]. Traditionally, a trusted subject is allowed to bypass MAC and access multiple security levels. Here we use it to distinguish applications from trusted resources and user downloaded.

[3] Modern browsers such as Chromium has multi-process architecture, where browser kernel process can be regarded as trusted and renderer processes are not trusted. However we observed that this architecture has not been widely adopted on mass-produced mobile devices.

As a typical service framework provides functions to other applications via APIs defined in library files, we identify all create- and write-like APIs for each framework, and track their implementing methods in the daemon. For a creating method, we insert a hook to label the new object with the integrity level of the requesting subject. For a write-like method, we insert a hook to check the label of target object and the requesting subject, and deny or allow the access according to pre-defined policies according to our integrity rules. Section 4 illustrates the details of applying this principle to some major service frameworks on our evaluation platform.

3.4 Integrity Rules

For integrity protection, it is critical to control how information can flow between high and low integrity entities. Without restriction, a process can accept low integrity data and write to high integrity data or send to high integrity processes. We propose a set of information flow control rules for different types of subjects. Our rules focus on create, read, and write operations[4].

Rule 1. *$create(s, o) \leftarrow L(o) = L(s)$, where $L(x)$ is the integrity level of subject or object x: when an object is created by a process, it inherits the integrity level of the process.*

Rule 2. *$create(s_1, s_2, o) \leftarrow L(o) = MIN((L(s_1), L(s_2))$: when an object is created by a trusted process s_1 with input/request from another process s_2, the object inherits the integrity level of the lower bound of s_1 and s_2.*

These two rules are *exclusively* applied upon a single object creation. Typically, Rule 1 applies to objects that are *privately* created by a process. For example, an application's logs, intermediate and output files are private data of this process. This rule is particularly applied to Type I trusted subjects and all untrusted subjects. Rule 2 applies to objects that are created by a process upon the request of another process. In one case, s_1 is a server running as a daemon process, the s_2 can be any process that leverages the function of the daemon process to create objects, e.g., to create a GPRS session, or access SIM data. In another case, s_1 is a common tool or facility program that can be used by s_2 to create object. In these cases, the integrity level of the created object is corresponding to the lower of s_1 and s_2. This rule is applied to Type III trusted subjects as aforementioned.

Rule 3. *$can\_read(s, o) \leftarrow L(s) \leq L(o)$: a low integrity process can read from both low and high integrity entities, but a high integrity process can only read from entity of the same level.*

Rule 4. *$can\_write(s, o) \leftarrow L(s) \geq L(o)$: a high integrity process can write to both low and high integrity entities, but a low integrity process can only write to entity of the same level.*

[4] We consider destroying/deleting an object is the same operation as writing an object.

As Figure 1(a) shows, these two rules indicate that there is no restriction on information flow within trusted entities, and within untrusted entities, respectively. However, these also imply that information flow is only allowed from high integrity entities to low integrity entities, which are, fundamentally, BIBA-like integrity policies. Note that a reading or writing operation can happen between a subject and an object, or between two subjects via IPC mechanisms. These two rules are applied to direct communication between Type I trusted subjects and all untrusted entities.

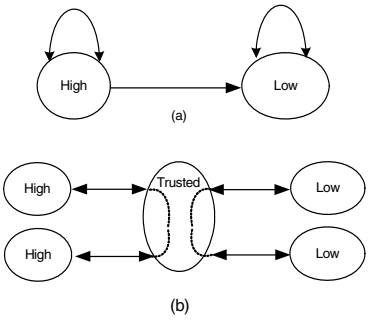

Fig. 1. Information flows are allowed between high and low integrity entities, directly or indirectly via trusted subjects

Rule 5. *can_read*$(s, o_1) \leftarrow L(s) \geq L(o_1) \wedge write(s, o_2) \wedge L(o_1) \geq L(o_2)$*: a high integrity process* s *can receive information from low integrity entity* o_1*, provided that the information is separated from that of other high integrity entities, and it flows to low integrity entity* o_2 *by the high integrity process* s*.*

As Figure 1(b) shows, this rule allows a trusted subject to behave as a communication or service channel between untrusted entities. This rule is particularly for the Type III trusted subjects, which can read/receive inputs from low integrity entities while maintaining its integrity level, under the condition that the low integrity data or requests are separated from high integrity data and handled over to a low integrity entity by the trusted subject.

