# Rooting Android – Extending the ADB by an Auto-Connecting WiFi-Accessible Service

**Conference Paper** · October 2011
DOI: 10.1007/978-3-642-29615-4_14



**3 authors**, including:

Assem Nazar
KTH Royal Institute of Technology
**1** PUBLICATION **5** CITATIONS

SEE PROFILE

# Rooting Android – Extending the ADB by an Auto-Connecting WiFi-Accessible Service

Assem Nazar[1,2], Mark M. Seeger[1,3], and Harald Baier[1]

[1] Center for Advanced Security Research Darmstadt (CASED)
Mornewegstraße 32
64293 Darmstadt, Germany
{mark.seeger, harald.baier}@cased.de

[2] KTH - The Royal Institute of Technology
Department of Computer & System Sciences
Forum 100, SE-164 40 Kista, Sweden
anhus@kth.se

[3] Gjøvik University College
Department of Computer Science
N-2818 Gjøvik, Norway

**Abstract.** The majority of malware seen on Android has a top-down approach often targeting application programming interfaces (API) of the financially rewarding telephony and short message service (SMS). In this paper we present a proof of concept of compromising an Android based smartphone by targeting the underlying Linux kernel.
We adopt an unorthodox bottom-up approach on modifying the operating system to allow an application to re-route the Android debug bridge (ADB) daemon onto a wireless link. We support our research using case scenarios to show how information can be extracted and inserted into the smartphone without the knowledge of the user. We discuss how the Android build environment can be changed to harness functionality from secured operations. We also discuss how an application can be designed to function with minimum resources, be hidden and perform operations without user consent or interaction. We also provide an overview of how a rooted Android operating system can be misused.

**Keywords:** Android; ADB; Mobile Operating System; Rooting; Bottom-Up

## 1 Introduction

Android is developed in an environment where security plays an important role and all features and services are designed around this focal point. The Android operating system (OS) is creating ripples in the mobile operating system world with its open, modular and secure operating system. Market shares for Android

have risen considerably within the past 12 months [1], however, research associated with it has not seen a similar rise. Limited and unidirectional academic research in the field of security issues associated with Android has formed the motivating source for our research and the paper at hand. We develop a simple proof of concept for deployment and implementation of an Android application on an Android based smartphone with negligible user interaction.

The Android operating system is based on a modified Linux kernel that is used as an abstraction layer between the hardware and software stack. Core services like security, memory, process management, network stack and driver models are migrated from a Linux kernel version 2.6 [2]. A simple Android application comprises the following components:

- Activity: Almost all of the functionality of the application is fulfilled in the Activity class `android.app.Activity`. The Activity class has the sole responsibility of interacting with the user.
- Service: A service is an application component that is responsible for communicating with other applications, running in the background and providing threading support.
- Broadcast Receivers: Components built into an application that respond to system-wide broadcast announcements. Applications can also initiate broadcasts depending on the nature of data being handled.
- Content Providers: Android uses the Dalvik Virtual Machine (VM) where each application is run in its own instance of the virtual machine. The virtual machines are equipped with all necessary libraries and APIs required for the application to run normally.

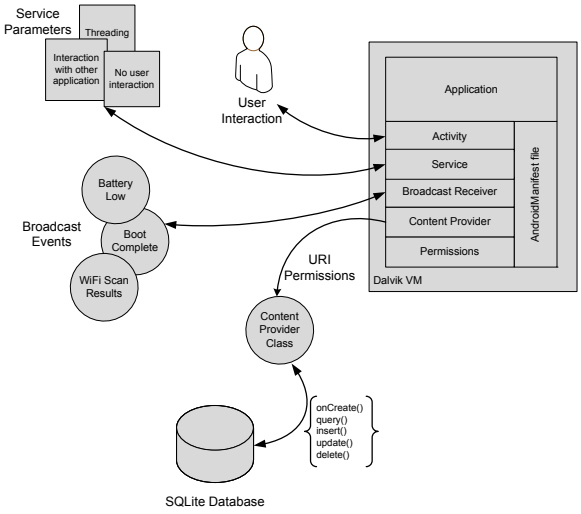


**Fig. 1.** Android application fundamentals.

The AndroidManifest.xml[4] file contains all the permissions and access rights. These rights are usually cross-checked by the Linux kernel upon installation and presented to the user if necessary. A visual representation of an Android application can be seen in Fig. 1.

