Improving security Solutions in modern Smartphones

Fredrik Bugge Lyche
Jørgen Haukedal Lytskjold

Master of Science in Communication Technology
Submission date: June 2014
Supervisor: Van Thanh Do, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title:	Improving security solutions in modern smartphones
Student:	Fredrik Bugge Lyche & Jørgen Haukedal Lytskjold

Problem description:

Smartphones have become immensely popular in very short time. It has enabled on-the-go use of features normally associated with computers, making communication and workflow happen at an immediate pace. The quick adoption rate, however, has given opportunities for malicious individuals who look to exploit this vast new market, resulting in a rapid increase of smartphone related threats and attacks. The smartphone, being a relatively young technological device, has features that still need to be fully developed, and one of these is security.

The main goal of this thesis is to assess what improvements can be made to security in smartphones both in general and the ones running the Android OS in particular. In addition, existing protective measures will be identified and evaluated. To achieve the described goals, the most prevalent attacks will also be studied thoroughly.

Responsible professor:	Do van Thanh, ITEM
Supervisor:	Do van Thanh, ITEM

Abstract

Since the introduction of modern smartphones in 2007, their popularity has soared, and today smartphones are ubiquitous. People all over the world have embraced new ways to work, shop and socialize by means of their smartphone. Resulting from this is that all sorts of personal and sensitive data is communicated through, and stored on the devices. Whether the data is business documents or payment card details, it results in the smartphone being an attractive target for maliciously intended people.

This thesis aims to identify new solutions that may improve smartphone security. It provides fundamental knowledge of the Android platform to help understand the attacks and security measures discussed later. Both system architecture and application structure is covered. The existing security solutions included utilize many of the aspects from the system architecture, but also consist of external mechanisms, like the application vetting.

To determine what new security solutions to suggest, the most prominent and notorious attacks have been examined. Trojans, attacks through advertisement and attacks through WebView were identified as the greatest threats. The information acquired from assessing these was also used for evaluating the effectiveness of the proposed security measures. As a means of quantifying the evaluation of the new measures, a threat model has been used.

The thesis concludes with determining the effect the new features will have for different types of smartphone users, and also ranks the suggested features according to which were deemed most influential.

Sammendrag

Siden den moderne smarttelefonens inntog i 2007, har dens popularitet økt voldsomt og ført til at smarttelefoner i dag er allemannseie. Mennesker over hele verden har tatt i bruk nye måter å jobbe, handle og sosialisere på ved hjelp av smarttelefonen. Et resultat av dette er at mye sensitiv data kommuniseres gjennom og lagres på enheten. Hvorvidt det gjelder dokumenter som er sensitive for en bedrift, eller detaljer for et betalingskort, er konsekvensen at smarttelefoner utgjør et attraktivt mål for angripere.

Oppgavens mål er å identifisere nye løsninger som kan forbedre sikkerheten for smarttelefonbrukere i dag. For å gjøre de angrep og sikkerhetstrusler som diskuteres mer forståelige, presenteres sentrale elementer ved og i Androids plattform, som for eksempel systemarkitektur og applikasjonsstruktur. De eksisterende sikkerhetsløsningene som er omhandlet i oppgaven utnytter mange av aspektene ved systemarkitekturen for å fungere godt, men består også av eksterne mekanismer, som for eksempel undersøkingen av alle applikasjoner som legges ut på den offisielle markedsplassen.

For å avgjøre hvilke nye sikkerhetsløsninger som skulle foreslås ble de mest dominerende og beryktede angrepene undersøkt. Trojanere, angrep gjennom reklame, samt angrep gjennom WebViews ble funnet å være de største truslene. Analysen av disse ga grunnlag for evalueringen av effektiviteten til de foreslåtte sikkerhetsløsningene. En trusselmodell ble også brukt i denne sammenheng, med formål å gi kvantifiserbare data.

Oppgaven avslutter med å avgjøre hvilken effekt de nye løsningene kan ha for ulike brukere av smarttelefoner, samt en rangering av løsningene på bakgrunn av deres totale innvirkning på dagens sikkerhetssituasjon.

Preface

This report is the result of a master's thesis in communications technology, at the Department of Telematics at the Norwegian University of Science and Technology (NTNU). Our supervisor has been Professor Do van Thanh. The targeted readers are those who wish to learn more about how security is employed to mitigate the most prominent threats to smartphones today. No particular prerequisite knowledge should be required, albeit an interest for the topic should provide for an easier read.

We would like to thank our supervisor Do van Thanh for guidance and valuable feedback. We are also thankful for the feedback and insightful comments given by our co-students, and the security researchers that took time out of their schedule to discuss smartphone security with us.

Fredrik Bugge Lyche &
Jørgen Haukedal Lytskjold
Trondheim, Wednesday 11th June, 2014

Contents

List of Figures

Glossary

Address Space Layout Randomization Address Space Layout Randomization randomly arranges the position of key data in memory to protect against buffer overflow attacks.

API Application Programming Interface is a set of rules and tools for making software applications.

APK Android application package file. Each Android application is compiled and packaged in a single file that includes all of the application's code (.dex files), resources, assets, and manifest file. The application package file can have any name but must use the .apk extension, e.g. myExampleAppname.apk. For convenience, an application package file is often referred to as an APK.

bloatware Bloatware is software pre-installed on a device that is deemed unnecessary by the user. It is often not possible to remove these applications without elevated privileges..

C&C server Short for Command and Control server. These servers are centralized machines that are able to send commands and receive outputs of machines that are part of a botnet. Any time an attacker wishes to launch a DDoS attack, he/she can send special commands to their botnet's C&C servers with instructions to perform an attack on a particular target, and any infected machines communicating with the contacted C&C server will comply by launching a coordinated attack.

CSP CSP is short for Communications Service Provider. A CSP is a service provider that transports information electronically, be it in the telecom, Internet, cable or satellite businesses.

DNS Domain name Server is any computer registered in a Domain Name System (a standard for managing names for web sites).

IETF Internet Engineering Task Force develops and promotes internet standards.

IFRAME The `<iframe>` specify an inline frame used to embed another document within the current HTML document.

IMEI International Mobile Station Equipment Identify is a global identifier number used for mobile communication.

IPC Inter-Process Communication is methodes for data exchange between multiple threads in a process.

malware The term malware is a merging of the words 'malicious' and 'software'.

NFC Near Field Communication is a type of communication used by smartphones (and some other similar devices). It is a radio communication that requires very close proximity.

No eXecute No eXecute is used by CPUs to protect against buffer overflow attacks.

RAM Random Access Memory is a type of computer memory that stores information in a random order. It is often used as a device's main memory.

ransomware Ransomware is a form of malware where the infected device or its data is held ransomed by an attacker by encrypting it. To free the device or its data, the owner is asked to pay a ransom. When paid, the cryptographic key needed to unlock the device is provided. Known occurrences have shown some ransomware to fulfil the agreement, but some don't.

risk Risk, as defined by the International Organization for Standardization (ISO), is the product of likelihood and possible impact.

SD card Secure Digital card is a memory card formate used by portable devices.

SDK Software Development Kit is a set of development tools used to create applications for software frameworks, computer systems, hardware platforms and similar systems.

SIM Subscriber Identity Module is an integrated circuit used to identify and authenticate subscribers using a mobile device.

Snort Snort is network intrusion detection and prevention system. It is open sourced.

SoC SoC is short for System on a Chip. It is an integrated circuit that incorporates all components of a computer or other electronic system into a single chip.

SOP Same-Origin Policy is a concept in web security, and it restricts how documents/scripts in one origin interact with resources from another origin.

URI Uniform Resource Identifier (URI) is a string of characters used to locate a web resource.

URL Uniform Resource Locator is a character string referencing to a resource, and in communication networks and Internet it is better known as a web address.

VM Virtual Machine is a software emulation of a computer.

vulnerability Internet Engineering Task Force (IETF) defines the term vulnerability as: *"A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy"*.

zero-day Zero-day is an attack that uses previously unknown exploits.

List of Acronyms

API Application Programming Interface. *Glossary:* API.

APK Android Application Package. *Glossary:* APK.

AV Antivirus.

BYOD Bring your own device.

C&C server Command and Control server. *Glossary:* C&C server.

CPU Central Processing Unit.

CSP Communications Service Provider. *Glossary:* CSP.

DNS Domain Name Server. *Glossary:* DNS.

GB Gigabyte.

GPS Global Positioning System.

IDS Intrusion Detection System.

IETF Internet Engineering Task Force. *Glossary:* IETF.

IMEI International Mobile Equipment Identity. *Glossary:* IMEI.

IPC Inter-Process Communication. *Glossary:* IPC.

IPS Intrusion Prevention System.

ISO International Organization for Standardization.

.jar Java file.

MB Megabyte.

NFC Near Field Communication. *Glossary:* NFC.

NIDS Network Intrusion Detection System.

OEM Original equipment manufacturer.

OS Operating System.

RAM Random Access Memory. *Glossary:* RAM.

SD card Secure Digital card. *Glossary:* SD card.

SDK Software Development Kit. *Glossary:* SDK.

SIM Subscriber Identity Module. *Glossary:* SIM.

SMS Short Message Service.

SoC System on a Chip. *Glossary:* SoC.

SOP Same-Origin Policy. *Glossary:* SOP.

UI User Interface.

UID User Identifier.

URI Uniform Resource Identifier. *Glossary:* URI.

URL Uniform Resource Locator. *Glossary:* URL.

VM Virtual Machine. *Glossary:* VM.

Chapter 1

Introduction

A smartphone is in many ways the quintessential technical device of our time. Ever since Apple introduced the iPhone in 2007 and defined what we have come to expect of a modern smartphone, popularity has soared. Seven years have passed since, and people all over the world have embraced new ways to work, shop and socialize by means of their smartphone. The extent is such that no businesses have been left unaffected. In short, the ways to utilize a smartphone seem limitless.

However, with a wide array of functionality, comes an equally wide array of vulnerabilities. A smartphone has the capabilities of a computer and phone combined into one, making it a device suitable for both work and private use. The two have previously been largely disjoint, and the overlap provided by the smartphone presents severe challenges to security and privacy. While users may exert caution in business settings, they often let down their guard in off work settings, installing applications of unknown reputation or browsing the far corners of the web. Mixing business and leisure is not the only security threat, though. A smartphone is a highly personalized asset, trusted by users to store information of who their friends are (contacts), what they speak of (messages and chat), their schedule (calendar) and payment details (application store or banking app). The result is a device with a potential value way beyond its retail price. If the information above were to fall into the wrong hands, businesses could buckle or users lose control of their digital identity. The impact is potentially huge, making the importance of security features evident.

So how can phone manufacturers, communications service providers (Communications Service Provider (CSP)s), application developers, or even the users themselves, ensure the smartphone environment is a safe and secure one?

It is a difficult task. The manufacturers are responsible for and control the hardware side of it, the application developers the software side, and the CSPs control the communication to and from a device. Ultimately, it is the users who operate the phone, however, and the way each of them use theirs is harder to predict.

With smartphone penetration worldwide amounting to 1.75 billion devices this year (2014, [1]), it is clear that smartphone users must come in all shapes and colors. From [2] it is seen that Android users are quite evenly spread in the range of ages 18 through 55, albeit slightly skewed towards the young. Additionally, differences in education attainment and household income fail to describe the Android user, as opposed to iPhone users, who are more likely to score high on both statistics.[2] Thus, finding one single persona to embody a typical Android user can not be done. It might be that a solution where each user is responsible for securing their own device could prove effective, with the current solutions also in place, of course. Making sure every user knows how best to provide the additional security is an ungrateful task, however, as the facts above state that it essentially means educating *everyone*.

Even so, the fact remains that more than 95 % of all malware infecting smartphones today (i.e. Trojans), could not be deployed without the user interacting at some point.[3]. With users unable to provide security to their own devices, as stated above, that leaves improving smartphone security to the hardware manufacturers and software developers. Their task is then to make sure the user environment is as safe as possible, keeping the users from being able to accept and run malicious code on their devices.

Smartphone security is still a relatively young field of study, but has in many ways surpassed regular computers' security features already. A lot has been learned from decades of security work with computers, and this experience has been brought along when working out security solutions for the smartphone platform. Where computers were not initially designed to be common property, smartphones always were. Thus, measures to safeguard the user on a smartphone permeates the system as a whole, and they help reduce the risk a user is subjected to a great deal.

Yet, numbers provided by Juniper Networks[4] show malware findings in smartphones has increased almost exponentially, growing with 155 % in 2011 to 614 % from March 2012 to March 2013. Sophos reveals the same trends in its mobile security threats report issued this year (2014, [5]), effectively saying that there is room for improvement to the existing solutions in smartphone security.

This report is aimed to answer how.

