2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications

Messing with Android's Permission Model

André Egners, Ulrike Meyer
Research Group IT Security
RWTH Aachen University
Aachen, Germany
Email: {surname}@umic.rwth-aachen.de

Björn Marschollek
Chair of Medical Engineering
RWTH Aachen University
Aachen, Germany
Email: bjoern.marschollek@rwth-aachen.de

***Abstract*—Permission models have become very common on smartphone operating systems to control the rights granted to installed third party applications *(apps)*. Prior to installing an app, the user is typically presented with a dialog box showing the permissions requested by the app. The user has to decide either to accept all of the requested permissions, or choose not to proceed with the installation. Most regular users are not able to fully grasp which set of permissions granted to the application is potentially harmful. In addition to the knowledge gap between user and application programmer, the missing granularity and alterability of most permission model implementations help an attacker to circumvent the permission model. In this paper we focus on the permission model of Google's Android platform. We detail the permission model, and present a selection of attacks that can be composed to fully compromise a user's device using inconspicuously looking applications requesting non-suspicious permissions.**

***Index Terms*—Android, Permission, Attack, Smartphone**

I. Introduction

The mobile phone market today performs very well. In many countries, especially in Western Europe and North America, the number of cell phone subscriptions exceeds the population count. Moreover, the number of landlines is decreasing, while the mobile phone sales steadily rose over the past years. According to the Gartner market research firm, handset sales in 2010 have increased by 31.8 percent compared to the year before [1]. With an increase of 72 percent, the smartphone market grew fastest and is increasingly dominated by the Android operating system.

The remarkable history of *Android* starts in 2005, when Google acquired the 2003-founded startup Android Inc. Until then, only little was known about the young organization's work, whose main business was developing software for mobile handsets. In 2007, Google announced the *Open Handset Alliance*, a consortium of firms which aim to develop open standards for mobile devices. Among those firms are handset manufacturers, mobile operators, and software and semiconductor companies. The Open Handset Alliance at the same time announced the development of Android, which features a complete software platform for mobile handsets including an operating system, middleware and key mobile applications.

As the other current smartphone operating systems (e.g., Microsoft's Windows Phone 7, Apple's iOS), Android allows for the installation of third-party applications by the user. A permission model is used to control the rights granted to installed third party applications (apps). Prior to installing an app, the user is presented with a dialog box showing the permissions required for the app to function fully-featured. The user has to decide either to accept all of the requested permissions or choose not to proceed with the installation.

In this paper, we discuss the permission model chosen by the Android smartphone operating system and present a selection of attacks against it. We show how these attacks can be composed to silently root the targeted smartphone. In particular, we show how to mount a UI takeover, how to start (malicious) applications after installation without knowledge of the user, how to start (malicious) applications after boot, and how to obtain a two-way Internet communication for an application that did not request the permission to access the Internet. Putting these attacks together we show how the device can be silently rooted after successfully establishing an Internet connection by downloading a root exploit and running it against the system.

After covering related work on Android security and on attacking Android's permission model in particular, we briefly introduce the Android operating system. We then detail Android's permission model. In Section V we present the core part of this paper: our novel attacks against the permission model.

II. Related Work

The security of the Android platform has attracted the attention of a broad range of security researchers, not only from academia, covering different aspects of Android's security model. For instance, the GTalkSevice connection of Android devices [2] and vulnerabilities in the newly started web-based Android Market [3] were inspected. [4] focuses on inter-process communication, while [5] inspects address space layout randomization (ASLR) on mobile devices. A generally valuable overview of current mobile malware in the wild was presented by Felt et al. [6] in 2011.

Besides the comprehensive Android developer documents [7], [8], various other articles on the web [9], [10], [11], [12], [13], [14], [15], [16] are discussing a broad range of security issues of the Android operating system. Also studies with a general focus on Android exist, e.g., by Enck et al. [17], [18] and Shabtai et al. [19]. Other researchers explicitly focus on a single security mechanism, e.g., the construction of Android botnets [20], and inter-application communication [4].

978-0-7695-4745-9/12 $26.00 © 2012 IEEE
DOI 10.1109/TrustCom.2012.203

Other researchers explicitly focus on a single security mechanism, e.g., the permissions used by Android apps [21], [22], [23], [24], [25], [26]. The work of Vidas et al. [26], which discusses the permission model as part of a general description of the Android security model as a whole. The authors define a taxonomy on the historic attacks that have been observed for Android devices thus far. In addition, they discuss Android's patch-cycle and the resulting fragmentation in its different versions across the multitude of devices. The patch-cycle turns out to be one of the most important security issues of Android: even though vulnerabilities in the operating system are eventually patched, not all device will timely be updated, and some will not be updated at all. This is due to the fact that the responsibility for pushing updates to the devices resides at the device manufacturers.

The Android operating systems confines its 3rd party applications in a defacto sandbox using process IDs and POSIX file system permission. As a consequence, system level exploits must escalate the application's privileges in order to gain from exploiting an application. The work of Davi et al. [24] provides insight into privilege escalation attacks on the Android operating system. The authors show that both, benign applications exploited at runtime, as well as malicious applications are able to escalated their respectively given permissions. Their work presents an instantiation of a privilege escalation attack exploiting a known vulnerability of the Android Scripting Environment running on Android version 1.6.

A more general approach, also involving details on the permission model of the Android operating system, is taken by Höbarth and Mayrhofer [23]. In particular, the authors develop a framework for on-device privilege escalation exploits. They extend their framework such that arbitrary temporary root exploits can be used to gain permanent root permissions.

