International Journal of Security and Its Applications
Vol. 9, No. 5 (2015), pp. 113-122
http://dx.doi.org/10.14257/ijsia.2015.9.5.12

Implementation of Privacy-Enhanced SMS Provider on the Android Platform

Min-woo Park[1], Jung ho Eom[2*] and Tai-Myoung Chung[1]

[1]*Department of Electrical and Computer Engineering, Sungkyunkwan University, Chunchun-dong 300,Jangan-gu, Suwon, Kyunggi-do, Republic of Korea*
[2]*Military Studies, Daejeon University, 62 Daehakro, Dong-Gu, Daejeon,*
mwpark@imtl.skku.ac.kr, eomhun@gmail.com, tmchung@skku.edu

Abstract

The Android platform stores basic telephony data such as contacts, call logs, schedules, and SMS messages. These basic telephony data are managed by ContentProvider, which is one of the core components of Android applications along with Activities, Services, *and BroadcastReceivers. If an Android application requires basic telephony data, it requests queries such as query, insert, update, and delete operations to ContentProvider. In the Android platform, every operation for which there is a possibility of misuse is protected by permissions. Generally, every application with proper permissions can request a protected operation from the Android platform. Database operations which access a database through ContentProvider are also protected by READ and WRITE permissions. However, this security policy has a critical flaw: it is impossible to differentiate the permissions of individual contacts in the Android Platform. If one application has READ permission for contacts, it can read every contact stored on an Android device. When the entities are not equal value, this flaw becomes a critical flaw. In the particular case of SMS, the problem is more serious because SMS messages can include financial information, authentication tokens, or privacy information. To address this security problem, we have designed and implemented a privacy-enhanced SMS provider. In this paper, we show how to hide sensitive SMS data from untrusted applications.*

Keywords: *Android Security, Android Access Control, Content Provider Security*

1. Introduction

Smartphones are currently among the most influential computing devices that have changed our lifestyle. According to Lenovo [1], during the past few years, the smartphone market has shown significant growth. In 2013, smartphone shipments reached 1.03 billion units and accounted for 56% of all mobile phone shipments. The popularity of smartphones has a great impact on other industries. Linkage of early smartphones with other devices consisted merely of transferring the display output. However, recent devices that can be linked with smartphones have become complex and diverse. Smartphones can control drones such as the Quadcopter and communicate with the Internet of Things. Furthermore, services to target smartphones are increasing. Many companies have created mobile Web pages and distributed mobile applications. Our lives have become more convenient through these converged services. In the near future, we might be able to check electricity consumption or the amount of food in the refrigerator by using smartphones.

However, the need for stronger security is greater than anything else among these converged services owing to two reasons. First, it is possible for security

* He is corresponding author of this paper.

ISSN: 1738-9984 IJSIA

vulnerabilities of one device to threaten the entire service. For example, if a smartphone has vulnerability at the operating system (OS) level, an adversary can take control of a drone connected to the smartphone by exploiting the OS vulnerability. To protect entire services, each device must be managed safely; the security of the master device, which is the commander, is most important. Second, when a security incident occurs, it is difficult to directly investigate the cause. If malicious behavior is detected, it is difficult to locate the base of the problem.

Android is the most popular platform for smartphones and embedded devices [2]. Android is a completely open smartphone OS. All source code from the kernel code to the framework code is freely available. Therefore, it is easier than proprietary platforms to understand and modify. Moreover, the Android OS approves more powerful permissions to applications than other platforms when these applications have been given certain rights by users. As a result, the Android platform is popularly used in various devices—smart TVs, tablets, mini set-tops, etc. However, the fact that its users are so numerous makes Android a target for adversaries. In fact, the mobile platform of malicious code that targets Android is overwhelmingly high. According to a Kaspersky Laboratory report, 1,363,549 unique Android attacks have been found; this number is four times higher than that in the previous year [3].