By structure of an Android environment, each application is separated from other running applications by its sandbox. Application-specific data is restricted to access by the corresponding application alone, and sensitive global information (e.g. contacts and calender events) is governed by special containers known as content providers. Fig. 2 shows how a content provider works.

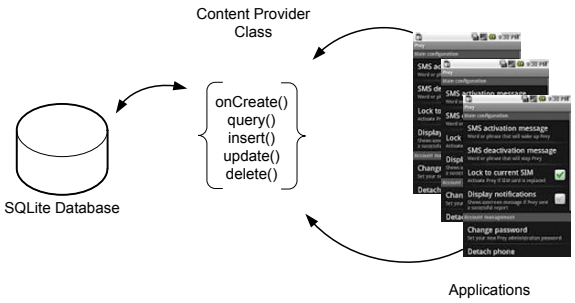


**Fig. 2.** How a typical content provider works.

Applications can access sensitive data by including relevant content providers and invoking them during run time. These content providers require permissions that have to be confirmed by the application user, and are only mentioned during installation. These permissions are never revoked during the lifetime of an application and there is no indication of these permissions being used when the process takes place.

The Android Market[5] provides another attack vector where a malware can be disguised as a seemingly novice application [3]. Reverse engineering an Android application is a lengthy process and because the Android Market neither requires a developer authentication (e.g. in contrast to Apple's AppStore) nor does contain original source codes, determining what an application actually does can be a daunting task.

As the complete Android operating system is developed in Java and open to public view, it can be used to generate an attack vector. The source code can be tinkered with to collect data, share data and even create back doors for malicious purposes. Android operating systems developed by third party vendors such as HTC, Samsung and Acer are usually closed source, however, these are

[4] The manifest file presents essential information about the application to the Android system.

[5] The Android Market is an online software store developed by Google for Android powered devices. http://market.android.com

stemmed from the same Android operating system designed by Google and can be replaced easily.

In this paper we present a novel proof of concept to gain unauthorized access to sensitive user information. We perform an attack against the Android system by using its source code and native applications packed alongside the operating system. We show that by tweaking the operating system and adding escalated privileges, a system can be compromised easily. To assess the effectiveness of our approach a modern smartphone[6] was flashed[7] with an optimized operating system, and information was pulled and pushed off the device without any user interaction.

The remainder of this paper is structured as follows. A short terminology is provided to assist with technical terms used in this paper in Section 2. An overview of related work that has been done in the field of Android security is presented in Section 3. A technical groundwork to our approach is defined in Section 4 and Section 5. We outline our approach along with the application design in Section 6. Section 7 contains a detailed technical account of the implementation of the scenario. We define the experiments that we undertook, their results and a brief overview of the outcome in Section 8. An analysis of results is discussed in Section 9. Section 10 concludes this paper.

## 2 Terminology

Many technical terms are used extensively in this paper and can be taken out of context. The following definitions are provided with regard to the Android system.

**Android System:** The Android system is the complete environment: It comprises the operating system, the firmware it interacts with, all native applications and developer tools. This in fact comprises of the complete Android source code.

**Application:** A Java based application that interacts with the Android system.

**Operating System:** This indicates the operating system deployed on the smartphone. The Android operating system is upgraded at regular time intervals. We make use of the operating system 2.3 also known as Gingerbread. The operating system encompasses all the libraries, framework and APIs required for integration of applications with the hardware.

**Firmware:** The firmware is a set of hard-coded libraries that are unique to the hardware device. These are required to control the hardware components and interface them to the operating system. These libraries are developed by the organizations that manufacture the device, example HTC, ACER.

**Rooting:** Rooting is the process by which users/applications can gain access to privileged settings and APIs, allowing the user/appplication to bypass standard security checks in place.

[6] The smartphone used was Nexus S co-developed by Google and Samsung.

[7] The process of installing an operating system onto the device.

**Root Access:** Denotes the highest level of access rights an application/user can attain. By having root access, an application/user can traverse through parts of the Android system, that is not possible otherwise.

**Application Programming Interface (API):** An API is a set of coded rules and specifications that applications can use to communicate with each other or parts of the Android system.