The research of Shin et al. [22] presents an interesting flaw of the permission model, which exploits the absence of naming rules for developer-defined permissions. Due to the absence of naming rules, two different permissions with the same name can be in use. Permissions that have once been granted to applications remain in the system, even after removal of the application that originally requested the permission. The authors show that this flaw can lead to the permission being overwritten by another application. Thus, a new application is falsely granted a permission which has been originally granted to another application that is not existent anymore.

In their work on inter-process communication in the Android operating system, Chin et al. [4] also discuss the so-called *broadcast theft*. They show that malicious applications can easily eavesdrop on public broadcast messages from other applications. These broadcasts are considered implicit intents, which are not protected by sufficiently strong permissions, i.e., *signatureORsystem* or *signature*.

A recent study of Felt et al. [21] presents findings on Android's permission model, as well as its usage among developers. The authors profiled 940 applications with respect to permission usage, with the goal to identify so called *permission over-usage*. Besides elaborating on the implementation of permission enforcement, their work also includes the tool *Stowaway* which allows mapping API calls of compiled applications to their respective permission.

Another study by Enck et al. [18] provides an empirical overview on Android application security and Android malware Another interesting study was proposed by Zhou et al. [27] in 2011, in which the authors attempt to detect malware in Android Markets - also by using permission based filtering.

The article by Enck et. al [17] from 2009 offers a general understanding of Android security. We thus refrain from a detailed discussion of the general security framework of Android, but rather recapitulate the permission model as it is the most relevant security feature for this present work.

III. Overview on Android

Android is free and in large parts open source software based on a 2.6 Linux kernel. It combines the benefits of Linux such as file access rights and efficient memory management, with Java's type-safety. The platform is highly customizable and therefore additionally attractive to handset manufacturers. Many manufacturers take the chance to ship their devices with their own standard user interface. This results in a very inconsistent user experience, even though the very same operating system version is used.

Figure 1 shows the four layers of the Android operating system. The monolithic Linux kernel resides on the lowest layer. It is responsible for process and memory management, handles device drivers and additionally provides the hardware abstraction layer to other parts of the system [7]. The kernel has been highly optimized to meet the requirements of a mobile device.

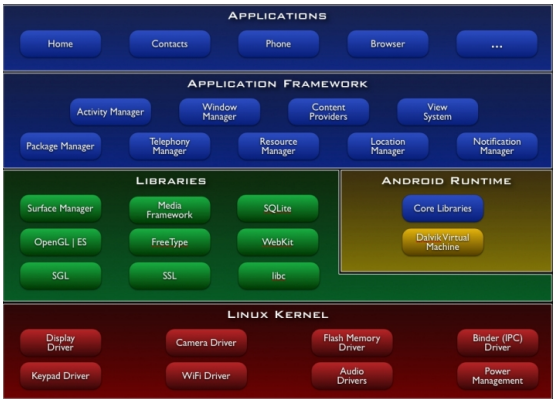

Fig. 1. Android's four-layer model [7]

The second layer of the hierarchy contains the native libraries of the Android operating system. The libraries are written in C/C++ and can be uses freely by any component of the remaining two upper layers. Included in the system are a custom C library (`libc`), media codecs to play a large variety of multimedia formats (e.g., MPEG4, H.264, MP3, AAC, and JPG), the native web rendering engine *WebKit*[1] and

[1]The WebKit Open Source Project: http://www.webkit.org/

2D/3D-graphics libraries, as well as SQLite for data storage purposes. The lack of the GNU C library `glibc` which is very widespread in the PC world, introduces some challenges running existing Linux applications on Android. Android uses a lightweight C library called `bionic`.

On the same level, next to the native libraries, the Android runtime is located. Its main component, the *Dalvik Virtual Machine*, is an optimized variant of a Java Virtual Machine. Dalvik programs, unlike pure Java programs, only use one `.dex` (Dalvik Executable) file to store all of the compiled source code. This is considered to be more efficient bytecode, and is better suited for the needs of resource constraint devices. The Dalvik Core Libraries are entirely written in Java and contain collection classes, utilities, and I/O libraries.

The application framework layer contains the Java frameworks which essentially represent the APIs that can be accessed by application developers. Among these frameworks are the *PackageManager*, which keeps track of the installed applications and their capabilities, the *LocationManager*, which provides the current location to applications, and the *View System*, which provides user interface elements and handles event dispatching and drawing.

Applications reside on the topmost layer and can use the underlying frameworks and libraries. Many applications are preinstalled and provide the main functions of the device such as telephony and SMS. The Telephony application enables the device to initiate and receive calls, the Contacts application allows to browse the address book and the home application takes care of the stand-by screen. All third-party applications run on this layer as well.

IV. Android's Permission Model

Android employs a very fine-grained application permission model. As of API version 11, 116 different permissions are predefined [8]. For an exhaustive discussion of the permission model we refer to the work of Felt et al. [21]. Some examples are shown below:

- `INTERNET` - allows accessing the Internet
- `RECEIVE_SMS` - for monitoring, recording, or processing incoming SMS
- `RECORD_AUDIO` - for recording audio messages
- `READ_FRAMEBUFFER` - directly reading the framebuffer (e.g., screenshots)
- `DUMP` - allows retrieving a state dump of system services

In addition to these predefined permissions, developers can define and add custom permissions, e.g, in order to control the access to their own applications by other applications. For example, an application using an SQLite database to store addresses, may request to allow other applications to access the datasets, but only if the user agrees. An application has to explicitly request every single permission it is going to use in its *Manifest* file. This should prevent applications to hide undesired activities. While it may sound reasonable for a Tetris game to request Internet access to maintain an online highscore list, it would be suspicious if it would also requested the permission to read the address book. Listing 1 shows a declaration in the manifest file (`AndroidManifest.xml`).