The Android platform has a permission policy to protect itself from malicious applications. Generally, Android applications operate in a separate environment, and it is impossible to access the resources of other applications. If an application does not have proper permissions, it cannot call protected APIs to request geographic information, access received SMS messages, connect to the Internet, etc. The Android platform defines 152 permissions at API level 22 to protect individual harmful APIs. An API is harmful if it could be misused by an adversary; we call these *protected APIs* in this paper. To use protected APIs, an Android application must be authorized by the user upon installation. When an application calls protected APIs, *PackageManager* verifies the correctness of the permissions. According to the security policy, only the legitimate application that has received the user's permission can be running.

However, the permission policy of the Android platform has a critical flaw. The permission policy is coarse-grained because it was made by abstraction to the behavior of applications [4, 5]. Thus, it is impossible to set precise restrictions. For example, access to files stored on the external SD card is governed by only two permissions—*READ_EXTERNAL_STORAGE* and *WRITE_EXTERNAL_STORAGE*. Thus, an application that is authorized for *READ_EXTERNAL_STORAGE* can access all files stored on the external SD card. If the user stores sensitive information on the external SD card, any installed application with *READ_EXTERNAL_STORAGE* permission can leak sensitive information. However, owners generally do not know about this flaw and do not consider this threat when installing an application. Considering that most mobile malware is of the Trojan variety, this security flaw is a major problem. In the Kaspersky report, Trojans represent 53% of all threats [3].

In particular, this flaw in the permission policy is abused by SMS. Ten out of the twenty Android malware applications are SMS Trojans [3]. Generally, owners think that SMS is used to send simple messages that are not sensitive, so *SMS_RECEIVED* permission can be easily given to SMS Trojans. For example, most users agree to requests from an SMS Trojan that is disguised as a useful application for *SMS_RECEIVED* permission to verify its authentication code by SMS. This is very dangerous. Currently, nearly everyone has a mobile phone, whether it is a smartphone or not. To avoid going through the Internet, authentication codes are generally sent using SMS, which is transmitted via the mobile network. These authentication services may be performed under the

assumption that SMS messages cannot be seen by others. However, in the Android platform, SMS Trojans can easily trick the owner and steal SMS messages.

In this paper, we propose a privacy-enhanced SMS provider (PESP) for the Android platform. We modified the Android framework that plays the role of reference monitor to protect sensitive SMS message from SMS Trojans. The owner can choose whether to adopt the sensitivity policy or not. When the owner wants to use PESP, all received SMS data are checked and filtered. We will design, implement, and demonstrate our model on the Nexus 5. In Section 2, we describe the Android security mechanism, and Section 3 presents PESP. Section 4 demonstrates our model. Finally, we conclude the paper in Section 5.