1.1 Goal and scope of the report

This thesis is meant to shed light on the security mechanisms in place to allow a smartphone owner safe use of his or her device, as well as evaluating additions that may improve the security.

Seeing as Android is the platform targeted by close to all smartphone threats (97 %, [6]), it has naturally been the focal point of our research. Following this, the report in its entirety concern Android powered smartphones in particular, but some solutions (chapter 5) and discussions (chapter 6) may also be valid for other platforms.

As pointed out in the introductory text, smartphones comprise of features from both computers and cellular phones, and thereby inherit strengths and weaknesses of both. Ideally, all such weaknesses would be scrutinized in this thesis, but time constraints have not allowed it.

Hardware attacks were ruled out because they require the attacker to possess the device, which in turn means that the user must have lost their smartphone or that is has been stolen. For most users, the greatest cost of such an attack would be the price of acquiring a new device, and likewise, the thief or finder likely cares more for the value of the smartphone than what it holds. Additionally, there are already measures to protect the data stored on a lost device in place, e.g. screen lock and the option to encrypt data. Besides forcing users to apply features that already exist and making them take better care of their phones, there is not much to be done to mitigate this threat.

Network attacks were also ruled out. There are arguably improvements to be made in the way some smartphones handle wifi connections, but security in the network domain lies primarily outside the connecting device.

The decision made was thus to aim attention at software attacks. Software attacks can be executed from anywhere in the world to anywhere in the world, and are steadily increasing both in complexity and occurrences. The attacks presented in this report is a selection of noteworthy attack vectors that has been chosen based on relevance and notoriety.

These attacks were used as the basis for the new solutions suggested later. The proposed new solutions are not restricted to software solutions, however. Security mechanisms in both hardware and software are represented, as software attacks at some point must be run by the hardware, and thus can be mitigated by it.

1.2 Methodology

The thesis work has not included any experiments or tests, but consisted mainly of study of literature. Each topic explored throughout the report has been thoroughly discussed both internally and with the supervising professor before being included in the finished work. None of the explicit solutions mentioned are the authors' own, but included because they were deemed viable both in their own right and as natural enhancements for today's solutions. Discussions and conclusions in chapters 6 and 7 are based on subjective impressions and opinions.

Smartphone security has proven to be a difficult topic with regards to finding good, scientific research that is not obsolete. Published reports discussing threats and attacks or the state of affairs in smartphone security, are written by interested parties/corporations with stakes in the business. This means some of the numbers and threats may have been exaggerated in order to sell their own security solutions to the reader, resulting in their findings being taken with a pinch of salt.

We attended an information security conference (Paranoia in Oslo, Mars 2014), discussing the topic of smartphone security with renowned researchers like Brendan O'Connor[1] and Corey Nachreiner[2], that provided invaluable insight.

Also, emails have been exchanged with the authors of some of the sources in the reference list. This helped provide deeper knowledge of the material as well as determining whether or how their research had been followed up by the relevant party.

1.3 Thesis chapter outline

To provide a solid understanding of how a smartphone functions, the Android platform is explained in detail in **Chapter 2**. The chapter is meant to provide the reader with general insight into the Android system, expanding on subjects such as the ecosystem, application marketplace solution, Operating System (OS) architecture and data handling. We also introduce a threat model in order to make the attacks and their corresponding countermeasures more easy to follow.

Chapter 3 discusses what measures are in place today to mitigate the threat towards attacks against smartphones. Some measures utilize the architecture explained in chapter 2, and some mechanisms are in place outside the smartphone device itself.

In **Chapter 4**, we present what we believe to be the most noteworthy and notorious attacks today.

Chapter 5 is where new solutions are presented and analysed.

The effectiveness of both existing and new solutions is discussed in **Chapter 6**, before we conclude our findings in **Chapter 7**.

[1]Brendan O'Connor is a security researcher and law student from the US. He has, among other things, developed surveillance systems to demonstrate the far-reaching implications of passive eavesdropping on data exchange that occur constantly between wireless devices.

[2]Corey Nachreiner has written thousands of security warnings and articles for his employer Watchguard during the last 15 years. His security tutorial videos have generated more than 100.000 views, and he is regularly cited in online media like C|NET, eWeek and Slashdot.

Chapter 2 The Android platform

The purpose of this chapter is to provide the reader with basic knowledge about what parts of the Android system an attacker may exploit. First, the so-called ecosystem is explained, followed by how Android organizes its application marketplace and how the system's structure and architecture is designed. Applications are what separate smartphones from feature phones, and are described next, followed by information on Android's way of storing data. In-app advertisement is central to both threats and security improvements in the thesis, and is explained in some detail here. The chapter is concluded by the introduction of a threat model, which will serve as a visual aid to the explanations and analyses later on. By studying this information, it should become easier to understand how attacks on a smartphone can be executed.

2.1 Ecosystem

Android is an OS originally developed with smartphones in mind, by a team led by Andy Rubin.[7] It was bought by Google in August of 2005,[8] and the first Android device was released in October 2008 with Android OS 1.0.[9] Since then, the Android OS has increased its market share and is now the leading the race with a current market share of 78.4 % worldwide.[10]

A platform's ecosystem refers to the distribution of different OS versions, as well as the selection of devices that run them. The Android platform is used by lots of smartphone (and tablet) manufacturers, the most prominent being Samsung, Sony, LGE, Motorola, HTC and perhaps Google itself. Depicted in figure 2.1, is the distribution of market share among Android manufacturers, according to numbers from mixpanel [11].

Fragmentation is not only an issue relating to devices, but also in OS versions and the rate at which the users update theirs. Google's policy of keeping Android an open, free-for-all platform has led to its superior market share. A consequence of this is the sheer number of distinct devices running Android, which is partly at

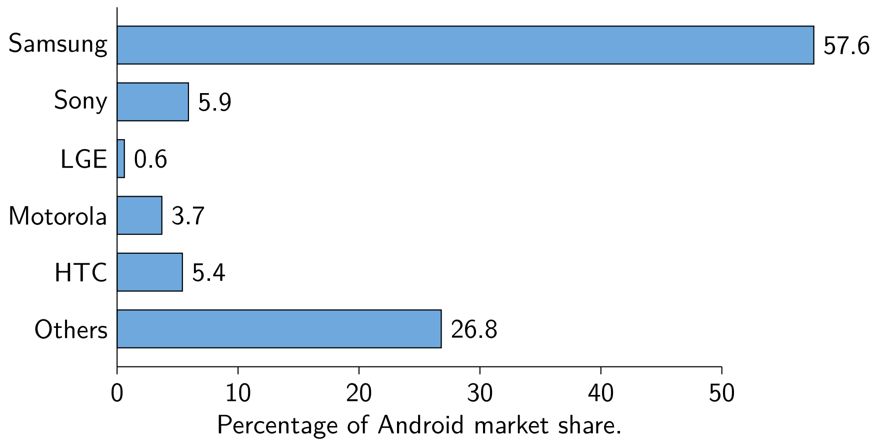


Figure 2.1: Bar chart of market share among manufacturers of Android devices. [11]

fault for the OS fragmentation. Additionally, the manufacturers implement slightly different versions of the Android platform to fit their own devices better. Pushing updates to newer OSs thus happens at a pace set by the manufacturers (and the telecommunication service providers), according to when they are able to port the updated OS to their environment. These aspects contribute to the current status of Android OS fragmentation, as seen in the figure 2.2. In comparison, Apple's similar figures see iOS 7 (the most recent) racking up 87 percent, with the penultimate iOS 6 garnering 11 percent, according to Apple. [12]

Why and how are these statistics relevant to the topic of smartphone security? The answer is that the wide array of Android devices and the OSs they are running make it practically impossible to create an application native to each device's system (hardware and software combination). This in turn effects the amount of vulnerabilities present or attack vectors possible, in applications on each platform.

2.2 Application store solution

Applications are the main feature of smartphones that distinguish them from regular phones. Google have their own centralized marketplace for distribution of applications, namely Google Play Store (formerly Android Market)[1]. Google has been known to embrace an open and available platform, and this is also true for its attitude towards developers uploading to the Google Play Store. The only requirement for uploading applications to the Play Store is an email account, with no proof of identity necessary.

[1]The two names will both be used throughout the report, depending on what time a given incident occurred.

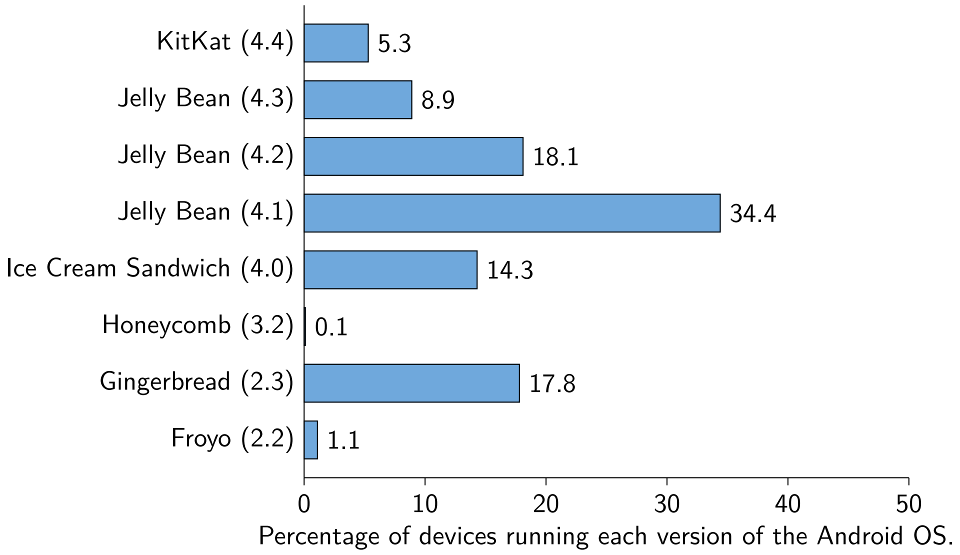


Figure 2.2: Note: Because this data is gathered from the new Google Play Store app, which supports Android 2.2 and above, devices running older versions are not included. However, in August, 2013, versions older than Android 2.2 accounted for about 1 % of devices that *checked in* to Google servers (not those that actually visited Google Play Store).[13]

Google's Play Store boasts the highest amount of applications in its store, with over a million in total.

When an app has been uploaded to Google Play for approval, an automated tool called *Bouncer* performs a check for suspicious activity. If nothing suspicious is discovered, the app becomes available to the public. The *Bouncer* is discussed in more detail in chapter 3.

There are other stores than Google Play Store available to the Android user. All it takes is checking a system setting ('Unknown sources'), thereby allowing for downloads from places other than the Google Play Store, such as the Amazon application store, or other third party stores.

2.3 Structure

Android is by far the most targeted platform for smartphone attacks[3], and one of the main reasons is the open nature of the OS. Google has created an environment that is attractive to developers, but because of its openness, it also has weak points that are being targeted by attackers.[14]

2.3.1 Architecture

The source code of the Android system is open source, and that means it is open to the public and anyone who wants to review, modify or build their own Android system can do that.[2] [15] Though, more recently, Google has slowly stopped their contribution to more and more open-source applications, rather moving development to their own proprietary versions in order to increase revenue and user lock-in.[16] As seen in figure 2.3, the Android system architecture is based on four different layers, each of which are elaborately explained in the next sections.

Kernel

At the core of the Android system is the Linux based kernel that provides hardware interaction for applications (it contains the hardware drivers), as well as managing processes and memory usage.[15] The hardware devices include display, camera, keypad and audio, among others.

Libraries and Android runtime

The second layer is divided into two sections; Libraries and Android Runtime. The Libraries part contains native libraries available for applications, e.g. OpenGL (graphics), SQLite (database) and WebKit (for rendering web pages).[15] The second part contains core libraries and the Dalvik Virtual Machine (VM). Core libraries provide the applications with basic Java functionalities and the Dalvik VM is designed to run applications on a mobile device (i.e. with limited processing power and memory).[15]

Application framework

Applications interacts with this layer directly, which provides a variety of functionalities. It contains some of the basic tools a developer needs to build an application, e.g. Activity Manager (manages application life cycle), Content Provider (manages data sharing between applications), Telephony Manager (manages all voice calls) and Resource Manager (manages the various resources used in applications).

Application

The final layer is where the applications reside, and for an average user this is the layer they interact with the most. Several standard applications are included in the operating system by default, among others; Contact manager, Web Browser and SMS client. The applications are usually written in Java and converted into Dalvik byte code by the Software Development Kit (SDK).[15] By communicating with the

[2] source code is available at http://source.android.com/source/downloading.html

layers previously mentioned, the applications can achieve the functions expected in a smartphone.

