Android Operating System: Architecture, Security Challenges and Solutions

Working Paper · March 2016

DOI: 10.13140/RG.2.1.4966.3126

1 author:

Ahamed Shibly
South Eastern University of Sri Lanka
46 PUBLICATIONS **10** CITATIONS

SEE PROFILE

Android Operating System: Architecture, Security Challenges and Solutions

FHA. Shibly, Lecturer in IT, South Eastern University of Sri Lanka, Oluvil, Sri Lanka, E-mail :- shiblymis@gmail.com

1.0 Introduction

As smart phones and tablets become more popular, the operating systems for those devices become more important. Android is such an operating system for low powered devices that run on battery and are full of hardware like Global Positioning System (GPS) receivers, cameras, light and orientation sensors, Wi-Fi and UMTS (3G telephony) connectivity and a touch screen. Like all operating systems, Android enables applications to make use of the hardware features through abstraction and provide a defined environment for applications.

Unlike on other mobile operating systems like Apple's iOS, Palm's webOS or Symbian, Android applications are written in Java and run in virtual machines. For this purpose Android features the Dalvik virtual machine which executes its own byte code. Dalvik is a core component, as all Android user applications and the application framework are written in Java and executed by Dalvik. Like on other platforms, applications for Android can be obtained from a central place called Android Market.

The platform was created by Android Inc. which was bought by Google and released as the Android Open Source Project (AOSP) in 2007. A group of 78 different companies formed the Open Handset Alliance (OHA) that is dedicated to develop and distribute Android. The software can be freely obtained from a central repository [01] and modified in terms of the license which is mostly BSD and Apache. [02]

The development of Android takes place quickly, as a new major release happens every few months. This leads to a situation where information about the platform becomes obsolete very quickly and sources like books and articles can hardly keep up with the development. Sources that keep up with the pace are foremost the extensive SDK documentation, documentation in and the source code itself as well as blogs.

2.0 Background & History

Android is described as a mobile operating system, initially developed by Android Inc. Android was sold to Google in 2005. Android is based on a modified Linux 2.6 kernel. Google, as well as other members of the Open Handset Alliance (OHA) collaborated on Android (design, development, distribution). Currently, the Android Open Source Project (AOSP) is governing the Android maintenance and development cycle [03].

To reiterate, the Android operating system is based on a modified Linux 2.6 kernel [04]. Compared to a Linux 2.6 environment though, several drivers and libraries have been either modified or newly developed to allow Android to run as efficiently and as effectively as possible on mobile devices (such as smart phones or internet tablets). Some of these libraries have their roots in open source projects. Due to some licensing issues, the Android community decided to implement their own c library (Bionic), and to develop an Android specific Java runtime engine (Dalvik Virtual Machine – DVM). With Android, the focus has always been on optimizing the infrastructure based on the limited resources available on mobile devices [05]. To complement the operating environment, an Android specific application framework was designed and implemented. Therefore, Android can best be described as a complete solution stack, incorporating the OS, middleware components, and applications. In Android, the modified Linux 2.6 kernel acts as the hardware abstraction layer (HAL). To summarize, the Android operating environment can be labeled as:

- An open platform for mobile development
- A hardware reference design for mobile devices
- A system powered by a modified Linux 2.6 kernel
- A run time environment
- An application and user interface (UI) framework

Android Architecture

Figure 1 outlines the current (layered) Android Architecture. The modified Linux kernel operates as the HAL, and provides device driver, memory management, process management, as well as networking functionalities, respectively. The library layer is interfaced through Java (which deviates from the traditional Linux design). It is in this layer that the Android specific libc (Bionic) is located. The surface manager handles the user interface (UI) windows. The Android runtime layer holds the Dalvik Virtual Machine (DVM) and the core libraries (such as Java or IO). Most of the functionalities available in Android are provided via the core libraries.