2. Security Model and Limitations of the Android Platform

2.1. Security Model of the Android Platform

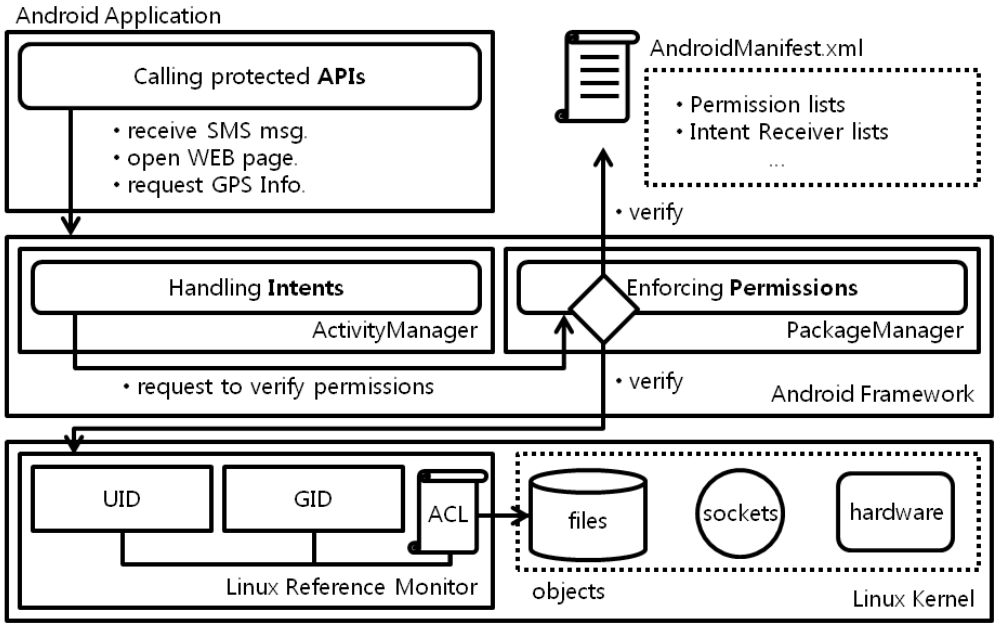

Figure 1. Security Model of the Android Platform

The architecture of the Android platform is layered—1) Linux kernel, 2) Android framework, and 3) application layer. Firstly, the Android platform is constructed based on the Linux kernel, which is responsible for direct interaction with the hardware. The Linux kernel performs the low-level OS functions such as file system handling, memory management, and process management. The Android framework comprises various managers to run applications and is located above the Linux kernel. *ActivityManager* and *PackageManager* are the most important components in the Android framework. *ActivityManager* handles inter-process communications by using intent, which is a specific signal component in the Android platform, and *PackageManager* handles install or uninstall processes for applications. Additionally, *PackageManager* plays the role of *ReferenceMonitor* in Android's permission model. When an Android application calls protected APIs using intent, *ActivityManager* requests permission from *PackageManager*. The application layer is located on top of the Android platform.

The security model of the Android platform comprises the permission model of the Android framework and the access control of the Linux kernel. Figure 1 shows the security model of the Android platform. Each application runs in an isolated

environment. To provide the isolated running environment, the Android platform uses the access control of the Linux kernel [6, 7]. Each application is assigned a unique user ID (UID) by *PackageManager*. Thus, the individual resources of each application are protected by the access control of the Linux kernel. When an application attempts to access another application's resources such as individual files, the Linux kernel blocks this behavior because access permission is violated. The Android platform uses group IDs (GID) for public system resources such as Socket, Bluetooth, and external SD cards. Upon installation, if a user agrees to authorize an application to access the external SD card, *PackageManager* adds the UID of the application to the GID, which is *sdcard_r* or *sdcard_all*, of the external SD card. Thus, if the application requests to read files on the external SD card, the Linux kernel approves this behavior when the application is a member of group *sdcard_r* or *sdcard_all*.

In other cases, API permissions and IPC permissions are controlled by the permission model in the Android framework. The process of being authorized with permissions is performed only during installation. If an application requires permissions of protected APIs, the developer must specify a list of permissions in its *AndroidManifest.xml* file, which contains metadata about the Android application. *PackageManager* reads *AndroidManifest.xml* and shows the list of permissions to the user during installation. If the installation is allowed, *PackageManager* completes the installation process. At runtime, when the application calls protected APIs or IPCs, *ActivityManager* requests permission enforcement from *PackageManager* using methods such as *enforceCallingPermission*(). *PackageManager* then compares the list of permissions in *AndroidManifest.xml* to the requested intent from *ActivityManager*.

2.2. Limitations of the Security Model

In theory, the Android platform is always safe because all application behavior is performed according to the user's consent. However, this is not true. The permission model in the Android framework has a critical flaw. The permission policy was made by abstraction to the behavior of applications. It is easy for the user to grant more privileges than expected to an application, because permissions are too comprehensive and ambiguous [9, 10]. In general, users do not correctly comprehend the overall permission policy. Furthermore, if the user understands that the application's permission request is too comprehensive and excessive, the only options are to install or not install the application. Thus, the permission model of the Android platform is vulnerable to malicious code such as Trojans. According to statistics, 1,363,549 unique attacks have been found by Kaspersky, and Trojans account for 53% of all malicious code [3].

In particular, the limitations of the security model are most critical in regard to SMS messages. The Android platform provides the *ContentProvider* component for handling database-type data, which has a massive volume with a uniform structure and is shared with other applications. The Android framework has several native *ContentProviders* for handling public telephony data such as contacts, call logs, schedules, and SMS messages. Table 1 shows a list of native *ContentProviders*.