Rule 5 requires that any input from untrusted entities does not affect the runtime behavior of the high integrity subject. Therefore not every subject can be trusted for this purpose. Typically, communications between applications and service daemons can be modelled with this rule. For example, on a mobile phone device, a telephony daemon can create a voice conversation between an application and wireless modem, upon the calling of telephony APIs. A low integrity process cannot modify any information of the connection created by a high integrity process, thus preventing stealthily forwarding the conversation to a malicious host, or making the conversation into a conference call. For another example, data synchronizer from device manufacture or network provider can be trusted to read both high and low integrity data without mixing them.

Rule 6. *change_level* : $L'(s) = L(o) \leftarrow read(s, o) \wedge L(s) > L(o)$*: when a trusted subject reads low integrity object, its integrity level is changed to that of the object.*

This rule is dedicated for the Type II trusted subjects, which usually read untrusted data (e.g., Internet content or media files). As these subjects do not communicate with other trusted subjects, downgrading their integrity level does

not affect their functions and system performance in our design. This is the only rule that changes a subject's integrity level during runtime.

3.5 Dealing with IPC

According to Rule 1, most IPC objects inherit the integrity level of their creating processes, including domain sockets, pipes, fifo, message queues, shared memory, and shared files. Therefore, when a low integrity process creates an IPC object and write to it, a high integrity process cannot read from it, according to our integrity rules. In many mobile Linux platforms such as LiMo, OpenMoko, GPE, Maemo, and Qtopia, D-Bus is the major IPC, which is a message-based communication mechanism between processes. A process builds a connection with a system- or user-wide D-Bus daemon (`dbusd`). When the process wants to communicate to anther process, it sends messages to `dbusd` via its connection. The `dbusd` maintains all connections of many processes, and routes messages between them. A D-Bus message is an object in our design, which inherits integrity level from its creating process. According to Rule 5, Type III trusted subject `dbusd` (specified by policy) can receive any D-Bus message (low or high integrity level) and forward to corresponding destination process. Typically, a trusted process can only receive high integrity messages from `dbusd`. Also, according to Rule 5, if a process is a Type III trusted daemon, like telephony or message server daemon, it can receive high and low integrity messages from `dbusd`, and handle them separately within the daemon. Next section illustrates the implementation details of secure D-Bus and other phone service daemons based on our design.

3.6 Program Installation and Launching

An application to be installed is packaged according to particular format (e.g., the .SIS file for Symbian and .ipk for many Linux-based phone systems), and application installer reads the program package and meta-data and copies the program files into different locations in local filesystem. As the application installer is a Type III trusted subject specified by policy, it can read both high and low integrity application packages. Also, according to our integrity Rule 2 and 5, it writes (when installing) to trusted part of the filesystem when reads high integrity software package, and writes to untrusted part of the filesystem when reads low integrity package.

Similar to installation, during the runtime of a mobile system, a process is invoked by a trusted program called program launcher, which is also a Type III trusted subject according to policy. Both high and low integrity processes can be invoked by the program launcher. All processes invoked from trusted program files are in high integrity level, and all processes invoked from untrusted program files are in low integrity level. Compare to traditional POSIX-like approaches, where a process's security context and privileges typically are determined by a calling process, in our design, a process's integrity level is determined by the

integrity level of its program files including code and data[5]. On one aspect, this enhances the security as a malicious application cannot be launched to a privileged process, which is a major vulnerability in traditional OS; on the other aspect, this simplifies policy specification in a real system, which can be seen in next section.

3.7 Dealing with Bluetooth/MMS/Browser and Their Received Code/Data

As aforementioned in Section 2.1, SEIP regards MMS agents and mobile browsers as Type II trusted subjects via security policy. According to integrity Rule 6, their integrity level is changed to low whenever they receive data from outside, e.g., reading message or browsing web content. Any code or data received from Bluetooth, MMS, and browser is untrusted by default as during runtime these subjects only can write untrusted system resources such as filesystems. Thus, any process directly invoked from arbitrary code by MMS agent or browser is in low integrity level, according to our integrity Rule 1. Further, any code saved by these subjects is in low integrity level and it cannot be launched to high integrity processes, as the program launcher is Type III trusted subject following integrity Rule 5. Thus it cannot write to trusted resources and services, such as corrupting system binaries or changing platform configurations. More fine-grained policy rules can be defined to restrict phone related functions that an untrusted process can have, such as accessing phone address book, sending messages, and building Bluetooth connections, which prevent further distribution of potentially malicious code from this untrusted subject.