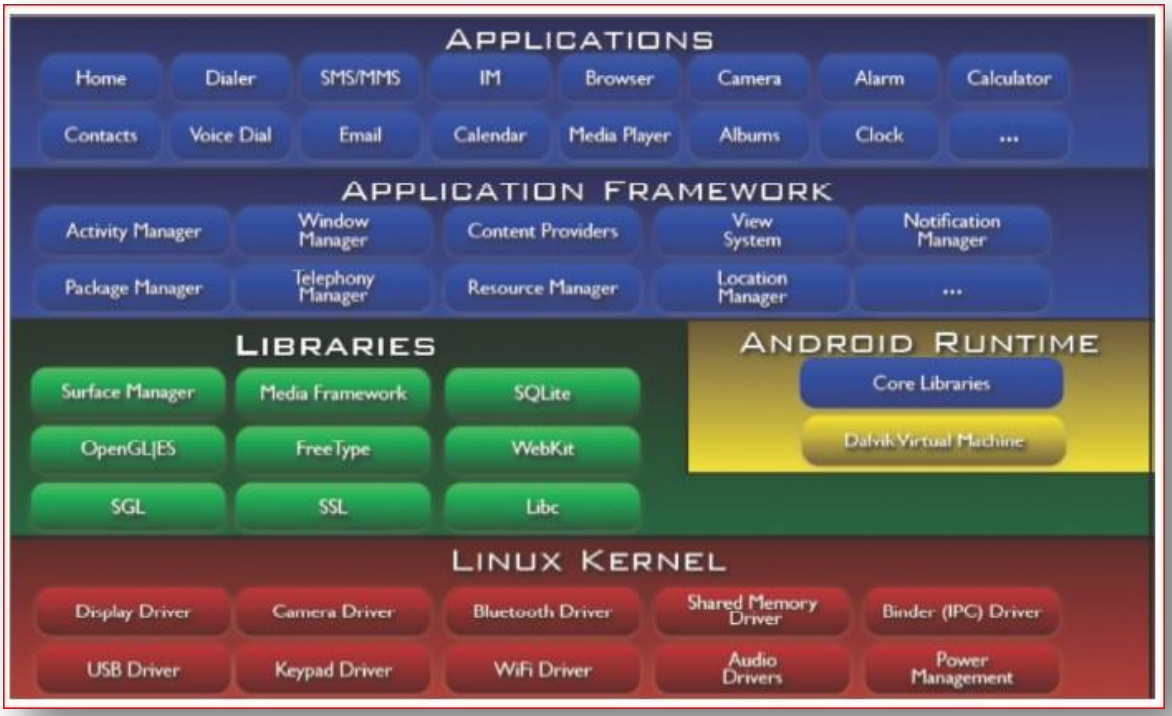

Figure 1 :- Android system architecture. Green items are written in C/C++, blue items are written in Java and run in the Dalvik VM. Image taken from [06, What is Android?].

Application layer: It is the most upper layer in android architecture. All the applications like camera, Google maps, browser, sms, calendars, contacts are native applications. These applications works with end user with the help of application framework to operate. **Application framework:** Android applications which are developing, this layer contain needed classes and services. Developers can reuse and extend the components already present in API. In this layer, there are managers which enable the application for accessing data. These are as follows: [5] **Activity manager:** It manages the lifecycle of applications. It enables proper management of all the activities. All the activities are controlled by activity manager. **Resource manager:** It provides access to non-code resources such as graphics etc. **Notification manager:** It enables all applications to display custom alerts in status bar. **Location manager:** It fires alerts when user enters or leaves a specified geographical location. **Package manager:** It is use to retrieve the data about installed packages on device. **Window manager:** It is use to create views and layouts. **Telephony manager:** It is use to handle settings of network connection and all information about services on device. **Android runtime:** In this section, all the android applications are executed. Android has its own virtual machine i.e. DVM (Dalvik Virtual Machine), which is used to execute the android application. With this DVK, users are able to execute multiple applications ate same time.

Libraries: Android has its own libraries, which is written in C/C++. These libraries cannot be accessed directly. With the help of application framework, we can access these libraries. There are many libraries like web libraries to access web browsers, libraries for android and video formats etc. **Linux kernel:** This layer is core of android architecture. It provides service like power management, memory management, security etc. It helps in software or hardware binding for better communication. **Security in android app:** According to review, there is a research paper on security issues on android smart phones. Paper is Taming Information Stealing Smartphone Applications (TISSA). In this paper, TISSA is a system which is used to provide security to the contacts, call logs etc. By using TISSA, user can easily protect its contacts and call logs by filling all the permissions. After giving all the permissions, user can easily access its own data in very privacy mode. TISSA is evaluated with many of android apps which are affected by leakage of private information of user. TISSA uses efficient CPU, memory and energy etc. In TISSA, there are main three components are used which provides security to the user for securing call logs and contacts. **These main components are: Privacy setting content provider:** It is used to provide current privacy

setting for an installed application. **Privacy setting manager**: It is for the user that he/she can easily update the privacy setting for the installed application. **Privacy aware components:** These are enhanced to regulate the access to user's information which also includes contacts, call logs and locations. TISSA starts works when user sends request through installed app to the content provider. It holds the request and check current privacy settings for app. It matches all the stored information in database and then send result back to the content provider. If all the information is correct then it allows the user to access the data otherwise it will reject the request.

Dalvik Virtual Machine

Android based systems utilize their own virtual machine (VM), which is known as the Dalvik Virtual Machine (DVM) [7]. The DVM uses special byte-code, hence native Java bytecode cannot directly be executed on Android systems. The Android community provides a tool (dx) that allows converting Java class files into Dalvik executables (dex). The DVM implementation is highly optimized in order to perform as efficiently and as effectively as possible on mobile devices that are normally equipped with a rather slow (single) CPU, limited memory resources, no OS swap space, and limited battery capacity. The DVM has been implemented in a way that allows a device to execute multiple VM's in a rather efficient manner.

It also has to be pointed out that the DVM relies on the modified Linux kernel for any potential threading and low-level memory management functionalities. With Android 2.2, some major changes to the JVM infrastructure were implemented. Up to version 2.2, the JVM was an actual interpreter, similar to the original JVM solution deployed with Java 1.0. While the Android solution always reflected a very efficient interpreter, it was still an interpreter and hence, no native code was generated. With the release of Android 2.2, a justin-time (JIT) compiler has been incorporated into the solution stack, which translates the Dalvik byte-code into much more efficient machine code (similar to a C compiler). Down the road, additional JIT and garbage collection (GC) features will be deployed with Android, further busting (potential) aggregate systems performance.

Kernel and Startup Process

It is paramount to reiterate that while Android is based on Linux 2.6, Android does not utilize a standard Linux kernel [04],[08]. Hence, an Android device should not be labeled a Linux solution per se. Some of the Android specific kernel enhancements include:

- alarm driver (provides timers to wakeup devices)
- shared memory driver (ashmem)
- binder (for inter-process communication),
- power management (which takes a more aggressive approach than the Linux PM
- solution)
- low memory killer
- kernel debugger and logger

During the Android boot process, the Android Linux kernel component first calls the *init* process (compared to standard Linux, nothing unusual there). The *init* process accesses the files *init.rc* and *init.device.rc* (*init.device.rc* is device specific). Out of the *init.rc* file, a process labeled *zygote* is started. The *zygote* process loads the core Java classes, and performs the initial processing steps. These Java classes can be reused by Android applications and hence, this step expedites the overall startup process. After the initial load process, *zygote* idles on a socket and waits for further requests.