## 3 Related Work

Academic research on the Android operating system and its security framework is covered in depth by Shabtai et al. [4]. The team explains the different components that make up the operating system and how security mechanisms are incorporated within each of these components. Possible attack vectors and their mitigation is also addressed. Although the paper gives a detailed account of different mechanisms governing the security and any problems it can pose, the focus point remains the application level and the framework it comes in direct contact with. Our research differs as we follow a bottom-up approach targeting the Linux kernel and moving up towards the application layer.

Other research appearing simultaneously in the IEEE Security and Privacy issue of Jan-Feb 2009 by Enck et al. [5] gives a thorough security architecture of applications running on the Android platform. The authors discuss how applications network, exchange data and interact through components. The paper also gives an account of protected APIs, permissions used by content providers, broadcast and intent providers. The process of sharing data, permissions and security mechanisms governing this process is also defined in great depth here. This paper provides a base in understanding the basic functionality of applications, their interaction with the environment and applications. A basic working of permissions for applications, how they are invoked and implemented is also given in the paper presented by Shin et al. [6].

Coupled with information about the internals of the Dalvik VM obtained from the intricate blog by Ehringer [7], helped gain insight into the internals of the life cycle of an Android application. Major technical breakthroughs have been conducted by individuals that have contributed to the success of our experiments and these include Thomas Cannon, who has implemented attacks related to Android using the Android Market [8][9], reverse engineering [10] to obtain important information and bypassing security locks on an Android based smartphone [11]. However, the methods and tools employed during any research on an Android based smartphone assume a clean off-the-shelf operating system, which can be dreadful if the firmware has been modified or has security holes.

The Internet is flooded with modified Android operating systems and each of them differs in something from the further ones. Unlike the Android OS provided by Google most of its vendors do not reveal their source code and are thus distinguishable from the original Android OS. We took this idea a step further by retaining the original design and functionality of the original operating system, but extracting more privileges, so that a formal comparison shows no difference.

In the following sections we provide a detailed description and analysis of how we achieved this.

## 4 Technical Aspects

The Android operating system is a software stack for mobile operating systems developed in the Java programming language. Android also ships with a set of core developer tools written in native C/C++. The notable element is that this entire source from core Google-developed applications to specialized developer tools are open to general public.

The complete Android source is divided into multitudes of layers, and we make use of the following ones:

- Core applications that the system is shipped with, for example Google Maps, e-mail, calender.
- Developer tools to assist developers, for example monkeyrunner[8], ADB, Logcat[9], sqlite3 among others. [12]
- Libraries needed for interaction between hardware layer and software layer example media libraries for audio-video support, 3D libraries for graphics, among others.

We use these sections of the source code to inject our application, alongside the core applications and extract vital information from the user using the tools and system libraries.

### 4.1 PIDs and UIDs

The Android environment abstracts application permissions from the Linux kernel it is running on and divides applications depending on permissions assigned. Android applications are primarily written in Java and compiled into a custom byte-code (DEX)[10]. Each application is run in a separate virtual machine and having a unique userID (UID) [13].

Inherited from the Linux kernel underneath, files are divided into either application files or system files. System files are usually owned by the root user or the system user whereas application files are owned by application-specific users [14]. Unlike the Linux desktop environment, every application is run by a different user, if not explicitly defined differently, and this provides an extra level of security where applications cannot interact with each other. The possibility of gaining access to components or public information of other applications is only possible through specialized containers called content providers. Permissions to these content providers should also be defined during development process of

[8] Provides an API for writing program to control emulators.
[9] The Android logging system collects and views system debug output.
[10] The Dalvik VM executes `.DEX` files, which are converted from compiled Java-bytecode.

the application. Fig. 3 shows how content providers and applications interact to handle data.

In a typical Android environment the number of users usually equals the number of applications. Other than these, system daemons, processes and privileged applications are divided among root user, system user and radio user.

Android daemons running as root include `init`, `mountd`, `debuggerd`, `zygote`, and `installd` [14]. System and radio level user IDs are hardcoded into the kernel to provide additional privileges. Radios are usually used to connect applications to specialized hardware like bluetooth, wireless, broadcast receivers, GSM among others.

