Hindawi Publishing Corporation
Mobile Information Systems
Volume 2015, Article ID 746930, 13 pages
http://dx.doi.org/10.1155/2015/746930

Research Article

Messaging Attacks on Android: Vulnerabilities and Intrusion Detection

Khodor Hamandi, Alaa Salman, Imad H. Elhajj, Ali Chehab, and Ayman Kayssi

Department of Electrical and Computer Engineering, American University of Beirut, Beirut 1107 2020, Lebanon

Correspondence should be addressed to Imad H. Elhajj; ie05@aub.edu.lb

Received 31 October 2013; Accepted 26 February 2014

Academic Editor: David Taniar

Currently, Android is the leading mobile operating system in number of users worldwide. On the security side, Android has had significant challenges despite the efforts of the Android designers to provide a secure environment for apps. In this paper, we present numerous attacks targeting the messaging framework of the Android system. Our focus is on SMS, USSD, and the evolution of their associated security in Android and accordingly the development of related attacks. Also, we shed light on the Android elements that are responsible for these attacks. Furthermore, we present the architecture of an intrusion detection system (IDS) that promises to thwart SMS messaging attacks. Our IDS shows a detection rate of 87.50% with zero false positives.

1. Introduction

Smartphones have become an integral part of our lives along with their enormous capabilities. At the time of writing this work, the state-of-the-art smartphone is the Samsung Galaxy S4. The S4 is effectively a small, yet extremely powerful device capable of performing very sophisticated computing and communications tasks. On the software side, the Android operating system (OS) has been, for several years now, the OS of choice for Samsung phones. Recent statistics clearly show that Android dominates the smartphone market in both Europe and China with 70.4% [1]. Google claims that more than 1 million new Android devices are activated worldwide each day [2]. Android was first released in 2008, and since then seven codenames and more than nine versions were presented [2]. Android users are using the Gingerbread version of Android (36.4%), followed by version 4.1 of Jelly Bean (29%), and then Ice Cream Sandwich (25.6%). Android is, by design, an *open* OS and is governed by two licenses. The GNU Public License v2 (GPLv2) is used for the kernel and the Apache 2.0 license for the rest. The reason behind using the Apache license is due to its fewer restrictions on freely published work such as not requiring the source code to be disclosed, a very appealing aspect for manufacturers [3]. On the other hand, users are able to access the OS source code and can use the publicly available application programming interfaces (API) to build applications and publish them on the Android application market [2]. This philosophy has enriched the Android market with over 675,000 applications by September 2012 [4]. Furthermore, Google announced in May 2013 that more than 48 billion apps were already downloaded from the market, with 2.5 billion app installs per month [5]. With this popularity comes a dark side of Android; in fact, eight million new mobile applications were identified by McAfee as malware between April and June of 2012 [6]. In addition, F-secure reported that Android has the largest share of malware, an estimation of 96% in the last quarter of 2012, with a twofold increase from the preceding quarter [7].

The report in [6] added that the newly discovered problems emanated from short messaging service- (SMS-) based malware, mobile botnets, spyware, and destructive Trojans. As an example, "Trojan!SMSZombie," an SMS Android Trojan, discovered in July 2012, infected 500,000 phones in China. Many financial transactions and payments are processed using SMS in China; the ability to intercept SMS payloads and to have the privilege to send SMS messages has granted this Trojan the ability to execute numerous attacks [8]. In September 2012, a malicious Android game app was detected and its developers were fined 77,500 USD. This app used SMS in order to make users subscribe to a nonfree

service and was estimated to have collected 397,000 USD [9]. Also in the same month, "FakeInst" was discovered. It is a Trojan that masquerades as a basic text exchange app while secretly subscribing to premium-rate services by sending SMS messages silently. This Trojan was reported to have stolen 10 million USD [10].

In this paper, we discuss the main features and vulnerabilities of the Android OS that allow the development and infection by SMS malware. In addition, we demonstrate how Android-based smartphones can be exploited by deploying a malware that uses the SMS service as its medium of operation. Typically, this type of malware relies on vulnerabilities from two parties: the telecom-operator on one hand and the Android OS on the other hand. For the former, a good number of mobile operators use SMS text messaging to transfer units/credits between two mobile users without requiring any form of validation/authorization beyond the message being sent from the phone. The units/credits refer to the user balance or airtime that can be used to make phone calls, to send/receive SMS messages, or to access the Internet. For example, users who want to transfer units build a structure-defined text by entering two elements: the amount they want to transfer from their balance and the mobile number of the beneficiary. After drafting the message, the user sends it to a specific number and then the operator completes the transaction through balance transfer. This can be extended to other potential financial transactions that are made through SMS as well. As for the Android-specific vulnerabilities, the problem consists of two design features relating to the way SMS messages are sent and received on Android. To demonstrate these vulnerabilities, we were able to build and test a proof-of-concept malicious mobile app. This malware masquerades as a normal messaging application while in reality it would be covertly conducting credit transfers without user knowledge, while at the same time suppressing any confirmation messages received from the operator.

Furthermore, as part of the evolution of Android security, version 4.2 of the OS employs a new mechanism in an attempt to prevent SMS premium-rate attacks. We will discuss how this mechanism works and present a vulnerability in the design of the new procedure. This vulnerability allowed us to implement a workaround through a specific attacker model. Hence, the new mechanism is not efficient and keeps the users at risk of attacks of credit transfer, premium-rate SMS, and SMS-based botnets.

In addition, we will present a new attack on unstructured supplementary service data (USSD) transfer. Most operators rely on USSD in order to perform many operations such as recharging, credit transfer, balance checking, and many other operations. For this reason, USSD represents a high value target to hackers in order to reap great benefits similar to the ones seen in SMS attacks. We will demonstrate, through a proof-of-concept app design, how a benign phone dialer can have malicious components that allow for USSD-based attacks.

Based on a survey we conducted, we identified more than 20 mobile operators that use SMS as a service or a variation of it for unit/credit transfer between users. These operators are spread geographically in 28 countries (some operators have presence in more than one country). As such, the number of vulnerable users and operators at risk is significant. Similarly, USSD use is widespread around the world.

