University of Amsterdam

MSc System and Network Engineering

Research Project 1

Exfiltrating Data from Managed Profiles in Android for Work

Authors:
Tom Curran
Ruben de Vries

February 8, 2016

Abstract

In this paper we address whether it is possible to access data between managed and unmanaged profiles on Android devices enrolled on Mobiel Device Management platforms. To answer this question we analyse the implementation of data separation between profiles on a Nexus 7 (2013) tablet with root privileges. During the course of our research we were able to move access data due to lack of file-based encryption on the device, and provide a simple application for demonstration. Another attack scenario via the Binder IPC is also proposed. Finally, recommendations to mitigate such scenarios are given to help improve upon these issues.

Contents

1. Introduction

The demand for "Bring Your Own Device" (BYOD) schemes, where enterprise software is installed on employees' personal devices has grown significantly due to the increasing adoption of smartphones and tablets in the workforce, combined with productivity gains and cost savings[13]. As such, the risk of leaking confidential data stored on employees' devices is a serious concern. Android for Workaims to solve this problem by allowing Android users to operate managed and personal profiles that integrate seamlessly on the same device, whilst securely isolating the data stored under each profile.

Android for Workfrom Google is an Enterprise Mobility Management (EMM) framework that Mobile Device Management (MDM) providers can use to create solutions that allow businesses to remotely manage security policy, applications and content on employees' mobile devices. It is the combination of several features that have been gradually built into the Android operating system as it continues to evolve, offering administrators features such as the ability to set enterprise-wide security policy, wipe data remotely, separate enterprise from personal data and enable full-disk encryption by default.

Google strives to provide Android developers with the right tools to provide the best experience for their users by facilitating interoperability across applications. This feature however may be less desirable on an enterprise device that stores data that is sensitive and private by nature. Android for Workhandles this by allowing administrators to disable sharing content from the work profile to the home profile.

Enterprises have limited control over how employees use their devices in their personal life and mobile malware is becoming and more pervasive. Storing confidential information on a device with an uncertain exposure to attack represents a significant business risk for organisations. As such, ensuring that there can be no data exfiltrated is imperative.

Several vulnerabilities in Android's recent past have enabled malicious users to exfiltrate sensitive information using techniques such as privilege escalation to execute arbitrary code or intercept messages as they pass between running processes, [16][2].

Intuitively one would expect a trade-off when combining integration with strict data isolation between applications. Therefore, we hypothesise that there are avenues through which data leakage can occur between user profiles.

1.1 Problem Statement

Android is the most widely used mobile operating system holding a share of 82.8% of the total smartphone market whilst its largest competitor, Apple's iOS, holds 13.9% [12]. Contrast this to the Enterprise space where Android and iOS represents 26% and 72% of total device activations, respectively [19]. Such a significant difference between market share when moving from the personal to enterprise space indicates a lack of confidence in Android's ability to meet their needs, significant one being security.

Android's open-source nature has been a significant factor contributing to its widespread usage among mobile devices. Device manufacturers are able to produce and sell devices running Android for a low price to consumers, and developers are able to write applications for it using popular programming languages they are already familiar with such as Java and C++, facilitating an expansive app ecosystem.

Whilst making the code open source has contributed to its success, it has also opened the system up to many vulnerabilities, as anyone is able to study its source code for flaws. Furthermore, as Android is built upon the Linux kernel, vulnerabilities found in the Linux kernel will also affect millions of Android devices. Responding to vulnerabilities is a slower process for Android than other operating systems as device manufacturers modify the code to support their hardware platforms. Users are thus dependent on the speed with which vendors are able to offer updates, if the user even cares to update at all.

Many exploits in the past have aimed to root the device, permanently or temporarily, and thereby gain unrestricted access to the system and its components. Though a number of security-enhancing features have been introduced to mitigate rooting and prevent applications from accessing files without the user's explicit permission, a significant proportion of the Android community want to legitimately root their devices for greater control. Therefore, despite whatever controls are set in place to prevent rooted devices, there will always be groups trying to circumvent them. The existence of remote exploits such as Towelroot and Stagefright make it possible for devices to be rooted without the user's awareness and target components common to every device running the operating system [7, 4]. Such exploits have already been incorporated into malware as a means to obtain control over a victim's device, without their knowledge [14].

Android for Work was first introduced in version 5.0 and is able to support devices running version as far back 4.0 through the use of separate application. Remote exploits exist for devices up to 5.0 [7, 4]. Figure 1.1 below shows the distribution of versions that make up Android's ecosystem [1], highlighting that approximately 80% of devices today are vulnerable to such exploits.

The heterogeneous landscape makes it difficult for enterprises to adopt Android devices in their BYOD policies as they lose a degree of control and face uncertainty over what exactly is connected to their network. This increases a business' exposure to risks such as the leaking of confidential information or unprivileged access to company equipment, which could in turn incur financial loss.

MDM vendors all include root detection mechanisms as part of their solutions for companies, however they can eventually be thwarted. A systematic survey of the most popular MDM solutions available for Android devices revealed that all root checks thus far can be detected by either disassembling the applications

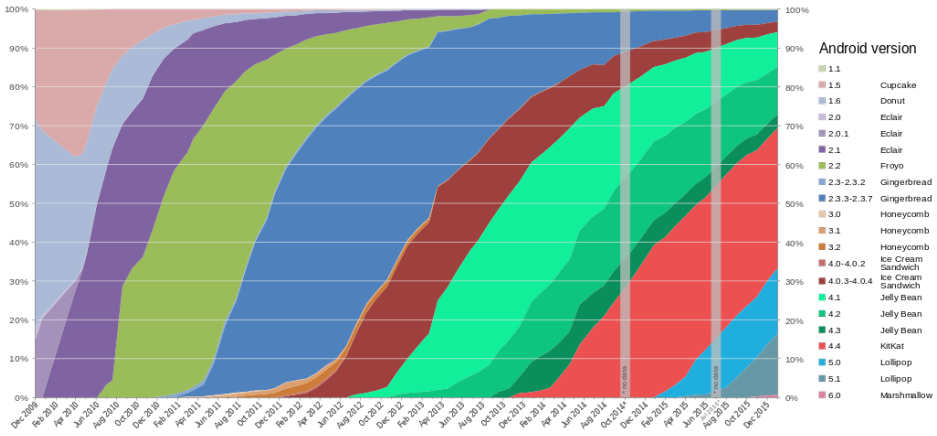

Figure 1.1: Android version distribution, collected over 7-day period ending on 4th January 2016

Source: Google, 2016

and viewing the source code or running it through a debugger and analysing what is loaded into the memory at runtime [6]. Once an attacker knows how root checks are implemented, it's possible to bypass them.