When an application component attempts to perform an action which is protected by a particular permission that the application does not have, the Android system raises a *SecurityException*, resulting in the action failing. Depending on the type of component, permission checks can by enforced by the application calls to protected functions in the system, e.g., when starting a new activity to disallow the start of third-party applications. At any time it is possible for developers to check whether the calling process has more fine-grained permissions, by calling the context's `checkCallingPermission()` method. Besides the developer, permission checks are spread throughout the Android API, which may also lead to redundant permission checks. Other permission can also be enforced by the underlying UNIX file system permissions [21].

```
<permission
android:name="com.myapp.permission.DEADLY_ACTIVITY"
android:label="@string/perm_deadlyAct"
android:description="@string/permdesc_deadlyAct"
android:permissionGroup=
"android.permission-group.COST_MONEY"
android:protectionLevel="dangerous"/>
```

Listing 1. Permission Declaration in Manifest

A. URI Permissions

Android employs so called *Content Providers*, which are responsible for storing and retrieving data, and additionally make the data accessible to all applications. These content providers are often protected by additional permissions. Applications may wish to pass a *URI* (Uniform Resource Identifier) to another application in order to be able to exchange data. For example, an email application usually protects its emails from being read by other applications using additionally defined permissions. However, a third-party image viewer may want to display an image attachment of a specific email. Following the principle of least privilege, a benign image viewer should not hold the permissions to read emails directly. In this case, the image viewer should rather be handed a URI to the data by using the `Intent.FLAG_GRANT_READ_URI_PERMISSION` flag set by the callee of the function. This enables the receiver, i.e., the image viewer, to read the data at the given URI. Thus, only the information which is absolutely necessary is passed from the email application to the viewer applications. Only a very small set of operations, e.g., playing sounds, can be executed without requesting a single permission.

B. Permission Protection Levels

The Android system defines four protection levels, level 0 to 3, for permissions.

- **Level-zero (0)** permissions are so called *normal* permissions. They pose a low-risk factor and typically only affect the application's scope. Examples are setting a timer or making the phone vibrate. Level-zero permissions are granted by the system automatically without explicit approval of the user. Optionally, the user can

request to be notified of the permission request prior to the installation of the application.

- **Level-one (1)** permissions are referred to as *dangerous* permissions. They are higher-risk permissions that, e.g., allow costly access to services such as initiating phone calls or access to the device's sensors, the Internet, or sensitive user data. An interesting permission on Level-one is the permission to read the device's log files (cf. Section IV-D). Prior to the installation, the package installer displays the set of requested dangerous permissions to the user, which decides to either grant or deny the set permissions (see Figure **??**). Only if the user gives his consent to *all* of the requested permissions, the application can successfully be installed.
- **Level-two (2)** permissions are only granted if the application that is being installed is signed with the private key corresponding to the same certificate as the application that originally defined this permission. These so called *signature* permissions can be used by developers, e.g., to share information between their own applications while preventing applications of other developers to gain access to this information. So even if the user would consent, a signature permission cannot be granted to applications signed with the private key corresponding to another certificate.
- **Level-three (3)** permissions can be granted by the system to applications that are contained in the system's image, or by applications that have been signed with the same certificate as the system image. Permissions of this highest category are reserved for handset manufacturers and OS developers. Representatives of this category are the permission to install new application (packages) or to change security settings.

1) Granularity: Even though Android employs a very fine-grained permission model, its suffers from a number of flaws. The most important one is that the user is only able to grant or deny all permissions at once. Granting or denying a particular permission (or a particular subset of the requested permissions) is not possible. Thus, the user is effectively forced into refraining from installing an application which might be useful, but requests too many or a suspicious set of permissions. Returning to the Tetris example, it would suffice to deny the access to the contacts, but grant Internet access to participate in online highscores. However, Android does not allow this differentiation. Also, the user can choose to grant or deny permissions only prior to installing the application. Permissions that have been granted once, can only be revoked by uninstalling the affected application. This is in fact a strong plea to the user's discipline. Users tend to try out applications while not paying too much attention to the permissions dialog.

C. Dalvik VMs & Permissions

Dalvik VMs are often confused with Android's sandboxing mechanism. Each application on Android runs in its own Dalvik VM. The VM does not prevent an application from escaping its VM. Android deliberately supports native code execution through the *Java Native Interface* (JNI). Every application is able to run native code *within* its current set of permissions via JNI. This can be beneficial for performance-critical procedures, e.g., those creating a heavy CPU load. In order for an application to call a native method from inside the Dalvik, the method must be declared using the keyword `native`. Additionally, the library containing the native method's implementation must be loaded prior to the actual function call. From a security perspective an application does not gain anything from leaving its virtual machine. Once it leaves it virtual machine, the application's rights are still enforced by means of the standard POSIX file and process permissions. This results in the applications only being able to operate within its prior given rights. In order to run privileged code, a privilege escalation exploit is still necessary.

D. Known Vulnerabilities

Known vulnerabilities of Android's permission model include vulnerabilities related to log permissions, FAT32 formatted SD cards, and WebKit or other browser engine. Note that this is not an exhaustive list, but rather a selection of the most prominent vulnerabilities that have been uncovered over the past years.