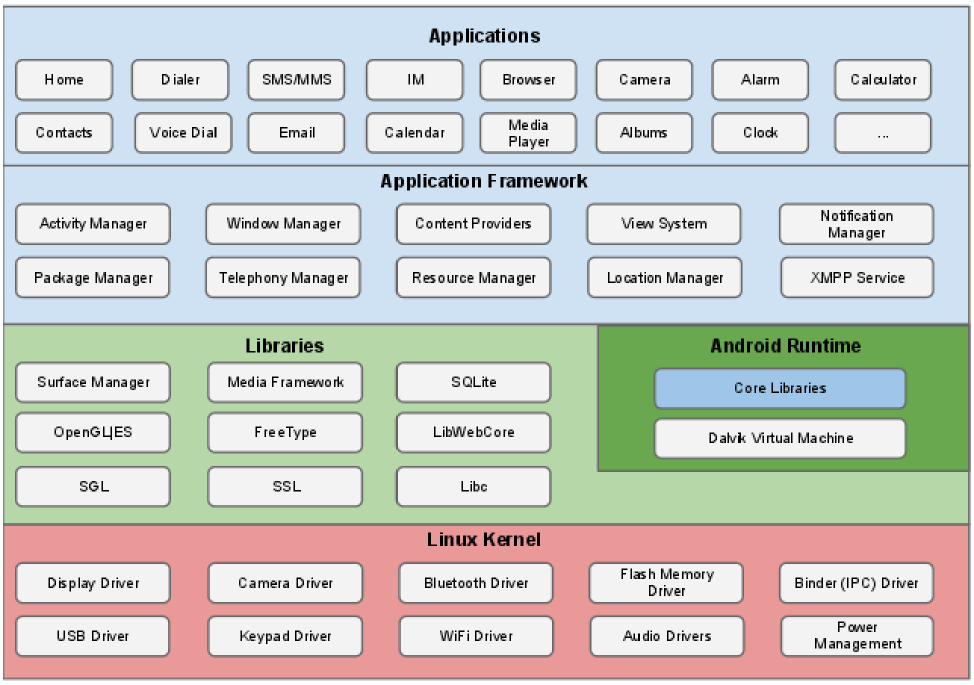


Figure 2.3: Android four level system architecture[17].

2.4 Applications

Application Components

Applications provide the user with all the features normally associated with a smartphone. It is essentially what separates a smartphone from a feature phone. Applications consist of four components: [15]

Activities is the core part of an application. An activity is displayed on the screen (takes care of creating windows) and handles interactions with the user.[18]

Content Providers manage the access to application specific data for other applications and internal components[15] - the standard interface to connect data within one process with code in another process.[19]

Broadcast Receivers receive and respond to intents.[15] (Intents are explained in the next section.)

Service handles long running operations in the background, and does not provide a User Interface (UI).[20]

Application communication

To perform interprocess or intercomponent communication, an asynchronous message known as *intent* is used.[15] Intents provide a runtime binding between code in different applications, and its most prominent use is to launch activities and act as the glue between activities.[21] An example of intent use is illustrated here in figure 2.4.

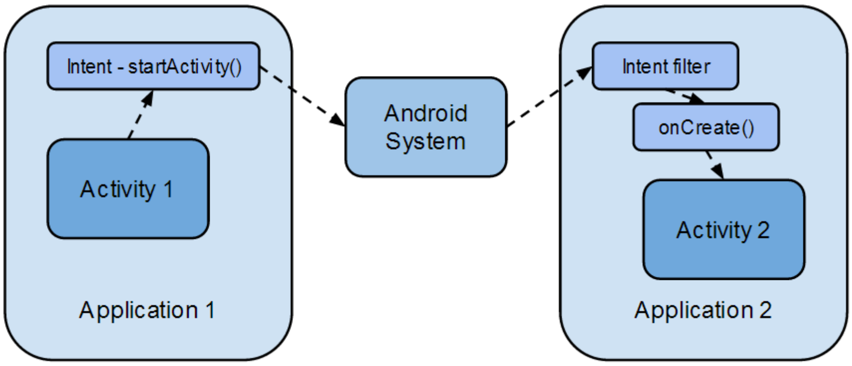


Figure 2.4: Activity 1 launches an intent and passes it to `startActivity()`. It is then passed to Android system, which finds the application (Application 2) with the matching intent filter. Android System then forwards the intent to the appropriate application (Application 2) and the intent then invokes the `onCreate()` method to launch Activity 2.[22]

There are also three additional Inter-Process Communication (IPC) methods used in the Android system, Binder, Service and ContentProvider, but intents are the primary option.

2.5 Data storage

The Android system provides two ways to store data storage; internally and externally.

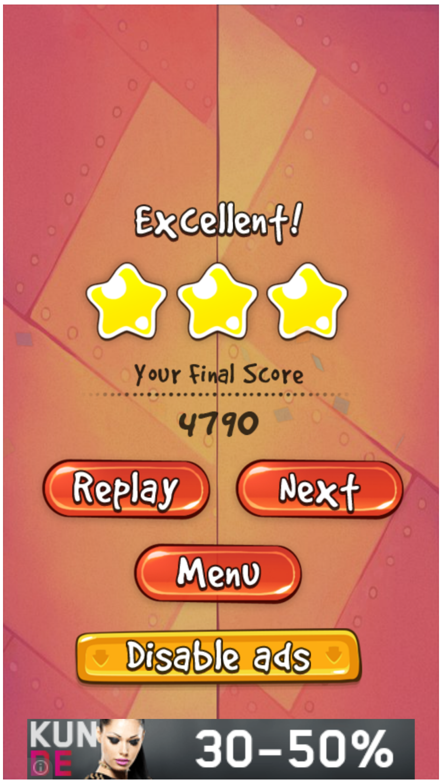
Internal storage is something every application has, and this storage space is only accessible to that specific application.[15] Files stored in this space are private to the application and deleted when the application is uninstalled.[23] The access restriction is based on the application's User Identifier (UID).[15]

External storage is usually in the form of an Secure Digital card (SD card) and often removable.[15] Data stored in external storage are world readable, and all applications have access to read them. Also, it is possible to insert the storage device in another compatible system to access the information.[15]

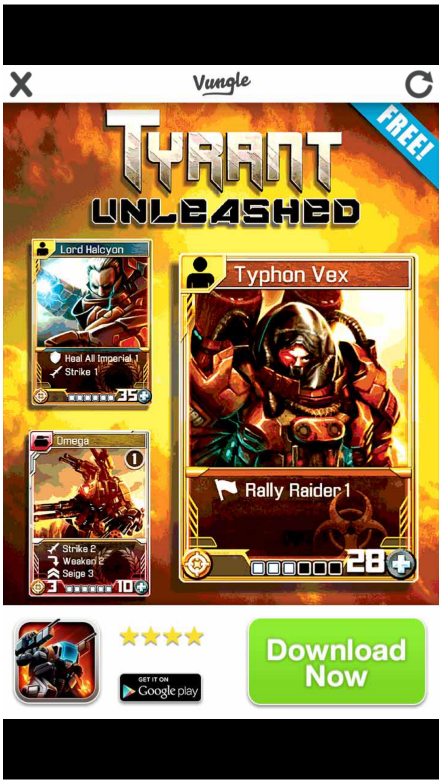
2.6 In-application advertising

A very popular way for developers to get paid for their work is to include advertisements in their applications. By incorporating an advertisement library, they can release the application for free and still generate revenue.[24] The result is some sort of a banner, e.g. at the bottom of the screen as seen in figure 2.5a, or an advertisement covering the whole screen for some seconds, as in figure 2.5b. At run-time, the advertisement library connects with the advertisement server and receives advertisements to be displayed. The advertisement network pays the developers based on the exposure of the advertisement, either per time its displayed, per click or some other measure. [24]

Advertisement is explained quiet thoroughly here, which is because an in-depth understanding of how it works is recommended to better follow the information presented on the subject in later chapters.



(a) Banner advertisement.



(b) Full screen advertisement.

Figure 2.5: Screenshots illustrating the two main types of in-application advertisement.

There are four main participants in the application advertisement process, which are listed below. Figure 2.6 provides a graphical explanation of the interplay.

Advertisers are persons or companies that want to advertise. They contact the Advertisement Publishing Network and provide them with ads to be sent to users.

Publishing Networks receive advertisements from Advertisers and then those ads are given to the Advertisement Libraries to be displayed. Publishing networks are the ones that pay the application developer.

Advertisement Libraries are SDKs incorporated into the application, and used by the developer to request advertisements (from the advertisement publishing networks) and display them. The two primary types of in-application Advertisement Libraries are *mobile web* and *rich media*, both described further below.

Applications host and display the advertisements.

As stated above, there are two primary types of Advertisement Libraries.

Mobile Web libraries work as front end for web based advertisement networks. They are requested from the server and displayed using standard web technologies. [24] An advertisement's display is normally just text or a banner, and the libraries have little interaction with the user's device itself. Mobile web libraries is the most common advertisement library type. [24]

Rich Media libraries are similar to the mobile web library, but include more advanced features. Through the use of powerful Application Programming Interface (API)s for both developers and advertisers, they support more advanced types of advertisements, such as active content like (JavaScript and HTML5), video and interstitial advertisements.[24] Rich media libraries are not as common as mobile web libraries, but some of the most popular Advertisement Libraries use them.[24]

Common for both types of libraries is that they contain user-interface code, used to display the advertisement, and network code, used to request and handle advertisements sent from the server to the device. [24]

2.6.1 Example implementation of AdMob libary

One of the most popular advertisement Publishing Networks is Google's AdMob, which uses the Google Mobile Ads SDK as Advertisement Library. If a developer

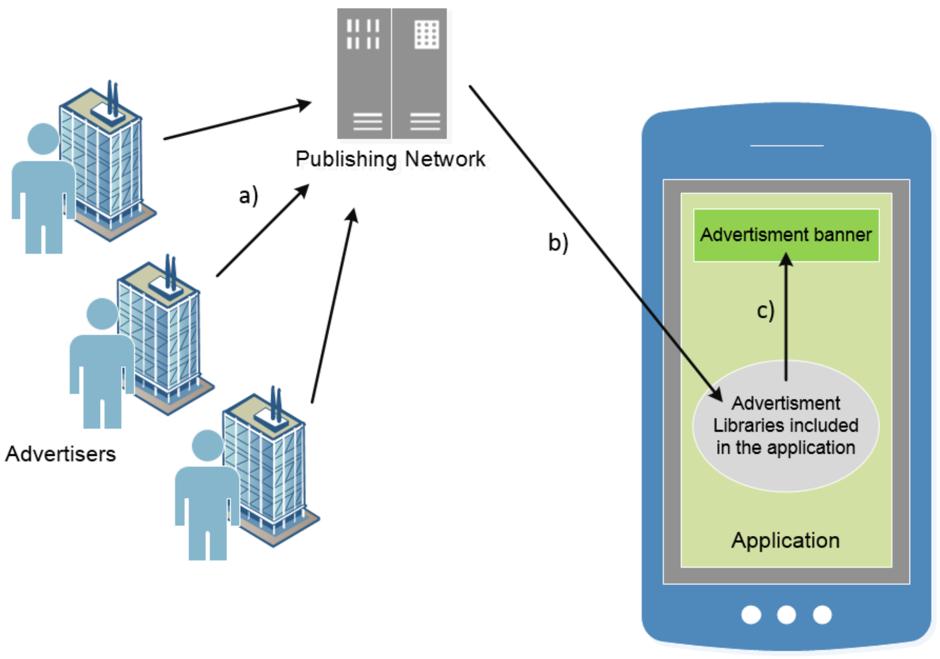


Figure 2.6: Interaction between all the participants in the advertisement process. **a)** Advertisers send ads to the Publishing Network. **b)** The Publishing Network provides ads to Advertisement Libraries. **c)** Advertisement Libraries work together with the Application to display the ad.

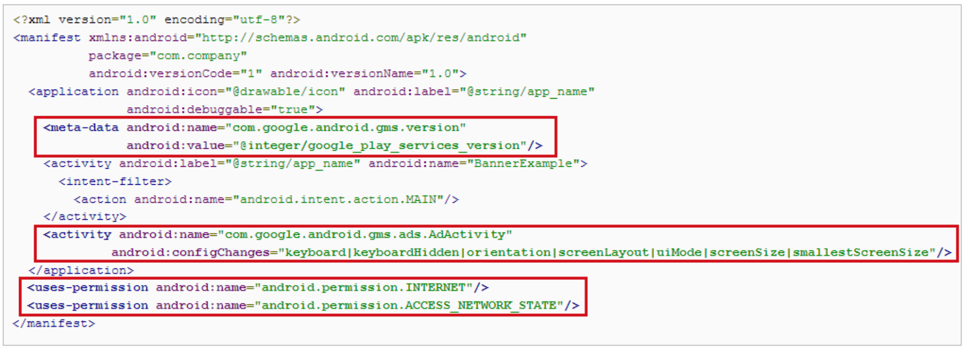
wants to take advantage of the services provided by AdMob, he or she needs to incorporate the Advertisement Library into the relevant application. To accomplish this, some changes to the main application is necessary, the first of which is to modify the manifest file. The following changes are required (also depicted in figure 2.7): [25]

Include a metadata tag The metadata tag is included because it provides an effective and generic way of setting configuration values. It works particularly well with API keys that vary across different applications.[26]

Declare the activity that contains the advertisement logic Every activity in an Android application has to be declared in the manifest file, and so is also the case for the AdMob activity.