Table 1. List of Native *ContentProviders* and their Source Codes

Name	Database	Source Code (*.java*)
CalendarProvider	*calendar.db*	*CalendarDatabaseHelper*
ContactsProvider	*contacts2.db*	*ContactsDatabaseHelper*
MediaProvider	*internal.db*	*MediaProvider*
	external.db	

TelephonyProvider	*mmssms.db*	*MmsSmsDatabaseHelper*
	telephony.db	*TelephonyProvider*

In this paper, we focus on the problem of coarse-grained permissions for the SMS native *ContentProvider* because it is the most serious [8]. Ten out of the twenty malware applications are SMS Trojans [3]. The cause of this concentration is that SMS messages can contain sensitive information. Sensitive online services—such as banking, online commercial, and government services—are used in combination with additional authenticating means other than IDs and passwords for more precise authentication. Moreover, SMS messages are used for such two-channel authentication because SMS messages are transmitted through mobile networks that are isolated from the public Internet. These authentication services may be performed under the assumption that SMS message are safe. Unfortunately, SMS messages are not safe owing to the coarse-grained permissions of the Android platform. According to our investigation of the 2,000 applications registered on GooglePlay in 2015, a total of 221 applications request *RECEIVE_SMS* permission. Furthermore, 10% of the investigated applications request the authority to act as an SMS Trojan.

3. Privacy-enhanced SMS Provider

3.1. *SMS_RECEIVED* Action in the Android Platform

Mobile traffic is received via the mobile network interface, which is managed by the Linux kernel. If a new SMS message is received, the Linux kernel delivers it to the radio interface layer (RIL) daemon in the Android framework. The RIL daemon handles all incoming traffic from mobile network interfaces. When the RIL daemon receives a new SMS message, it calls the *processUnsolicited*() method. *processUnsolicited*() parses the incoming traffic to fit the *SmsMessage* class, which is an internal data object to handle SMS messages, and *processUnsolicited*() calls *SMSDispatcher*, which handles *SmsMessage* according to the message type. If *SMSDispatcher* receives a new SMS message, it delivers *SmsMessage* to *InboundSmsHandler*. Finally, *InboundSmsHandler* requests the broadcasted intent, *SMS_DELIVER*, which is used to notify the default SMS application of the incoming SMS message. The broadcasted intent is performed by *ActivityManager*. When *ActivityManager* receives the broadcasted request, *ActivityManager* collects the information about *BroadcastReceiver* from *PackageManager* and broadcasts the *SMS_DELIVER* intent. Only registered applications can receive broadcast intent. Figure 2 shows the *SMS_RECEIVED* action in the Android platform.