In light of this uncertain environment, and the growing demand for BYOD schemes among businesses, we deem it valuable to investigate how data is secured on devices using the Android for Work framework in presence of a rooted device. Given the history of exploits in the past, it is not unrealistic to assume that more will be discovered in the future.

An aspect of Android for Work of particular interest is the implementation of multiple user profiles that run concurrently on the device. Support for multiple users was first added to Android in version 4.2 [9], though it only permitted one user to be running on the device at any one time. Android for Work offers seamless integration between the two profiles, for example running at the same time, whilst also promising secure data isolation.

1.2 Research Question

The previous discussion lends itself to our main research question:

- Is it possible to exfiltrate information from a managed profile to a personal profile in Android for Work?

To answer this, the following sub-questions will be answered:

- Is it possible to intercept information belonging to the managed profile to a personal profile using the Binder IPC of Android?
- Is it possible to record the activities of the managed profile from an application initiated from the personal profile?

1.3 Structure

The remained of this paper is structured as follows. In the next chapter we discuss the related research on Android security. In Chapter 3 we provide the background necessary for the reader to understand the components of the Android system our research targets followed by an outline of our methodology in Chapter 4. Chapter 5 contains the bulk of our research as we discuss our findings regarding encryption and interprocess communication on the device. We then proceed to discuss the implications of our research in Chapter 6, offering suggestions for future work in Chapter 7. Finally, in Chapter 8 we summarise our work and conclude our findings.

2. Related work

We were inspired to conduct our research by studies that had successfully implemented them in previous years, [16] [2] [15]. At the time of writing, there is no academic literature that explores the mechanics of Android for Worknor whether data exfiltration is possible between concurrent user profiles.

Design flaws have been found in the implementation of multitasking, a key component enabling developers to offer rich user experiences left the system vulnerable to *task hijacking* attacks[16]. In this study the careful selection of activity attributes, intent flags, and call-back functions could allow attackers to hijack state transitions in applications and conduct stealthy spoofing and phishing attacks.

Ratazzi et al.[15] conducted a systematic study of multi-user support to identify critical places where access controls were not present or properly implemented and were able to describe attack scenarios where secondary users with limited privileges were able to spy on the primary user without his knowledge.

Finally, malware researchers presented at a Blackhat conference [2] how they were able to hook into the Binder service facilitates Inter Process Communication between apps and the kernel and successfully parse, utilise, and exfiltrate data as it passed through, demonstrating the efficacy of the technique in conducting attacks such as key logging, data grabbing, and SMS interception without any indication on the device.

3. Background

The security features of Android for Work make use of features already built into the Android OS. In this chapter we provide an overview of the Android architecture and then describe the components relevant to our paper.

3.1 Android Architecture

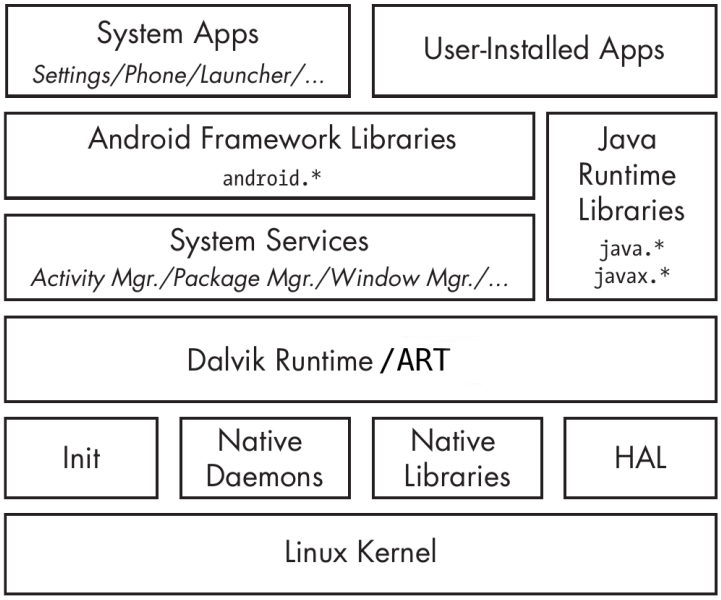

Figure 3.1: Android Architecture

Source: [5]

Android OS consists of four main layers of abstraction: applications, framework services and libraries, native libraries, and the Linux kernel. The kernel provides drivers for hardware, networking, file-system access, and process management. On top of the kernel sits the native userspace layer consisting of the *init* binary, native daemons, and native libraries that are used throughout the system. The hardware abstraction layer is a software layer that interacts drivers located inside the Linux kernel and is responsible for handling aspects such

as the camera, Wi-Fi, and Touch Panel. Native applications, daemons, and libraries can be run directly on the system without the full Java stack.

Android is mostly implemented in Java, and is executed by a Java Virtual Machine (JVM). Dalvik Virtual Machine (DVM) was Android's initial JVM implementation in which applications ran *Dalvik Executables* (DEX files). Java bytecode generated by the Java compiler is converted to Dalvik bytecode (*.dex* files). Since Android 2.2, Dalvik has used a just-in-time compiler [1].

A new runtime, the Android RunTime (ART) was introduced in Android 4.4 and has replaced Dalvik as the default runtime since 5.0. In contrast to Dalvik, ART features ahead-of-time (AOT) compilation. Compilation takes place when an application is installed on the device, process generates native code from the Dalvik bytecode stored in the application. Precompilation significantly improves the performance of applications in comparison to just-in-time compilation.

Android's core Java libraries sit on top of the runtime layer. Some have native code dependencies and will communicate with the native libraries lower down using the *Java Native Interface (JNI)*[5], which allows Java code to call native code and vice versa. The Java runtime libraries layer is directly accessed both from system services and applications.

3.1.1 Services

System services make up the core of Android and are responsible for implementing features such as display and touchscreen support, telephony, and network connectivity. Rather than interacting with kernel directly, applications call a remote interface defined by the relevant system service which talks to the kernel. Services can also be defined and implemented by application developers and are used to provide background functionality such as managing downloads in the background, or playing music whilst using another application.

Client applications communicate with services in a client-server format via interprocess communication (IPC). The programming interfaces used by both parties are defined in Android Interface Definition Language. AIDL files (*.aidl*) generate Java interface files at compilation time which define a client-side *Proxy* and server-side *Stub* interfaces.

3.1.2 The Binder IPC

Android uses kernel-level sandboxing to prevent applications from accessing the memory space of others. At installation time, applications are assigned a unique user ID that then run under. Whenever an application needs access to a resource it doesn't implement itself it must contact the relevant service using the Binder IPC at the core of the Android.