In order to address the SMS-based attacks on Android, we propose an anomaly-based intrusion detection system (IDS). Anomaly detection identifies unacceptable deviation from expected behavior. The typical way was to collect "normal profiles" and use them in order to detect outliers. In recent years, complex techniques have been introduced to enhance detection. Examples of these techniques involved the use of recurrence quantification analysis along with machine learning in order to enhance the anomaly capture when faced with a complex and nonlinear network traffic dynamics [11]. In addition, some of these techniques also tackle the problem of obsolete gained information used in determining normal behavior and accordingly detecting anomalous behavior. For that, a restricted Boltzmann machine was used in order to make intrusion detectors learn dynamically anomalous behavior [12]. Regarding solutions for Android, the research community has seen many proposed solutions in the past that are in fact very appealing but unfortunately cannot be used by end-users effectively due to the high requirements of the IDS. These requirements come in the form of high privilege, or major modifications to the OS, and are typically OS version specific and may demand a patch that sometimes takes months to become available. For that purpose, we propose an IDS solution that resides at the app level. Our IDS monitors many features from the system and builds a per-app score in order to pinpoint the aggressors.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 presents some Android background relevant to the work done, and Sections 4, 5, and 6 present the different attacks. We detail the IDS design, testing, and analysis in Section 7. Finally, we conclude the paper in Section 8.

2. Literature Review

Golde was able to find a number of vulnerabilities in SMS implementations that are used by the majority of the feature phones on the market despite the closeness of these phones' operating systems [13]. He showed how SMS vulnerabilities could be used to disconnect the phone from the network, end calls, crash, and reboot. In addition, Denial-of-service (DoS) cases were also seen because of some tests that made the phone crash before the SMS is acknowledged, so the network was under the impression that the message did not get delivered and hence continuously retransmitted it. Moreover, he showed how a subscriber identity module (SIM) data download (a management tool used by operators to remotely manage SIM cards), through which SMS is directly sent to SIM (or USIM, the SIM card used in third generation or 3G networks), can be manipulated so the attacked phone will send SMS from the phone to any number the attacker specifies, a process through which users' units/credits can be drained slowly. Furthermore, he showed how this last feature could be used to carry out a DoS attack on a

specific mobile number. Traynor et al. demonstrated how SMS messaging can be malicious and harmful to the network [14]. Many mobile operators provide an Internet-based SMS service through which users can send SMS messages directly from the web to a mobile phone connected to the operator network. This service, if exploited, can lead to DoS attacks, and thus preventing mobile users from making phone calls in a targeted city. Mulliner and Miller presented a general framework that can be used specifically with smartphones for testing and monitoring of SMS messages [15]. Although many smartphones were investigated in [15], our main interest is the Android-based ones. In their work, they introduced a way to inject messages and monitor telephony by modifying the serial line that the radio interface layer uses to communicate with the modem. Castiglione et al. [16] were among the first to pinpoint potential threats emerging from smartphones and the interconnection of the Internet to the telephone networks. They demonstrated how the use of the enhanced messaging service (EMS), an enhanced version of SMS, can carry malicious binary data. In a simpler form, WAP-Push SMS, pretending to be originated from the operator, can be used to deliver a URL of a malicious application. In order to accomplish the latter attack, they demonstrated how identity theft can be carried out by injecting an SMS message directly to an SMSC by using a web portal that allows users to specify the senders. The authors presented potential countermeasures to thwart the presented attacks including the involvement of cryptographic operations that are possible with powerful devices like smartphones.

From a different perspective, the Android permission system has been under study and proved to be vulnerable to privilege escalation attacks. Davi et al. presented a method whereby applications can place phone calls without having such a privilege [17]. Although they added that this problem was solved, they showed a proof-of-concept scenario through which much more harm can be done whereby a "nonprivileged vulnerable app" was used to execute Tcl commands, which ended up sending some 50 SMS messages to any number the attacker specified. As a result, not understanding the difference between different Android permissions can put users at risk. This was also confirmed in [18] where an online survey and interviews were conducted with a group of Android users and the results showed that only a minority of these users were able to understand the reason and the difference between the various permissions required by applications. For SMS in particular, it is very important to understand the difference between the four available permissions: "SEND_SMS," "RECEIVE_SMS," "READ_SMS," and "WRITE_SMS."

The Android operating system was first released in September 2008. By that time, more than 280 smartphone malwares were already identified and SMS was a target of such malware in order to steal users' money, according to [17]. In addition, the authors were able to develop a malware in the form of a native "Linux binary" that can be stored in an image, for example, and that can be used to bypass Android permissions. Bypassing a specific permission allows an intruder to do any operation, including sending SMS messages.

A vulnerability in the application launching flow has been demonstrated by Armando et al. [19, 20]. The vulnerability allows any app, without requesting any permission, to send a "malicious fork command." These commands bypass security checks and create child processes that are not bound to any app and accordingly not removable. Amplified, the child processes fill the device's memory and accordingly force it to restart. Furthermore, two countermeasures were proposed. The first consists of checking the requesting party and allowing only legitimate requests from system server and root. The second consists of fixing the Zygote socket by lowering its permissions. In addition, in [20], they extended their security assessment to two other sockets in Android OS, the keystore and property_service sockets. These sockets have been found to possess high permissions, similar to Zygote. After tests, the keystore and the property_service sockets turned out to be resilient against attacks. Although any app can try to connect to them, they do not accept commands from nonlegitimate users by using a socket credential mechanism and allowing only specific users to obtain access.

Furthermore, three major works were able to survey Android-specific attacks. Vidas et al. presented a survey on Android attacks in [21]. The paper surveyed the different attacks that are due to the weaknesses in the Android security model. Many attacks arise from the fact that permissions used in apps are confusing, and the design of intent broadcasting makes it further complicated. The attacks were divided into four major categories depending on whether or not the attacker has physical access to the phone and on the availability of the Android debug bridge (ADB) tool. The ADB tool has the ability to install malicious apps directly from a PC to the phone through a USB cable, and the installed apps do not have to pass through the official market. The main problems of the unprivileged attacks are due to the unnecessary permissions granted to apps. Many of the malicious apps were repackaged versions of legitimate apps. Other attacks, as discussed, stem from the Android recovery mode and boot process whereby the attacker creates a malicious recovery image, which leads to a privileged access on a device. On top of all that, detection of malicious apps is problematic on Android, since antivirus and firewall software need higher privileges than malicious ones, and hence they are not very effective. Moreover, despite the fact that Google mitigates vulnerabilities by releasing patches, the release takes months to reach devices after the detection is made. Numerous mitigations were proposed, including reducing the patch cycle, allowing some applications such as antiviruses to get higher privileges in order to make them more effective, and authenticating Android market downloads and ADB. In [22], the authors tried to understand the incentives behind malwares on iOS, Android, and Symbian. For this purpose, they collected 46 pieces of malware samples from January 2009 till June 2011. The results showed that 61% of the samples collect user information, 52% send premium-rate SMS, and many others use attacks such as sending spam SMS. The authors also noted that root exploits for these OS are publically available 74% of the time. Out of the 46 samples, 15 were shown to change their behavior based on commands received from command and control (C&C) servers. In