It is possible that some software and data received from Bluetooth, MMS, and browser are trusted. For instance, a user can download a trusted bank application from his PC via Bluetooth or from a trusted financial service provider's website via browser. For another example, many users use Bluetooth to sync calender and contact list between mobile devices and PC. SEIP does not prevent these types of applications. Usually, with extra authorization mechanism such as prompting via user actions, even one application or some data is originally regarded as untrusted, it can be installed or stored to the trusted side, e.g., by the application manager or similar Type III trusted subjects on the phone. Similar mechanism can be used for syncing user data or installing user certificate via browser.

4 Implementation and Evaluation

We have implemented SEIP on a real LiMo platform [4]. Our implementation is built on SELinux, which provides comprehensive security checks via Linux Security Module (LSM) in kernel. Also SELinux provides domain-type and role-based policy specifications, which can be used to define policy rules to implement high

[5] Superficially this feature is similar to the setuid in Unix/Linux. However setuid is mainly for privilege elevation for low privileged processes to complete sensitive tasks, while here we confine a process's integrity aligning with its program's integrity level.

level security models. However, existing deployments of SELinux on desktop and servers have very complex security policies and usually involves heavy administrative task; furthermore, current SELinux does not have an integrity model built-in. On one side, our implementation simplifies SELinux policy for mobile phone devices based on SEIP. On the other side, our implementation augments SELinux policy with built-in integrity consideration.

4.1 Trusted and Untrusted Domains

Figure 2 shows a high-level view of the filesystem and memory space layout in our evaluation platform. All Linux system binaries (e.g., `init`, `busybox`), shared libraries (`/lib`, `/usr/lib`), scripts (e.g., `inetd`, `network`, `portmap`), and non-mutable configuration files (`fstab.conf`, `inetd.conf`, `inittab.conf`, `mdev.conf`) are located in a read-only cramfs filesystem. Also, all phone related application binaries, configurations, and framework libraries are located in another cramfs filesystem. All mutable phone related files are located in an ext3 filesystem, including logs, tmp files, database files, application configuration files, and user-customizable configuration files (e.g., application settings and GUI themes). By default, all codes and data downloaded by user, e.g., via USB, Bluetooth, MMS, or browser, are stored and installed under `/app/usr` of the ext3 filesystem and in `/mnt/mmc` (which is mounted when a flash memory card is inserted), unless explicitly prompted by the user to install to the trusted ext3 filesystem.

As Figure 2 shows, all read-only filesystems and part of ext3 filesystem where phone related files are located are regarded as trusted, and user writable filesystems are regarded as untrusted. By default, processes launched from trusted filesystems are trusted subjects, and processes launched from untrusted filesystems are untrusted subjects. Note that our approach does not prevent trusted user application from being installed on the device. For example, a trusted mobile banking application can be installed in the trusted read-write filesystem, and the process launched from it is labelled as trusted. We also label any process invoked by message agent (e.g., MMS or email agent) or browser as untrusted. According to SEIP, trusted subjects can read and write to trusted filesystem objects, and untrusted subjects can read all filesystem objects, but can only write to untrusted objects. Figure 2

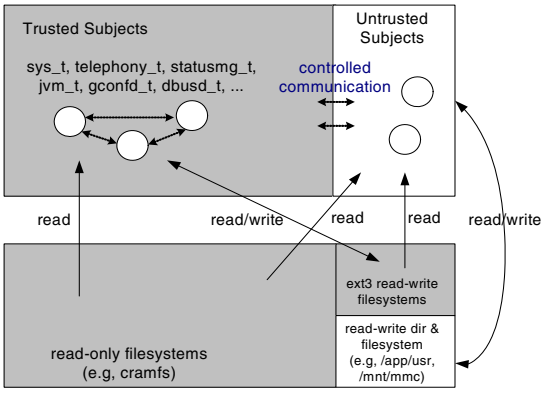

Fig. 2. Trusted and untrusted domains and allowed information flow between them on our evaluation platform

also shows the information flow between subjects and filesystem objects. The access controls are enforced via SELinux kernel level security server.

As created by kernel, virtual filesystems like `/sys`, `/proc`, `/dev` and `/selinux` are trusted. Similar to the ext3 filesystem, `/tmp` and `/var` include both trusted and untrusted file and directory objects, depending on which processes create them.