Every Android application runs in its own process environment. A special driver labeled the *binder* allows for (efficient) inter-process communications (IPC). Actual objects are stored in shared memory. By utilizing shared memory, IPC is being optimized, as less data has to be transferred. Compared to most Linux or UNIX environments, Android does not provide any swap space. Hence, the amount of virtual memory is governed by the amount of physical memory available on the device [03].

Storage Media & File System

When it comes to configuring and setting-up mobile devices, traditional hard drives are in general too big (size), too fragile, and consume too much power to be useful. In contrast, flash memory devices normally provide a (relative) fast read access behavior as well as better (kinetic) shock resistance

compared to hard drives. Fundamentally, two different types of flash memory devices are common, labeled as NAND and NOR based solutions [5]. While in general, NOR based solutions provide low density, they are characterized as (relative) slow write and fast read components. On the other hand, NAND based solutions offer low cost, high density, and are labeled as (relative) fast write and slow read IO solutions. Some embedded systems are utilizing NAND flash devices for data storage, and NOR based components for the code (the execution environment).

From a file system perspective, as of Android version 2.3, the (well-known) Linux ext4 file system is being used [9]. Prior to the ext4 file system, Android normally used YAFFS (yet another flash file system). The YAFFS solution is known as the first NAND optimized Linux flash file system. Some Android product providers (such as Archos with ext3 in Android 2.2) replaced the standard Archos file system with another file system solution of their choice. As of the writing of this report, the maximum size of any Android application equals to a low 2-digit MB number, which compared to actual Linux based systems has to be considered as being very small. This implies that the memory and file system requirements (from a size perspective – not from a data integrity perspective) are vastly different for Android based devices compared to most Linux systems.

Power Management

In the mobile device arena, power management is obviously paramount. That does not imply though that power management should be neglected on any other system. Hence, power management in any IT system, with any operating system, is considered a necessity due to the ever increasing power demand of today's computer systems. To illustrate, to reduce and manage power consumption, Linux based systems provide power-saving features such as *clock gating*, *voltage scaling*, activating *sleep modes*, or *disabling memory cache*. Each of these features reduces the system's power consumption (normally at the expense of an increased latency behavior) [9]. Most Linux based systems manage power consumption via the Advanced Configuration and Power Interface (*ACPI*).

Android based systems provide their own power management infrastructure (labeled *Power Manager*) that was designed based on the premise that a processor should not consume any power if no applications or services actually require power. Android demands that applications and services request CPU resources via *wake locks* through the Android application framework and native Linux libraries. If there are no active *wake locks*, Android will shutdown the processor.

Android Applications

Android applications are bundled into an Android package (.apk) via the Android Asset Packaging Tool (AAPT). To streamline the development process, Google provides the Android Development Tools (ADT). The ADT streamlines the conversion from class to dex files, and creates the .apk during deployment. In a very simplified manner, Android applications are in general composed of:

- *Activities* (needed to create a screen for a user application – classes with a UI)
- *Intents* (used to transfer control from one activity to another)
- *Services* (classes without a UI, so they can be executed in the background)
- *Content Providers* (allows the application to share information with other applications)

3.0 Security Issues in Android OS

Security has always been a major concern for consumers, but it's especially important for enterprise users. In far too many cases, malicious hackers are able to take control of a computer, steal sensitive information or use it against millions across the world. On the Web, hackers take every opportunity to try and take down sites or turn them into their own personal Trojan-delivering friends. However, in recent years, a new threat has emerged that, at least so far, few people know enough about: mobile security. From Android to Symbian and even, in some cases, iOS, operating systems across the mobile market are being targeted by malicious hackers. Users of those operating systems, meanwhile, do little to safeguard themselves from those threats. In a world where the trend to bring user's own device (BYOD) is becoming the norm, the worlds of consumer and business security are starting to collide. Interestingly, over the last year or so, Android has become the chief target for malicious hackers. According to several reports, cyber-crooks are targeting the Android operating system since it's essentially open and the sheer number of people using the platform makes it a worthwhile option. But there are still many people that don't believe Android security is a major threat to them. Those people need to wake up. Here's a look at why Android security is a growing threat. [10]

Android remains the world's most widely used operating system, based on market and usage share statistics, used by hundreds of millions of customers worldwide.

But, according to a new document obtained by Public Intelligence, the U.S. Dept. of Homeland Security (DHS) and the Federal Bureau of Investigation (FBI) are increasingly aware of the threats its law enforcement users and officials face at a federal, state, and local level in using older versions of the mobile platform.

According to the roll call release — marked as unclassified but "for official use only," and designed for police, fire, emergency medical services (EMS) and security personnel — upwards of 44 percent of Android users worldwide are still using Android versions 2.3.3 to 2.3.7, which still contain security vulnerabilities fixed in later versions.

The document does not state, however, how many U.S. government staff use Android, let alone older versions of Android, on its networks. Android continues to be a "primary target for malware attacks due to its market share and open source architecture," the document says, and an uptick in mobile device use by government users "makes it more important than ever to keep mobile [operating systems] patched and up-to-date."