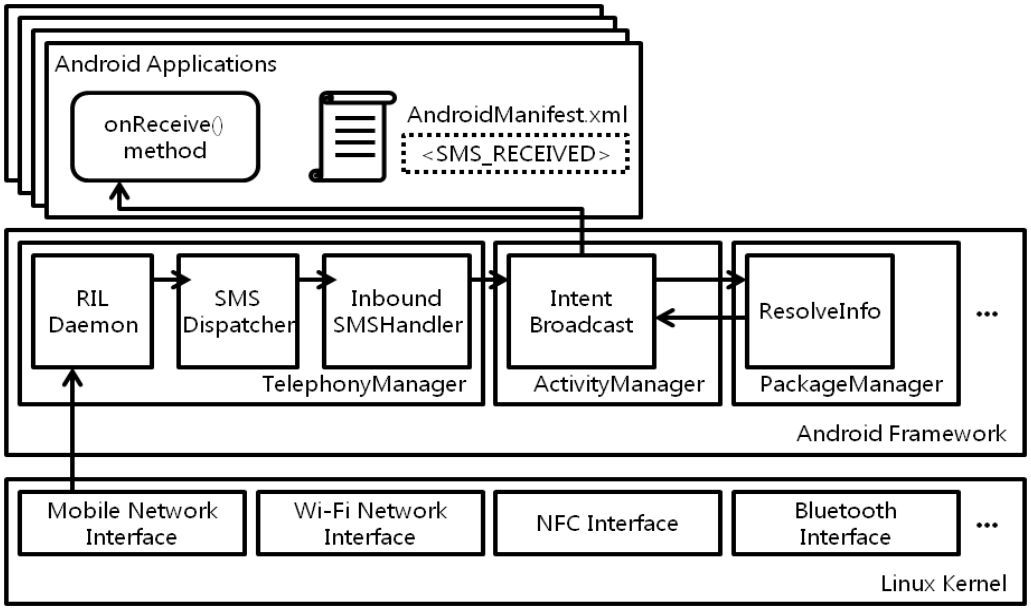

Figure 2. SMS_RECEIVED Action in the Android Platform

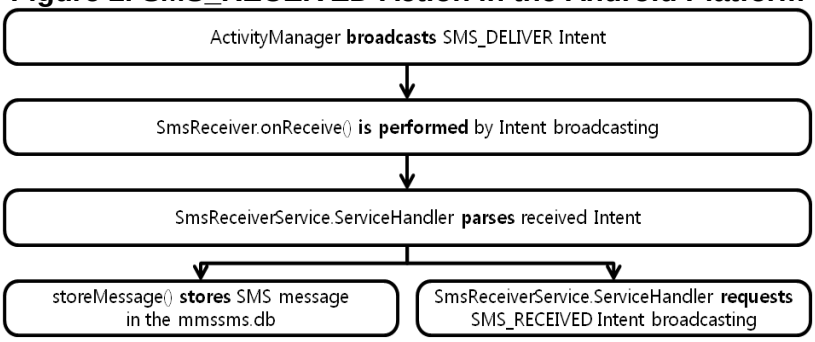

Figure 3. Handling Process of basic SMS application (*Mms.apk*)

When *ActivityManager* broadcasts *SMS_DELIVER* intent, only the default SMS application can receive this intent. By default, the basic SMS application is the default SMS application. Figure 3 presents the handling process of the basic SMS application. The *onReceive*() method of *SmsReceive* is performed by the *SMS_DELIVER* intent. When a new message is incoming, the basic SMS application requests the *SMS_RECEIVED* broadcasted intent to *ActivityManager* and stores the SMS message in *mmssms.db* by using the SMS native *ContentProvider*.

3.2. Function of Privacy-enhanced SMS Provider

The purpose of PESP is to protect sensitive SMS messages from SMS Trojan applications. The core functions of PESP are follows:

- First, PESP blocks sending *SMS_RECEIVED* intent that has sensitive information except from the application specifically added by the user.
- Second, PESP hides received sensitive SMS message from third-party applications.

To implement PESP, we modified the Android framework *ActivityManager*. We designed an additional reference monitor to watch incoming sensitive SMS messages in *ActivityManager*. In this paper, we define sensitive SMS messages to be those that include an authentication token in the body and are transmitted from well-known source addresses of mobile certification agencies.

3.3. Design and Implementation

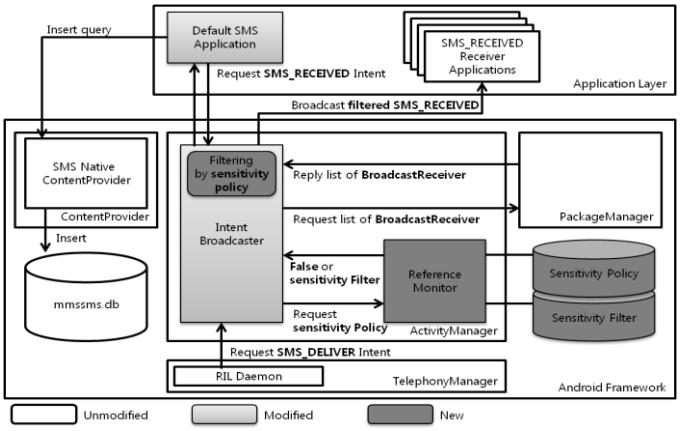

Figure 4. Architecture of PESP

Figure 4 shows the architecture of PESP. To implement PESP, we modified *IntentBroadcaster* in the *broadcastIntentLocked* class of *ActivityManager*. In addition, we added *ReferenceMonitor* and sensitivity policies. We also modified the basic SMS application to hide sensitive SMS messages from third-party applications.