Add permissions The permissions needed are INTERNET and ACCESS_NETWORK_STATE, but the last one is optional and used to check the internet connection before making an ad request (for further information regarding permissions, refer to section 3.1).[25]

After editing the manifest file, the developer has to create a new activity (the one declared in the android manifest) where the advertisement logic is implemented. AdMob supplies the activity that contains functions for requesting and displaying the advertisement from the advertisement network. Only minor changes are required by the developer.[27] The activity AdMob provides is included in appendix A.



```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.company"
          android:versionCode="1" android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name"
               android:debuggable="true">
    <meta-data android:name="com.google.android.gms.version"
               android:value="@integer/google_play_services_version"/>
    <activity android:label="@string/app_name" android:name="BannerExample">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
    </activity>
    <activity android:name="com.google.android.gms.ads.AdActivity"
             android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|uiMode|screenSize|smallestScreenSize"/>
  </application>
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
</manifest>
```

Figure 2.7: Example AndroidManifest.xml file with highlighted sections required for in-application advertisement.

2.7 Introducing the Threat Model

The Threat Model, of which an example is shown in figure 2.8, will be used throughout the report to serve as an illustration of how different smartphone security solutions handle different threats and attacks. The model itself is found on the next page.

Each attack has some goals it seeks to achieve, e.g. earning a profit, gathering information and so on. In general, the means through which an attack can attain these intended goals, are specific features, or assets, of the smartphone.

Below is a list of assets, divided into four groups according to what impact an attack that is utilizing them can have:

- **Telephonic features**
  - Read/send SMS
  - Call function
- **User data**
  - Camera
  - Global Positioning System (GPS)
  - Microphone
  - Contacts
  - Email
  - Calendar
  - Phone number
  - SD card
  - International Mobile Equipment Identity (IMEI) number
  - Gyro-/accelerometer
- **Internet communication**
  - Remotely impose instructions (Push option)
  - Remotely retrieve information (Pull option)
- **Computing Power**
  - Central Processing Unit (CPU)

The boldfaced items in the list are the category headers, summing up their underlying content. A smartphone contains a lot of features not included there, but the ones above represent those with the greatest risk of being targeted in an attack. For example, many attackers would probably like to access other applications' internal storage and resources, but the existing security solution (as seen in section 3.1) renders this option so unlikely that the risk is negligible, and thus this target has not been included in the list above.

All attacks and security measures explained from here on out, have distinct versions of the Threat Model figure below (2.8) accompanying them. An important note on this particular figure shown here, is that it displays all possible graphic elements, each of which is explained below the figure.

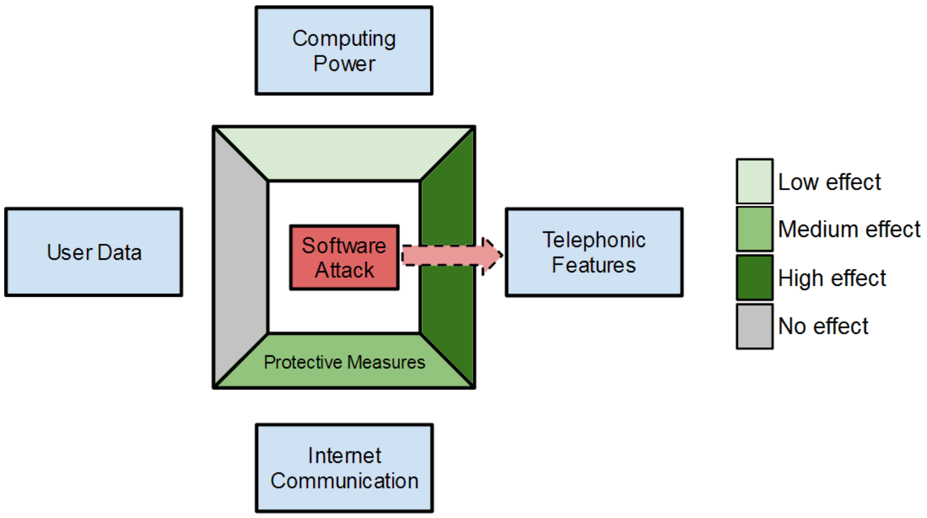


Figure 2.8: The fundamental Threat Model.

An attack class is situated in the middle and is colored red to indicate its malicious nature.

The blue boxes serve as the asset groups.

The red arrow represents an attack from the given attack class towards an asset group (here: Telephonic Features).

The four edges that constitute a frame, symbolize protective measures restricting the attack's access to the smartphone's assets. Each edge is colored according to the protective measure's level of effectiveness in preventing the attack.

NOTE: In later figures, if a direction is not covered by an arrow, all edges in that direction is colored medium green. This means no assessment has been made of an attack from that given class towards the assets in that direction.

Chapter 3 Existing security solutions

There are number of security measures operating in smartphones today. This chapter introduces those employed for the Android platform. First comes the mechanisms related to security that are available through the OS. Second is Google's system for vetting applications available in their Play Store. Third, Antivirus (AV) solutions are examined. This last feature is optional, but users can opt-in by downloading the software. It is thus an existing security feature, and is therefore presented here.

3.1 Operating system structure

3.1.1 Kernel

The kernel is the base for the mobile computing environment and provides the Android system with several security features, all of which are explained here. [17]

Permission access

On installation all applications are given a unique UID, which the kernel uses to distinguish applications from each other. That ensures an application can not access resources belonging to other applications or use hardware components that the application doesn't have permission to access. [15]

Processes isolation

Each application is separated based on the UID and run in its own process. The kernel protects the processes from each other, with the result that a given User A is unable to read User B's files, cannot use up User B's memory or CPU resources, and also cannot exhaust User B's peripheral assets (e.g. GPS, bluetooth). [17]

Secure IPC

To perform secure communication between processes the Android system has four methods; binder, services, intents and content providers.[17] The most frequently used is intents, as explained in section 2.4.

Sandbox

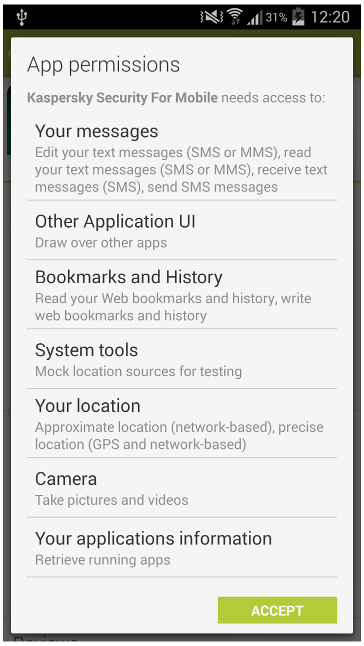
The security features provided by the kernel enforces a security barrier between applications and the system at a process level. This creates a sandbox.[17] The sandbox stops applications from performing interactions they do not have permissions to, and is the core of the Android security. Because the sandbox is enforced at the kernel level in the architecture (see figure 2.3) all software above this level is run within a sandbox.[17] This includes OS libraries, application framework and application runtime.

3.1.2 Application Security

Application Framework layer is the second place access control is imposed. To get access to restricted functionalities provided by this layer an application has to declare a permission in its manifest file (AndroidManifest.xml).[15] Figure 3.2 illustrates the relationship between applications, permission and restricted system resources. An example of such a permission is to grant an application internet access, which the manifest file is displayed as `android.permission.INTERNET`. Specific permissions later in this report will drop the "android.permission." prefix and only be written with the last capitalized name.

On installation these permissions are presented to the user, and the user has then the option to accept or decline the installation. A screen shot of the permission accept screen is shown in figure 3.1. The user can not choose to accept some permissions and decline others, all permissions has to either by approved or rejected. [17]

If an application tries to access a system resource that it does not have the necessary per-


App permissions

Kaspersky Security For Mobile needs access to:

Your messages
Edit your text messages (SMS or MMS), read your text messages (SMS or MMS), receive text messages (SMS), send SMS messages

Other Application UI
Draw over other apps

Bookmarks and History
Read your Web bookmarks and history, write web bookmarks and history

System tools
Mock location sources for testing

Your location
Approximate location (network-based), precise location (GPS and network-based)

Camera
Take pictures and videos

Your applications information
Retrieve running apps

Figure 3.1: The screen presented to the user during the installation process of an application, where the user has to decide to accept or decline the permissions

missions for, a security exception is thrown back to the application.[17] Not all system features are accessible by the use of permissions, for instance an application can never modify the Subscriber Identity Module (SIM) card. A list of all the permissions available to the Android manifest to can be found at the Android developer web site (see [28]).

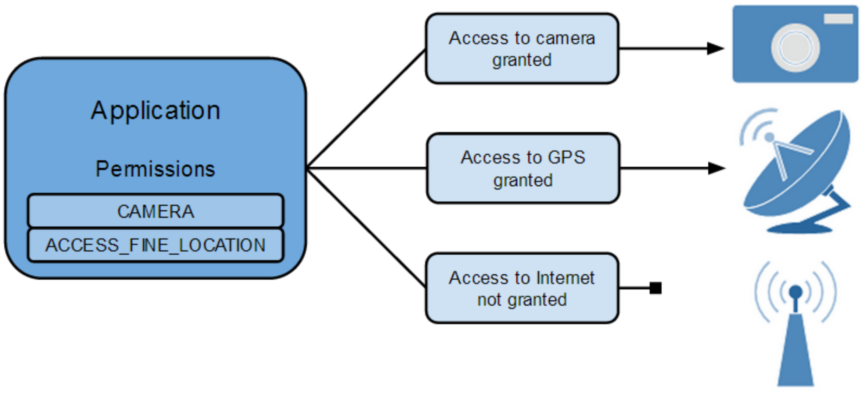


Figure 3.2: An application using permissions to access system resources. The system will check if the application has the necessary permissions, and accept or decline access accordingly.

3.1.3 Memory Security

In addition to the security provided by the permission based model, Android contains some further safety features to prevent memory exploitation. Different solutions have been added with different Android distributions, and two of the most important are: [17]

No eXecute was added in Android 2.3 and prevents memory attacks by setting heaps and stacks of memory to nonexecutable.[15]

Address Space Layout Randomization was added in Android 4.0 and updated in 4.1. It randomizes key section of memory and that way protects against exploitation of memory corruption.

3.1.4 Operating system structure and the Threat Model

The security measures provided by Kernel, Memory Security, Application Security together form two green borders in the Threat Model; Permissions and System Security. Figure 3.3 illustrate an example scenario of a malicious application trying to access a User Data asset. Say the asset is the GPS location of the device, if the attack wants this information it must have the permission ACCESS_FINE_LOCATION to pass the Permissions layer in the Threat Model.

Another scenario is if the asset is login information stored in a banking application on the device. In this case the banking application's internal storage space is protected by the System Security, and the attack has to break through this to acquire the information. Both these security measures are highly effective, and are therefore (for this type of attack) graded dark green.

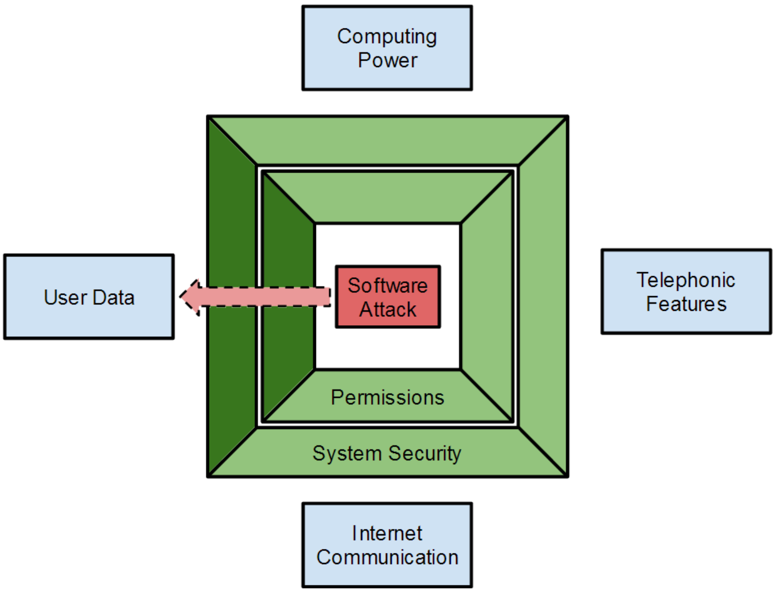


Figure 3.3: A software attack trying to bypass the system security to gain access to the User Data asset. The asset is protected by both Permissions and the System Security.