As many will know, staying ahead of the Android security curve requires actively ditching existing handsets and buying a new device, particularly in a bring-your-own-device world where this falls down to the responsibility of the user. Many manufacturers and carriers do not issue the latest Android versions for older devices. [09]

Some highlights from the report:

- 79 percent of malware threats affect Android, with 19 percent targeting Symbian. Windows Mobile, BlackBerry, iOS, and others all peg in at less than 1 percent each. (The source of the figures is not known.)

- SMS text messages represent "nearly half" of the malicious applications circulating today on older Android operating systems. Users can mitigate by installing Android security suites on their devices.

- Rootkits also pose a massive threat. The DHS/FBI document notes that in late 2011, a popular rootkit Carrier IQ was installed on millions of devices, including Apple iPhones (though Apple later removed the software) and dozens of Android devices. These rootkits often go undetected and can log usernames, passwords, and traffic without the user's knowledge — a serious security risk in a government enterprise setting.

- Fake Google Play domains are sites created by cybercriminals, the document notes, which replicate the Android application store to trick users into installing fake or malicious apps. DHS/FBI note that only IT approved updates should be allowed, hinting that IT department should ensure secure IT policies from back-end mobile device management services.

The open nature of Android and its large user base have made it an attractive and profitable platform to attack. Common exploits and tool kits on the OS can be utilised across a wide number of devices, meaning that attackers can perform exploits en masse and re-use attack vectors. It is obvious why Android is a target, but why is it vulnerable? Google did take measures in the development of the Android kernel to build security measures in; the OS is sandboxed, preventing malicious processes from crossing between applications. Whilst this attempt to eliminate the concept of infection is admirable in some regards, it fails to address the issue of infection altogether. Android is a victim of its own success, not just in the way it has attracted malicious attention, but in its very nature. One of the reasons the OS has succeeded in gaining market share so rapidly is that it is open source, it is essentially free for manufacturers to implement (patent settlements excluded!). Additionally this has led to substantial fragmentation of Android versions between devices and means that vendors have been reluctant to roll-out updates, presumably out of some concern regarding driving demand for future devices.

There is little value to the manufacturer in updating a device, something that to date Google has tried to encourage but been largely unsuccessful in doing so. Where updates do occur, manufacturer specific software on top of Android (such as HTC's Sense or Motorola's Blur) and even network provider bloat ware, serve only to further delay patch management. After Google release an update this must then be customized by the manufacturer and network before release, unless of course it is a vanilla device such as the Nexus range. As a result vulnerabilities are left un patched in stock ROMs, and advanced users are turning to flashing custom ROMs on their devices which raises a whole host of other issues. In an enterprise environment, who is responsible for patching a connected consumer device? And what of the users? Increasingly employees want to be able to use their smart phones at work, they want to access their email on the go, may need to access a content management system, and might prefer to log on to the corporate network than use 3G. Where Blackberry went from enterprise to consumer in terms of market penetration, Android is doing the inverse (much as iOS has) – consumers are buying these devices for personal use but wanting to utilise them in a professional capacity as well but without regard for the impact. So what does this mean for security? What threats are there to corporate information assets?

Security Issues in Android OS

Taking specific malware out of the equation, what are some of the threats/vulnerabilities on Android devices that might be cause for concern? These certainly are not comprehensive, but do cover a significant range of the vulnerabilities and risks that may be exploited on the Android OS:

User as admin

Install apps, grant app permissions, download data, and access unprotected networks - The user can reign free over their Android domain without restriction.

The Android Market

Google's verification processes for applications entering their market have been shown to be woefully lacking over the last year or two, leading to a number of malware-infected apps and games being made legitimately available to users.

Gateway to PC

HTC devices have long been able to utilize a VPN, but increasingly other applications are becoming available for remote access – Go to Meeting, Team Viewer, Remote Rack space. Although secured, these third party services still provide a line in to the corporate network and may be implemented fairly easily on to an endpoint. Any Android device can be connected to a PC via a USB cable, laying out the contents of its SD card for read/write/delete. The SD card itself as removable storage can be easily accessed directly as well. Indeed these methods could be utilized themselves for bringing malware in to a corporate network, for downloading malicious content on to a PC or sucking up data as soon as it is connected.

Application permissions

In the form of a pop up, the user may see these notifications as a nuisance, a delay in accessing the newly downloaded Angry Birds levels. Or they may simply not understand the nature of the requests. Common permissions that may (read: should!) raise an eyebrow would include 'Read/Send SMS', 'Access Fine Location', 'Access IMEI, phone identity', 'Brick' (required to disable the device in trace and wipe apps), 'Access camera', and so on. Such requests may be integral to functionality, but could equally be recording calls and transmitting sign-in credentials.

Malicious application injections

Data/process transfers between virtualised application environments are handled by a protocol of implicit and explicit intents. Transmission or interception of an intent by a malicious application can result in data being compromised as the target app will respond to the string, potentially resulting in data loss.

Third party applications

One of the great things about Android is choice in terms of standard functionality, such as address books, messaging, keyboards, etc. I'm sure no one in the information security industry would need an explanation as to why it might not be a good idea to use an untrusted third party keyboard or password manager. In a rapidly growing OS environment it can be difficult to identify reputable vendors, and considering the nature of the Android community, can you trust a bedroom programmer with user's credentials? Even reputable services can get mobile applications wrong, both Facebook8 and Twitter9 transmit mobile app data in the clear, i.e. without encryption, on nearly all devices. This happens despite the development of such security measures for web app versions.