READ_LOGS Permission: For logging purposes, Android defines a permission called `READ_LOGS`. It enables applications to read the four virtual log devices that are supported by the Android system. The log devices reside at `/dev/log/` and are called `events`, `radio`, `main` and `system`. The event log is for diagnosis purposes, e.g. used by core system developers. The radio log contains radio stream data. The main log contains all log information from applications categorized into the five log levels, *Verbose, Debug, Information, Warning, Error*. The log can also be read via the Android Debug Bridge (adb) via USB, or directly from the device by executing the `logcat` shell command. Lineberry et al. [28] demonstrated how to gain equivalents for several permissions just by reading the system logs. For instance, it is possible to collect the list of recently used activities (originally protected by the `GET_TASKS` permission), since every intent that is used to start an activity is logged by the system. It is also possible to obtain the crash dump data of applications. This task normally requires the `DUMP` permission. However, all this data is also written to the log and can therefore also be obtained without this permission. The `READ_SMS` permission can be gained on system images with higher debug level and even the `ACCESS_COARSE_LOCATION` equivalent is also found in the logs. Up to Android 2.1, the cell ID and area ID of the cell phone tower were logged. This is sufficient to look up the exact location of the tower, and thus the user, in a database.

FAT32 formatted SD Cards: Many of the available Android devices allow to insert an SD card in order to increase the available storage of the device, e.g., for music files and pictures. For compatibility reasons, the SD card's memory of Android devices have to be formatted using the FAT32 file system [29]. This file system can be read and written by the three most popular desktop operating systems Linux,

Mac OS X and Windows. However, FAT32 does not support file system permissions, i.e., Android's standard sandboxing technique of POSIX file permissions does not apply to the SD card. Data on the SD card can thus be accessed and modified by every application. This is the most prominent reason why the permission to write the external storage is automatically requested without explicit declaration. The user has to approve this request upon installation. It is therefore advisable not to store any sensitive data on the SD card, as it can be entirely read by every application.

WebKit and Browser Engines: Android and Apple's iOS use the same basis for their browser software, namely the open-source WebKit rendering engine. WebKit has been initially branched by Apple from the open-source KHTML library. It powers current desktop browsers such as Google Chrome, Apple Safari, as well as the smartphone variants *MobileSafari* on iOS, and *Browser* on Android.

The browser is traditionally one of the easiest entry points for attacking a system. Users can be tricked into opening websites with malicious content exploiting vulnerabilities of the web browser engines. With the increasing popularity of URL shortening services or security services provided via SMS (e.g., mobile TANs), users are more likely to trust short URLs, despite the actual URL being obscured. The user can be tricked into opening malicious websites via social engineering or, e.g., from an application that an attacker distributes. Android permits opening websites (using the Browser) even if the calling application does not have the permission to access the Internet. This is the central vulnerability used in our attack to open a two-way Internet communication described in Section V.

The fact that WebKit is open-source, makes it easy to understand and it is indeed well-understood by developers as well as attackers. Consequently, there have been attacks exploiting vulnerabilities in the WebKit engine: Most recently, researches of the security enterprise CrowdStrike have demonstrated an elaborate attack exploiting a new WebKit vulnerability purchased from the gray market[2] demonstrating the severe effects of WebKit vulnerabilities. Cannon demonstrated a vulnerability in WebKit allowing *stealing data* from the device [30]. He was able to read all files that the browser is able to access. In this case, the attack was not overly serious for two reasons. First, the browser runs in a sandbox and can thus only access files within the scope of the application. However, as the Linux file system permissions do not apply on FAT32-formatted disks (cf. Section IV-D), reading the SD card could be done by an arbitrary application. In theory, attackers can upload the entire SD card to a chosen destination by tricking the user to open a website. In combination with the technique used in our attack (cf. Section V-E), i.e. only opening a browser window if the screen is off, data stealing can effectively be done covertly. In addition, the user may have to deal with increased cost due to excessive mobile data transfers. The attacker has to know the exact path of the file to be stolen. However, obtaining the location of interesting data of the target applications is no obstacle as the application's storage patterns can simply be researched by the attacker..

All Android versions up to 2.2 are affected by this vulnerability. Android 2.3 *Gingerbread* was supposed to fix this particular vulnerability. Unfortunately this has not been fully successful and proven by a similar attack [13]. The final fix has been distributed with Android version 2.3.4.

In November 2010, Keith released a WebKit exploit which allowed *remote code execution* [14]. Both Android and iOS were affected and opened a shell to an attacker-controlled IP address. However, the shell only ran within the application's sandbox and required an additional privilege escalation exploit to become dangerous. The vulnerability has been patched with Android 2.2 and iOS 4.1, respectively.

V. Attacks

In this section we detail an attack path that allows an attacker to silently root a user device. This attack path is composed of several smaller attacks based on different vulnerabilities of the permission model. While each of these smaller attacks already is a threat on its own, the composition of these attacks is devastating. As a basis for our attacks we assume that the user device is non-jailbroken and that applications installed by the user have limited permissions.

A. Attack Outline

In the beginning, our attacker takes over the UI (cf. Section V-B) in order to trick the user into installing a malicious application crafted by the attacker. In the next step the attacker ensures that his malicious application is directly started after it has been installed on the device (cf. Section V-C). In order to also be started after the phone has been rebooted, the attacker needs to make sure that the application is started directly after the completion of the boot process (cf. Section V-D). Ultimately, the attacker ensures that the application obtains a two-way Internet communication in order to be able to connect to, e.g., a dropzone to deliver user data, or to a command and control (C&C) server of a botnet (cf. Section V-E). Following the successfully established outside communication, the malicious application will be able to download exploits via this connection in order to silently root the user device, and thus fully compromise the system (cf. Section V-F).

B. UI Takeover

As mentioned in the previous section (cf. Section V-A), the first step of our attack targets the take over of the UI. This attack does not require the application to request any permission. By itself, the attack constitutes a denial-of-service attack which itself is harmless, but annoying. However, in our context, this typically harmless attack is the basis for an elaborate attack path. The idea of the UI takeover is to intercept all key presses and keep the current application in the foreground. Thus, the device is effectively blocked as there can only be one activity running at a time. We implemented this in the so called *KeyIntercepter* as a proof of concept.