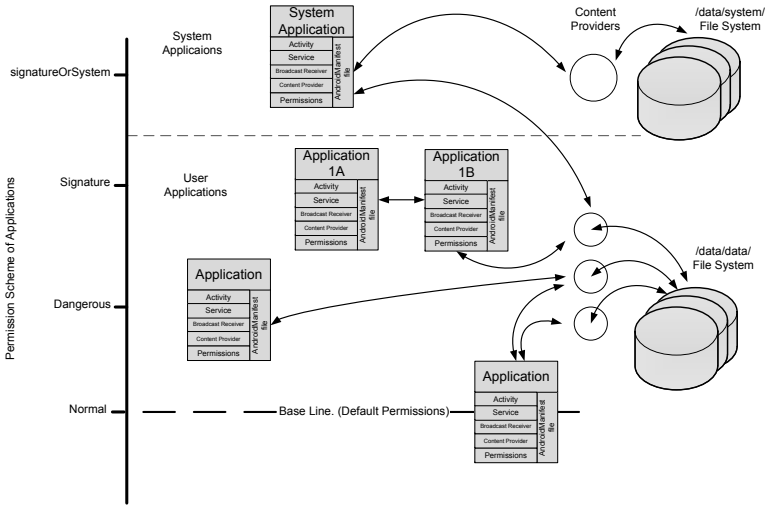


**Fig. 3.** Distribution of permissions among applications.

### 4.2 File Level Permissions

As described in ample detail by Enck et al. [5], system files and application files are stored separately and are accessible by their respective applications. System applications along with their data are stored under the path `/data/system`, whereas default applications would go under `/data/data` [5]. This can also be seen from Fig. 3.

### 4.3 Application Permissions

Inheriting from the Linux environment, permissions are assigned to files in tuples. Another permission scheme followed by Android is for applications where applications are assigned degrees of security ranging from normal to system level

permissions [15] as depicted in Fig. 3. Also built into the Android development environment are system level APIs which are used to change system and security settings. These APIs are only accessible to applications that have required permissions of `signatureOrSystem`.

Android also protects sensitive information like call logs, contacts through specialized containers known as content providers. A content provider may want to protect itself with read and write permissions, and permissions to use these content providers are declared in the manifest file during application development [15]. All the permissions and providers the application will use are provided to the user at install time.

### 4.4 Initiating Permission

When the Android system boots, it sets permissions for files, critical settings and defines which core applications and daemons are running at which security level. Of particular importance are the following two files that are used during compilation of the source code:

1. File: *main.mk*
   File path: `android/build/core/main.mk`
   The main make file contains all the different properties that are assigned to different builds of the Android system, which subdirectories to create and any tags assigned to modules.

2. File: *init.rc*
   File path: `android/system/core/rootdir/init.rc`
   This is the top level initialization file that ends up being executed by the init process at boot time. The file sets up the environment, creates mount-points, directories, assigns read and write permissions and composes the basic file structure. Alongside it also tells the system, which properties to set at boot time, which ports to open or block, and lists the daemon processes that should be running.

We make use of both files to create a user build that runs on a standard smartphone with escalated privileges for the ADB daemon. The `init.rc` file builds on the output of the make file and so it is necessary to change properties to suit to our needs. The following properties are of great importance:

1. `persist.service.adb.enable`: This property determines whether the ADB daemon is persistently enabled to listen for open connections or not.
2. `ro.debuggable`: This property determines whether the device is debuggable or not. If the device is not debuggable, dissecting applications will not be possible. It also helps in acquiring the status of executing applications at runtime.
3. `ro.secure`: This property determines whether the smartphone can run root commands or not. By default all builds have shell access. Privileges can be escalated by issuing the `su` command.

   All these properties are represented by a single bit: `0` or `1`.

## 5 Android Debug Bridge

The Android software development kit (SDK) is used for developing applications in conjunction with Java. Google also provides a set of tools to aid development and debugging of applications. One such tool is the Android Debug Bridge (ADB). ADB forms the basis of our attack vector as it suitably covers all aspects of deploying and extracting information from the smartphone. It comprises three blocks:

- ADB Daemon: The ADB daemon is part of the Android system on every smartphone and runs as a background process.
- ADB Server: Running on the development machine, the ADB server handles communication between the daemon and the client.
- ADB Client: The client runs on the development machine as well and is invoked by shell commands. It connects to the ADB daemon through standard communication ports of the Universal Serial Bus (USB) and the Transmission Control Protocol (TCP)[11].