4.2 Securing IPC via D-Bus

D-Bus is the major IPC mechanism for most Linux-based mobile platforms. The current open source D-Bus implementation has built-in SELinux support. Specifically, a `dbusd` can control if a process can acquire a well-known bus name, and if a message can be sent from one process to another, by checking their security labels. These partially satisfy our integrity requirement: a policy can specify that a process can only send message to another process with the same integrity level. However, as in mobile devices, including our evaluation platform, both trusted and untrusted applications need to communicate with framework daemons via `dbusd`, e.g., to make phone calls or access SIM data, or set up network connections with connectivity service. Therefore, existing D-Bus security mechanism cannot satisfy this requirement.

Following our integrity rules, we extend D-Bus built-in security in two aspects. Firstly, each message is augmented with a header field to specify its integrity level based on the process which sends the message, and the value of this field is set by `dbusd` when it receives the message and before dispatches it. As `dbusd` listens to a socket on connection requests from other processes, it can get the genuine process information from the kernel. Note that as `dbusd` is a Type III trusted process, both high and low integrity processes can send messages to it. Secondly, according to our integrity rules, if a destination bus name is a Type I trusted subject, it can only accept high integrity messages; otherwise, it can accept both high and low integrity messages.

With these, each message is labeled with a integrity level, and security policies can be defined to control communication between processes. When a message is received by a trusted daemon process, its security label is further used by the security mechanism inside the daemon to control which object that the original sending process can access via the method call in the message. We explain this in next subsections.

Our implementation introduces less than 200 line of code based on the D-Bus framework of LiMo platform, mainly for adding a security context field of D-Bus message header, setting this filed by `dbusd` message dispatcher, and security check before dispatching based on the integrity level of a message and its destination process. Specifically, as each client process (the sender) connects `dbusd` with a dedicated socket, the dispatcher calls `dbus_connection_get_unix_process_id()` to obtain the pid of the sender, then the security context of the sender with `getpidcon()`, and then sets the value into the header field. This field is used later when the message arrives at a destination trusted subject, i.e., to make access control decision of whether the original sender of the message can invoke a particular function or access an object.

4.3 Securing Phone Services

The telephony server provides services to typical phone-related functions such as voice call, data network (GSM or UMTS), SIM access, message services (SMS and MMS), and GPS. General applications calls telephony APIs (TAPI) to access services provided by the telephony server, which in turn connects to the wireless modem of the device to build communication channels. In LiMo platforms, message framework and data network framework are dedicated for short message and data network access services. An application first talks to these framework severs which in turn talk to the telephony server. Security controls for those services can be implemented in their daemons.

Different levels of protection can be distinguished in voice call. For example, one policy can only allow trusted applications can make phone calls, while untrusted application cannot make any phone call, which is the case in many feature phones in market nowadays. For another example policy, untrusted applications can make usual phone calls but not those of premium services such as payment-per-minute 900 numbers. Different labels can be defined for telephone numbers or their patterns. In our implementation, we allow untrusted applications to call 800 toll-free numbers only. Similar design is used in message framework. Figure 3 shows the workflow for a typical voice call. A client application calls `tapi_call_setup()` to initialize a phone call with $TelCallSetupParams\perp$, which includes the target phone number and type (voice call, data call, or emergency call), and a callback function to handle possible results. The telephony server provides intermediate notifications including modem and connection status to the client. Once the call is established with the modem, the telephony server sends the connected indication to the TAPI library which in turn notifies the application via the registered callback function about the status of call (connected or disconnected), and then the application handles the processing. We insert a security hook in the telephony server daemon which checks the integrity level (security context) of calling client from D-Bus message and decides to allow or deny the call request. For simplicity the `dbusd` is ignored in Figure 3.

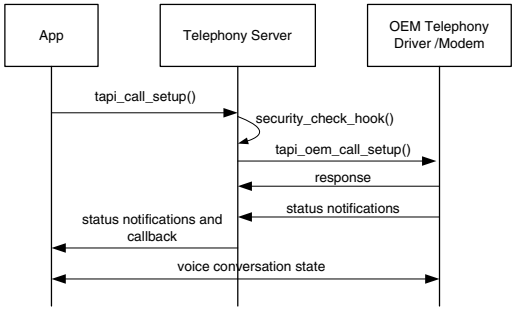

Fig. 3. Secure telephony server. A security hook in telephony daemon checks if a voice call can be build for the client.