addition, malicious apps tend to request more permissions than nonmalicious apps. On average, these numbers were 6.18 permissions for malicious apps versus 3.46 for regular apps. Finally, they mentioned how different entities can benefit from stolen data or attack, and they gave monetary values for each attack. The number of Android-based malwares has been increasing tremendously, which leads to the work in [23] whereby 1,260 Android malware samples were captured and evaluated from August 2010 till October 2011. The main focus was on the mechanisms through which malware propagates, the activation procedures, the permissions required, and the events also known as the "broadcasted intents" that will be listened to by the malware. The results showed that 86% were repackaged versions of legitimate apps, 36.7% use root exploits in order to obtain a high privilege, 93% turn the phone into a botnet, and 51.1% harvest users' information. Also, results have shown that 21 malware families, out of a total of 49 families, listen to incoming SMS messages, the second most used broadcast after boot broadcast. Moreover, 45.3% of the malware samples "tend to subscribe to premium-rate services with background SMS messages." In addition, some were found to do some filtering of SMS messages and some were discovered to reply to the received messages. Furthermore, the antiviruses were not able to demonstrate a full ability to detect malware beyond a best-case detection rate of 79.6%.

Facing such an enormous wave of attacks along with the associated losses on Android-based systems, a stream of work has been developed in order to build tools that can detect and prevent such attacks. For example, in [24] the authors present mobile-based IDS that uses a knowledge-based temporal abstraction (KBTA) method, promising to outperform signature-based antimalwares. The KBTA takes input time-stamped parameters and events to get context-based interpretation. In order to evaluate the KBTA, an Android-based IDS was developed that primarily consists of an Agent Service that is responsible for handling all the IDS activities. The Agent Service talks to processors (e.g., KBTA) that feed the output to a weighting unit, then to an alert manager, and finally back to the Agent Service. The IDS monitors the system, collects and processes parameters and events at regular time intervals. The number of monitored raw parameters was 160. In addition, 4 malwares based on 4 classes of attacks were also developed: unsolicited information, theft of service, information theft, and DoS. In [25], a behavior-based detection framework for Android-based devices, "Andromaly," was presented. This IDS is host-based. It monitors and collects different system metrics that are fed to a detection unit, which uses machine learning and a classification algorithm. The main goal is to check if "Andromaly" can correctly classify an app as a game or a tool based on its behavior, which promises to detect abnormal behavior. The work presented is close to [24], and the same framework with the Agent Service is utilized. Our work is relatively close to [24], but instead it is anomaly-based. On the other hand, the main problem with the IDSs in [24, 25] is the fact that there is lack of concrete evidence on how the detection is done. Similarly, "TaintDroid" is presented in [26] as a solution for tracking how third-party apps acquire user sensitive data by using "system-wide dynamic taint tracking and analysis" at a very low-level. In [27], "QuantDroid" which is based on "TainDroid" tries to monitor interprocess communication (IPC) between apps in order to mitigate privilege escalation attacks. Monitoring the communication is done by modifying the "android.os.Binder" module since high-level messages, such as intents, are sent using a low-level binder with involvement of high-level middleware. A threshold related to the size (in bytes) of information transmission between apps is used to detect a violation or potential information leakage. The major drawback in these two works, TaintDroid and QuantDroid, is the fact that they require a deep modification to the Android OS, so they cannot be deployed at the app-level.

3. Android Background

Android provides developers with a software developer kit (SDK) that exposes the API needed to build applications. A comprehensive documentation along with the SDK is provided on a dedicated website [2]. In what follows, we will describe selected topics from the SDK that are relevant to our work.

3.1. Activities. An activity is the component that provides a user with a graphical user interface (GUI) [2]. Every application should have at least one activity, the "main" activity. Usually, an application has many activities and it can start another activity; each activity holds a state (e.g., stopped or paused).

3.2. Services. Services are components that do not provide any user interface, and they usually run in the background performing long-running operations. Services are started by other application components, so an activity or a service can start a service, and a service life is independent of the component that started that service [2].

3.3. Permissions. For security reasons, some subsets of the API are not accessible by an application without having permission for it. Usually, this permission granting must be statically declared in a "Manifest.xml" file when developing the application. These permissions can be used for many reasons such as filtering in the Android store, notifying the user (at installation time only), and guarding these critical APIs from malicious use [2].

3.4. Broadcast Receiver. Broadcast receiver is a mechanism that defines how Android forwards data to applications. The main usage of these broadcast receivers is interprocess communications and tracking of specific events (e.g., arrival of an SMS message to the phone). Applications declare statically or dynamically their interest in receiving a certain type of information and accordingly the OS will try to deliver the requested data when available. For the procedure of sending information, Android uses "intents" which are data structures that should be passed to "sendBroadcast," for example. Android defines two types of broadcast receivers:

normal and ordered. The normal ones are asynchronous so there is no defined order according to which apps would receive the data. As for the ordered ones, a priority can be set to require from the system to deliver the information to each app in a certain sequence, and as such some apps will get the information before others. This feature allows developers to capture and possibly modify the carried data before it reaches lower-priority consumers. In this case, an app can prevent other apps from getting specific data by aborting the received data [2]. According to our experiments, for an app to ensure that it will obtain an ordered broadcast, it has to be the first application to register to the desired intent with the highest priority. It is worth noting that an app can register for an intent with any priority it specifies with no constraints or limitations after being given permission.

3.5. SMS Manager. The SMS manager, part of the Android telephony stack, provides developers with the necessary functions to send messages. In order to send a text message, and apart from getting the right permissions, an app can send a text message at any time by a simple function call. The main function to send SMS messages is "sendTextMessage" [2]. Calling the send function displays no notifications on the phone; the sending process is seamless and transparent to mobile users.

3.6. Logcat. Android has a special logging system through which the OS stores the logs in several circular buffers (for radio, events, and main). Developers can benefit from these buffers to get information from the system and debug their apps. For that purpose, a special command (*logcat*) can be used in the ADB tool to extract data from the targeted buffer [2].

4. SMS-Based Attacks Prior to Version 4.2

In what follows, we will describe the design in terms of activities, services, permissions, and broadcast receivers required for each of the apps we developed. In that sense, each of the three attacks, SMS on Android prior to 4.2, SMS on Android 4.2 and beyond (Section 5), and USSD (Section 6), has its own app. For this purpose, we will show how each component of the app contributes to the attacking mechanism. Also, we will explain how the apps were tested.

Prior to the introduction of the Android version 4.2, Jelly Bean, users were not prompted when an app tries to send an SMS message. In this first part, we will show how an attack can be carried out.