The Android debug bridge was designed to aid developers in debugging applications directly on the device. It has all the advanced functions that a developer would require allowing the user to push and pull data, install and run applications, and execute scripts. We make use of these functions to carry out our attacks. The subsequent sections describe in detail the scenario and phases of our compromise.

## 6 Rooting and Application Design

For gaining root access on the operating system the following lines were added to the `main.mk` file mentioned in Section 4.4:

**Listing 1.1.** Code extract for gaining root access.

```
ADDITIONAL_DEFAULT_PROPERTIES += ro.secure=0
ADDITIONAL_DEFAULT_PROPERTIES += ro.debuggable=1
ADDITIONAL_DEFAULT_PROPERTIES += persist.service.adb.enable=1
...
tags_to_install += user
tags_to_install += debug
```

The first three lines add properties to the system allowing it to run root commands, the device to be debuggable and the ADB service to be started on boot. In addition to modifications of these parameters, an Android application is developed and included into the source by installing it when the operating system is pushed onto the device. This is done by including the application contents under the install path parallel to the other generic applications.

[11] Android explicitly uses the USB port for connecting to devices. The protocol can also be routed over the TCP but only through privileged commands.

### 6.1 Application Design

The application was designed, keeping in mind the requirements of a typical malicious application, where there is no interaction with the user, is stealthy in nature, does not have a fingerprint and has minimum processing load (memory and battery).

The application implements two broadcast receivers, one for receiving notification of a completed reboot process and another for delivering notification of wireless access point availability. This can be seen from Fig. 4, depicting our application. The application is also designed to extensively make use of the wireless API, included as part of the Android SDK, to scan for wireless networks, connect and bind to them.

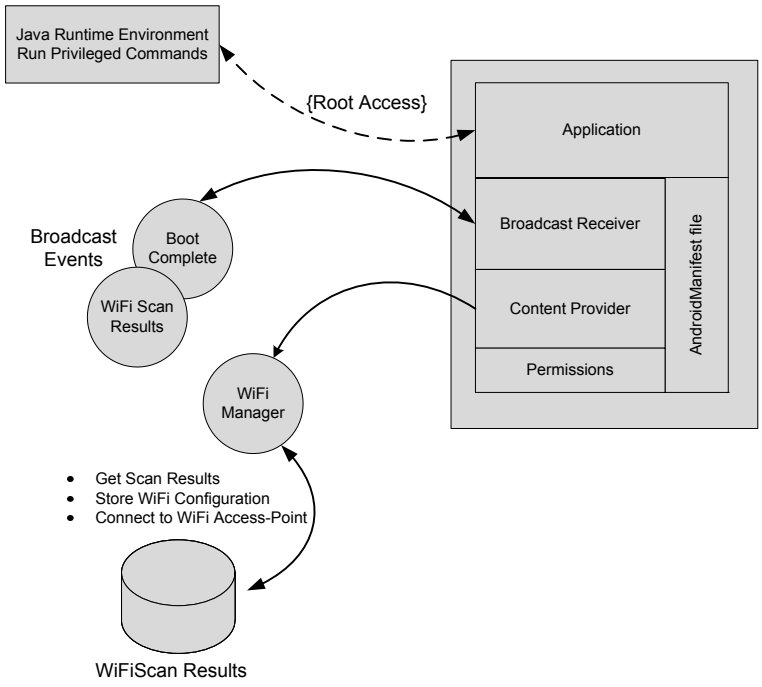


**Fig. 4.** Design of an application to extract information from an Android smartphone.

Our application is designed to not have a user interface and therefore no notifications are generated. This also ensures that there is no other direct communication with the user.

The application uses the Java Runtime Environment to pass commands to the Linux kernel. Using the Linux kernel underneath the application reroutes the ADB daemon to communicate over the TCP link. In normal cases, an application would not be allowed to do so because of the added restrictions from

the operating system. The complete application and all its modules are run as threads and therefore they consume considerably less processing power and have minimal footprint.