At booting time, *ReferenceMonitor* reads *SensitivityPolicy* and *SensitivityFilter*. Table 2 presents examples of *SensitivityPolicy* and *SensitivityFilter*. *SenstivityPolicy* comprises three tags—intent, check-list, and package-list. The intent tag has two attribute names for the monitored intent and type—black-list or white-list. The name attribute of the check-list tag has only two values in PESP. Only when the check-list is listed is its corresponding filter applied. When the type attribute of the intent tag is *BLACK*, the name attribute of the package-list tag refers to the filtered package name. *SensitivityFilter* comprises only the filter tag. The filter tag has two attributes—name and rule. The name attribute is *SMS_HEAD_CHECK* or *SMS_BODY_CHECK*, and the rule attribute compares the string for filtering. In this example, when Android receives an SMS message that is sent from "1588-2486" and contains a "passcode," "pin code," or "auth" string, *ReferenceMonitor* determines that this SMS message is sensitive.

Table 2: Examples of *SensitivityPolicy* and *SensitivityFilter*

<?xml version="1.0" encoding="utf-8"?> <sensitivities> <intent name="android.provider.Telephony.SMS_RECEIVED" type="BLACK" > <check-list name="SMS_BODY_CHECK" /> <check-list name="SMS_HEAD_CHECK" /> <package-list name="com.handcent.nextsms" /> </intent> </sensitivities>	<sensitivityFilters> <filter name="SMS_HEAD_CHECK" rule="15882486" /> <filter name="SMS_BODY_CHECK" rule="passcode" /> <filter name="SMS_BODY_CHECK" rule="pin code" /> <filter name="SMS_BODY_CHECK" rule="auth" /> </sensitivityFilters>
SensitivityPolicy	*SensitivityFilter*

When a sensitive SMS message is incoming, *IntentBroadcaster* checks the filtering policy. If the filtering target is on the list, *IntentBroadcaster* excludes it from the receiver list. *IntentBroadcaster* then adds the "hiding" string value in the *SMS_DELIVER* intent to notify the default SMS application of the sensitivity. Detailed algorithms of *IntentBroadcaster* are presented in Table 3.

Table 3. Algorithms of *IntentBroadcaster*

Algorithms	
Receive: SMS_DELIVER intent broadcasting request 1. ***if*** (sensitive policy is set) 1. ***case***(check-list) ***SMS_HEAD_CHECK|SMS_BODY_CHECK:*** 1. checking filtering rules about address 2. checking filtering rules about body strings 3. ***if*** (both rules are satisfied) 1. add hiding tag to Intent ***SMS_HEAD_CHECK:*** 1. checking filtering rules about address 2. ***if*** (rule is satisfied) 1. add hiding tag to Intent ***SMS_BODY_CHECK:*** 1. checking filtering rules about body strings 2. ***if*** (rule is satisfied) 1. add hiding tag to Intent	Receive: SMS_RECEIVED intent broadcasting request 1. ***if*** (sensitive policy is set) 1. ***case***(check-list) ***SMS_HEAD_CHECK|SMS_BODY_CHECK:*** 1. checking filtering rules about address 2. checking filtering rules about body strings 3. ***if*** (both rules are satisfied) 1. ***if*** (type is BLACK) package removed 2. ***else*** making new list ***SMS_HEAD_CHECK:*** 1. checking filtering rules about address 2. ***if*** (rule is satisfied) 1. ***if*** (type is BLACK) package removed 2. ***else*** making new list ***SMS_BODY_CHECK:*** 1. checking filtering rules about body strings 2. ***if*** (rule is satisfied) 1. ***if*** (type is BLACK) package removed 2. ***else*** making new list

We also modified *SmsReceiverService* in the default SMS application. When *SmsReceiverService* receives an *SMS_DELIVER* intent, it confirms that an additional field exists that has a "hiding" string. If the *SMS_DELIVER* intent has a "hiding" string, *SmsReceiverService* requests a modified query from the native SMS *ContentProvider*.