[2] http://www.webcitation.org/65yxer3I3

The Android operating system allows applications to intercept all key presses for further processing. As soon as a key press occurs, the current foreground activity is queried whether it wants to handle the key press, or if the event should be forwarded to the next receiver in the queue. For this purpose, the activity's `onKeyDown()` method is called automatically. In this method, the application will handle the key press and finally indicates if the application did indeed handle the key press, or if it is supposed to be forwarded. If the method returns `false`, the press will be forwarded, otherwise it will be discarded. The proof of concept KeyIntercepter application uses this mechanism to intercept all key presses. However, it will just discard them and thus disables all keys by indicating to handle them, but doing nothing.

The `onKeyDown()` method is called for every key except the *Home* button. If the Home button is pressed, the system will immediately pause the frontmost activity and return the user to the home screen of Android. Holding down the Home button for a few seconds will show a list of recently used applications. In either case, the current activity will be paused, because the Home screen or another application comes to the foreground. As a result, the `KeyIntercepterActivity` of the KeyIntercepter will be notified by automatically calling the `onPause()` method. In this method, the application uses an intent (the very same it was originally started with) to restart the KeyIntercepter and finishes.

However, there is a challenging fact. The Android operating system delays the start of new activities for a time period of approximately five seconds after the Home key was pressed. This happens only if there was no user input that triggered the start of the application, i.e., the activity was started from code but not the user. Thus, the activity is not restarted immediately, but after a maximum delay of five seconds. Still, this suffices to be very annoying after a short time and comes close to a denial-of-service attack. In order to be able to start new activities right away, the application must hold the `STOP_APP_SWITCHES` permission. In Android versions up to 1.5, this was even more annoying, as the five-second delay had not been implemented.

In combination with the option to start at boot time without requesting permission (cf. Section V-D), this application can be used as ransom ware by e.g. prompting users to pay a small amount of money in order to make the application disappear. However, as the application is a third-party application, users can boot the device into Android's Safe Mode which does not allow any third-party applications to run. The application can then easily be removed.

Despite there is the chance to register a URL scheme on the other platforms and launch the browser application with this scheme and make it restart the original application, this behavior would immediately be discovered on the review process causing the application to be rejected. Moreover, applications are suspended as soon as the screen is turned off, which means that this attack will not work due to the lack of multitasking.

When installing an application from the Android Market the user can browse and gather the author and the price of the applications. If the user decides to install the application, he taps the price tag which is also the install button. The button turns into the *OK* button and the necessary permissions of the application will be shown. Thus, the same button is used to present the details of an application as well as to approve permissions. This behavior can be exploited by the UI takeover attack. As we have detailed, some components of applications may run in the background. Thus, using the KeyIntercepter, an attacker can either force the user into only being able to press the button which he chose, or block the UI for an amount of time. This can be experienced as the device reacting sluggishly With a high probability the user will tap more than once thinking his first tap has not been recognized or he barely missed the button. When the system finally becomes responsive again, it interprets the second tap and installs the application.

C. Starting Applications after Installation

After the attacker has successfully installed the application on the user device, the next step is to ensure that the application is started. Note that the attacker cannot rely on the user starting the application as the user has no desire to do so as he did not want to install the application in the first place. Also note that any attempt of the user to uninstall the application (while not using the safe-mode) can be prevented by the attacker using the UI takeover detailed in the previous section (cf. Section V-B).

```
<receiver
android:name="com.google.android.apps.analytics.
    AnalyticsReceiver"
android:exported="true">
<intent-filter>
<action android:name="com.android.vending.
    INSTALL_REFERRER"/>
</intent-filter>
</receiver>
```

Listing 2. Declaration of Broadcast Receiver in the AndroidManifest.xml

The Android operating system sends out broadcast *intents* in order to notify installed applications of events that have taken place. Examples for these events are the date change at midnight, the battery status getting below a threshold, or the fact that the system has finished booting. These broadcasts are a type of inter-process communication.

Typically, users expect to have to explicitly interact with the device in order to run applications for the very first time. Applications can be started automatically by using the Google Analytics SDK [31]. This framework allows tracking of referrers in the Android Market since versions Android 1.6 of the operating system. For this purpose, the Android system sends the intent `INSTALL_REFERRER` right after the installation of an application to this particular tracking application. If the attacker's application implements a corresponding receiver method, it will also receive the intent and handle the referral information sent to it. The way it was originally intended to be used is in combination with the Google Analytics SDK, which itself is supposed to deal with the information for the user

in the final application. The Android operating system documentation suggests the use of the Google Analytics broadcast receiver by integrating it in the way shown in Listing 2.

However, as stated before, the Google Analytics broadcast receiver is not the only component able to receive this broadcasts. It is also possible to implement a broadcast receiver to listen to the `INSTALL_REFERRER` intent and handle it, e.g., in an attackers' application. It is only necessary to replace the name of the Analytics broadcast receiver in the first line of Listing 2 with the custom name chosen by the attacker. Since Android version 2.2 the broadcast receiver is directly notified after the installation of an application. Therefore, as soon as the custom broadcast receiver is instantiated, in order to be notified of the referral, it can execute any code it desires to, and hence is able to start the main activity in one line of code.

Thereby, applications can immediately be started after their installation. Although this is not protected by a particular permission in the Android operating system, this behavior is most certainly not intended by the operating system developers. Since hardly any application starts automatically after installation, the user does not expect applications to do so. Thus, the user will most likely not correlate dialogues popping up immediately to the freshly installed application. For instances, a malicious application could use this attack to mimic the Google Market interface which is used to sign in, and start asking for the user credentials. Along with a message that announces the Market session having timed out and a new login is required. This could trick users into entering their credentials again.