Rooting

Rooting an Android device is akin to jail-breaking an iPhone, it opens out additional functionality and services to users. The process of gaining root access, requires the device to be switched from S-On to S-Off (where S = security). Additionally, root is a common exploit used by malicious applications to gain system-level access to user's Android. DroidKungFu is one such threat that can root a system and install applications at that level, it escapes detection by utilising encryption and decryption to deliver a payload.

Wi-Fi

The vulnerability of Android devices running 2.3.3 to compromise on unprotected Wi-Fi networks apparently came as a surprise to many11 – it shouldn't have, when is this practice ever safe?! Beyond highlighting the need for better consumer security awareness, it leads to some other considerations around secure Wi-Fi access. Ideally sign in credentials should always be completed over a secured network, but sometimes this isn't enough. FaceNiff is an easily downloadable application that allows the user to intercept the social networking logins of any Android on their network12. The only way this exploit won't work is if the user is utilising SSL. Furthermore, devices running 2.3 (or rooted older

devices) can act as a Wi-Fi hotspot – as an Information Security Manager, how happy would you be about unverified users and devices connecting to a smartphone with a corporate footprint?

Privacy

By default, HTC devices geo-tag photos and Tweets. This is the primary issue with Android as a consumer device – functionality over security. Other applications claiming localised services could utilise GPS permissions for location tracking.

Manufacturer trust

Whatever their intentions, manufacturers play a significant role in user privacy. Uncovered recently is an application that sits at root level on new HTC devices (Evo, Evo 3D, Thunderbolt, Sensation) which collects and transmits a range of information on users including accounts, phone numbers, SMS, system logs, GPS locations, IP addresses, and installed apps14. It is bad enough that HTC feel it is appropriate to collect and use this data without notifying the user, it is even worse that it failed to secure it! Consequently any app with the 'Access internet' permission can access this data.

3.0 Solutions for Security issues of Android OS

3.1 Solutions for Android Developers

Android has security features built into the operating system that significantly reduce the frequency and impact of application security issues. The system is designed so you can typically build user's apps with default system and file permissions and avoid difficult decisions about security.

Some of the core security features that help you build secure apps include:

- The Android Application Sandbox, which isolates user's app data and code execution from other apps.
- An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure IPC. Technologies like ASLR, NX, ProPolice, safe_iop,

OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap_min_addr to mitigate risks associated with common memory management errors.

- An encrypted file system that can be enabled to protect data on lost or stolen devices.
- User-granted permissions to restrict access to system features and user data.
- Application-defined permissions to control application data on a per-app basis.
- Nevertheless, it is important that you be familiar with the Android security best practices in this document. Following these practices as general coding habits will reduce the likelihood of inadvertently introducing security issues that adversely affect your users.
- Storing Data

The most common security concern for an application on Android is whether the data that you save on the device is accessible to other apps. There are three fundamental ways to save data on the device:
Using internal storage By default, files that you create on internal storage are accessible only to user's app. This protection is implemented by Android and is sufficient for most applications.

You should generally avoid using the MODE_WORLD_WRITEABLE or MODE_WORLD_READABLE modes for IPC files because they do not provide the ability to limit data access to particular applications, nor do they provide any control on data format. If you want to share user's data with other app processes, you might instead consider using a content provider, which offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.

To provide additional protection for sensitive data, you might choose to encrypt local files using a key that is not directly accessible to the application. For example, a key can be placed in a KeyStore and protected with a user password that is not stored on the device. While this does not protect data from a root compromise that can monitor the user inputting the password, it can provide protection for a lost device without file system encryption.

Using external storage

Files created on external storage, such as SD Cards, are globally readable and writable. Because external storage can be removed by the user and also modified by any application, you should not store sensitive information using external storage.

As with data from any untrusted source, you should perform input validation when handling data from external storage. We strongly recommend that you not store executables or class files on external storage prior to dynamic loading. If user's app does retrieve executable files from external storage, the files should be signed and cryptographically verified prior to dynamic loading.

Using content providers

Content providers offer a structured storage mechanism that can be limited to user's own application or exported to allow access by other applications. If you do not intend to provide other applications with access to yourContentProvider, mark them as android:exported=false in the application manifest. Otherwise, set theandroid:exported attribute "true" to allow other apps to access the stored data.
When creating a ContentProvider that will be exported for use by other applications, you can specify a singlepermission for reading and writing, or distinct permissions for reading and writing within the manifest. We recommend that you limit user's permissions to those required to accomplish the task at hand. Keep in mind that it's usually easier to add permissions later to expose new functionality than it is to take them away and break existing users.

If you are using a content provider for sharing data between only user's own apps, it is preferable to use theandroid:protectionLevel attribute set to "signature" protection. Signature permissions do not require user confirmation, so they provide a better user experience and more controlled access to the content provider data when the apps accessing the data are signed with the same key.

Using Permissions

We recommend minimizing the number of permissions that user's app requests Not having access to sensitive permissions reduces the risk of inadvertently misusing those permissions, can improve user adoption, and makes user's app less for attackers. Generally, if a permission is not required for user's app to function, do not request it.

If it's possible to design user's application in a way that does not require any permissions, that is preferable. For example, rather than requesting access to device information to create a unique identifier, create a GUID for user's application (see the section about Handling User Data). Or, rather than using external storage (which requires permission), store data on the internal storage.

In addition to requesting permissions, user's application can use the <permissions> to protect IPC that is security sensitive and will be exposed to other applications, such as a ContentProvider. In general, we recommend using access controls other than user confirmed permissions where possible because permissions can be confusing for users. For example, consider using the signature protection level on permissions for IPC communication between applications provided by a single developer.