4. Demonstration of Privacy-enhanced SMS Provider

We implemented PESP on a Nexus 5 with Android OS version 4.4.2. To implement PESP, we modified and rebuilt *ActivityManager*, the default SMS application, and the native SMS *ContentProvider*. To demonstrate, we tested *SensitivityPolicy* and *SensitivityFilter*. For the sake of the experiment, we assumed that the Handcent SMS application is an abnormal application despite its benign nature.

We added the phone number of a Samsung smartphone to *SMS_HEAD_CHECK* of *SensitivityFilter* and assigned an "auth" string value to *SMS_BODY_CHECK*. We then sent two SMS messages. The first message was "Normal SMS Sending," and the second message was "Hidden SMS Sending (auth 0100)." As a result, in Figure 5, we can show that the unfiltered application (left, default SMS application) received the hidden SMS, whereas the filtered application (middle, Handcent) could not receive the hidden SMS.

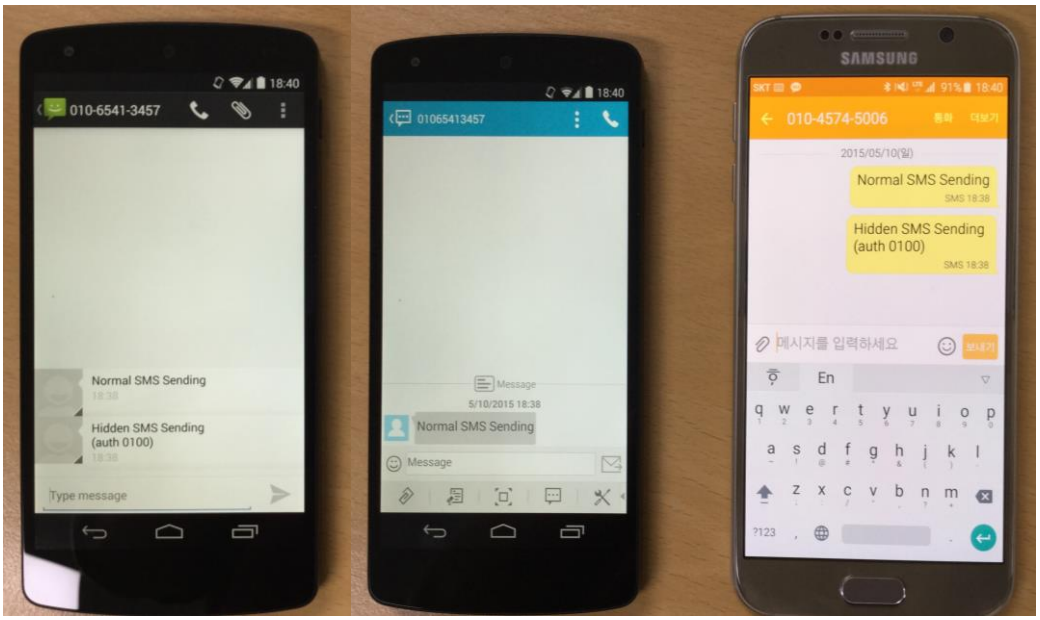

Figure 5. Access Hidden SMS Message

5. Conclusion

We proposed the architecture of a privacy-enhanced SMS provider to protect sensitive SMS messages. In general, SMS Trojans have two chances to steal SMS messages in the Android platform because the permission model is coarse-grained. The first is by using the *SMS_RECEIVED* intent, which is broadcasted when a new SMS is incoming. The second is by receiving *READ_SMS* permission from the owner. It is not difficult for an application to deceive the owner by claiming to need *SMS_RECEIVED* or *READ_SMS* permissions. If the application requests permission to access its own authentication message, the owner is easily convinced. For applications to extract useful information from SMS messages as a sort of card utilization history received by the SMS, it needs only *READ_SMS* permission. Therefore, it is possible to easily obtain *READ_SMS* permission when users accept it as if it were such an application. In this paper, we proposed PESP in order to protect these sensitive SMS messages from SMS Trojans. We additionally implemented *ReferenceMonitor*, which watches incoming SMS messages under the sensitivity policies. If a sensitive SMS is incoming, *ActivityManager* broadcasts the *SMS_RECEIVED* intent to filtered receivers, and the default SMS application requests the hiding SMS message from the native SMS provider. As a result, third-party applications cannot receive *SMS_RECEIVED* intent or access hidden SMS messages.