D. Starting Applications at Boot

After the user has been successfully tricked into installing the attacker's application using UI takeover (cf. Section V-B), and the application is forced to be started directly after having been installed (cf. Section V-C), the next step is the following: The application needs to be running even after rebooting the device, i.e, it needs to be started after the device boots.

As mentioned earlier, the Android operating system allows to send out broadcast *intents* to other applications or their components. Concerning the boot sequence of devices, Android uses the `BOOT_COMPLETED` broadcast intent. This intent is sent after the devices successfully booted, and apps can register a broadcast receiver to react on this intent.

In order to prevent applications from illegitimately starting at system boot, Android introduces the permission `RECEIVE_BOOT_COMPLETED`. Applications that want to receive this particular broadcast intent must explicitly request the according permission to do so. However, enforcing a check for this particular permission has been forgotten in Android. Applications can successfully register to listen for the `BOOT_COMPLETED` intent without asking for the necessary permission. The user will not notice that this intent has been sent at all, unless he reads the log devices. A common user will most likely never read logs, most user will not even be aware of the log functionality, unless the user happens to be a developer.

E. E.T. Calling Home

Summarizing, up to this point the attacker has now managed to install his application, start the application after installation, and restart the application each time the device reboots. Next, in order to produce valuable output for the attacker, the application needs to establish bidirectional outside communication, e.g., to a specified dropzone delivering user data, or to a command and control (C& C) server of a botnet. In the attack described in the following, we will enable a two-way communication channel misusing another user-installed application with the required Internet permission.

The Android operating system restricts access to the Internet by forcing applications to request a permission called `INTERNET`. If the permission is granted, the requesting application may freely access the Internet without being restricted by the system in this regard anymore. Otherwise, any incoming or outgoing Internet connections are denied for this particular application. However, even without possessing the `INTERNET` permission, it is still possible to achieve a two-way Internet communication channel for an application. This particular issue is caused by the *transitivity* of Android's permission model.

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.
    parse("http://malicious-site.net")).
setFlags(Intent.FLAG_ACTIVITY_NEW_TASK));
```

Listing 3. Starting a browser instance for http://malicious-site.net

Applications are able to define their own custom permissions and protect their components by requiring other applications to hold particular permission if they want to use a protected component of this application. For instance, when the browser component of Android accesses the Internet, it requests the `INTERNET` permission. The browser is also intended to be launched by other applications that desire to display a website. However, using the browser is not protected by a separate permission, although it in fact enables Internet access. After firing the `VIEW` intent for a website URI (cf. Listing 3), the Android operating system will launch a new browser activity and load the requested page. This transitivity is not detected by Android. Despite not holding the `INTERNET` permission to access the Internet, the application which originally invoked the `VIEW` intent for the browser, is able to load a potentially malicious website inside the browser. Actually, the browser application would need to separately protect its components using the `INTERNET` permission as well. In particular, it would have to protect every component that uses a permission that the browser itself holds with this permission, as long as the system does not remove the implicit transitivity of application permissions.

Using this design flaw enables the attacker to construct outgoing communication. Data can be sent to any destination, e.g., using simple HTTP GET requests, for free by every application. It suffices to start a browser intent and load an appropriate URL, which can be done for free by every application as well.

However, in order to be able to receive incoming data as well, a custom URI scheme needs to be registered for the attacker's application. URI schemes are used to deliver data to applications, e.g., the `tel` scheme. By opening the URI `tel:+1234567890` in a browser, the Android operating system determines the application handling the `tel` scheme. By default, the Android telephony application is going to be launched and be given the telephone number +1234567890 pre-entered in its dialer component.

URI schemes that an application desires to handle must be defined in the application's manifest file. However, these schemes are never shown to the user. Once the application is installed and has been launched, it can launch a browser and have it start loading an attacker-controlled website. This website or script can evaluate data that has been sent from the user's device and return data by redirecting the browser to the custom URI scheme which the application listens to. This makes the application come to the foreground again and evaluate the data. For instance, the script could return an HTML document like in Listing 4. This redirects the browser to itsecscheme:itsec?data=secret, which launches the application that registered for `itsecscheme`.

A remaining challenge is that of the browser having to be in the foreground whilst loading the attacker's website. However, most of the time devices idle, e.g., in the user's pocket, and thus the screen is turned off. The state of the screen can be obtained for free from Android's *PowerManager*. The attacker can simply run a service in the background which constantly monitors the state of the phone's screen. Thus the attacker can make sure that the home screen is launched and thus the browser is hidden as soon as the screen is turned on again. In the meantime, the application (which does not need to use a user interface at all) can perform its malicious transactions. We successfully implemented a proof of concept exploiting this particular behavior and called it *SilentCommunicator*.