4.1. Application Design. The application is designed to look like a normal SMS application that has the ability to send and receive SMS. There are in fact many such applications for Android users, and many of them are popular (such as Go SMS Pro) due to the fact that users can replace the plain SMS application that comes with the OS with more sophisticated and user-friendly ones. The popularity of these applications demonstrates the feasibility and ease with which a malicious messaging application can be deployed simply by masquerading as a user-friendly SMS application.

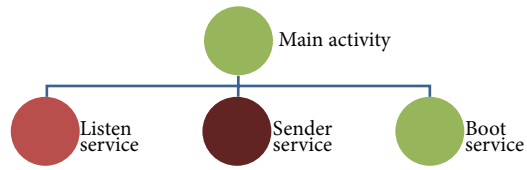


Figure 1: The components of our app.

Malicious code was added to the application in order to specifically target the unit/credit transfer through the SMS service. For that purpose, the application needs at least a single activity and three service components as shown in Figure 1.

4.1.1. Main Activity. The main activity component has the entire graphical user interface to read and send SMS messages. On Android phones, SMS messages are inserted and read using database queries to the content provider "content://sms/". In addition, this component is the base component that launches the listen service and the sender service, at least for the first time. On the other hand, the boot service runs on its own after the first launch of the main activity. Also, note that although we are describing a single activity application, this is by no means a limitation as an application design can have many activities.

4.1.2. Listen Service. This component listens for incoming SMS messages and takes actions according to predefined criteria. Since this service needs to listen for incoming messages, it has to be registered as a broadcast receiver. When an SMS message arrives, this component gets notified. Incoming messages are then parsed and checked. If the newly received message is not an acknowledgment related to the "illegal" unit/credit transfer, the component allows the message to pass unmodified or it can directly insert it in the messaging database. If, on the other hand, this message is related to the malicious activity, it will be suppressed and will never reach the database or any other application. It is important to note that the priority option available for a registered broadcast receiver makes the suppression very efficient and we believe it to be vulnerability in the Android OS. In addition, this service could be made "not closable" which will allow it to rerun, even if the user intentionally closes it.

4.1.3. Sender Service. This component is the main malicious part of the application. It is made to work in a silent manner and to perform credit transfers at random times. Of course, this component will not allow its messages to be stored in the database, so the user will not be able to track when the transfer was made. Also, this service is not closable. Many refinements can be added to this service; for example, it can be modified in such a way to monitor the activity level of the user and then to execute the malicious transfers during the busiest periods when the user is actually using the phone for

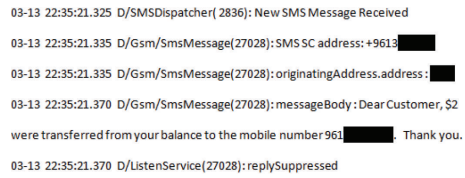


Figure 2: Output from main buffer showing logs of the receipt and suppression of SMS.

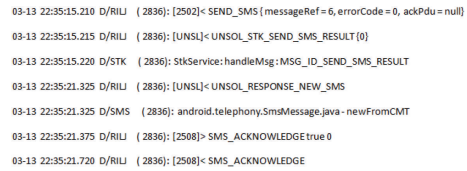


Figure 3: Output from radio buffer showing logs of the sending and receiving of SMS.

calls and/or SMS. This will result in less likelihood of the user noticing the reduction in credit.

4.1.4. Boot Service. This component is needed to make the previously mentioned services run at the launching of the OS. This is achieved by registering as broadcast receiver to "BOOT_COMPLETED" event.

4.1.5. Permissions. The minimum permissions needed to carry out the malicious activities are the "RECEIVE_SMS" and "SEND_SMS" which are typically used by any SMS application. Note that the most popular SMS applications surveyed on the Android market use additionally the "READ_SMS" and the "WRITE_SMS" permissions. Therefore, the request for these permissions would not alert the user to the malicious behavior.

4.2. Testing. In what follows, we demonstrate how we tested the application, the types of security checks that were performed on the application, and the results that were obtained.

4.2.1. Implementation. The testing was made using two mobile phones. Although the application needs an Android-based smartphone to run, it is not a must to use two smartphones. In fact, we used a Samsung Galaxy SII smartphone and a Sony Ericsson K770i feature phone. The application was installed on the victim phone, the Samsung Galaxy SII. The Android OS version preinstalled on this phone is 4.0.3 (Ice Cream Sandwich). On the other hand, the Sony Ericsson holds the SIM card of the attacker who will get all the transferred credits.

The credit-transfer operation in our experiment is done by building a message that has the following format: `ReceiverNumberTAmount`. This message is usually sent to a special dedicated 4-digit number. Once the message is sent, the credits are removed from the sender balance and added to the receiver balance. Finally, both sender and receiver get a message informing them that the credit-transfer transaction was completed.

Figure 4: Results of VirusTotal scanning our application.

In order for the victim not to notice the transaction, we had to suppress the sent and received messages related to this operation. Hence, we used the "logcat" tool to check that the operation is being done correctly. The output from the "main" buffer showed that the transfer is being carried out. A sample output is shown in Figure 2.

Additional output from the "radio" buffer having the same timestamp confirmed the operation and a sample is shown in Figure 3.

4.2.2. Antimalware. In order to check if this malware is detectable, a freely available service called "VirusTotal" was used. This service provides an online scanning for URLs and files, including Android-apk files, using a set of the major antimalware currently available on the market [28]. The report that was generated is partially shown in Figure 4 and it confirms that none of the 43 antimalwares was able to detect the fact that this application is malicious. This is not surprising since most of these tools are rule-based.