Since a process cannot access the memory space of another process, it must use the Binder interface exposed by the kernel. A client will send requests to the Binder driver located in the kernel, */dev/binder*, who then passes it on to the target server. A server, will hold a pool of Binder threads that wait and process requests as they are received from clients. Client applications interact with the Binder driver using the native `libbinder.so` library that is loaded into every process.

Before contacting the Binder driver, a process will use the `libbinder.so` to wrap the data into a "Parcel". This is a flattened data structure made up of

primitives data types that the kernel can handle. Once the Parcels have been created they are copied into Binder driver's memory space via "ioctl()" syscalls and then copied by the driver into the memory space of the receiving application. Once received, the Parcels are reconstructed and the request is processed. As data objects are flattened before they are sent, any data type can be sent over the Binder.

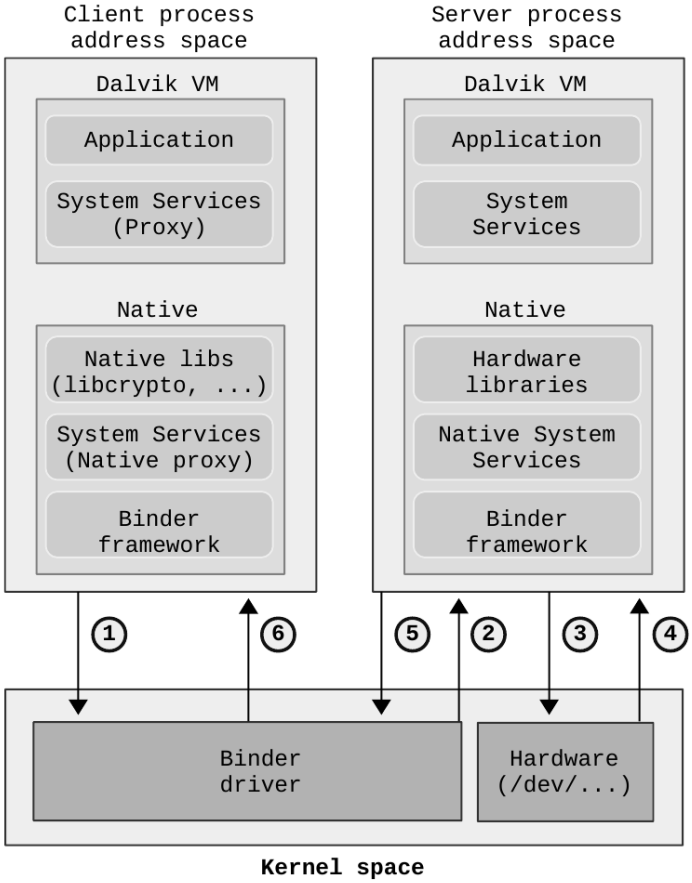

Figure 3.2: Interprocess communication using the Binder

Source: [2]

The client's address space contains an instance of the DVM, which runs the application and proxies for Java system services. These Java system services communicate with their native counterparts via JNI using the Binder. After marshalling the data object to be sent into Parcels, the Binder framework calls an `ioctl()` syscall with the file descriptor of `/dev/binder` as a parameter and transfers the data to the kernel (1). The driver then looks up the requested service and copies the data into the system server's address space, then wakes up a waiting thread in the server process to handle the request(2). After unmarshalling the "Parcel" objects and verifying that the client process has the

relevant permissions to carry out the required task, the server performs the requested service and, if necessary, calls into the kernel to interact with the relevant hardware (3) and receive a response from it (4). After this, the copy of `libbinder.so` which is loaded within the server's address space marshals the response data and sends it back to the driver(5), which hands it back to the client process (6) [2].

3.2 Security Features of Android for Work

3.2.1 Mobile Device Management

Devices are managed remotely by IT administrators using an EMM console provided by Google or a trusted partner vendor. Account provisioning must first take place before a device can be managed (See Appendix for detailed steps). During this process, a *work profile* is created on the device and a *device policy controller* (DPC) is installed. The DPC enforces policies set by the administrator such as requiring strong passwords, wiping data, or preventing clipboard sharing between device profiles.

There are two modes of operation in device provisioning, depending on whether the device is corporate or employee-owned. In the former, the DPC is configured in *device owner* mode. This gives the DPC complete control over the device, allowing it to perform device-wide actions such as configuring device-wide connectivity, global settings, and performing factory resets. In the latter (the BYOD case), the DPC is configured in "profile owner" mode, and is only able to manage the work profile.

Administrators are able to configure all devices enrolled in the system through the EMM console. Once installed, the DPC periodically contacts the company server with status information to ensure policy-compliance. If a device is non-compliant, an alert is displayed to the administrator and pre-defined responses carried out. In the past it has been possible for a device with root access to block regular contact to the server and send spoofed information so as not to raise any flags [6].

3.2.2 Root Detection

Corporate policy typically prohibits the use of rooted devices as they have the ability to undermine security measures by manipulating the underlying operating system. Application-level root detection is implemented by the MDM vendors via the DPC, though it is a non-trivial task as the root user has control the system environment being checked. Indeed, Evans et al (2014) systematically analysed all major MDM apps on offer from vendors in 2014 and was able to identify and bypass all checks [6]. Root checks can be discovered by decompiling application binaries and searching for hard-coded strings. For example, a check for the presence of the `su` binary or a search for common root applications amongst a list of installed packages can avoided by simply renaming the binary and application being searched for. Once an attacker is aware of the checks that are made, he simply changes the environment to remain undetected.

It's also interesting to note that many MDM providers (including Google) do not prevent rooted devices from connecting to EMM platforms by default.

Furthermore, some companies may require rooted devices to run proprietary applications. In both cases, the security policies can be intentionally or unintentionally undermined.

3.2.3 SELinux

Android has historically relied upon the Linux kernel's Discretionary Access Control (DAC) for enforcing application sandboxing and restricting access to system resources, even though its shortcomings are well documented [18]. SELinux was developed by the NSA as a Mandatory Access Control (MAC) mechanism to overcome these shortcomings. Unlike DAC, SELinux enforces a system-wide security policy over all processes, objects and operations on the basis of security contexts that define what they are allowed to do. As it operates at the kernel-level, it is able to restrict services and users that run with *root* or *system privileges* [17].

SELinux as enabled in *permissive mode*, where policy violations are logged but not enforced, when it was first introduced in version 4.2 [1]. However, it is now enabled in *enforcing mode* by default on version 5.0 and later, and is a requirement for devices enrolled in an Android for Work framework. When enabled in enforcing mode, policy violations are actively blocked and logged.