```
<html>
<head>
<meta http-equiv="Refresh" content="0;
url=itsecscheme:itsec?data=secret" />
</head>
<body>
redirecting...
</body>
</html>
```

Listing 4. Website redirecting to custom URI scheme

The *SilentCommunicator* application constantly checks the status of the screen. If the screen is turned off, the application is certain that it can send data using HTTP GET to the attacker controlled server and use the registered custom URI scheme to receive data without being spotted. Sending the data is realized by opening a browser window like in Listing 3, but with the URL adjusted to the address of a script on the attacker-controlled server and appending the data. The script is able to return data to the application using the custom URI scheme `itsecscheme`. Thereby, the application polls the server in regular intervals, e.g., the SilentCommunicator implementation polls every five seconds. The application's main activity (`SilentCommunicatorActivity`) only contains the `onCreate()` method which is called as soon as the activity has been created.

Two threads are running simultaneously. One constantly monitors the status of the screen in an infinite loop. This was outsourced into a new thread in order to not block the UI thread and sustain the responsiveness of the user interface. If the thread detects that the screen is off, it can start the transmission if none has been started yet. The transmission is also done in a new thread (`transmissionThread`), which must be stoppable from the application. If, however, the screen-supervising thread detects that the screen is on, it has to make sure that the browser gets hidden quickly, if it was visible before. The `transmissionThread` first delivers data to the script running on the attacker-controlled server using HTTP GET. In our implementation, it transmits the value `secret` for the variable `harvestedData` to the script `silentData.php` hosted at http://www.muffi.eu/itsec/silentData.php by opening the URL with variable/value pairs http://www.muffi.eu/itsec/silentData.php?harvestedData=secret.
The script evaluates the data and redirects the browser to the custom URI. The system checks which application wants to handle the `itsecscheme` scheme and finds the application that registered for it. This results in the `ReceivingActivity` coming to the front and retrieving the data. In our implementation, it is only going to log the data.