## 7 Software Distribution and Attack Scenario

The exploit begins by distribution of the rooted operating system. This distribution can be done through the Internet, spoofing a person to download an updated version of the operating system. Another method of distribution can be through a dedicated software used for flashing ROMs. The scenario developed for carrying out the attack was a simple one as shown in Fig. 5. Since there is no visual difference between the original operating system and the rooted one provided by us, we assume that a typical non-technical user cannot notice any discrepancies. If the ROM is distributed through traditional clients this assumption holds even stronger. The following steps highlight the attack process:

1. The user updates the smartphone with custom ROM off the internet or through a dedicated application like Samsung Kies[12].
2. The device boots up with the rooted ROM under normal procedures.
3. The rogue application runs in the background, waiting for a broadcast signaling successful boot operation.
4. The application starts scanning for wireless access points.
5. It sorts the retrieved access points into non-authenticated networks and authenticated ones.
6. The application traverses through open networks trying to connect to them.
7. To mitigate use of web authentication, we use the application to poll for a random public address.
8. On confirmation of successful connection to the network, the application runs root commands to forward the ADB daemon onto TCP port 5555.
9. The attacker present on the same network scans the network for an Android device using for example Nmap[13].
10. The attacker connects to the device using the ADB client (Command: adb connect <ipaddress>)
11. The attacker carries out attacks to compromise confidentiality, availability or integrity of the system.

### 7.1 Internals of the Scenario

Internally there is a lot happening on the application and middle layers once an Android system starts up. Fig. 6 shows the typical behavior of an application if it does not have escalated privileges. All access to public and private information is

[12] Client-side software from Samsung for connecting to smartphones. `http://www.samsung.com/ca/kies`

[13] A widely used network scanning tool. `http://www.nmap.org`

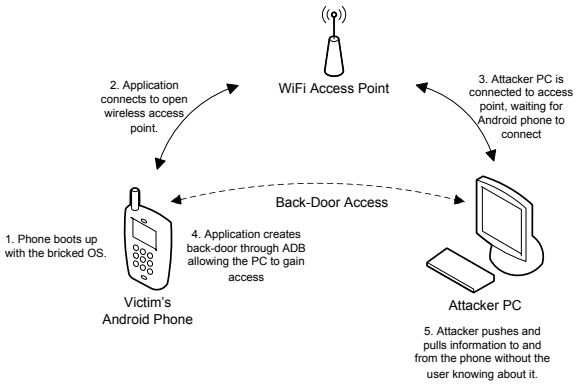


**Fig. 5.** A visual representation of the attack scenario.

controlled through permissions. The AndroidManifest file contains permissions to content providers and broadcast receivers, the application requires to get access to. Accordingly, it also contains permission levels from either normal level application to system level application.

The Android system assigns all due resources to the application only after confirmation from the user at install time. These resources are never verified again during the run time of the application. Fig. 6 visualizes the application functions in a sand-boxed environment controlled by the Dalvik VM. The virtual machine coordinates between the application and the kernel.

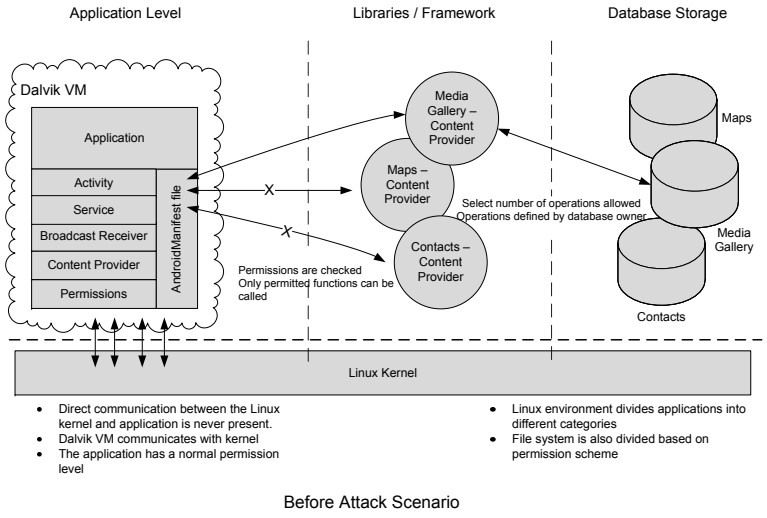


**Fig. 6.** Normal communication flow of an Android application.

Once we have injected our modified operating system, the Android system overlooks certain permission schemes in the AndroidManifest file and assigns escalated privileges to the applandroidsecassessmentication. Once the application has gained these privileges it uses the Java Runtime Environment to issue commands at the kernel level. This is depicted in Fig. 7.