SELinux has made it significantly more difficult to root devices. Some of the latest tools require custom patched kernels that disable SELinux before a device can be rooted. However, the policies in SELinux have been notoriously difficult to configure properly in the past, often due to confusing syntax of policy definitions and considerable efforts have been spent on making it easier to implement properly [18, 17].

Though SELinux makes exploitation much harder, it is still not impossible. Gong (2015), was able to gain access to a shell running with `system_server` privileges in the presence of SELinux by chaining together a series of exploits found in different system services and utilising their different security contexts to access the next link in the chain [8].

3.2.4 Full-Disk Encryption

Full-disk Encryption (FDE) promises ensures that everything on disk is stored encrypted, including operating system files, cache, and temporary files. In Android, it is based on *dm-crypt*, a kernel feature that operates at the block device layer [10]. It uses 128 AES with cipher-block chaining (CBC) and ESSIV:SHA256. FDE has been enabled by default on Android since version 5.0 and is a requirement to enrol in Android for Work systems.

Upon first boot, the device generates a 128-bit key which is then encrypted using an additional key encryption key (KEK) that can be either stored in a hardware module such as a TPM, or derived from a passphrase entered by the user on each boot.

Limitations

Disk encryption only protects data at rest; that is, when the device is turned off. As disk encryption is transparent and handled at the kernel level, after an encrypted volume is mounted, it is indistinguishable from a plaintext volume

to user-level processes. Therefore, disk encryption does not protect data from malicious applications running on the device [5].

4. Methodology

This chapter focuses on our approach for conducting our analysis. Specifically, we detail our test environment and our method for addressing our research question and sub-questions stated in Chapter 1.

4.1 Environment Setup

To simulate an Android for Work deployment we first created an MDM platform compatible with Android for Work and enrolled a device. Android for Work operates as a a framework which partner vendors incorporate into their existing EMM platforms [11]. Though EMM platform implementations vary, they interact with devices using a standard API provided by Google. We chose to focus on the framework itself rather than implementation-specific features, using Google's own MDM platform, *Google Apps for Work*.

We used a rooted Google Nexus 7 tablet (2013), running Android 5.1.1. We used version 5.1.1. over the latest version (6.1) as this currently makes up the largest proportion of devices running Android and, in the context of BYOD schemes, is likely to be more representative of current BYOD schemes that could be deployed in enterprises today. We wanted to test how data is secured on the device in the presence of a malicious user who has compromised a device configured with an MDM platform. A common next step for Android malware once installed on a device is to download exploits from compromised servers to remotely root the device and gain unrestricted privileges. In addition, Android has also suffered from a number of vulnerabilities in the past that enable remote code execution with escalated privileges [7, 4]. Taken together with the relative ease a sophisticated attacker can circumvent root detection mechanism [6], we chose to use a rooted device to see how secure a device is in such a scenario.

As mentioned in 3, the Device Policy Controller can be installed as either a *device owner* or *profile owner* at time of enrolment. We test the latter, as it applies to scenarios where an employee brings in his personal device to be enrolled with the company's MDM platform. This gives the employee device-wide control and limits the MDM to managing only the work profile.

4.2 Approach

Recall our stated research question is to access information stored under the managed profile from the personal profile. To answer this question we proposed two methods:

1. Using an application installed by the user profile
2. Using the Binder IPC mechanism

We now discuss the structure of each approach in the following sections.

4.2.1 Application Approach

We first explored the file system of the enrolled device to understand how data separation between the two profiles is achieved. We spawned remote shells to interact with the device using the Android Debug Bridge (`adb`) command line tool that is included as part of the Android SDK [1]. This tool also enabled us to pull and push files to the device as well as run commands such as `logcat` and `ps`, that display real-time information on system logs and running processes.

After mapping out the file system in the context of Android for Work, we then used the Android Studio suite to develop a proof-of-concept application to demonstrate our findings.

4.2.2 Binder Approach

Previous work has already demonstrated that attacks via the Binder IPC are both viable and effective [2]. We chose to follow the approach outlined by Artenstein to see whether it is possible in context of BYOD schemes via managed and unmanaged user profiles. Essentially, we are testing for a scenario in which the attack has root access to a managed device and wishes to install a discrete listener via the Binder to intercept data passing through the Binder for as long as possible.

It consisted of the following steps:

1. Identify a target shared system service running on the device.
2. Download the Android AOSP source code and create a hook inside `libbinder.so` to dump information passing through calls made to the kernel driver to a file on the device.
3. Upload the modified `libbinder.so` to the device.
4. Inject the code into our target service by setting the `LD_PRELOAD` environmental variable and forcefully restarting the target service using the root shell.
5. Parse the acquired "Parcel" data to reveal its contents.

A full practical implementation of this attack was not possible due to time-constraints, though the results we were able to obtain are presented in the following chapter.

5. Analysis

This chapter describes the findings from our research. We first analyse how user data is separated on the file system, later presenting a simple application that is able to transfer data from the work profile to the user profile. We then analyse the Binder IPC in the context of multiple users and present a theoretical attack that aims to intercept data as it is exchanged between processes. Note that we use the terms *profiles* and *users* synonymously throughout.

5.1 Separating Data Between Users Profiles

The first user created on a device is set as the *device owner* and is assigned all privileges on the device, enabling it to create and delete other users as well change system-wide settings that affect all other users. It is assigned userID (UID) 0. Any subsequent users that are added as secondary users and lack the ability to configure system-wide settings on the device. The first secondary user is assigned UID 10 and subsequent UIDs are assigned incrementally from 10. In the context of our Android for Work BYOD scenario, where an employee brings in his own device, the work profile is added to the device as a *secondary user* by the Device Policy Control (DPC) application.

Each user on the device has a dedicated user directory containing meta-data detailing their installed appliations, permissions, and policy restrictions. Private data directories are created for each application and user and protected, in the absence of SELinux, by the regular UNIX file permissions. Support for multiple users was first implemented in Android 4.2, though it only supported tablets and was unable to run user profiles concurrently [1]. We can verify that both work and personal profiles are running concurrently by issuing a `pm list users` command in an user shell.

```
1 $ pm list users
2 Users:
3   UserInfo{0:Os3:13} running
4   UserInfo{10:Work profile:30} running
```

5.1.1 Applications

Applications for the Managed profile in Android for Work are controlled by the IT administrator. Using the rather ironically named *Google Play for Work Store*, administrators can whitelist applications that an employee is allowed to

install on his or her own device. Applications from the regular Google Play Store can be approved by the administrator and made available. Applications can be silently pushed to devices by the administrator and a managed user is unable applications.