```
public class ReceivingActivity extends Activity {
  /*
  * This activity only logs the received data and
  * quits. This would be the place to handle
  * any incoming transmissions.
  */
 protected void onCreate(Bundle savedInstanceState)
 {
   Uri uri = Uri.parse(getIntent().toURI());
   super.onCreate(savedInstanceState);
   Log.e("Receiving Activity", "Received data: " +
       uri.getQuery());
   finish();
 }
}
```

Listing 5. Implementation of the ReceivingActivity

Using the mechanisms we have just detailed, downloading binary files from unprivileged applications is easily possible by opening a browser with the according URL. On the Android operating system, downloaded files are automatically stored to the SD card and a notification is displayed. While the file can be easily deleted from the external storage (because every application automatically requests the permission to write it), the notification cannot be removed without permission. Leaving it would attract the user's attention and the operation could probably not be done silently.

In order to circumvent this small inconvenience while transferring binary files, the Base64 encoding schemes can be used. These schemes can transfer binary data into ASCII representations which can then be sent back to the application using the URI scheme. Decoding, saving and, if necessary, merging the received strings will then be done by the attacker's

application. This opens up to the possibility to extend the application to periodically poll the server, transmit details on the phone and software and download corresponding payload. The server could use the operating system version to deliver the Base64-encoded variant of the current jailbreak for the running system version. This way, the application can ensure that the device it is running on is always rooted, and that the application is able to spawn a root shell, if a jailbreak exists.

This attack is serious and compromises the entire device security while requiring only little social engineering efforts. The attacker's application does not request any permission and would not make many users suspicious, e.g., if the rootkit functionality is hidden behind some useful routines or a neat game. Hidden like this, there is hardly any chance for the user to detect the compromise at all.

F. Silently Rooting Android

After the attacker has managed to installed the application, started the application after installation, started the application every time the device boots, and established a bidirectional outside communication channel, an optional goal can be to fully compromise the device, i.e., rooting (jailbreaking) it.

A very popular method for rooting Android is the *zimperlich*[3] jailbreak which is able to root devices running Android versions prior to 2.3. Our proof of concept application, *Installer*, uses a slightly modified version of the zimperlich jailbreak. While the original version creates a copy of the default shell `/system/bin/sh`, our version in Installer sets the setuid bit on the shell executable which is owned by `root`. That is, every instance of the shell runs with `root` privileges. Real world malicious developers would probably not use this particular technique, since it would enable users to easily uninstall the application. Instead, the root access would probably be restricted to the own application. For the sake of simplicity of this demonstration, setting the setuid bit suffices.

The resulting exploit binary must be compiled from `zygote.c` to run on Android devices. This requires the presence of a copy the complete Android source code that has been built successfully. Once the source code of the native jailbreak code is compiled, it must be named accordingly with a `lib` prefix and carry the file extension `.so`. The file must be placed in the `libs/armeabi` directory of the Android application project.

Our proof of concept implementation uses a single activity, the `JailbreakActivity`, featuring a single button which triggers the jailbreak by executing the native code in the runtime. As soon as the button is pressed, the application starts an infinite loop which executes the jailbreak binary over and over again until it succeeds. *Succeeding* refers to the Android operating system throwing an exception, due to the current user that belongs to the application exceeding its maximum amount of allowed processes. Recall, the `setuid()` calls of the `zygote` application which intended to change the owner of the newly forked process to the user that has been assigned to the calling application will fail. Thus, the new process will keep its `root` permissions that it inherited from the process it was forked by, the `zygote`. Now that the maximum number of allowed processes is reached, it only remains to have the `zygote` fork a new process. This means that there must be a new application component that is instantiated as soon as the exception is caught. We realized this using starting a new *ContentProvider* by the name of `RootProvider` that is queried. Upon its instantiation, RootProvider (now running as `root`) will the binary again. The binary detects that it runs as `root` user and sets the setuid bit of the current shell `sh`. In the `query()` method, Android's PackageManager command line tool is used to install the `installme.apk` package that has been copied to the SD card by the `JailbreakActivity`. The `root` user is allowed to install applications and the device owner will not notice the installation. Note that the PackageManager is also invoked in the legitimate application installation process after the approval of the requested permissions through the user. This means that the interface-less PackageManager will not ask again and just perform the installation process.

In our implementation, we install a dummy application (*Permissions*), which only requests the permissions to access the Internet and to initiate phone calls. The Installer demonstrates that it is possible to root the device and install a package requesting any permission completely without user consent or even notification. It is clear that this behavior can be hidden behind useful functionality and triggered silently. Recall that in the grand scheme of our attack, this could simply be embedded into the application used in Section V-B, or be downloaded by using the established bidirectional communication channel as shown in Section V-E.

With a `root` shell, it is already possible to read all desired data like contacts, SMS messages and device internal information. Installing additional applications makes it more comfortable to use Android's frameworks. If an attacker aims to steal data without an additional application, he can for instance open the SMS and MMS database, which is located at `/data/data/com.android.providers.telephony/databases/mmssms.db` and stored in SQLite format. The contacts database can be found at `/data/data/com.android.providers.contacts/databases/contacts2.db`.

Summarizing, a user's device can be silently jailbroken using the steps described in Section V-B, Section V-C, Section V-D, Section V-E, and finally Section V-F without requiring many or suspicious permission.

VI. Conclusion

In this paper we presented a selection of vulnerabilities associated to the application permission model of the Android operating system. We showed how these vulnerabilities can be used – either in combination or stand alone – by an attacker to mount various attacks. These attacks included tricking the user into installing an attacker crafted application, starting applications immediately after installation, as well as after

[3]CSkills: http://c-skills.blogspot.com/2011/02/zimperlich-sources.html

booting the device without possessing the necessary permissions to do so. Building upon an attacker's inconspicuous application deliberately installed by the user, we detailed an attack by which this application is able to cheat its way into obtaining a bidirectional communication channel to the Internet, without declaring the `INTERNET` permission. This bidirectional channel could, e.g., be used for communication with a botnet command and control server. This allows to download additional exploits that allow to silently root the device, leading to a full compromise of the system. Summarizing, it becomes obvious that the permission model does not allow to sufficiently secure a user against malicious applications. Even though applications are not possessing the necessary permission, they are able to take actions which should be prevented by the permission model. As such, elaborate attacks using inconspicuously looking applications requesting non-suspicious permissions can be used to fully compromise a user's device.

REFERENCES

[1] Gartner, Inc, "Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010," Gartner Press Releases, February 2011, Retrieved on 2011-05-19, archived at http://www.webcitation.org/5yn8K0GjA.

[2] J. Oberheide, "A Peek Inside the GTalkService Connection," June 2010, Retrieved on 2011-05-09, archived at http://www.webcitation.org/5yY0FxfjH.

[3] ——, "How I Almost Won Pwn2Own via XSS," http://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/, March 2011, Retrieved on 2011-05-09, archived at http://www.webcitation.org/5yXyUFBhy.

[4] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application Communication in Android," in *Proceedings of the 9th ACM MobiSys '11*.

[5] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM CCS '04*.

[6] 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 SPSM '11*.

[7] Android Developers, "What is Android?" Android Developers, http://developer.android.com/guide/basics/what-is-android.html, May 2011.

[8] "The Developer's Guide," http://developer.android.com/guide/, 2011, Retrieved on 2011-05-09.

[9] E. Schonfeld, "Touching The Android: It's No iPhone, But It's Close," TechCrunch, http://techcrunch.com/2008/09/23/touching-the-android-its-no-iphone-but-its-close/, September 2008, Retrieved on 2011-05-19, archived at http://www.webcitation.org/5ynWx29YF.

[10] J. Oberheide and Z. Lanier, "TEAM JOCH vs. Android: The Ultimate Showdown," Washington, DC, January 2011.

[11] J. Oberheide, "Remote Kill and Install on Google Android," June 2010, Retrieved on 2011-05-09, archived at http://www.webcitation.org/5yYCucC6N.

[12] K. Mahaffey, "Security Alert: DroidDream Malware Found in Official Android Market," March 2011, archived at http://www.webcitation.org/5yYBUjtUS.

[13] X. Jiang, "Android 2.3 (Gingerbread) Data Stealing Vulnerability," January 2011, archived at http://www.webcitation.org/5youAnT9n.

[14] D. Goodin, "Researcher outs Android Exploit Code," November 2011, archived at http://www.webcitation.org/5yoxrzjcq.

[15] A. Gingrich, "Malware Monster: DroidDream Is An Android Nightmare, And We've Got More Details," March 2011, archived at http://www.webcitation.org/5ynIuuEgO.

[16] ——, "The Mother Of All Android Malware Has Arrived: Stolen Apps Released To The Market That Root Your Phone, Steal Your Data, And Open Backdoor," March 2011, archived at http://www.webcitation.org/5ynIYoxLb.

[17] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *Security and Privacy*, 2009.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security, SEC'11*.

[19] S. et al., "Google Android: A Comprehensive Security Assessment," *Security and Privacy'10*.

[20] C. X. et al., "Andbot: towards advanced mobile botnets," in *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats, LEET'11*.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM CCS '11*.

[22] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "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 '10*.

[23] R. M. Sebastian Höbarth, "A framework for on-device privilege escalation exploit execution on Android," in *IWSSI'11*.

[24] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android," in *Proceedings of the 13th international Conference on Information Security, ISC'10*.

[25] "Permission re-delegation: Attacks and defenses," in *20th Usenix Security Symposium*, 2011.

[26] 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 Conference on Offensive Technologies, WOOT'11*.

[27] Y. Z. et al., "Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the 19th Network and Distributed System Security Symposium, NDSS '11*.

[28] A. Lineberry, D. L. Richardson, and T. Wyatt, "These aren't the Permissions you're Looking for," DEFCON 18, Las Vegas, NV, 2010.

[29] Microsoft Corporation, "Extensible Firmware Initiative FAT32 File System Specification, FAT: General Overview of On-Disk Format," 2006.

[30] T. Cannon, "Android Data Stealing Vulnerability," November 2010, archived at http://www.webcitation.org/5yorcAUc8.

[31] Google Inc., *Google Analytics SDK for Android*, 2011, http://code.google.com/intl/de/mobile/analytics/docs/android/. Retrieved on 2011-05-24.