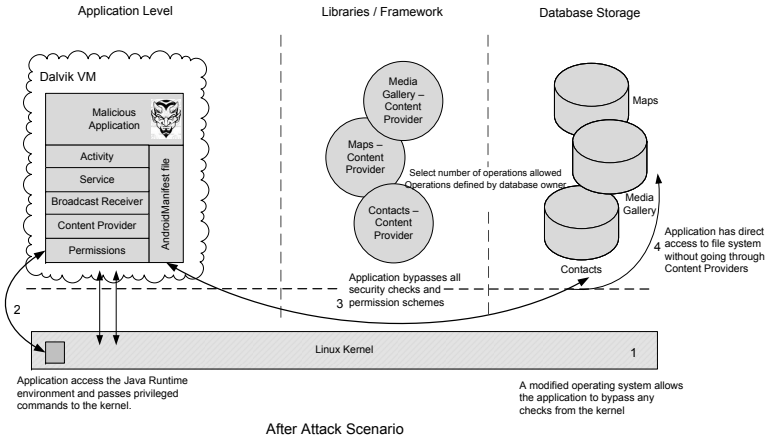


**Fig. 7.** Communication flow after privilege escalation took place.

The application is therefore allowed to create a tunnel to the data storage and gets unhindered access to public and private data. Commands issued at the kernel level change parameters of the file system. As discussed in Section 4.2, system files and data files are stored separately. By escalating privileges of the application from a lowly 'normal' to the highest 'system', the application can access files with no restrictions.

The application allows a wireless access to consequently make use of the Android system tool ADB to pull and push data, install applications and run scripts at will.

## 8 Results

As a result of connections to the Android device, we were able to perform operations that are not usually possible over a normal off-the-shelf Android based smartphone. We conducted a number of experiments to determine an effective security breach:

**Use-Case 1** Pulling data from the device

We were able to successfully pull data from the device without notifications to the smartphone user. In order to succeed we used the traditional ADB pull command. We extracted important information with respect to call records

made from the phone. The information is stored in an SQLite Database[14] and a simple open source SQLite GUI Browser[15] was used to read the information.
File extracted: contacts2.db
Directory: `/data/data/com.android.providers.contacts/databases`
Command: adb pull `/data/data/com.android.providers.contacts/databases/contacts2.db /hdd08/Desktop/Extracted/`

**Use-Case 2** Modification of information
Extracted databases can be easily changed and used to replace originals on the device. We were able to add records to the call logs, modify existing ones and delete records at will. This modified database was pushed onto the device replacing the original one.
File modified: contacts2.db
Directory: `/data/data/com.android.providers.contacts/databases`
Command: adb push `/hdd08/Desktop/Extracted/contacts2.db /data/data/com.android.providers.contacts/databases/contacts2.db`

**Use-Case 3** Pushing data to the device
As an example for pushing information onto the device, we were able to push a picture to the memory card and view it later on the device. It should be noted that this file could as well be a script, a Trojan horse or a reconnaissance application.
File: cased-logo.png
Directory: `/sdcard`
Command: adb push `/hdd08/Desktop/cased-logo.png /sdard/`

**Use-Case 4** Installation of an application on the device
The ADB is a rich tool and allows most of the operations that a debugging tool would also allow. Using native commands of ADB we were able to successfully install Android applications onto the device. The troubling part of this exercise is that there is no notification to the user if such a process has occurred. Any application installed using ADB also circumvents any permission rules, and can therefore have high level privileges to monetary applications like SMS and phone dialer without the user knowing about it.
Application: Angry-Birds.apk
Command: adb install Angry-Birds.apk

A thorough study of the ADB tool gave an insight of other manipulations that can be achieved using the ADB on an Android based smartphone:

– Logcat: We were able to monitor behavior of applications using the debugger. A complete device log can be seen with all protocol initiations, all clear text

[14] SQLite is a software library that implements an SQL database engine. http://www.sqlite.org/
[15] http://sqlitebrowser.sourceforge.net/

communications that take place between modules and lower level hardware interaction of applications.
- Reboot: We were able to successfully reboot the device without knowledge or confirmation of the user.
- Uninstall Applications: We were also able to uninstall applications once the required task is achieved. This process does not show any indication to the user of any operation being carried out.
- Device details: We were able to pull information about the device like serial number and product name.
- Wiping off complete data partitions from a remote desktop while the device is connected over a wireless link is also possible by running a single command (`adb -w` or `adb -d`).