4.2.3. Android Market (Play Store). The final test was to check the response of the Android Market (now renamed as Play Store). The test goal was to check whether or not the store could detect that the application has a malicious component. For that purpose, and with a modification to guarantee that the risk of accidental release of the application is minimal, the application was published successfully for a period of time as shown in Figure 5 and then unpublished as seen in Figure 6 in order to make sure that it does not get downloaded by any user.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ? >
<premium-sms-policy >
        <package name=" com.myapp.smsapp42" sms-policy=" 3" / >
</premium-sms-policy >
```

Box 1

Figure 5: Android developer console showing publishing of application.

Figure 6: ADC showing removal of the application.

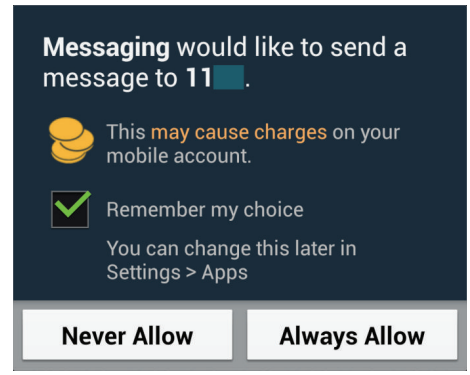


Figure 7: Dialog that appears on SMS sending to a 4-digit number.

5. SMS-Based Attacks Beyond 4.2

The telephony manager is the application framework responsible for providing SMS messaging API and directly responsible for security of SMS. As part of the evolution of Android, this framework has seen important modifications in the newest release of the OS, namely, version 4.2 (Jelly Bean). The major change defined in the new release dictates that any app trying to send an SMS message to a premium-rate number will make the system prompt the user to authorize such action. This mechanism aims at curbing the huge volume of SMS-based attacks.

This feature of Android 4.2 was tested on a Samsung Galaxy S4 running the 4.2.2 version of Jelly Bean. In this experiment, we tried a credit transfer through the native app. This was done as demonstrated earlier, by sending a message to a 4-digit number with a drafted message formed of the beneficiary phone number concatenated with a letter `T` and then the amount of credit to transfer. Once the send button is clicked, a dialog similar to the one presented in Figure 7 is observed.

Accordingly, a user can choose one of three options: simply authorize sending, or check to remember with never allowing, or always allowing such sending. If any of the last two options is chosen, the user will not be prompted again for this particular app. From this point on, this app can send premium-rate SMS similar to any SMS app before 4.2. To modify or see the choice selected, the user can go to the settings menu under the specific app.

Under the hood, Android OS maintains this security feature by storing each app decision for premium-rate numbers in *"/data/misc/sms/premium_sms_policy.xml"*.

A sample file looks as shown in Box 1 where the value "3" of "sms-policy" means "always allow" for app "com.myapp .smsapp42".

In order to secure the access or modification of this file, a high privilege (root) is required. An experimental trial to modify this file was done on a rooted Nexus phone. In this experiment, the xml file was changed, but in order to take effect a restart was required. Accordingly, a malicious app that wants to survive the new modification has to gain root access, then modify the xml file, and finally restart the phone. Based on [23], many of previously seen malwares do a privilege escalation attack, thus gaining root access. For the previously presented pre-4.2 app, the only additional component needed for Android 4.2 is a service that checks if the phone is already rooted, and if not, apply a privilege escalation attack.

Beyond rooting, social engineering can be used as well to make users choose the "always allow" option as part of the app installation guideline. In conclusion, the new premium-SMS attacks prevention mechanism requires a stronger attacker model but can still be surmounted.

Another shortcoming of the new mechanism is the fact that regular SMS messages can still be sent without authorization from users. Similarly, hiding of received SMS messages is still possible by aborting broadcasted intents. This has many implications including having the SMS playing the role of a channel for many attacks. As stated in [22, 23], one of the primary targets is to steal private information and accordingly SMS can still be used toward that. Furthermore, SMS as a medium for propagation of command and control (C&C) messages is still easily possible, and accordingly this keeps Android OS as a haven for botnets.

6. USSD Attacks

USSD is a protocol used by operators worldwide to execute specific functionalities between users and operators. Examples of such functions include credit check and credit transfer. In what follows, we will show how a benign phone dialer app can be used to execute attacks through USSD. We will

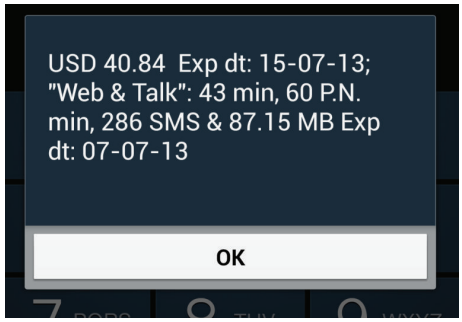


Figure 8: Returned message upon finished USSD transaction.

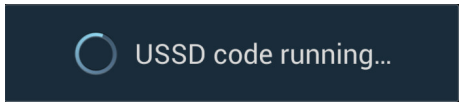


Figure 9: USSD in-progress dialog.

focus on how each component will help towards that goal. In fact, there are many such apps that are quite famous, such as "Dialer One."

6.1. Main Activity. The main activity component comprises the entire graphical user interface to make phone calls. On the other hand, the boot service runs on its own after the first launch of the main activity. Note that although we are describing a single activity application, similar to our previously described apps, this is by no means a limitation as an application design can have many activities.

6.2. USSD Interceptor Service. This component listens for incoming USSD responses and acts based on a predefined mechanism that is, of course, dependent on whether or not the response is due to an attack. To intercept a USSD response, a service should be bound to "IExtendedNetworkService" that is included under the "com.android.internal.telephony". This service provides methods to mainly modify the response message given to the user (e.g., Figure 8) and the message that is generated while the USSD is in process (e.g., Figure 9).

Accordingly, an attack on USSD can do one of two things: remove any response or return a fake response. In the first case, almost nothing is shown on the screen.

6.3. Boot Service. This component will ensure that the previously mentioned service runs at the launch of the OS and prevents the default case where "PhoneUtils" bind to it. This is achieved by registering as broadcast receiver to "BOOT_COMPLETED" event.

6.4. Permissions. The minimum permissions needed to carry out the malicious activities are the "RECEIVE_BOOT_COMPLETED" and "CALL_PHONE." Obviously, these permissions are typical for a dialer app. Therefore, the request for these permissions would not alert the user to the malicious behavior.

6.5. Testing. A proof-of-concept app was built based on the design presented in this section. The device hardware is a nonrooted Samsung Galaxy SII smartphone running the Ice Cream Sandwich Android version (4.0.3). As a sample experiment, we tried to check the balance of the user. The local operator uses the number 220 in order for subscribed users to check their balance. Since it is a USSD, the full message is composed of ∗220#. Accordingly, all the users have to do is to dial this number and a short message is received back from the operator with the requested information. Under the hood, the OS recognizes that this is a USSD from the format of the message and treats it as such. To test our app, we made it dial this number on its own from the service and information similar to that shown in Figures 8 and 9 was obtained, *but no USSD dialog box appeared on the screen*.

7. Intrusion Detection System