Our future research will focus on extending our security model to cover more general permissions. Finally, we will continue to conduct performance evaluation through simulations.

Acknowledgement

This work was supported by the ICT R&D program of MSIP/IITP. [2014-044-072-003 , Development of Cyber Quarantine System using SDN Techniques]

References

[1] "Global Smartphone Market Analysis and Outlook: Disruption in a Changing Market," CCS Insight, **(2014)** June, pp. 3.

[2] "Smartphone OS Market Share", Q4, 2014, http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[3] Kaspersky Labs, "Kaspersky Seuciry Bulletin 2014", Overall statistics for 2014, **(2014)** December.
[4] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin and D. Wanger, "Android Permissions: User Attention, Comprehension, and Behavior," Proceedings of the Eighth Symposium on Usable Privacy and Security, Article no. 3, **(2012)**.
[5] W. Enck, D. Octeau, P. McDaniel and S. Chaudhuri, "A Study of Android Application Security," Proceedings of the 20th USENIX conference on Security, **(2011)**, pp. 21–21.
[6] J. J. Drake, P. O. Fora, Z. Lanier, C. Mulliner, S. A. Ridley and G. Wicherski, "Android Hacker's Handbook," John Wiley & Sons, Inc.
[7] J. Six, "Application Security for the Android Platform," O'Reilly Media.
[8] H. Lee, M. Park, J. Lim, J. Kim, J. Eom and T. M Chung, "Advanced Content Security for Android Platform: Security Analysis and Implementation Issues," Proceedings of IJCC, AACL, vol. 04, **(2015)**, pp. 1–4.
[9] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu and G. Kaiser, "Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems", Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, **(2014)**, pp. 113-129.
[10] J. Li, D. Gu and Y. Luo, "Android Malware Forensics: Reconstruction of Malicious Events," Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops, **(2012)**, pp. 552-538.

Authors

Min Woo Park, is currently a Doctor candidate at Sungkyunkwan University, Suwon, Korea. He received his B.S. degrees in Information & Communication Engineering from the SungKyunKwan University, Suwon, Korea, in 2008, respectively. He received his M.S. degrees in Department of Electrical and Computer Engineering from the SungKyunKwan University, Suwon, Korea, in 2010, respectively. His research interests include security of Sensor Networks, Cloud Computing and Smartphone.

Jung ho Eom, received his M.S. and Ph.D. degrees in Computer Engineering from Sungkyunkwan University, Suwon, Korea in 2003 and 2008, respectively. He is currently a professor of Military Studies at Daejeon University, Daejeon, Korea. His research interests are information security, cyber warfare, network security.

Tai-Myoung Chung, received his first B.S. degree in Electrical Engineering from Yonsei University, Korea in 1981 and his second B.S. degree in Computer Science from University of Illinois, Chicago, U.S.A. in 1984. He received the M.S. degree in Computer Engineering from University of Illinois in 1987 and a Ph.D. degree in Computer Engineering from Purdue University, W. Lafayette, U.S.A. in 1995. He is currently a professor at Sungkyunkwan University, Suwon, Korea. He is now vice-chair of the Working Party on Information Security & Privacy, OECD, and a senior member of IEEE. He also serves as a Presidential committee member of the Korean e-government, the chair of the

information resource management committee of the e-government. He is an expert member of Presidential Advisory Committee on Science & Technology of Korea, and is chair of the Consortium of Computer Emergency Response Teams (CERTs). His research interests are in information security, network, information management, and protocols on the next-generation networks such as active networks, grid networks, and mobile networks.