3.1.5 Other Security Features

The Android system contains additional security features, some are listed here: [17]

- Screen lock
- Cryptographic APIs for application use
- Full filesystem encryption, and the encryption key is based on the screen lock password (Android 3.0 and later)
- Remote wipe (Android 2.2 and later)

For additional information on any of the items listed here, see the Android Security Overview web page [17].

3.1.6 Acquiring root access

The process of acquiring root access on an Android device is often referred to as *rooting*. This has the effect of giving the user more privileges, but it is not only the user that gains those rights. Access to resources and files are normally restricted by the security model, as seen above. However, if an application is allowed to run under the root user, this security model breaks down. Applications run as root user bypass the permission checks and can access any file. Needless to say, a skilled attacker with this much influence can damage a device a great deal.[15]

Many people in the Android community see rooting as a good way for the user to get more control over the device. It allows a user to *re-rom* the device, which is installing a custom OS. This is desirable to many because they can be free of so-called bloatware and let the user update to the latest versions of the Android OS immediately upon release. A drawback is that it enables malicious software to use the same techniques to gain control of the device.[15]

3.2 Application vetting

Google's Bouncer is among the security mechanisms that helps prevent malware spreading through the Google Play Store. As mentioned in chapter 2, it is an automated (scanning) tool that vet applications being submitted to the Play Store. All applications have to go through the same process. If the vetting process reveals malicious activity in an application, it will not be accepted into the store. When Google first announced its tool for application vetting in February 2012, it described how Bouncer works in the following manner:

"once an application is uploaded, the service immediately starts analyzing it for known malware, spyware and trojans. It also looks for behaviors that indicate an application might be misbehaving, and compares it against previously analyzed apps to detect possible red flags. We actually run every application on Google's cloud infrastructure and simulate how it will run on an Android device to look for hidden, malicious behavior. We also analyze new developer accounts to help prevent malicious and repeat-offending developers from coming back.".[29]

Summed up, Google said Bouncer's characteristics were: [30]

– It's automated
– It scans both new applications and those already in the market.
– It looks for known malware immediately upon upload by a developer
– It's also behaviour based.
– It's run on Google's cloud.
– It simulates Android's runtime
– It looks for "hidden, malicious" behaviour.

A more technically detailed description of how it works has not been published by Google, probably because that would make it much easier to design code with special care taken to steer clear of the scans. However, others have researched and tested how the Bouncer operates.

At the Black Hat convention in 2012, the two researchers Nicholas J. Percoco and Sean Schulte, then with Trustwave SpiderLabs, presented their research. Their intention was to test the limits of the Bouncer tool, and see how rigorous its scans were. Full details are found in [30], a summary is provided here.

Phase 0 The researchers initially created a benign application called "SMS Bloxor", and uploaded it to the Play Store with a newly registered developer account. The purpose of the application was to let the end user define a phone number from which to block incoming Short Message Service (SMS) messages. Thus, they justified requesting some of the more intrusive permissions without raising suspicion, in case the Bouncer checked for unnecessary permission requests. In order to check whether Google's statements were true concerning Bouncer's attributes, Percoco and Schulte included code that reported home if run, with data such as IP address of its current location and what device it was run from.

Within a few minutes of uploading, the application phoned back to the control server, revealing exactly what was expected. Bouncer ran the application in an emulator from an IP address in a network belonging to Google. The researchers then made a minor update to the application and uploaded it once more to check whether Bouncer actually scanned each time. This resulted in the same occurrence, only from a different IP address, albeit still the same network.

Phase 1 through 7 At this point, Percoco and Schulte took action and included several features to avoid getting caught. First, they wrote code to disable malicious activity if the application was run from within Google's IP network. Second, they included Javascript to enable a Javascript bridge. With these features in place, they gradually increased the maliciousness of the application's payload.

Step by step, the application became more and more invasive and aggressive. Functionality like copying all contact info, extracting SMS messages, pictures and call records and even forcing the user to any web page of the attacker's choice, were added. On all but one occasion, Bouncer scanned the updated application, but at no time were there warnings or did it detect the malevolent activity. The researchers even enabled remote controlling of the device running the application from a Command and Control server (C&C server).

Conclusions A few weeks passed without updates, the application still residing in the Play Store, yet Bouncer scanned it once nonetheless. The result was no different that time, and the researchers decided they wanted to try to get caught. A minor update to the application was uploaded, the change being that the block to Google's IP network was removed. Bouncer scanned the application within minutes of the update, but to their surprise, it did not find the malicious payload.

At this point, the application was phoning home all the above-mentioned details at boot and then every 15 minutes. To provoke additional scans, the interval was reduced to 1 second, acting way more aggressively than it would in real life. Upon the next scan by Bouncer, the researchers received the Bouncer emulator's contact list, phone number, IMEI number, SIM serial number and more. This time, the scanner was run 19 times within 6 minutes. Approximately 24 hours later the application was removed and an email from Google notified the developers that their Google Play Store account was terminated.

Google has not made any further comments or announcements concerning their application vetting. What is evident though, is the need for a more thorough examination of the uploaded code. It seems the scanner only runs the code, and does not perform any review of the individual methods or functionalities which would have made it able to assess such hidden functionality as seen above.

3.2.1 Application vetting in the Threat Model

The application vetting security mechanism is featured in the Threat Model below (figure 3.4). It is the first security mechanism an application encounters.

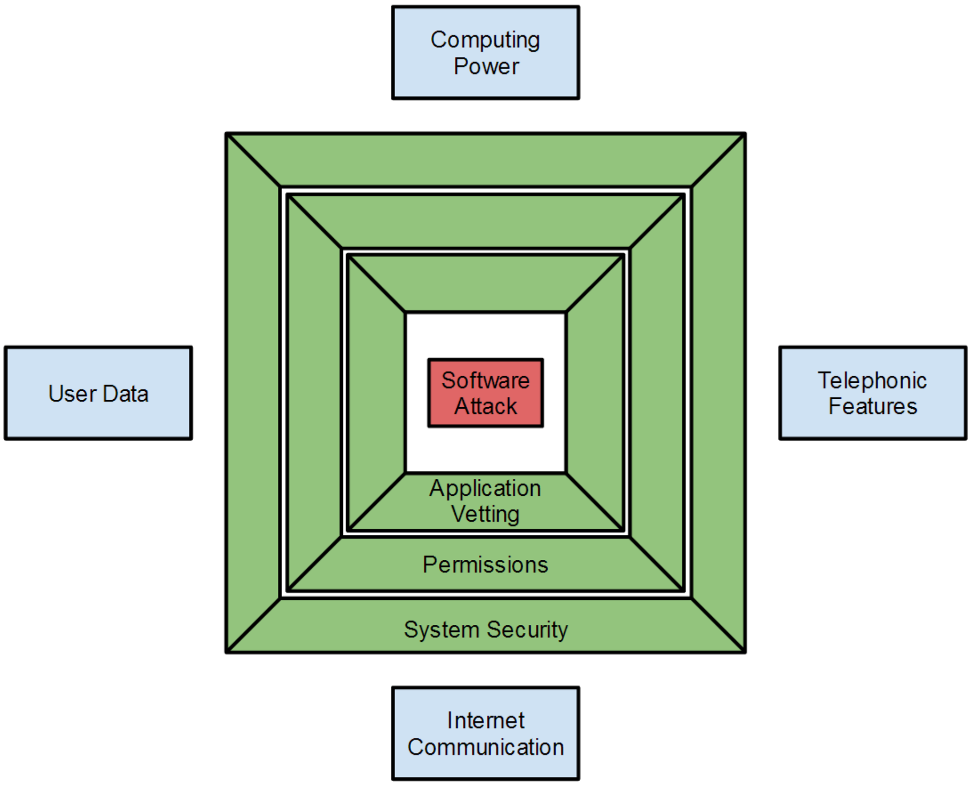


Figure 3.4: The application vetting security solution is the first security mechanism an application encounters and occupies the innermost frame.

3.3 Anti-virus software

An AV program differs from the other security measures featured earlier by being optional. In practice, this means most users do not download and install this layer of security, but it is an available option, and is therefore evaluated here. There are a number of commercial anti-virus solutions available to the Android user, and most, if not all big corporations in PC security have them available. The features in place to protect the user are generally the same, with some variations as to which less critical ones are included.

The AV software typically include: [31]

Malware detection and protection
The traditional signature-based detection is the obligatory feature, one that all AV solutions employ. Signatures are used to compare and validate software and data. IETF defines a digital signature as: *"A value computed with a cryptographic algorithm and appended to a data object in such a way that any recipient of the data can use the signature to verify the data's origin and integrity"*[32].

In a signature-based detection scheme a signature is made for known malicious software and then distributed to the AV programs. The AV programs then check the received signatures against signatures of the software on the device, and if a match is found, the AV program flags that software as malicious. A problem with this kind of approach is that it will only detect already known malicious software, and it requires the AV program to update frequently to stay effective.

Each commercial actor have their own database of known malware, to which they compare the program files they scan on the device. Thus, the precision of the malware scan is the only feature separating different anti-virus programs, apart from which package of features is included.

Theft protection
It does not protect you from thieves, but rather mitigates the impact should the smartphone be stolen or lost. This threat is not within the scope of the thesis, but the features protecting users in such cases are an essential part of what AV solutions offer, and are mentioned here for that reason.

Typical components are remote wipe, remote lock and locate device. Due to Android's system architecture, the remote wipe function is limited to handle data logging a user's tracks, i.e. contact data, text and email messages, browser history and bookmarks, user defined dictionary and the like.

Remote lock and locate device are simpler features. The first enables you to remotely lock the device should the screenlock initially be disabled, making it necessary to type a given password to reopen it. Locating the device is reliant on either GPS or wifi-positioning being enabled. If the smartphone is simply mislaid, an option to sound an alarm, or "scream", is often available. The ability to sound alarms and/or take mugshots of whoever enters the wrong code to unlock the device three times is also common.

Safe web surfing

Protects the user whilst surfing the Internet, primarily against phishing attack and threats. This mode is often accompanied by a parental control mode, where it is possible to define a blacklist or whitelist of specific websites.

Notable other features

There are lots of other features available to the user, depending on which company delivers the product. Notable others are anti-spam, SIM-lock[1], data backup, and call & text filtering. Some companies may also include the ability to monitor the device's resource use, warning the user should an activity seize a suspiciously large amount.

Nevertheless, AV programs for smartphones are quite limited, as they are just another application with all the limitations that follow. To effectively scan the entire device, root access is necessary, but without an agreement with either Google or the Original equipment manufacturer (OEM), that can't be granted without compromising the entire device (as seen in section 3.1.6). For these reasons, the most valuable features AV programs offer, are malware detection and phishing attack protection, which both rely on the individual developer's analysis tools and databases.

[1]If a new SIM card is inserted, the device is locked

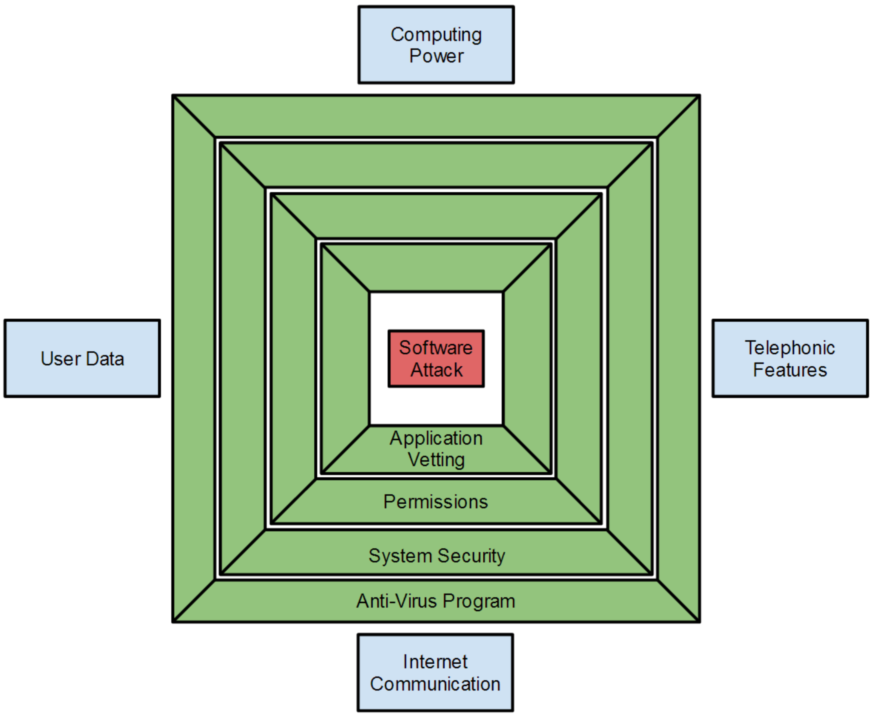


Figure 3.5: The AV security solution is an optional security mechanism, active and is therefore shown as the innermost frame.