Do not leak permission-protected data. This occurs when user's app exposes data over IPC that is only available because it has a specific permission, but does not require that permission of any clients of it's IPC interface. More details on the potential impacts, and frequency of this type of problem is provided in this research paper published at USENIX: http://www.cs.be rkeley.edu/~afelt/felt_usenixsec2011.pdf
Creating Permissions Generally, you should strive to define as few permissions as possible while satisfying user's security requirements. Creating a new permission is relatively uncommon for most applications, because the system-defined permissions cover many situations. Where appropriate, perform access checks using existing permissions.
If you must create a new permission, consider whether you can accomplish user's task with a "signature" protection level. Signature permissions are transparent to the user and only allow access by applications signed by the same developer as application performing the permission check. If you create permission with the "dangerous" protection level, there are a number of complexities that you need to consider:

- The permission must have a string that concisely expresses to a user the security decision they will be required to make.
- The permission string must be localized to many different languages.
- Users may choose not to install an application because a permission is confusing or perceived as risky.
- Applications may request the permission when the creator of the permission has not been installed.
- Each of these poses a significant non-technical challenge for you as the developer while also confusing user's users, which is why we discourage the use of the "dangerous" permission level.

Using Networking

Network transactions are inherently risky for security, because it involves transmitting data that is potentially private to the user. People are increasingly aware of the privacy concerns of a mobile device, especially when the device performs network transactions, so it's very important that user's app implement all best practices toward keeping the user's data secure at all times.

Using IP Networking

Networking on Android is not significantly different from other Linux environments. The key consideration is making sure that appropriate protocols are used for sensitive data, such as HttpsURLConnection for secure web traffic. We prefer use of HTTPS over HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on networks that are not secured, such as public Wi-Fi hotspots.

Authenticated, encrypted socket-level communication can be easily implemented using the SSLSocket class. Given the frequency with which Android devices connect to unsecured wireless networks using Wi-Fi, the use of secure networking is strongly encouraged for all applications that communicate over the network.

We have seen some applications use localhost network ports for handling sensitive IPC. We discourage this approach since these interfaces are accessible by other applications on the device. Instead, you should use an Android IPC mechanism where authentication is possible such as with a Service. (Even worse than using loopback is to bind to INADDR_ANY since then user's application may receive requests from anywhere.) Also, one common issue that warrants repeating is to make sure that you do not trust data downloaded from HTTP or other insecure protocols. This includes validation of input in WebView and any responses to intents issued against HTTP.

Using Telephony Networking

The SMS protocol was primarily designed for user-to-user communication and is not well-suited for apps that want to transfer data. Due to the limitations of SMS, we strongly recommend the use of Google

Cloud Messaging (GCM) and IP networking for sending data messages from a web server to user's app on a user device.

Beware that SMS is neither encrypted nor strongly authenticated on either the network or the device. In particular, any SMS receiver should expect that a malicious user may have sent the SMS to user's application—Do not rely on unauthenticated SMS data to perform sensitive commands. Also, you should be aware that SMS may be subject to spoofing and/or interception on the network. On the Android-powered device itself, SMS messages are transmitted as broadcast intents, so they may be read or captured by other applications that have the READ_SMS permission.

Performing Input Validation

Insufficient input validation is one of the most common security problems affecting applications, regardless of what platform they run on. Android does have platform-level countermeasures that reduce the exposure of applications to input validation issues and you should use those features where possible. Also note that selection of type-safe languages tends to reduce the likelihood of input validation issues.

If you are using native code, then any data read from files, received over the network, or received from an IPC has the potential to introduce a security issue. The most common problems are buffer overflows, use after free, andoff-by-one errors. Android provides a number of technologies like ASLR and DEP that reduce the exploitability of these errors, but they do not solve the underlying problem. You can prevent these vulneratbilities by careful handling pointers and managing buffers.

Dynamic, string based languages such as JavaScript and SQL are also subject to input validation problems due to escape characters and script injection.

If you are using data within queries that are submitted to an SQL database or a content provider, SQL injection may be an issue. The best defense is to use parameterized queries, as is discussed in the above section aboutcontent providers. Limiting permissions to read-only or write-only can also reduce the potential for harm related to SQL injection.

If you cannot use the security features above, we strongly recommend the use of well-structured data formats and verifying that the data conforms to the expected format. While blacklisting of characters or

character-replacement can be an effective strategy, these techniques are error-prone in practice and should be avoided when possible.

Handling User Data

In general, the best approach for user data security is to minimize the use of APIs that access sensitive or personal user data. If you have access to user data and can avoid storing or transmitting the information, do not store or transmit the data. Finally, consider if there is a way that user's application logic can be implemented using a hash or non-reversible form of the data. For example, user's application might use the hash of an an email address as a primary key, to avoid transmitting or storing the email address. This reduces the chances of inadvertently exposing data, and it also reduces the chance of attackers attempting to exploit user's application.

If user's application accesses personal information such as passwords or usernames, keep in mind that some jurisdictions may require you to provide a privacy policy explaining user's use and storage of that data. So following the security best practice of minimizing access to user data may also simplify compliance.

You should also consider whether user's application might be inadvertently exposing personal information to other parties such as third-party components for advertising or third-party services used by user's application. If you don't know why a component or service requires a personal information, don't provide it. In general, reducing the access to personal information by user's application will reduce the potential for problems in this area.

If access to sensitive data is required, evaluate whether that information must be transmitted to a server, or whether the operation can be performed on the client. Consider running any code using sensitive data on the client to avoid transmitting user data.