## 9 Analysis

From our analysis we find that critical user information stored on the device is not encrypted, for example call logs and contact details. Application developers can, however, encrypt any data that is stored in databases. Depending on the hash algorithms used, data loss could be prevented.

No notifications are generated once the device is polled for files, or in the case an application is installed on the device. Indicators about anything malicious happening on the device are not given, unless a strong malware detection system is in place as shown by Shabtai et al. in [14] and [16]. Highly privileged commands such as wiping data, installing and executing scripts is also allowed once connected to the device through the ADB protocol.

Since the attacks are carried out over a wireless link, the time required to carry out the attacks are only subject to how close the device is to the access point and the data rate achieved over the wireless link.

## 10 Conclusion

In this paper we presented a bottom-up approach in compromising an Android system by changing build parameters of the original Android operating system. We devised an Android application to harness resources from the system that a normal application would not have access to. These extra privileges allowed us to create a back-door into the operating system which can be accessed over a wireless link. Piggy-backing on a root access provided by the back-door, we were able to extract, modify and use sensitive information pertaining to the user and the environment. This clearly shows how dangerous a back-door can be, where access to the file system is authenticated only during install time.

Countering such an attack can be difficult since there is no indication to the user that an operation has taken place. Once integrity and confidentiality of information on the device has been compromised, it is imperative that stricter rules for modifying system parameters have to be in place.

## References


1. Elmer-DeWitt, P.: Needham: Android's Market Share Peaked in March. http://tech.fortune.cnn.com/2011/06/21/needham-androids-market-share-peaked-in-march/ (June 2011) (Cited: July 01, 2011).
2. Google Android: What is Android? http://developer.android.com/index.html (2011) (Cited: June 22, 2011).
3. BBC News: Android Hit By Rogue App Malware. http://www.bbc.co.uk/news/technology-12633923 (March 2011) (Cited: May 18, 2011).
4. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google Android: A Comprehensive Security Assessment. IEEE Security & Privacy **8** (March 2010) 35–44
5. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. IEEE Security & Privacy **7** (January 2009) 50–57
6. Shin, W., Kwak, S., Kiyomoto, S., Fukushima, K., Tanaka, T.: A Small but Non-negligible Flaw in the Android Permission Scheme. In: Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2010), Fairfax, VA, USA, IEEE Computer Society (July 2010) 107–110
7. Erhinger, D.: The Dalvik Virtual Machine Architecture. Technical report (March 2010) (Cited: July 02, 2011).
8. Cannon, T.: Android Market Security. http://thomascannon.net/blog/2011/02/android-market-security/ (February 2011) (Cited: June 01, 2011).
9. Cannon, T.: Android Data Stealing Vulnerability. http://thomascannon.net/blog/2010/11/android-data-stealing-vulnerability/ (November 2010) (Cited: June 01, 2011).
10. Cannon, T.: Android Reverse Engineering. http://thomascannon.net/projects/android-reversing/ (November 2010) (Cited: June 01, 2011).
11. Cannon, T.: Android Lock Screen Bypass. http://thomascannon.net/blog/2011/02/android-lock-screen-bypass/ (February 2011) (Cited: June 01, 2011).
12. Google Android: Tools. http://developer.android.com/guide/developing/tools/index.html (2011) (Cited: June 22, 2011).
13. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically Rich Application-centric Security in Android. In: Proceedings of the $25^{th}$ Annual Computer Security Applications Conference (ACSAC 2009), Honolulu, HI, USA, IEEE Computer Society (December 2009) 340–349
14. Shabtai, A., Fledel, Y., Elovici, Y.: Securing Android-Powered Mobile Devices Using SELinux. IEEE Security & Privacy **8** (May 2010) 36–44
15. Google Android: Security and Permissions. Online http://developer.android.com/guide/topics/security/security.html (2011) (Cited: June 22, 2011).
16. Shabtai, A.: Malware Detection on Mobile Devices. In: Proceedings of the $11^{th}$ International Conference on Mobile Data Management (MDM 2010), Kanas City, MO, USA, IEEE Computer Society (May 2010) 289–290