Chapter 4 Attack vectors

Although smartphones are protected from attacks by many security features, malware can arrive on a mobile device through just about any attack vector commonly associated with other endpoint devices. The most typical entry point is through a downloaded application, but attacks can also be executed through visits to malicious websites, spam, malicious SMS messages, and malware-bearing ads.

4.1 Trojans

Trojans are the most prevailing attacks by far.[3] They are programs or code that disguise as something useful or enticing, then deliberately perform harmful actions once installed. They do not exploit technical weaknesses in the OS or device itself in order to be deployed, but make use of people's credulity to get a foothold for further/future misconducts. As such, new security mechanisms have not been able to assuage fears of malign Trojans, as the concern is not primarily technical. Human errors are very hard to eliminate, whereas technical vulnerabilities are constantly being rectified by manufacturers and developers. Together, these elements have contributed to the widespread success of Trojan attack vectors.

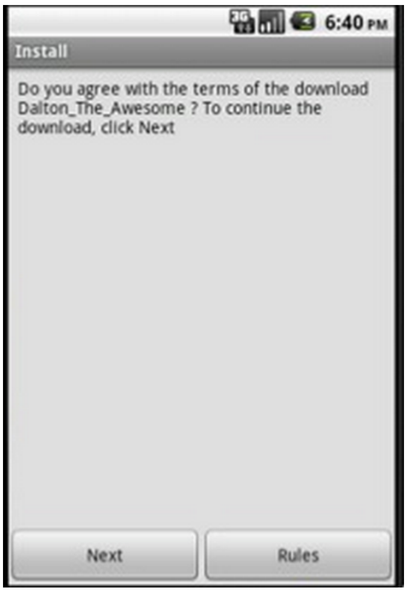
4.1.1 How they work

There are different ways to perform a Trojanesque attack, and below is a segmentation into three categories.

Disguised as a legitimate application is the most common form. A user that searches for a popular application through internet search or in a third party store, is the typical victim. If the search leads to a site looking official, with descriptions, screenshots, user reviews and application icon all looking authentic to a user, he or she is likely to download it. Even the installer file is usually named to look trustworthy, like some form of com.«program-name».apk.

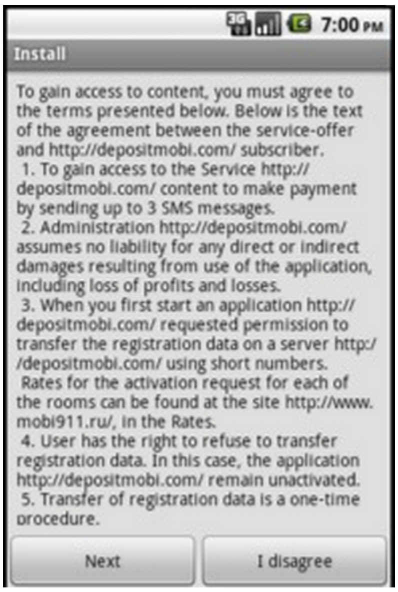
Wrapped together with installer for a legitimate application is another form, but is a rarer occurrence. It exploits the fact that users often go searching for free alternatives to paid applications. The attacker wraps the malicious payload together with the installer for a popular app, and uploads it to the application store as a free alternative. The price-conscious user is now likely to take the bait.

Claiming to be an update of some critical or popular software. Malicious websites might display a prompt imitating a notice of some critical system update, or update for a popular application. If the user accepts, the installer is downloaded.



Do you agree with the terms of the download Dalton_The_Awesome ? To continue the download, click Next

(a) Accept terms outright or view them.



To gain access to content, you must agree to the terms presented below. Below is the text of the agreement between the service-offer and http://depositmobi.com/ subscriber.
1. To gain access to the Service http://depositmobi.com/ content to make payment by sending up to 3 SMS messages.
2. Administration http://depositmobi.com/ assumes no liability for any direct or indirect damages resulting from use of the application, including loss of profits and losses.
3. When you first start an application http://depositmobi.com/ requested permission to transfer the registration data on a server http://depositmobi.com/ using short numbers.
Rates for the activation request for each of the rooms can be found at the site http://www.mobi911.ru/, in the Rates.
4. User has the right to refuse to transfer registration data. In this case, the application http://depositmobi.com/ remain unactivated.
5. Transfer of registration data is a one-time procedure.

(b) Terms and conditions.

Figure 4.1: The two images shown illustrate how (a) the initial install prompt, and (b) the terms and conditions, might look like when installing malware. Even though the permissions it requests are stated outright, the interface is very similar to what a legitimate application (on the same OS) would look like, thus fooling most users. The images were taken from [33].

Drive-by downloading[1] is a technique where a download is initiated automatically upon a visit to a webpage specified by the attacker.[34][35] The only action required is loading the webpage, which most likely appears as innocent or authentic. In fact, drive-by downloading is not malicious in itself, and legitimate services may also employ this technique.

[1]Drive-by downloading is perhaps deserving of its own section along with Trojans, advertisement and WebView attacks, but is simple in execution (making for a short section) and exploits user naivety in a manner similar to Trojans, and is placed here on that basis.

A drive-by download attack works great in tandem with advertisements. Normally this method is used to target and infect traditional computers, but there are known incidents where attackers have started to target mobile devices.[36] An example from June 2011 is a malware known as GGTracker that exploited an in-app advertisement to lead users to a webpage that initiated the download of a malicious payload immediately.[37] Another example is NotCompatible, the first instance of a web server used specifically for drive-by targeting mobile devices.[36]

After the payload is downloaded by either of the above methods, the critical part of the attack still remains. Since most Trojan attacks are done through downloads outside the official Google Play Store, installation does not ensue automatically, but must be executed by the user. When initiated, a prompt appears asking the user to accept the requested permissions, an example of which is provided in figure 4.1. The prompt displayed at this point does not necessarily list the permissions right away, often requiring a deliberate action should the user wish to see them. This is in accordance with McAfee's report [33], which states that the interface can be complicated and confusing. Sometimes the user is forced to comply to the terms immediately by pressing a 'next' or 'agree' button with no other options, or isn't given any prompts at all, before installation ensues. The installer might display a progress bar after accepting the terms, but the legitimate application is seldom installed along with the malware.[33]

Evolving complexity

While Trojans have proven to be a successful way of distributing malware, it is important to clarify that most occurrences are through 3rd party application stores and other unofficial fora. The official sales channels have security mechanisms to counter or mitigate the threats, for example the application vetting process mentioned in section 3.2. In the face of this, malware writers have been known to add smartness and complexity to their attacks, challenging even the most advanced security in place. Some Trojans have made it through to Google's own marketplace, e.g. DroidDream and BadNews (in sections 4.1.2 and 4.2.3, respectively).

An example of enhancements made to bypass virus scanning is polymorphic servers. Polymorphic servers have the ability to respond differently to the same URL request when coming from different sources. By making use of the victim's IP address as a seed, the server can create a unique Android Application Package (APK) file, with slight changes to the code implementations without altering the end product. This is a clear advantage for the attacker, because a unique APK file will generate a unique hash value and be regarded as a new application by the application store. Thus, a malicious payload can be used multiple times, even though security scans have detected the malware on previous occasions.

Obfuscation and recompilation is another technique. It means naming all variables and methods with arbitrary characters, making reverse-engineering next to impossible. Ultimately, it has the same effect. By altering the code, but keeping the payload, "new" applications are made. Some Trojans include anti-reversing techniques to avoid dynamic analysis and prevent malware from running in an emulator, as seen in section 3.2.[33]

4.1.2 Different impacts

Trojans can be divided further into sub-groups, depending on the type of impact they have, the most common being information theft, sending premium SMSs and hijacking device resources. Note, however, that these ways to impact a user are not restricted to result from Trojan attack vectors only.

Fake installers

F-Secure stated in their mobile threat report from Q3 2013 that 4 in 5 mobile threats were profit-motivated.[3] Among these, sending premium SMSs seems to be the adversaries' choice for monetizing their malware, which have come to be known as fake installers. Its name is similar in meaning to that of Trojan, but are a subgroup due to its specific end purpose. Lookout [34] state that as much as 80 percent of the malware they detected in June 2012 was in the fake installer category, and Juniper's [4] corresponding share as of March 2013 was 73 percent. Several sources thus attest to fake installers being the most widespread form of malware today.

FakePlayer is notable for being the first SMS Trojan found in the wild that targeted Android (discovered August 2010). It employed the first of the mentioned attack vectors, disguising as a media player, its icon reminiscent of the Windows Media Player icon. The user had to manually download and install the application, at which point FakePlayer specifically asked the user for permission to send SMS messages. Running the application first time would show a text in Russian reading «Wait, requesting access to the video library ...», while actually trying to send an SMS to Russian premium SMS services, the activity hidden from the user.[38]

The area of impact being Russia and Eastern Europe is no coincidence. These countries tend to have less strict legal regulations regarding premium SMS, and additionally, Google has a weaker market position. China's status is the same, and all of them have third party markets dominating in terms of user consumption. Several hundred million users is a significant amount, and lacking the security features accompanying the official sales channels, the possibility of infecting many users is higher. These are attributes making the application market malware prone, since malware writers have increased likelihood of making money e.g. through premium SMS services.

Figure 4.1 above shows both screens in English. More often this will be in Russian or Chinese, which is only natural since the attack is more widespread there.

Spyware

Spyware is software that collects information from the user's device without the user's consent. [39] A smartphone contains a lot of sensors that make it a very useful spy tool, such as camera, GPS locator, microphone and accelerometer, as well as the personal data stored like contacts, email, in the calendar and other applications. Until the beginning of 2012 spyware was the most common form of malware. The decline in percentage is due to the heavy increase in premium SMS targeted malware, and not the fact that the number of spyware attacks have diminished (it has actually increased). [34]

Spyware can be separated into two subgroups; untargeted and targeted spyware.

Untargeted Spyware is often an apparently safe and trusted software, which once installed abuses the privileges granted by the user to gather information. [40] An example of this can be a weather forecast application that requests the user's location and Internet access. This can be used to gain information about the weather for the user's current location, but it can also be used to share the user's location with an advertisement server so they can send the user targeted advertisements. [40] Untargeted spyware is not tailored to spy on one user, but instead is designed to gather information from a large user base, hence the name. It is likely to collect things like location data, contact information, browser data and keyboard log. [40]

Targeted Spyware on the other hand, focus on specific individuals, as the name implies. A typical example is parents spying on their kids or monitoring spouses. [39] Targeted spyware tend to collect more data than untargeted spyware does and may have the ability to remotely control the device (see section 4.1.2). Targeted spyware is often harder to deploy on a person's device than the untargeted is, and in many cases the attacker needs physical access to the device to properly install the software. [40]

TapSnake is an example of a simple targeted spyware originating in 2010. The application was available in the Android Market, disguised as a harmless "snake" game, but also collected the victim's GPS position. It thus falls into the category where the malicious payload is wrapped together with a legitimate application. In this case, the game actually played the way it was expected to. The installation process required that the attacker had physical access to the phone, because a pin code was needed to identify the victim later. The one spying then had to buy a second application, which once installed would provide the GPS location of the victim, identified by the pin code from earlier.[41][42][43]

Hijacking device resources

In their mobile malware report based on data from 2013 [44], G Data claims that approximately 10 percent of smartphone attacks now deploy so-called backdoors, the number rising from previous years. A backdoor has several purposes for an attacker, who can use it to remotely access the device and issue instructions to it. That enables the attacker to either steal information stored on the phone, or take advantage of its hardware resources. One infected device alone may not be very useful to an attacker, but when malware like this is successfully deployed to a number of users, the resources available to the attacker becomes significant. The word for such a network of compromised machines running under a command and control infrastructure, is 'botnet', where each infected device is a software robot - a 'bot'.[45]

Botnets are capable of executing computationally demanding tasks in practical time, but may also be used in a distributed denial-of-service attack. As such, botnet-Trojan malware come with a varying degree of functionality and complexity. An example is DroidDream from 2011, that managed to spread through the official Android Market, its malicious payload wrapped together with more than 50 different applications.[46] After first infecting a device, it was capable of stealing information and also download other applications. Thus, the first wave of the attack was not truly damaging in itself, but the second application installed would make the device a part of a botnet. DroidDream is noticeable because it managed to spread through Google's own marketplace, as well as being the first malware to urge Google to employ its ability to remotely wipe software off of a user's device.[46][47]

The emergence of digital currencies, or cryptocurrencies, has provided new ways to monetize malware. BitCoin, LiteCoin and other such currencies are "mined" by use of computational efforts, the more power the higher reward. It is easy to see how a botnet can be exploited to increase someone's mining ability. In February this year, malware claiming to be a popular radio application, TuneIn Radio Pro, used infected smartphones to mine for the cryptocurrency called Dogecoin.[48] To avoid suspicion from the user, the radio application it disguised as was fully functional, and only when the smartphone was inactive was the CPU used to mine Dogecoins. A vigilant user may have noticed a shorter battery life or a heated device due to the hard-working processor, but under use, the smartphone would operate seemingly normal, so that noticing the infection would be harder. Additionally, in the smartphone's system logs, the process was named "Google Service", making the threshold for regular users to end it high.