Android introduced the new SMS premium-rate policy in the Android 4.2.2, and, at the time of writing this work, as few as 4.0% of used devices have this version of the OS. Accordingly, the vast majority of users are still at risk of SMS-based attacks. And, as was demonstrated, the few using the 4.2.2 are still at some risk. Based on these facts, we propose an intrusion detection system (IDS) in order to detect malicious apps. Our IDS is anomaly-based and runs as an application (at the app level) and without any high privilege, which makes it suitable for any OS version and device.

7.1. IDS Components. The IDS consists mainly of four components: a data collector, an event collector, an app profiler, and a detector. Since our IDS is anomaly-based, a learning phase is required whereby apps are monitored in order to learn their associated behaviors. This behavior is quantitative and the collected values are used in the detection phase. Below is a description of each component followed by the algorithm description.

7.1.1. Data Collector. The data collector is responsible for gathering specific information relevant to our detection algorithm. This data includes running processes information, CPU consumption of each process, and package related information. This procedure is done every 5 seconds for the first two and only once for the latter. The obtained pieces are stored in a "SQLite" database for ease of retrieval and processing at later stages.

The activity manager application framework is the Android class responsible for the apps lifecycle and for providing a shared activity stack. Accordingly, it holds in a stack all the running processes, and, by inspecting the top of the stack, one can learn which app runs in the foreground or in the background. From the ActivityManager, we can obtain the list of running processes, their PIDs, their UIDs, names, and foreground/background states.

Similar to Linux, the Android OS maintains information about system resources under the "/proc" virtual file system that resides under the root directory. For a specific process with an associated PID, the CPU consumption is obtained by

parsing the file under "/proc/PID/stat". The numerical values obtained are expressed in Jiffies, which are directly related to the CPU processing times.

The package manager application framework is the Android class that allows a certain app to access information about other installed apps on the device. The manager retrieves a list of installed apps with their corresponding metadata including UID, name, and requested permissions.

As can be deduced, the data from packages and processes can be matched through the UIDs and names. We should note at this point that a timestamp is added to the collected data. The CPU consumption can be matched with a process using PIDs and timestamps.

7.1.2. Event Collector. In addition to the previously collected data, there is a great benefit in monitoring and collecting some actions or states seen on the phone, which could help in detecting malicious activities. Such actions or states are events including the screen state (on/off) and the SMS data store change seen through the "ContentObserver." The first is obtained by registering to a "BroadcastReceiver" that listens to intents sent by the OS when the screen changes state while the latter is obtained by registering to the specific content observer.

7.1.3. App Profiler and Detector. The app profiler builds a model that holds two types of information. The first type is obtained from the learning phase while the second type of information is filled on the spot during the detection phase. Accordingly, for each app, a specific model is obtained. Finally, the detector simply runs the detection algorithm.

7.2. The Detection Algorithm. The main goal of the detection algorithm is to make use of the collected data during the learning phase in order to detect if a malicious SMS was sent. The algorithm periodically checks SMS being sent from the Radio Log Buffer every 5 seconds. The reason for this check is the fact that this buffer always holds a trace that an SMS was sent. In particular two specific tags are always present: RILJ associated with the Radio Interface Layer (RIL), and SMS. It is more efficient to use the SMS tag because fewer logs are present which requires less memory and processing at each check.

If an SMS was seen in the buffer, a check is made to the sent SMS database. And if a corresponding SMS was added, then the assumption is that there is no threat since the app that sent this SMS message did not try to hide it.

On the other hand, if an SMS message was not added, each app having the permission to send SMS (android .permission.SEND_SMS) is checked.

Concerning the collection of data, we use the following notations:

(i) running process (ρ);

(ii) CPU consumption $\complement$ per process ρ of app P in snapshot ε as ($\complement_{\rho,\varepsilon}$);

(iii) the collection of all CPU consumptions $\complement_{\rho,\varepsilon}$ is the set $\mathbb{C}$.

We assume $C_{\min}$ to be the minimum amount of CPU consumption needed to send an SMS message. $C_{\min}$ can be measured in a standalone service by simply calling the "sendTextMessage" without doing any other computation or allocating any object.

We go through the pool of apps in $\mathbb{C}$: for every related consumption $\complement_{\rho,\varepsilon}$ that is higher than $C_{\min}$, we proceed with further checks, and we designate the pool of such apps as the restricted pool (P_r).

We define $C_{P_r,\min}$ and $C_{P_r,\max}$ to be the minimum and maximum amounts, respectively, of CPU consumption for a particular app in P_r within a specific time window. For each app, we compute the difference between these two values, which should indicate how much activity this app has undertaken in this specific window.

From this point on, a score is computed for each app in P_r based on its CPU consumption level, number of permissions required, screen state, and background/foreground state. Each app score is a summation of 3 components.

(1) The absolute value of the difference $|C_{P_r,\max} - C_{P_r,\min}|$:

(i) if the app is in foreground, this value is multiplied by an attenuation factor α in order to eliminate the GUI effect and accordingly get a fairer comparison; the value of α is computed as the difference between $C_{P_r,min}$ and the minimum consumption value observed when the app is in background; $\alpha \in [0, 1]$; and If an α value is not available, we take a default value of 1;

(ii) the final value in all cases (background/foreground) is normalized to a maximum of 1.

(2) The number of permissions used:

(i) if this number is bigger than 6.18 [22], a unit is added;

(ii) otherwise, a value equal to $NumberofPerm - issions/6.18$ is added.

(3) If screen is OFF, a unit is added.

Finally, the scores obtained are divided by $N = 3$ (the number of score components considered in the summation). This normalization step to the range of $[0, 1]$ makes it easy to incorporate into the algorithm additional factors in the future. From the array of scores, the top two apps having values higher than the 50% threshold will be flagged as being potentially malicious.

7.2.1. Algorithm Pseudocode. See Algorithm 1.

7.3. Testing and Results. In this work, the proposed IDS was installed on a Samsung Galaxy SII smartphone running the Ice Cream Sandwich (version 4.0.3) of Android. The phone is connected to a local operator with data connectivity available. At first, the IDS was turned into the learning mode, which was run for a full week, and provided 1,979,047 CPU consumption values along with other associated statistics.

if a log shows an SMS was sent **then**