With similar design principle, we have implemented SEIP in other system frameworks in our evaluation platform, including the device status management framework, which maintains all system status such as phone status, device status, memory status for out-of-memory, audio path, and volume status, and provides

get/set APIs for accessing them, and the GConf service, which stores configuration data for other applications. These mechanisms can prevent malicious modification of device status and configurations from untrusted applications. Due to space limit we omit the implementation details in this paper.

4.4 Performance Evaluation

Our policy size is less than 20KB including `genfscon` rules for filesystem labelling. Comparing to that in typical desktop Linux distributions such as Fedora Core 6 (which has 1.2MB policy file), our policy footprint is tiny. As aforementioned, the small footprint of our security mechanism is result from the simple way to identify the borderline between trusted and untrusted domains, and the efficient way to control their communications.

We study the performance of our SELinux-based implementation with microbenchmark to investigate the overhead for various low-level system operations such as process, file, and socket accesses. Our benchmark tests are performed with the LMbench 3 suites [5]. Our results show that for most operations, our security enforcement has less than 4% overhead, which is significantly less than the counterpart technology on PC [28].

5 Related Work

Information flow-based integrity models have been proposed and implemented in many different systems, including the well-known Biba [16], Clark-Wilson [19], and LOMAC [21]. Biba integrity property restricts that a high integrity process cannot read lower integrity data, execute lower integrity programs, or obtain lower-integrity data in any other manner. In practices, there are many cases that a high integrity process needs to read low integrity data or receive messages from low level integrity processes. LOMAC supports high integrity process's reading low integrity data, while downgrading the process's integrity level to the lowest integrity level it has ever read. PRIMA [25,31] and UMIP [27] dynamically downgrades a process's integrity level when it reads untrusted data. As a program may need to read and write to high integrity data or communicate to high integrity subjects after it reads low integrity data, it needs to be re-launched by a privileged subject or user to switch to high level. Although these approaches can achieve a platform's integrity status, they are not efficient for always-running service daemons on mobile devices.

Clark-Wilson [19] provides a different view of integrity dependencies, which states that through certain programs so-called transaction procedures (TP), information can flow from low integrity objects to high integrity objects. CW-lite [25,32] leverages the concept of TP where low integrity data can flow to high integrity processes via filters such as firewall, authentication processes, or program interfaces. Different with the concept of filter, UMIP [27] uses exceptions to state the situations that require low integrity data to flow to high integrity processes. A significant difference between these and our solution is that, we do

not sanitize low integrity information and increase its integrity level. Instead, our design allows a trusted process to accept low integrity data if it is separated from high integrity data thus does not affect the behavior of the process. This fits the requirements of framework services which provide functions to both high and low integrity processes on open mobile platforms.

UMIP [27] leverages discretionary access control (DAC) information of a Linux system to configure integrity policies, i.e., to determine high integrity files and programs. On many phone devices, there is single user [15], so DAC is not so helpful in this situation. Based on unique phone usage behaviors, our design considers user downloaded and received codes untrusted, which captures the existing major security threats of mobile phones.

Mulliner et al. [30] develop a labelling mechanism to distinguish data received from different network interfaces of a mobile device. However, there is no integrity model behind this mechanism, and this approach does not protect applications accessing data from multiple interfaces. Also, the monitoring and enforcing points are not complete, which only include hooks in execve(2), socket(2) and open(2) system calls, and cannot capture program launching via IPC such as D-Bus.

Android classifies application permissions into four protection levels, namely Normal, Dangerous, Signature, and SignatureOrSystem [1,20]. The first two can be available for general applications, while the last two are only available applications belonging to those signed by the same application provider. Most permissions that can change a system's configurations belongs to the last one, and usually only allows Google or trusted party to have. This is similar to SEIP: sensitive permissions that alter platform integrity are only available to trusted domain. However, SEIP does not have a complete solution to distinguish permission sets for general third-party applications as Android does, since SIP focuses on platform integrity protection only.