Also, make sure that you do not inadvertently expose user data to other application on the device through overly permissive IPC, world writable files, or network sockets. This is a special case of leaking permission-protected data, discussed in the Requesting Permissions section.

If a GUID is required, create a large, unique number and store it. Do not use phone identifiers such as the phone number or IMEI which may be associated with personal information. This topic is discussed in more detail in theAndroid Developer Blog.

Be careful when writing to on-device logs. In Android, logs are a shared resource, and are available to an application with the READ_LOGS permission. Even though the phone log data is temporary and erased on reboot, inappropriate logging of user information could inadvertently leak user data to other applications.

Using WebView

Because WebView consumes web content that can include HTML and JavaScript, improper use can introduce common web security issues such as cross-site-scripting (JavaScript injection). Android includes a number of mechanisms to reduce the scope of these potential issues by limiting the capability of WebView to the minimum functionality required by user's application.

If user's application does not directly use JavaScript within a WebView, do *not* call setJavaScriptEnabled(). Some sample code uses this method, which you might repurpose in production application, so remove that method call if it's not required. By default, WebView does not execute JavaScript so cross-site-scripting is not possible.

Use addJavaScriptInterface() with particular care because it allows JavaScript to invoke operations that are normally reserved for Android applications. If you use it, expose addJavaScriptInterface() only to web pages from which all input is trustworthy. If untrusted input is allowed, untrusted JavaScript may be able to invoke Android methods within user's app. In general, we recommend exposing addJavaScriptInterface()only to JavaScript that is contained within user's application APK.

If user's application accesses sensitive data with a WebView, you may want to use the clearCache() method to delete any files stored locally. Server-side headers like no-cache can also be used to indicate that an application should not cache particular content.

Handling Credentials

In general, we recommend minimizing the frequency of asking for user credentials—to make phishing attacks more conspicuous, and less likely to be successful. Instead use an authorization token and refresh it. Where possible, username and password should not be stored on the device. Instead, perform initial authentication using the username and password supplied by the user, and then use a short-lived, service-specific authorization token.

Services that will be accessible to multiple applications should be accessed using AccountManager. If possible, use the AccountManager class to invoke a cloud-based service and do not store passwords on the device. After using AccountManager to retrieve an Account, CREATOR before passing in any credentials, so that you do not inadvertently pass credentials to the wrong application. If credentials are to be used only by applications that you create, then you can verify the application which accesses the AccountManager using checkSignature(). Alternatively, if only one application will use the credential, you might use a KeyStore for storage.

Using Cryptography

In addition to providing data isolation, supporting full-filesystem encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using cryptography.

In general, try to use the highest level of pre-existing framework implementation that can support user's use case. If you need to securely retrieve a file from a known location, a simple HTTPS URI may be adequate and requires no knowledge of cryptography. If you need a secure tunnel, consider using HttpsURLConnection orSSLSocket, rather than writing user's own protocol.

If you do find yourself needing to implement user's own protocol, we strongly recommend that you *not* implement user's own cryptographic algorithms. Use existing cryptographic algorithms such as those in the implementation of AES or RSA provided in the Cipher class.

Use a secure random number generator, SecureRandom, to initialize any cryptographic keys, KeyGenerator. Use of a key that is not generated with a secure random number generator significantly weakens the strength of the algorithm, and may allow offline attacks. If you need to store a

key for repeated use, use a mechanism like KeyStore that provides a mechanism for long term storage and retrieval of cryptographic keys.

Using Interprocess Communication

Some apps attempt to implement IPC using traditional Linux techniques such as network sockets and shared files. We strongly encourage you to instead use Android system functionality for IPC such as Intent, Binder orMessenger with a Service, and BroadcastReceiver. The Android IPC mechanisms allow you to verify the identity of the application connecting to user's IPC and set security policy for each IPC mechanism.

Many of the security elements are shared across IPC mechanisms. If user's IPC mechanism is not intended for use by other applications, set the android:exported attribute to "false" in the component's manifest element, such as for the <service> element. This is useful for applications that consist of multiple processes within the same UID, or if you decide late in development that you do not actually want to expose functionality as IPC but you don't want to rewrite the code.
If user's IPC is intended to be accessible to other applications, you can apply a security policy by using the<permission> element. If IPC is between user's own separate apps that are signed with the same key, it is preferable to use "signature" level permission in the android:protectionLevel.

Using intents

Intents are the preferred mechanism for asynchronous IPC in Android. Depending on user's application requirements, you might use sendBroadcast(), sendOrderedBroadcast(), or an explicit intent to a specific application component.

Note that ordered broadcasts can be "consumed" by a recipient, so they may not be delivered to all applications. If you are sending an intent that must be delivered to a specific receiver, then you must use an explicit intent that declares the receiver by name intent.

Senders of an intent can verify that the recipient has a permission specifying a non-Null permission with the method call. Only applications with that permission will receive the intent. If data within a broadcast intent may be sensitive, you should consider applying a permission to make sure that malicious

applications cannot register to receive those messages without appropriate permissions. In those circumstances, you may also consider invoking the receiver directly, rather than raising a broadcast.

Using services

A Service is often used to supply functionality for other applications to use. Each service class must have a corresponding declaration in its manifest file. By default, services are not exported and cannot be invoked by any other application. However, if you add any intent filters to the service declaration, then it is exported by default. It's best if you explicitly declare theandroid:exported attribute to be sure it behaves as you'd like. Services can also be protected using theandroid:permission attribute. By doing so, other applications will need to declare a corresponding <uses-permission> element in their own manifest to be able to start, stop, or bind to the service.