  if the SMS sent was not added to datastore **then**
    for each $\mathbb{C}_{\rho,\varepsilon}$ **in** $\mathbb{C}$
      if $\mathbb{C}_{\rho,\varepsilon}$ is higher than $\mathbb{C}_{\min}$ **then**
        if P_r is in background **then**
          $score(P_r)$ += $Normalize(|C_{P_r,\max} - C_{P_r,\min}|)$
        else
          $score(P_r)$ += $Normalize(\alpha|C_{P_r,\max} - C_{P_r,\min}|)$
        if P_r has "Number of Permissions" > 6.18 **then**
          $score(P_r)$ += 1
        else
          $score(P_r)$ += Number of Permissions/6.18
        if screen is OFF **then**
          $score(P_r)$ += 1
        $score(P_r) = score(P_r)/N$
      flag top two P_r where $score(P_r) > 50\%$

Algorithm 1

Once the learning phase was over, the IDS was run in detection mode. In order to test our IDS, we considered many attacks that were executed repeatedly under specific conditions that will be highlighted next. We will present in what follows our experiments, some practical considerations, and finally the obtained results.

7.3.1. Practical Considerations. After data collection was done, we noticed that there are over 20 apps installed on the phone requesting the SEND_SMS permission. In addition, all except four of these apps are native apps that are installed by the phone manufacturer (Samsung in our case). The apps list include "com.android.email," "com.android.exchange," "com.android.mms," "com.android.phone," "com.android.settings.mt," "com.android.vending," "com.google.android.voicesearch," "com.handcent.nextsms," "com.jb.gosms," "com.samsung.map," and "com.whatsapp." For our tests, our main concern is to monitor untrusted third-party apps, rather than native apps.

Moreover, in order to conduct scheduled experiments, we had to rely on third-party apps that have such functionalities. On the Google Play Store, the two most famous SMS apps, at the time of writing this work, are "goSMS" (com.jb.gosms) and "HandcentSMS" (com.hancent.nextsms). Luckily, both possess scheduling functionalities and accordingly were employed in our IDS experiments.

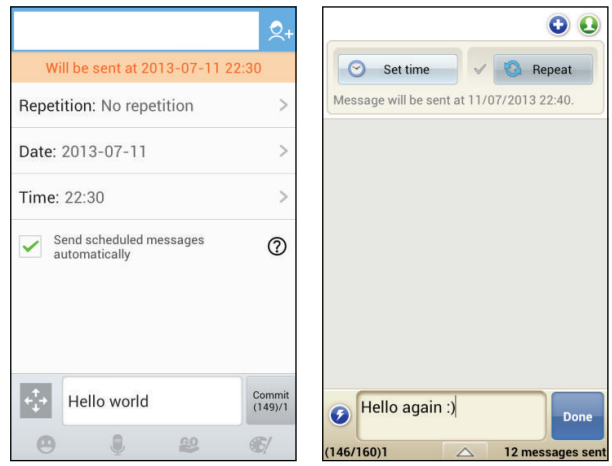
Furthermore, for each SMS app installed on the device, the algorithm uses a time window during which the CPU consumption was extracted. In order to find which apps have been made active and used the CPU, we look at CPU usage impulses. The impulses are determined by looking at the difference between the minimum and maximum CPU consumption in a window of time. Choosing the right window is a compromise since a small window would hinder detection, while a large window would make the detection of successive attacks difficult. After running several tests, it was found out that a 10- or 20-second window cannot give any valid results, while a window of 40 seconds gave very few positive results. We settled finally on a window size of 60 seconds centered on the timestamp of the radio log and providing up to 12 values of CPU consumption. In our algorithm, we are flagging the top n apps having a score higher than a predefined threshold. After extensive testing, we set $n = 2$ and a threshold value = 50%. The values of n and the threshold have a direct effect on the detection rate and on false positives. Having a high threshold will result in a low detection rate, while a low threshold will result in a high rate of false positives. Based on our tests, we concluded that malicious apps have scores higher than 50%. The value of n defines the maximum number of apps we can detect at a time. We believe that the likelihood that more than two apps attacking at the exact same time is very low.

7.3.2. Experimental Setup and Mechanisms. We carried out three types of attacks that we believe reflect real-attack models. In each attack, the involved apps attempt to send a random SMS message to a targeted user. Our main goal from these attacks is to study the capabilities of our IDS in terms of detection rate, false positives, and false negatives. In particular, we looked at SMS sending attempts under different conditions including while a single app is sending an SMS in the background and in the foreground and while multiple apps are sending SMS simultaneously. Furthermore, we were interested in seeing the smallest interval of time between which two different apps can execute an attack and that our IDS can still detect.

Background sending of SMS accounts for a large portion of SMS attacks. It was shown to be an attack in more than 50% of malicious apps in [22], and to be the attack responsible for stealing millions of dollars in [10]. In order to test the background cases, we used the scheduling SMS sending functionalities of the third-party chosen apps. Accordingly, in order to schedule an SMS, we use the app activities responsible for the scheduling and enter the text to be included in the SMS message along with the time at which the message should be sent. The app will handle simulating

Table 1: Experimental results.

App	Detection rate			
	Background	Foreground	Simultaneously (foreground and background)	Simultaneously (both in background)
goSMS	*100%*	*100%*	*60%*	*100%*
HandcentSMS	*80%*	*100%*	*60%*	*100%*



(a) (b)

Figure 10: Screenshots of scheduling components of goSMS (a) and HandcentSMS (b).

the attack from now on. We present some screenshots of the scheduling components in Figure 10.

For the foreground SMS sending attacks, the rationale behind it relies on the fact that malicious apps might try to send SMS while users are very busy or distracted. Such apps could be games that, for example, ask for SMS permission to share scores while in reality they are stealing credits through SMS. Such apps are similar to [9], where it is believed that hundreds of thousands of dollars were stolen. Executing this attack is similar to the way users send SMS from any regular app. This is done by entering the phone number of the target user along with the text and then directly sending from the GUI.

For the multiple apps experiment, we tried two different methods. In the first, we used two apps that send an SMS message at the same time. For one of the apps, the sending is scheduled and the attack is made in background. For the other app, we sent an SMS in the foreground. In the second method, we scheduled an SMS sending "attack" for both apps at the same time while both are in background. The purpose is to see in this case which app(s) would be detected.

7.3.3. Results and Discussion. At any time of the experiments, there were four third-party apps installed and usually running in background, listening for incoming or outgoing SMS. This is what is defined as normal behavior. Our IDS was run for a full day while experiments were being carried out. A set of 40 attacks were executed where 40 SMS messages were sent. Out of these attacks, 35 detections were observed which indicates a detection rate of 87.50%. For each detected app attack, an alert log is observed in the main Android log buffer similar as the following:

> *App com.jb.gosms has potentially tried to send an SMS with score 66.66% at 07-11-2013 22:22:38;*
>
> *App com.handcent.nextsms has potentially tried to send an SMS with score 66.66% at 07-11-2013 22:40:01.*

Since the IDS reads the radio log and even if a malicious app is not detected, the IDS can still detect that an unsolicited SMS was sent. In the statistics, however, we did not include those cases. We summarize our results based on experiment and related app in Table 1.

As we can see from Table 1, when a single app was acting at a time in the foreground, the IDS detected the SMS sending attempt every time. This is due to fact that the CPU consumption of that particular app rises during the activity period. On the other hand, the other SMS apps would be doing typical "minor" activity, not incurring enough CPU to influence the results. This was true for both apps.