When applications are installed for each user, they are assigned a per-user *effective ID* (EID) which is combination of a unique application ID (appID) and the UID of the user it is installed under. For example, the Skype EID listed below is a combination of `u10`, the work profile UID and texttta82, Skype's assigned appID.

```
drwxr-x--x u10_a82  u10_a82           2016-02-07 14:50 com.skype.
    raider
```

An applications installed under profile are not automatically installed for all profiles. This is enforced via the `package-restrictions.xml` file stored under the user's texttt/data/user/(UID)/ directory. An entry is added to this file whenever an application is installed by any user. If the application is installed by another user, an `inst` flag is set to `false`. Deleting this flag for an application in the `package-restrictions.xml` file will make the application available.

The ability to have applications installed from the user profile and make them available for the work profile is possible by a would-be attacker, though there is really nothing to gain as this would be flagged by the MDM app when receiving its routinely submitted list of installed packages and draw suspicion for further investigation.

As UNIX file permissions are the main mechanism preventing unauthorised acces to application data, a root user is able to access and view all files on the device. This means that any files that are unencrypted at the operating system level during normal operation can be read and highlights a need for extra layers of encryption to secure sensitive information.

5.1.2 Encryption

As mentioned in Chapter 3, full-disk encryption is limited to protecting data when the device is turned off. Once the device has been decrypted at start up, user-level processes are able to view files in plaintext format. This means that, unless data is stored within an encrypted container by the application, a user with root privileges is able to read this data.

To demonstrate this issue, we developed a simple application to extract email data from the Gmail app installed under the work profile.

5.1.3 Reading Work Profile Emails from the Personal Profile

The work profile's Gmail application store in the `/data/user/10/` directory has the following structure:

```
# ls
app_sslcache
app_webview
cache
databases
```

```
6 files
7 shared_prefs
```

The email database is stored as an SQLite datebase file under `mailstore.eric@themoves.nl.db`. After retrieving the file from the device and viewing it in an SQLite browser we were able to read the database structure and information about the received emails such as the sender's address, title of the message, and a snippet of the body. The main body field was however filled with $NULL$, due to compression from the *zlib* library. However, we can decompress these values and convert them into HTML using a Python script located in the Appendix.

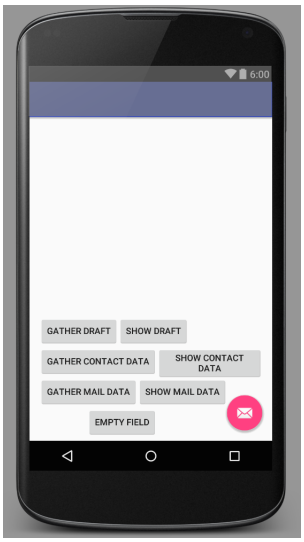

Figure 5.1: Application for extracting data from the managed profile to the personal profile

To demonstrate that a root user is able to access application data installed to a personal profile, a simple application was written 5.1. The application essentially automates the process just outlined. The purpose of the application is to show that the information can be read from another profile.

5.2 Binder IPC

As discussed earlier, the Binder IPC is the central point of communication flow between processes on Android. Whenever an application requests a service from another process, it passes through the Binder. To understand how data could be intercepted as it passes, we looked into the system services running on the device and documented process flow that takes place when a request is made.

To reduce confusion, we clarify some Binder terminology:

- Binder: *The* Binder refers to the overall Binder architecture, whilst *a* Binder refers to a particular implementation of the Binder interface.

- Binder Object: An instance of a class that implements the Binder interface. A Binder object can implement multiple Binders.
- Binder Protocol: The low-level protocol that the Binder middleware uses to communicate with the kernel driver.
- IBinder Interface: A well-defined set of methods, properties and events that a Binder can implement, described in AIDL.
- Binder Token: A numeric value that uniquely identifies a Binder.

5.2.1 Shared Resources

Services are shared between users on the device and provide access to various hardware resources in accordance with the calling application's permissions. They run in the background without direct interaction with the user, presenting an attractive opportunity for an attacker wishing to install a discrete listener on the target device.

Using the `service list` command on the device, we are able to view a list of running services and their corresponding interfaces.

```
1  location: [android.location.ILocationManager]
2  notification: [android.app.INotificationManager]
3  wifiscanner: [android.net.wifi.IWifiScanner]
4  wifi: [android.net.wifi.IWifiManager]
5  netpolicy: [android.net.INetworkPolicyManager]
6  textservices: [com.android.internal.textservice.
       ITextServicesManager]
7  network_management: [android.os.INetworkManagementService]
8  clipboard: [android.content.IClipboard]
9  statusbar: [com.android.internal.statusbar.IStatusBarService]
10 device_policy: [android.app.admin.IDevicePolicyManager]
11 mount: [IMountService]
12 input_method: [com.android.internal.view.IInputMethodManager]
13 bluetooth_manager: [android.bluetooth.IBluetoothManager]
14 input: [android.hardware.input.IInputManager]
15 vibrator: [android.os.IVibratorService]
16 content: [android.content.IContentService]
17 phone: [com.android.internal.telephony.ITelephony]
18 permission: [android.os.IPermissionController]
19 ...
```

Figure 5.2: Background services running on the device

As we can see in the output above there services are used to implement all of the core functionality of the device such as network connectivity, device sensors, and permission management. One instance of each service that runs on the device. To determine that the user profiles are sharing the same service a simple test can be conducted using the Keyboard application.

1. Use `ps` to identify the process ID (PID) of the keyboard application.
2. Verify that its EID is prefixed with the UID of the device owned (`u0`).

3. Open an application installed under the work profile that uses a keyboard interface.

4. Issue a `kill (PID)` command to kill the keyboard application.

5. Verify that the keyboard application has stopped both via `ps` and visually.

6. Restart the keyboard application by touching an input box for text in the work user application.

7. Check `ps` again and look for the keyboard application's PID

Conducting this test will confirm that the keyboard application is used by both profiles installed on the device. This is the example service that we wish to target for a potential keylogger and the process is outlined in the remainder of this section.

Keylogger Using the Binder

In order for an application to receive keyboard data, it must first register with an Input Method Editor (IME) server. An IME is the actual implementation of the keyboard and only one can be enabled on a device at a time. The default IME in most Android images is `com.android.inputmethod.latin`. When an application registers with an IME server to receive data from it, the client-server dynamic described in 3 is reversed: the application becomes the server and listens for remote requests from the Binder, whilst the IME acts as the client that sends Binder requests.

Binder interfaces are registered on the system via a *context manager* using a unique token that specifies their address. In Android, the *context manager* implementation is called *service manager*. The role of the *service manager* is essential to the Binder framework as each interface knows only its own address initially. When a Binder interface first becomes available, it will publish a name and its Binder token to the *service manager* so that other Binder objects can communicate with it via its interface.