Using Binder or Messenger is the preferred mechanism for RPC-style IPC in Android. They provide a well-defined interface that enables mutual authentication of the endpoints, if required. We strongly encourage designing interfaces in a manner that does not require interface specific permission checks. Binder and Messenger objects are not declared within the application manifest, and therefore you cannot apply declarative permissions directly to them. They generally inherit permissions declared in the application manifest for the Service or Activity within which they are implemented. If you are creating an interface that requires authentication and/or access controls, those controls must be explicitly added as code in the Binder or Messenger interface.

3.2 Solutions for Users

1. **Avoid Rooting -**It is popular for Android users to “root” their phone, a process that allows users to bypass Android security and unlock their phones, gaining access to settings and features often blocked. Do not do this. It reduces the Android security on user’s phone and ends user’s Android support warranty, preventing you from getting Android support from Google or the manufacturer.
2. **Avoid Side loading -**Some Android users choose to “side load” apps and programs that the Google Play Store does not offer and for which Google does not provide Android support. These apps usually come from unofficial Internet sources. Installing these apps is a major risk as many of them contain malware. Side loaded apps account for most attacks on Android security.

3. **Educate Yourself about App Permissions -**Each App you run on Android needs specific permissions to execute certain features. Know what permissions user's apps need and when an app is asking for permissions it should not. This will prevent Android security risks such as apps accessing personal information and sending it elsewhere. If you need help understanding permissions, you should seek Android support.
4. **Use the Android Browser or Google Chrome -**The specially designed Android browser and Google Chrome provide the best Android security when accessing the web. Studies show that third-party browsers are more exploitable and can be Android security risks. There is also less Android support for third party browsers.
5. **Keep user's Operating System Updated -**Because of software fragmentation—cell phone manufacturer's not releasing updates or providing Android support regularly-- this can be difficult. However, always install new updates when they are available. Each new version of Android includes new, more powerful Android security features.
6. **Backup user's Data -**Many of Rescuecom's Android support calls involve data recovery because a customer did not have proper backup or android support.

4.0 Conclusion

In these days, Android has become a very popular operating system for smart phones. There are some advanced features in android Smartphone, with which user can easily share applications via online market store i.e. Google market store. But, there are attacks and threats include in this platform, like malware applications are also attack on Android actual applications. Because malware on device can create number of risks, which creates problem while connectivity because of security issues. In this paper, it will be described that how security can be improve of Android Operating System so that users can safely used the android smart phones.

Elaborating on the major components that comprise the Android operating environment, this report focused on providing a comprehensive overview of the status quo. The very impressive, rapid evolution of Android resembles the great work done by the Linux community over the years. As discussed in this report, the android architecture has been discussed in a detail way and ideantified several security issues in using android devices and found some solutions for overcoming the security issues for both users and developers.

With such a rapidly developing environment, both in terms of product innovation and the threat landscape, other security considerations will rapidly develop in the months and years to come. The measures discussed in this text serve as a good starting point in providing a baseline of security on Android devices. The preferable solution would theoretically be not to allow personal devices on to the network at all, and this may prove an effective if sometimes unpopular decision. The risk-reward ratio is never going to be appealing to a security professional, however this is one of the lesser concerns amongst users. There is no one-stop effective security measure that can be implemented on an Android device. Certainly when it comes to corporate devices then one of the emerging products provides some much needed functionality to the mobile security tool kit. These solutions however are difficult for organizations to implement on personal devices, and don't really provide an effective solution on an individual handset. As a user then many of the actions described here can provide comparable functionality and protection. In the absence of a holistic solution then the enterprise or user must create a comprehensive suite of security controls and applications. The challenge here is maintaining that balance whereby security is seen as an enabler and does not impact too significantly everyday use of the device – failure to do so will lead to circumnavigation of security controls. As part of security education and awareness it would be advisable to discuss some of the core security implications associated with a Smartphone. Providing a suite of tools which can be installed on to a device, or offering an encrypted preloaded SD card, will ensure that exponential growth in mobile malware does not affect user's organization. A company is only as secure as the weakest supplier or user, and mobile devices create all kinds of opportunities for malicious activity – for cybercriminals the path of least resistance is going to be the most tempting, and in such a new technology area there are plenty of potential exploits and attack vectors, both known and unknown, to take advantage of.

References

[01] Google Inc.: Android Repository. http://android.git.kernel.org. Version: 2010

[02] Open Handset Alliance: Open Handset Alliance website. http://www.openhandsetalliance.com

[03] Android Wikipedia, 2013

[04] Johnson, “Performance Tuning for Linux Servers”, IBM Press, 2005

[05] Liang, “System Integration for the Android Operating System”, National Taipei University, 2010

[06] GOOGLE INC. (Hrsg.): Android Software Development Kit (SDK). Google Inc., http://developer.android.com/sdk/index.html. – Android 2.2, Release 2

[07] Bornstein, D., “Dalvik VM Internals”, Google I/O Developer Conference, 2008

[08] Heger, D., “Quantifying IT Stability – 2nd Edition, Instant Publisher, 2010

[09] www.zdnet.com .

[10] www.eweek.com

[11] Ryn Farmer, “A brief guide to Android”

[12] Dominique A. Heger, “**Mobile Devices - An Introduction to the Android Operating Environment Design, Architecture, and Performance Implications”**, DHTechnologies (DHT)

[13] Prof. Dr. Frank Bellosa**. “Analysis of the Android Architecture”,** Bearbeitungszeit: 2. June 2010– 6. October 2010