On the other hand, when the apps are doing an SMS sending attempt in the background, our IDS was able to detect that attempt with a combined detection rate of 90%. We noticed that the goSMS is more CPU needy which made it easier to be detected and this is reflected by the associated high percentage of detection rate of 100%. For the HandcentSMS, the IDS was able to detect it as well but with a lower probability of 80%.

However, for the case of two simultaneous attacks, the IDS was able to detect one of them in the first case where one is in the foreground while the other is in the background. We believe this result is satisfactory, especially because such a case is quite rare to occur. In particular, the results showed that the one in the foreground was detected because of its GUI effect. On the other hand, for the combination where the two apps were in the background, the results demonstrated a full detection rate of 100%. For false positives, we believe that the overall low rate observed is due to the checking step we perform on both the radio log and the sent SMS database, before the algorithm is executed. In fact, the radio log provides us with concrete evidence that an SMS was sent. On the other hand, we believe that a malicious app will not add its SMS message to the SMS sent database. Accordingly, no false positive was observed when an SMS was not being sent.

8. Conclusions

In this work, we presented two types of attacks targeting Android smartphones. The first is carried out by sending SMS in the background and suppressing network notifications in order to steal subscriber credits. Also, we showed how the security of the SMS framework in Android has evolved and yet we demonstrated how such an attack could still be carried out. The second attack uses a simple phone dialer app that uses the USSD protocol in the background in order to target users. In each of the attacks, we demonstrated how each component of the Android OS is responsible for allowing such attacks. Finally, we presented an anomaly-based intrusion detection system in order to thwart SMS-based attacks. Our IDS uses primarily CPU consumption levels along with other parameters in order to flag apps trying to send an SMS message without the user's consent. The IDS was able to detect 87.50% of cases with no false positives.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research was funded by TELUS Corporation, Canada.

References

[1] C. Jones, "Apple and Android Trading Smartphone Market Shares in the Largest Markets," July 2013, http://www.forbes.com/sites/chuckjones/2013/07/02/apple-and-android-trading-market-shares-in-the-largest-markets/.

[2] Android Developers, http://developer.android.com/index.html.

[3] A. Hoog, *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*, Syngress, 2011.

[4] S. Lowe, *Google Play celebrates 25 billion downloads with 25 cent apps, discounted books, music, and movies*, 2012, http://www.theverge.com/2012/9/26/3409446/google-play-25-billion-downloads-sale.

[5] M. Panzarino, "Google announces 900 million Android activations, 48 billion apps downloaded," 2013, http://thenextweb.com/google/2013/05/15/google-announces-900-million-activations-of-android-in-total-to-date/.

[6] J. Drew, "Malware growth maintains rapid pace as mobile threats surge," September 2012, http://www.journalofaccountancy.com/News/20126400.htm.

[7] Z. Whittaker, "Android accounts for most mobile malware, says F-Secure," 2013, http://www.zdnet.com/android-accounts-for-most-mobile-malware-says-f-secure-7000012261/.

[8] J. Russel, *Stealth SMS Payment Malware Identified in Chinese Android App Stores, 500,000 Devices Infected*, 2012, http://thenextweb.com/asia/2012/08/19/stealth-sms-payment-malware-identified-chinese-app-stores-500000-android-devices-infected/.

[9] C. Osborne, "SMS malware firm ordered to compensate victims," 2012, http://www.zdnet.com/sms-malware-firm-ordered-to-compensate-victims-7000003639/.

[10] S. Yin, "Will Your Android Device Catch Malware? Depends on Where You Live," September 2012, http://securitywatch.pcmag.com/none/302362-will-your-android-device-catch-malware-depends-on-where-you-live.

[11] F. Palmieri and U. Fiore, "Network anomaly detection through nonlinear analysis," *Computers & Security*, vol. 29, no. 7, pp. 737–755, 2010.

[12] U. Fiore, F. Palmieri, A. Castiglione, and A. de Santis, "Network anomaly detection with the restricted Boltzmann machine," *Neurocomputing*, vol. 122, pp. 13–23, 2013.

[13] N. Golde, *SMS vulnerability on feature phones [M.S. thesis]*, Berlin Institute of Technology, Berlin, Germany, 2011.

[14] P. Traynor, W. Enck, P. McDaniel, and T. la Porta, "Exploiting open functionality in SMS-capable cellular networks," *Journal of Computer Security*, vol. 16, no. 6, pp. 713–742, 2008.

[15] C. Mulliner and C. Miller, "Injecting SMS messages into smart phones for security analysis," in *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*, p. 5, 2009.

[16] A. Castiglione, R. D. Prisco, and A. D. Santis, "Do you trust your phone?" in *E-Commerce and Web Technologies: 10th International Conference, EC-Web 2009, Linz, Austria, September 1–4, 2009. Proceedings*, vol. 5692 of *Lecture Notes in Computer Science*, pp. 50–61, Springer, Berlin, Germany, 2009.

[17] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Information Security*, vol. 6531 of *Lecture Notes in Computer Science*, pp. 346–360, Springer, Berlin, Germany, 2011.

[18] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS '12)*, Washington, DC, USA, July 2012.

[19] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Would you mind forking this process? A denial of service attack on Android," in *IFIP SEC 27th International Information Security and Privacy Conference*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds., vol. 376 of *IFIP Advances in Information and Communication Technology (AICT)*, pp. 13–24, Springer, Heraklion, Greece, June 2012.

[20] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Breaking and fixing the android launching flow," *Computers and Security*, vol. 39, pp. 104–115, 2013.

[21] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: a survey of current android attacks," in *Proceedings of the*

5th USENIX Workshop on Offensive Technologies, p. 10, Berkeley, Calif, USA, November 2011.

[22] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, Chicago, Ill, USA, October 2011.

[23] Y. Zhou and X. Jiang, "Dissecting Android malware: characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy (SP '12)*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[24] A. Shabtai, U. Kanonov, and Y. Elovici, "Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method," *Journal of Systems and Software*, vol. 83, no. 8, pp. 1524–1537, 2010.

[25] A. Shabtai and Y. Elovici, "Applying behavioral detection on android-based devices," in *Mobile Wireless Middleware, Operating Systems, and Applications*, vol. 48 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 235–249, Springer, Berlin, Germany, 2010.

[26] W. Enck, P. Gilbert, B. G. Chun et al., "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, 2010.

[27] T. Markmann, D. Gessner, and D. Westhoff, "QuantDroid: quantitative approach towards mitigating privilege escalation on Android," in *IEEE International Conference on Communications (ICC '13)*, pp. 2144–2149, Budapest, Hungary, 2013.

[28] Virus Total, "VirusTotal—Free Online Virus, Malware and URL Scanner," https://www.virustotal.com/.