In case of the keyboard application, every time a key is pressed on the keyboard application, this data is wrapped into a Parcel using the `libbinder.so` and passed via the Binder to the application that is processing the input. A Binder-based keylogger would intercept the Parcels as they leave the keyboard application.

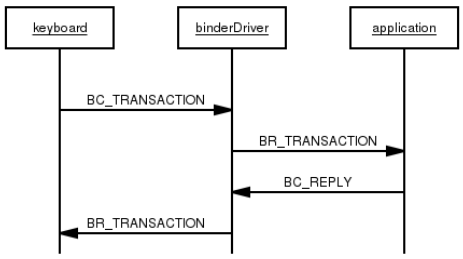

Figure 5.3: Binder Transaction Flow

Source: Created using mscgen_js

?? illustrates the same transaction flow between the keyboard application, Binder driver, and receiving application, only using Binder driver commands instead. Communication takes place in the form of `BC_TRANSACTION` and `BR_TRANSACTION`, and `BC_REPLY` and `BC_REPLY`. These are commands to the Binder driver that indicate what kind of message it is passing and keep track of transactions and their corresponding replies.

Such an attack, if successful could yield very valuable information for the attacker, at the expense of the enterprise. We now discuss a broad overview of the lower-level implementation.

5.2.2 Hooking the Binder

In order to intercept the Binder data via the `libbinder.so` library, the function responsible for transferring data from the client process to the kernel must be hooked. As published by Artenstein, this takes place in the `IPCThreadState::talkWithDriver` function exported by `libbinder.so`.

```
1 ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)
```

The parameters to this function are a kernel file descriptor for the process to write to, the command to send the Binder driver, and the address of the binder data structure containing the read and write buffers. Hooking this function would be in the form of the following pseudo-code.

```
1 int hooked_ioctl(int fd, int cmd, void *data) {
2   do_evil_deeds_with_transaction(data);
3   int ret = ioctl(fd, cmd, data);
4   do_evil_deeds_withreply(data);
5   return ret;
6   }
```

Whenever an `ioctl()` call is made to the driver, this hook function is called which parses the transaction data into a file stored locally on the device. It then performs the actual operation, before hooking the reply data that is sent back as a response. In this way data is intercepted as it flows through the Binder.

After adding the hook to the `libbinder.so` source code, one must compile the library and upload it to the device. Once this is done, configure the keyboard service to load the modified `libbinder.so` at runtime, and kill the current keyboard process. When the keyboard application starts up again, it will load the modified library and the intercept traffic.

```
1 # adb push libbinder.so /data/libbinder.so
2 # setprop wrap.com.google.android.inputmethod.latin LD_PRELOAD=/
    data/libbinder.so
```

This section provided a brief overview for how a keylogging application might be installed on a target device via the Binder. Its purpose was to highlight the mechanism through which is operates, rather than go into detailed exposition. For more details, we recommend reading [2].

5.3 Summary

To summarise this section, we analysed the way in which profile data is separated in Android for Work and demonstrated one method of accessing data across profiles. We then presented an overview of how a Binder-based attack would operate on the keyboard service, and motivated the case for why it should work. In the next chapter we discuss the results of our analysis.

6. Discussion

6.1 File Encryption

As we demonstrated in the first section of our analysis, after a device has been decrypted at boot time, data can be accessed in plaintext by processes running on the system. This occurs because the "dm-crypt" encryption takes place transparently, unbeknownst to the operating system. Though there are a significant number of measures in place to prevent attackers from gaining root privileges on a device enrolled with an MDM, previous work has demonstrated that it is more the case of *when* such an incident takes place, rather than *if*.

In order to help mitigate data leakage that could occur through this channel, we propose implementing file-based encryption inside applications. In this way, data would be encrypted and securely stored when the data is not in use and therefore protected from malicious applications. Of course, such a scheme would affect usability of the device as content must be decrypted before it is used each time. However, for sensitive applications such as storing confidential information, this is an acceptable trade-off.

6.2 The Binder IPC

Though we were unable to practically implement the Binder-based attack, recent work in this area has demonstrated that these types of attacks are still possible on Android. The issue is two-fold. Firstly, this type of attack is only possible because the data that is flowing across the Binder does so in plaintext. As all interprocess communication flows through the Binder it represents a single point of failure in the system. The second issue is stems from the notion of completely isolating data and services used by the managed and unmanaged profiles. There are currently one set of system services running on the device to access core functionality on the device. So data, necessarily, has to flow over potentially compromised channels.

In order to address Binder-based attacks, at least two things should be addressed. Firstly, the amount of data that is exchanged between applications handling sensitive data should be minimised, for example by implementing dedicated in-app services so they don't need to use the Binder. Secondly, data that is passed over the Binder should be encrypted. This of course is not so straight-forward to implement as there is a clear trade-off that needs to be made between security and practicality. Enabling on-the-fly encryption without a highly efficient encryption algorithm will be costly on both the devices resources (e.g. battery) and performance. Kaladharan et al. (2016), recently proposed

a way to encrypt traffic using a Lightweight Cryptography technique such as Sony's CLEFIA [3], though more research and auditing of secure encryption schemes must be done.

It is important that features such as file-based encryption and Binder encryption is eventually incorporated into the device. Application developers working at high-tech startups holds a different set of priorities from employees and business-owners as they are pressured to deliver their products faster and faster. As such they are unlikely to have the time nor the expertise to implement complicated encryption schemes themselves. Hence the overall security of the platform is undermined.

7. Future work

Unfortunately we were unable to implement the Binder-based attack proposed due to time constraints. An immediate point of future work would be to complete the implementation of this attack to prove that data from managed profiles can be intercepted as it passes through the Binder. Awareness of Binder-based attacks is growing and successful attacks have been demonstrated. Thus, there is a clear need for strategies that can prevent these types of attacks.

Research into file-based encryption techniques that could be implemented at the operating system level would secure sensitive data from malicious applications, and remove the reliance upon application developers to implement encryption schemes themselves. Furthermore, research into methods of encrypting traffic that passes through the Binder that doesn't hinder system performance or significantly drain device battery life could also be very promising.

Finally, research into alternative methods of implementing dual-persona devices that don't rely upon shared services. Perhaps through applications of virtualisation, or separate hardware modules designed to handle data and services for managed profile.