6 Conclusion

In this paper we present a simple but yet effective and efficient security solution for integrity protection on mobile phone devices. Our design captures the major threats from user downloaded or unintentionally installed applications, including codes and data received from Bluetooth, MMS and browser. We propose a set of integrity rules to control information flows according to different types of subjects in typical mobile systems. Based on easy ways to distinguish trusted and untrusted data and codes, our solution enables very simple security policy development. We have implemented our design on a LiMo platform and demonstrated its effectiveness by preventing a set of attacks. The performance study shows that our solution is efficient by comparing to the counterpart technology on desktop environments. We plan to port our implementation to other Linux-based platforms and develop an intuitive tool for policy development.

References

1. Android, http://code.google.com/android/
2. Gpe phone edition, http://gpephone.linuxtogo.org/
3. J2ME CLDC specifications, version 1.0a, http://jcp.org/aboutjava/communityprocess/final/jsr030/index.html
4. Limo foundation, https://www.limofoundation.org
5. LMbench-tools for performance analysis, http://www.bitmover.com/lmbench
6. Maemo, http://www.maemo.org
7. Mcafee mobile security report (2008), http://www.mcafee.com/us/research/mobile_security_report_2008.html
8. Mcafee mobile security report (2009), http://www.mcafee.com/us/local_content/reports/mobile_security_report_2009.pdf
9. Motomagx security, http://ecosystem.motorola.com/get-inspired/whitepapers/security-whitepaper.pdf
10. OpenEZX, http://wiki.openezx.org/main_page
11. Pandalab report, http://pandalabs.pandasecurity.com/blogs/images/pandalabs/2008/04/01/quarterly_report_pandalabs_q1_2008.pdf
12. Qtopia phone edition, http://doc.trolltech.com
13. Security in qtopia phones, http://www.linuxjournal.com/article/9896
14. Setools–policy analysis tools for selinux, http://oss.tresys.com/projects/setools
15. Understanding the windows mobile security model, http://technet.microsoft.com/en-us/library/cc512651.aspx
16. Biba, K.J.: Integrity consideration for secure computer system. Technical report, Mitre Corp. Report TR-3153, Bedford, Mass. (1977)
17. Bose, A., Shin, K.: Proactive security for mobile messaging networks. In: Proc. of ACM Workshop on Wireless Security (2006)
18. Cheng, J., Wong, S., Yang, H., Lu, S.: Smartsiren: Virus detection and alert for smartphones. In: Proc. of ACM Conference on Mobile Systems, Applications (2007)
19. Clark, D.D., Wilson, D.R.: A comparison of commercial and military computer security policies. In: Proceedings of the IEEE Symposium on Security and Privacy (1987)
20. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Security & Privacy 7(1) (2009)
21. Fraser, T.: LOMAC: MAC you can live with. In: Proc. of USENIX Annual Technical Conference (2001)
22. Heath, C.: Symbian os platform security. Symbian press (2006)
23. Hu, G., Venugopal, D.: A malware signature extraction and detection method applied to mobile networks. In: Proc. of 26th IEEE International Performance, Computing, and Communications Conference (2007)
24. Hypponen, M.: State of cell phone malware in 2007 (2007), http://www.usenix.org/events/sec07/tech/hypponen.pdf
25. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-reduced integrity measurement architecture. In: Proc. of ACM SACMAT (2006)
26. Kim, H., Smith, J., Shin, K.G.: Detecting energy-greedy anomalies and mobile malware variants. In: Proc. of the International Conference on Mobile Systems, Applications, and Services (2008)

27. Li, N., Mao, Z., Chen, H.: Usable mandatory integrity protections for operating systems. In: Proc. of IEEE Symposium on Security and Privacy (2007)
28. Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the linux operating system. In: Proceedings of USENIX Annual Technical Conference, June 25-30, pp. 29–42 (2001)
29. Lunt, T., Denning, D., Schell, R., Heckman, M., Shockley, M.: The seaview security model. IEEE Transactions on Software Engineering 16(6) (1990)
30. Mulliner, C., Vigna, G., Dagon, D., Lee, W.: Using labeling to prevent cross-service attacks against smart phones. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 91–108. Springer, Heidelberg (2006)
31. Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B.M., Jaeger, T.: Measuring integrity on mobile phone systems. In: Proc. of ACM SACMAT (2008)
32. Shankar, U., Jaeger, T., Sailer, R.: Toward automated information-flow integrity verification for security-critical applications. In: Proc. of NDSS (2006)
33. Venugopal, D., Hu, G., Roman, N.: Intelligent virus detection on mobile devices. In: Proc. of International Conference on Privacy, Security and Trust (2006)