Excessive wear to the CPU like that from Dogecoin mining, risk damaging it or causing it to malfunction sooner. A malfunctioning phone is costly for a user, having to pay for either a repair or perhaps a new phone.

4.1.3 Trojan attacks and the Threat Model

As said above, most Trojan attacks occur outside of the official application marketplace. This does not mean, however, that attackers fail to try. It means they fail the application vetting process. This is reflected in the Threat Model below (figure 4.2, where **application vetting** is classified as highly effective against all forms of Trojan attacks.

Permissions is colored light green for fake installer and spyware attacks, the reason being that some users may refrain from installing applications requesting intrusive permissions. Most users ignore that warning, though. Permissions do not thwart attacks that aim to hijack device resources, as using computational power requires no particular approval from the user.

System security, however, does make sure no application uses the entire hardware load. It also prevents a malicious app from accessing other application's internal data.

The effectiveness of **Anti-Virus programs** rely heavily on the issuing company's database. If a malware is known, the AV program may warn the user before any harmful actions are done, but it cannot ease the impact of the attack further.

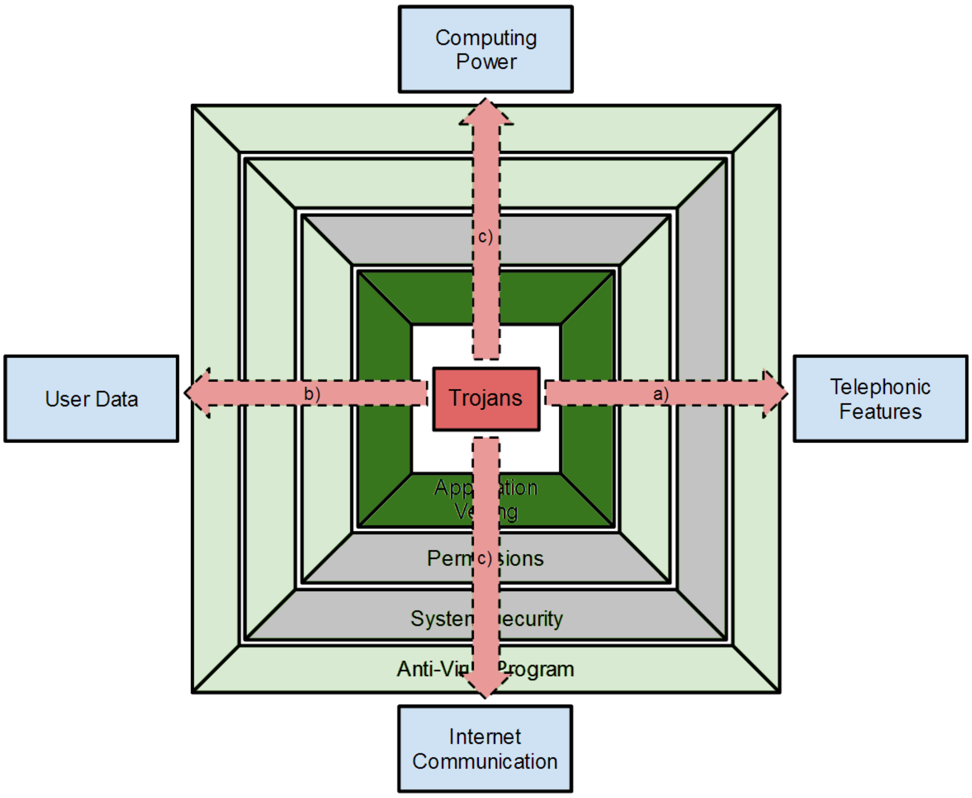


Figure 4.2: This Threat Model depicts the vulnerability picture for Trojan attacks. **a)** The arrow pointing right equals the fake installer attack. **b)** The arrow pointing left corresponds to spyware attacks, and **c)** the arrows pointing up and down represent attacks hijacking system resources.

4.2 Advertisement threats

Attacks through advertisement is different from Trojans, because they exploit weaknesses in the system and not a user's lack of security knowledge. The vulnerabilities targeted by advertisement attacks are related to how advertisement is integrated within applications (Advertisement Library) and their connection to servers that supply ads (Network Providers).

4.2.1 Permission abuse

The Android framework contains a predefined set of permissions used as a protection layer to limit the resources an application has access to. Permissions are used for Internet access, camera, GPS and read SMS, among others. Advertisement Libraries are implemented (see section 2.6.1) such that they are executed at the same time as the host application. This means they are run within the same environment, a Dalvik virtual machine, and the Advertisement Library will have the same UID. When an application wants to use a given system resource, the operating system checks to see if the application has the necessary permissions, and this check is based on the applications UID (see section 2.3). The Advertisement Library activity and the advertisement that is loaded into the application will thus have the same permissions as the host application. The process works both ways, so the host and the application will have access to the permissions provided by the Advertisement Library.

As a result of the shared permissions, many Advertisement Libraries will probe the application and try to discover what system resources they are granted permission to. [24] Malicious advertisements included in applications with a lot of permissions can have uncomfortable effects for the user. Two possible scenarios are:

Acquire user data An advertisement Publishing Network can gather the user's location (makes use of ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission) and then request (makes use of the INTERNET permission) and store that information on their serves. This information can in turn be used to send the user targeted advertisement.

Send spam A more aggressive scenario is where the advertisement Publishing Network access the users contact data (uses permission READ_CONTACTS) and send them spam. This can be done either from their servers or through the use of the email or SMS from the users phone (carried out by using either INTERNET or SEND_SMS permission).

The former is an invasion of the users privacy and uses borderline marketing techniques to increase their revenue. The latter is more of an opportunistic threat, because the advertisement looks for possible permissions and then abuse them with aggressive marketing.

Also worth noting is that the user, when looking at the permissions required upon installing the application, is unable to determine what permissions are for the application only (i.e. what the application utilize to function) and what the advertisements use. [24] This makes threats like these hard for the user to detect and avoid.

4.2.2 Fetching and loading dynamic code

Some advertisement networks are capable of fetching and downloading code (normally from the Internet), which poses a great threat for two reasons. [24] Firstly, the dynamically loaded code can not be analysed, thus bypassing any static analysis efforts. Secondly, the downloaded code can change at any time, making it very hard to predict and defend against its behaviour.

Plankton is an Advertisement Library that makes use of fetching and loading dynamic code. [49] In the Plankton case, only a small portion of the Advertisement Library included in the main application is malevolent. This portion is a service (see section 2.4) that contacts the remote server and shares a list of available permissions and the IMEI number. [24] In return the remote server provides the Advertisement Library with a Uniform Resource Locator (URL) to download a Java file (.jar) containing code specifically suited for the device. [24] This file contains most of Plankton's malicious code, and is dynamically loaded into the application using the `dalvik.system.DexClassLoader` object. [24] This object loads classes from .jar and APK files and can be used to execute code that is not installed as part of an application. [50] Once the download file is loaded into the application it will turn the host application into a bot that listens to remote commands. Several host applications subscribing to or being infected by this Advertisement Library have been removed from the official Android application marked. [24]

4.2.3 Malicious Advertisement Libraries

One opportunity that presents itself as a result of the malevolent possibilities associated with in-application advertisement is a type of malicious Advertisement Library. This is an Advertisement Library that masquerades as an legitimate network, but contains malicious payload that developers unknowingly include in their applications.

BadNews, detected in April 2013, is the first example seen of a malicious Publishing Network posing as a legitimate one. Developers would opt to include BadNews in their applications in the normal way, ignorant of the threat it posed. BadNews would then contact its C&C server server and share sensitive information from the device (e.g. phone number and IMEI) once an infected application was installed. In return it received instructions from the server to display fake news alerts to the user. A typical news alert could be to inform the user that an application needed an update, and then provide the user with a URL. This URL led to an installation file which contained toll fraud malware. The BadNews application was able to bypass the Google Bouncer, because the most malicious payload was pushed to the device after installing the application, thus after Bouncer's scans. BadNews appeared in 32 applications across four different developer accounts on Google, and was downloaded between 2 and 9 million times. [51]

4.2.4 Advertisement and the Threat Model

As illustrated by figure 4.3 advertisement attacks target three different assets - User Data, Telephonic Functions and Internet Communication. To reach these assets the threats has to pass through the different security layers in the model:

Application Vetting Advertisement attacks are hard to detect using application vetting, because they are delayed in striking and often the malicious code is not part of the host application. Malicious Advertisement Libraries and threats that fetches dynamic code may be detected with Application vetting, because the malevolent code is present in the application at publication.

Permissions Malicious advertisement often attach them self to legitimate applications, and then they just inherit the host application's permissions. This limits what actions the malicious advertisement can perform. Telephonic Features require more (and more uncommon) permissions than User Data and Internet Communication, thus it has a darker grade.

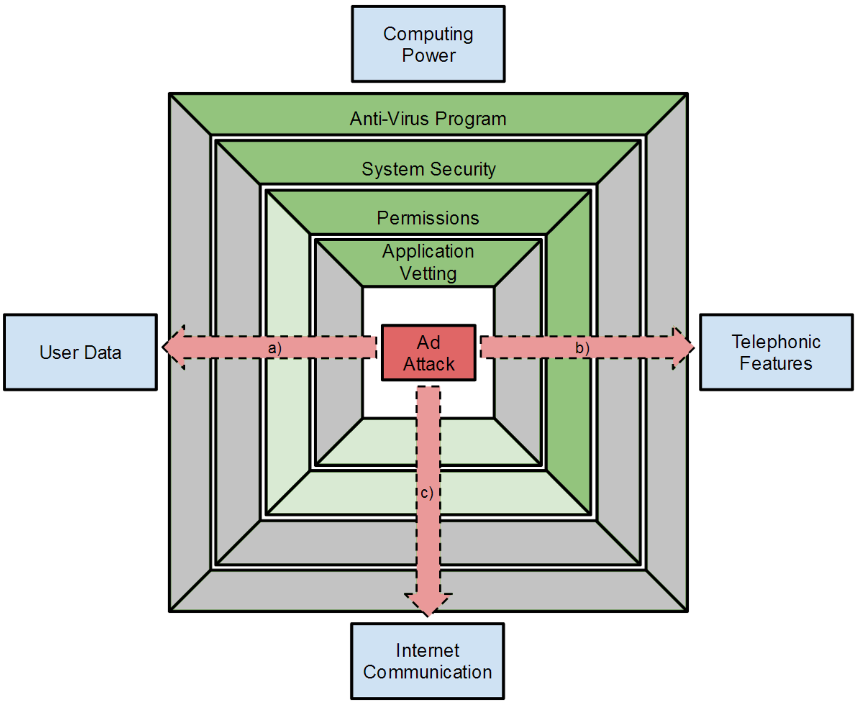


Figure 4.3: The Threat Model with advertisement attacks targeting User Data, Telephonic Features and Internet Communication assets. The arrows represents advertisement threats that target different assets: **a)** Acquire user data. **b)** Send spam. **c)** Fetching and loading dynamic code and Malicious Advertisement Libraries.

4.3 Attacks using WebView

Webviews are features in Android and iOS that enable applications to load and display their content from the Internet using the standard HTTP. [52] They allow the user to interact with the web content, and include basic browser functionalities like page navigation, page rendering and JavaScript execution. [52] In short, they act in the same way as real browsers (e.g. Firefox, Safari and Internet Explorer) only embedded into a host application. There are some major differences, however, as browsers are generic and designed to be independent from web applications. Webview based applications, on the other hand, are custom-fit to the intended web application.