8. Conclusion

In this work we addressed whether or not it is possible to access data between managed and unmanaged profiles on a rooted Android device. During our research, we were able to move able to access data using one method, and proposed another method that should be possible. In both cases, it is clear that encryption on the device does not protect data from malicious applications. If an attacker is able to successfully root a device from the unmanaged profile and remain undetected by EMM platforms, it can be relatively trivial to access the data stored by applications on the managed side. An attacker can also target system services that are shared by both profiles on the device. By hooking calls made to the kernel as processes exchange data, an attacker is able to silently copy data as it passes through.

The presence of full-disk encryption on the device protects data at rest, but can be viewed in plaintext by processes once the device has been decrypted at boot up. The key is stored on a separate hardware platform module and encrypted using a user-defined password. Only one password is used to decrypt the entire device, if a malicious user is able to gain root access and read the data of one profile, there is a significant chance that it can read data from the other. File-based encryption containers could be a possible solution to prevent this kind of attack, as data stored inside is encrypted when not in use, however it is the responsibility of the application developers to implement this kind of functionality.

All user profiles on a device share the same system services to access hardware functionality. Thus, if a attacker is able inject his own code that hook the target system services as they interact with the kernel, they can intercept data as it passes processes. This presents the attacker with an opportunity to install silent listening services on the device that record activity such as key loggers, and GPS trackers. This method of attack is currently limited only by the services that use the Binder. There are two suggestions to mitigate the likelihood and effectiveness of these attacks. Firstly by minimising the amount of traffic that flows across the Binder by implementing services that handle sensitive information, such as keyboards, within the application itself. Secondly, by encrypting sensitive traffic before it passes over the Binder. Both suggestions require the attention of application developers to implement, though a revision to the Android operating system that incorporates encryption into the Parcelling process would be an optimal solution.

We conclude that, under the assumption of root access to a device, it is possible to access data between Android profiles. The heterogeneity of Android versions, number of previous Android vulnerabilities, and possibility to bypass current root detection methods make this scenario an inevitability in the future.

Further research into incorporating file-based encryption into standard operating system as well as achieving on-the-fly encryption for data passing over the Binder is suggested.

Acknowledgements

We would like to express our gratitude to our supervisors Rick van Galen and Paul van Iterson for their feedback and support throughout the duration of this project at KPMG.

9. Appendix

9.0.1 Setting Up Android for Work

The Android for Work solution can be set up in two different ways. Either within Google's MDM platform (Google Apps for Work) or in any other MDM platform (AirWatch). In this appendix both methods are described.

Android for Work in Combination with Google Apps for Work

First step is to create a Google Apps for Work account at https://apps.google.com, which requires name credentials and most important; the business or organisation domain name. During this project the domain *themoves.nl* is used. Two methods were used in order for Google to verify whether the given domain is really ours; modify both homepage and MX records of the domain (*themoves.nl*). A meta tag containing a verification-token had to be added to the index.html file, whereas the MX records had to be changed to Google's mail servers.

It is mandatory to add users to the created work environment (admin panel) in order to use all Google apps for Work functionalities. An email address and password is automatically created when a new user (employee) is added.

After adding a user it is possible to enable Android for Work, which is available from the *get more apps and services* button under the *Common tasks* heading in the right sidebar. Here the *subscription* on the Android for Work service can be enabled, which makes it possible to generate an EMM token (a string of characters that EMM providers can use to connect or bind their Android for Work solution to the Google Account[**afwhelp**]) in the *Security* section of the admin control panel.

The binding between the Google apps for Work account and Android for Work is completed when the generated EMM token is filled in to the token field in the *Device management mobile settings*.

All previous steps were needed in order to enable the Android for Work solution on the MDM side. The following steps are performed at the device that a user wants to enroll.

A device controller app needs to be installed, so that the policies that are set at the MDM platform can be pushed to the device; *Google's Device Policy App*. The application will prompt the user to fill in his/her Google Business account, which is the account that is created by the IT administrator in the MDM platform. After this authentication the *Device Policy app* automatically detects whether or not the device is Android for Work compatible.

The device is now enrolled, and displays both personal as work applications on the menu.

Android for Work in Combination with AirWatch

Setting up Android for Work for a non-Google MDM provider is slightly more different than described in section [REF], since Android for Work recquires a Google Business account for every Androi For work user. Therefore, the non-Google MDM provider (AirWatch) has to generate an API call to create a Google Business account when a compatible Android for Work needs to be enrolled. We will see how this is achieved in the following steps.

The AirWatch MDM platform was already set up for us, so we can not provide information about how this is achieved.

The first step is again to create a Google apss for work account with the companies domain name. This time only the index.html verification is sufficient, since email is not mandatory to flow via Google when using AirWatch.

A project has to be created on the console.developers.com website (log in with Google Apps for Work account) where Google EMM, Admin SDK API and Auth2 credentials need to be enabled/created in order for AirWatch to make API calls to Google.

In the security section the Android for Work EMM token can be generated. This time, the token is filled in to the Android for Work settings on the AirWatch MDM platform, followed by adding the API credentials that were created. Now AirWatch is bound to the Google App for Work account which was created at step 1.

A device controller app has to be installed for an employee to enroll a device to the AirWatch MDM platfor; *AWMDM agent*. Again, automatically the app detects wheter or not the device is Android for Work ompatible, after the employee has authenticated with his/her AirWatch user credentials. If the device is Android for Work compatible, the user is prompted to create a Google Business account. This leads to the API call to generate this account, and store it in the user database of the given domain.

9.0.2 Application Code

```
1     public void buttonOnClickDraft(View v){
2         try {
3             Process proc = Runtime.getRuntime().exec("su && rm /
      data/local/draft.txt && cat /data/user/10/com.google.android.gm
      /databases/mailstore.eric@themoves.nl.db-wal | grep -aE '
      keyword' > /data/local/draft.txt && chmod 777 /data/local/draft
      .txt");
4             //proc.waitFor();
5         } catch (Exception e) {
6             Log.d("Exceptions", "Exception dropping caches: "+e);
7         }
8
9     }
10
11    public void buttonOnClickDraft2(View v){
12
13        File sdcard = Environment.getExternalStorageDirectory();
14        File file = new File("/data/local/", "draft.txt");
15
16        TextView tv = (TextView) findViewById(R.id.textView2);
17
18
```

```
19
20          if(file.exists()){
21              StringBuilder text = new StringBuilder();
22
23              try {
24                  BufferedReader br = new BufferedReader(new
    FileReader(file));
25                  String line;
26
27                  while ((line = br.readLine()) != null) {
28                      text.append(line);
29                      text.append('\n');
30                  }
31              }
32              catch (IOException e) {
33                  Log.d("Exceptions", "Here it is "+e);
34              }
35              //Set the text
36              tv.setText(text);
37          }
38          else{
39              tv.setText("-");
40          }
41
42      }
43
44      public void buttonOnClick(View v){
45          try {
46              Process proc = Runtime.getRuntime().exec("su && rm /
        data/local/contact.txt && cat /data/user/10/com.android.
        providers.contacts/databases/contacts2.db | grep -aE 'keyword'
        > /data/local/contact.txt && chmod 777 /data/local/contact.txt"
        );
47              //proc.waitFor();
48          } catch (Exception e) {
49              Log.d("Exceptions", "Exception dropping caches: "+e);
50          }
51
52      }
53
54      public void buttonOnClick2(View v){
55
56          File sdcard = Environment.getExternalStorageDirectory();
57          File file = new File("/data/local/", "contact.txt");
58
59          TextView tv = (TextView) findViewById(R.id.textView2);
60
61
62
63          if(file.exists()){
64              StringBuilder text = new StringBuilder();
65
66              try {
67                  BufferedReader br = new BufferedReader(new
    FileReader(file));
68                  String line;
69
70                  while ((line = br.readLine()) != null) {
71                      text.append(line);
72                      text.append('\n');
73                  }
74              }
```

```
75              catch (IOException e) {
76                  Log.d("Exceptions", "Here it is "+e);
77              }
78              //Set the text
79              tv.setText(text);
80          }
81          else{
82              tv.setText("-");
83          }
84
85      }
86
87      public void buttonOnClickMail1(View v){
88          try {
89              Process proc = Runtime.getRuntime().exec("su && rm /
        data/local/mail.txt && cat /data/user/10/com.google.android.gm/
        databases/mailstore.eric@themoves.nl.db | grep -aE 'keyword' >
        /data/local/mail.txt && chmod 777 /data/local/mail.txt");
90              //proc.waitFor();
91          } catch (Exception e) {
92              Log.d("Exceptions", "Exception dropping caches: "+e);
93          }
94
95      }
96
97      public void emtpyField(View v){
98          TextView tv = (TextView) findViewById(R.id.textView2);
99          tv.setText("");
100     }
101
102     public void buttonOnClickMail2(View v){
103
104         File sdcard = Environment.getExternalStorageDirectory();
105         File file = new File("/data/local/", "mail.txt");
106
107         TextView tv = (TextView) findViewById(R.id.textView2);
108
109
110
111         if(file.exists()){
112             StringBuilder text = new StringBuilder();
113
114             try {
115                 BufferedReader br = new BufferedReader(new
        FileReader(file));
116                 String line;
117
118                 while ((line = br.readLine()) != null) {
119                     text.append(line);
120                     text.append('\n');
121                 }
122             }
123             catch (IOException e) {
124                 Log.d("Exceptions", "Here it is "+e);
125             }
126             //Set the text
127             tv.setText(text);
128         }
129         else{
130             tv.setText("-");
131         }
132
```

```
133     }
```

9.0.3 Email extract script [pythonmail]

```
1  import sqlite3
2  import zlib
3
4  good_chars=' ,.0123456789
       @ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz~ '
5
6  conn = sqlite3.connect('mailstore.eric@themoves.nl.db')
7  cursor = conn.cursor()
8  cursor.execute("select _id, fromAddress, subject, bodyCompressed
       from messages")
9  rows = cursor.fetchall()
10
11 for row in rows:
12      fname = (str(row[0]) + row[1] + row[2])[:48]
13      fname = ''.join([c for c in fname if c in good_chars])
14      print fname
15      with open(fname + '.html', 'wb') as fout:
16          if row[3]:
17              data = zlib.decompress(row[3])
18              fout.write('<html><body>' + data + '</body></html>')
19
20 cursor.close()
21 conn.close()
```

Bibliography

[1] Android. *Android Developers*. 2016-07-01. unknown. URL: http://developer.android.com.

[2] Artenstein. *Man-In-The-Binder: He Who Controls IPC Controls the Droid*. 2014. URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf.

[3] Saketh Paranjape Dhinakaran Pandiyan. 2012. URL: http://rts.lab.asu.edu/web_438/project_final/Talk%208%20AndroidArc_Binder.pdf (visited on 11/01/2016).

[4] Joshua Drake. *Stagefright: Scary Code in the Heart of Android*. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf (visited on 15/01/2016).

[5] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014. ISBN: 9781593275815. URL: https://books.google.nl/books?id=y11NBQAAQBAJ.

[6] Nathan S. Evans, Azzedine Benameur, and Yun Shen. "All Your Root Checks Are Belong to Us: The Sad State of Root Detection". In: *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access*. MobiWac '15. Cancun, Mexico: ACM, 2015, pp. 81–88. ISBN: 978-1-4503-3758-8. DOI: 10.1145/2810362.2810364. URL: http://doi.acm.org/10.1145/2810362.2810364.

[7] geohot. *Towelroot by geohot*. URL: https://towelroot.com/ (visited on 15/01/2016).

[8] Guang Gong. *Fuzzing Android System Services by Binder Call to Escalate Privilege*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf.

[9] Google. *Android 4.2 APIs*. URL: http://developer.android.com/about/versions/android-4.2.html#MultipleUsers (visited on 15/01/2016).

[10] Google. *Android for Work Security White Paper*. 2015. URL: https://static.googleusercontent.com/media/www.google.nl/nl/NL/work/android/files/android-for-work-security-white-paper.pdf (visited on 16/01/2016).

[11] *Google Apps Administrator Help*. URL: https://support.google.com/a/answer/6178111?hl=en (visited on 12/01/2016).

[12] International Data Corporation (IDC). *Smartphone OS Market Share, 2015 Q2*. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 12/01/2016).

[13] Jeff Louck et al. *The Financial Impact of BYOD*. 1st ed. Cisco IBSG, 2013. URL: https://www.cisco.com/web/about/ac79/docs/re/byod/BYOD-Economics_Econ_Analysis.pdf.

[14] Carly Page. 2015. URL: http://www.theinquirer.net/inquirer/news/2433718/latest-android-adware-threat-is-virtually-impossible-to-remove (visited on 07/01/2016).

[15] Paul Ratazzi et al. "A Systematic Security Evaluation of Android's Multi-User Framework". In: *arXiv preprint arXiv:1410.7752* (2014).

[16] Chuangang Ren et al. "Towards discovering and understanding task hijacking in android". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association. 2015.

[17] Stephen Smalley. "The case for SE Android". In: *Linux Security Summit* (2011).

[18] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." In: *NDSS*. Vol. 310. 2013, pp. 20–38.

[19] Good Technology. "Mobility Index Report, Q3 2015". In: (). URL: https://media.good.com/documents/mobility-index-report-q3-2015.pdf (visited on 12/01/2016).