An example of a WebView based application is the "Facebook Mobile" application that is specifically tailored to interact easier and better with a mobile device user. That is why many prefer this over the traditional Facebook web interface when using a mobile device. [53]

User interaction and unique customization is made available through the use of APIs provided by the WebView. Another feature of the APIs that is perhaps more important, is enabling a two way interaction between the host application and a web page. The application can invoke or insert JavaScript code into web pages, and they can monitor and respond to events within the web page. The other side of this two way communication is that the application can register different interfaces, enabling the web page to use JavaScript to interact with the application and the device. This two way interaction makes the WebView more powerful than the traditional browser. [53]

With the introduction of WebView, the web security landscape has changed. A handful of traditional browsers had prominence, all developed by known companies that are trustworthy, but with WebView, this is no longer the case. Numerous unknown "browsers" are now present, and their trustworthiness is not certain. Traditional web applications rely on the browser to secure cookies, JavaScript code and HTTP requests, and with traditional browsers this is not a problem. The companies behind the browsers, spend a lot of time and effort to keep them reliable and secure. The introduction of unreliable WebViews has turned to a security concern.

Another security mechanism embedded in traditional browsers is the sandboxing of all web applications, thereby not allowing them to communicate outside the browser. To improve the interaction between applications and web pages, WebViews have made holes in this sandbox, and in the process introduced vulnerabilities that a potential attacker can take advantage of. [53]

4.3.1 Android Java and JavaScript interaction

JavaScript to Java

The WebView provides features to allow JavaScript code to invoke Android application Java code. By using an API named `addJavascriptInterface`, Android applications can register Java objects to the WebView, and then all public methods in these Java objects can be used by the JavaScript code inside the WebView. The code lines necessary are presented in figure 4.4 below. The ability to bind an interface to a WebView breaks the sandbox model used by traditional browsers. [53]

```
        wv.addJavascriptInterface(new FileUtils(), "FUtil");
        wv.addJavascriptInterface(new ContactManager(), "GC");
```

(a) `FileUtils` and `ContactManager` are registered.

```
    public int write (String filename, String data, boolean append);
    public String read (filename);
```

(b) `FileUtils` methods to access the Android's file system.

```
        public void searchPeople (String name, String number);
        public ContactTriplet getContactData (String id);
```

(c) `ContactManager` methods to access the user's contact list.

```
<script>
        filename = '/data/data/com.livingsocial.www/' + id +'_cache.txt';
        FUtil.write(filename, data, false);
</script>
```

(d) JavaScript code using `FileUtils` to write data to local file.

Figure 4.4: Java objects used to access the resources outside the WebView. [53]

Java to JavaScript

By using the WebView API known as `loadURL`, Java code (from the Android application) can be turned into JavaScript code (run on the web page inside the WebView). [53] If the URL from the Android application starts with `javascript:` followed by JavaScript code, the API will execute this code on the web page in the WebView. [53] The code lines necessary are presented in figure 4.5.

4.3.2 Attacks from web pages

Malicious web pages can deploy attacks through the use of WebViews. To achieve this, an attacker needs to trick the victim to load their web page into an application's WebView, which is easy for an accomplished developer. [53] It can be achieved using various means, including advertisements and emails. The relationship between server, WebView and application is illustrated in figure 4.6.

```
String str="<div><h2>Hello World</h2></div>";
webView.loadUrl("javascript:document.appendChild(str);");
```

(a) Code that adds "Hello World" to the page.

```
webView.loadUrl("javascript:document.cookie='';");
```

(b) Code that will set the page cookie to empty.

Figure 4.5: JavaScript code the API will execute inside the WebView. [53]

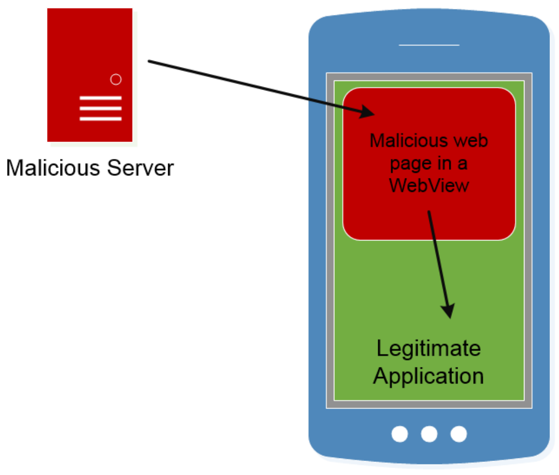


Figure 4.6: An attack from a web server using WebView to access the target device.

Attacks through the compromised sandbox

To protect against the risk of running untrusted JasvaScript programs, browsers implement a security feature called a sandbox. The sandbox is in place to achieve two security goals; separating web pages and the system, and separating web pages from other web pages. [53] The 'compromised sandbox attack' takes advantage of the previously described `addJavaInterface` to invoke Android application Java code and punch holes in the sandbox. [53] An example of the `addJavaInterface` JavaScript code is illustrated in figure 4.4. This results in the attacker gaining access to system resources, such as camera, GPS location, address book and other files, and breaks the separation barrier of the sandbox between web pages and the OS. [53] Additionally, once an interface is registered to the WebView (through the use of `addJavaInterface`) it becomes global, resulting in that all web pages loaded into the WebView can access this resource and the data maintained there. This enables two web pages of different origins to affect each other, effectively dismantling the last part of the sandbox protection. [53]

4.3.3 Attacks from applications

An attacker can also use WebView to target web pages. After a user has installed the malicious application, it is possible to load a target web page into its WebView. Up to this point, no harm is done, and many legitimate applications utilize this feature as well. [53] An example is the Android application known as *FriendCaster for Facebook* (not developed by Facebook), which uses a WebView to browse Facebook. [53] The relationship between server, WebView and application is illustrated in figure 4.7.

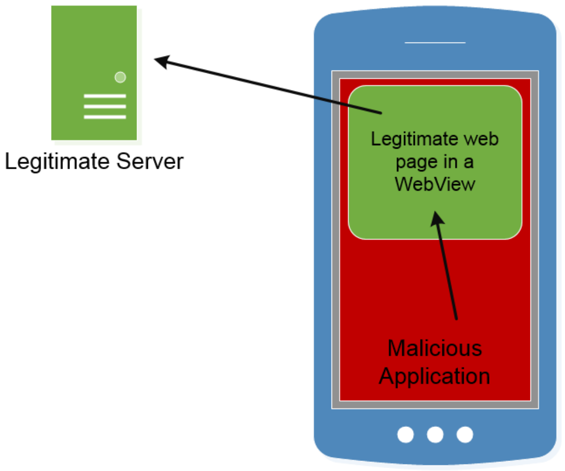
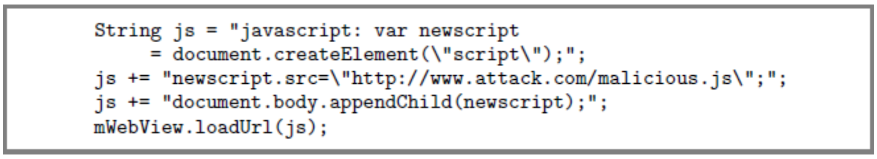


Figure 4.7: An attack from a application using WebView to access the target web server.

There are many ways to execute this type of attack, but the two main categories are *JavaScript injection* and *Event sniffing and hijacking*. [53]

JavaScript injection

JavaScript injection can be done by using a WebView to deploy JavaScript code into the targeted website. An application can make use of the WebView `loadUrl()` API to inject JavaScript code into web pages. [53] See figure 4.5 for an illustration of how to use `loadUrl()`. JavaScript code introduced by WebView and `loadUrl()` that is executed in a web page, has the same privileges as the rest of that page's JavaScript code. [53] This means that the injected JavaScript code can manipulate both cookies and the page itself. Figure 4.8 is displaying the code lines necessary to execute a simple JavaScript injection attack.



```
String js = "javascript: var newscript
     = document.createElement(\"script\");";
js += "newscript.src=\"http://www.attack.com/malicious.js\";";
js += "document.body.appendChild(newscript);";
mWebView.loadUrl(js);
```

Figure 4.8: Code illustrating a JavaScript attack. [53] The Java code constructs a string containing the JavaScript code. When this code is run in the context of the web page, it fetches additional code from an external server and executes it.

Event sniffing and hijacking

Event sniffing and hijacking takes advantage of the APIs used for interaction between the WebView and the web page. [53] By listening and intercepting these APIs, the attacker can complete attacks from outside of the WebView. [53] The WebViewClient class has a set of methods (see the Android developer page for a full list, [54]) that an application can use to register event handlers to a WebView. If a user does something inside the WebView that triggers an event, the application will be notified.

Event sniffing When a page is loaded, the `doUpdateVisitedHistory` is called. It is used to notify the host application to update its visited links database. This trigger can be used by the application to get a list of URLs the user has visited. A more severe case is when the application is able to intercept and alter the event that is triggered.

Event hijacking `shouldOverrideUrlLoading` is triggered when a new URL is about to load. It gives the application the ability to intercept and modify the URL. The application then has the ability to send the user to a different web site than what is expected. Added to that, the application can alter the scheme name of the URL, like using `http://` instead of `https://`.

4.3.4 WebView attack and Threat Model

WebView attacks are very effective, and has very few protective measures. Permissions can be a little effective. In an attack from a web page the host application defines the permissions, and the attack is limited by what permissions are available.

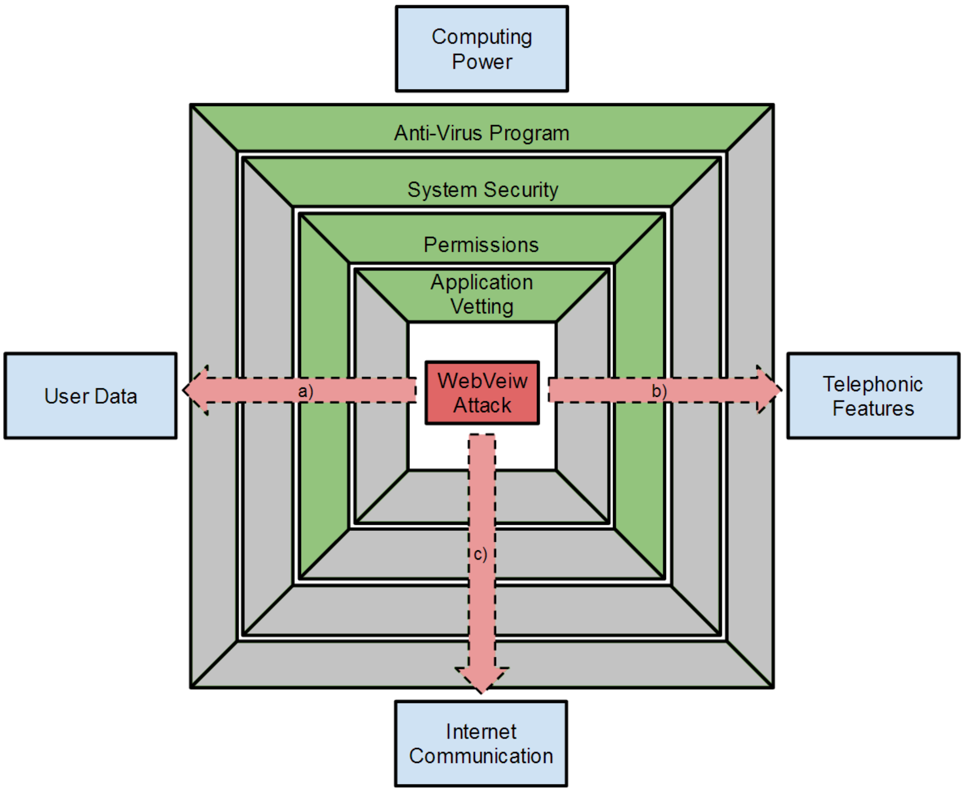


Figure 4.9: Attacks related to WebViews and their effect illustrated using the Threat Model.

Chapter 5

New and future solutions

With the ever increasing number of registered cases of malware, ways to improve smartphone security is needed.[3][5] In this chapter, some possible answers to this challenge are presented.

The first two solutions suggested are both based in hardware. Android smartphones are manufactured by a range of companies, and so hardware solutions need to be easy to materialize and put into action. Advertisements are subject to be exploited in attacks today, which is explained by the way the advertisements are handled in the Android system architecture. Three separate ways to fix this issue are provided here. The final area of improvement proposed here is an overhaul of the WebView technology used in many popular applications today.

Each change proposed in this chapter is compared to the current security solution only. For a comparison of the new solutions, see chapter 6.

5.1 Hardware based solutions

Anyone who wish to can make smartphones running the Android OS. Chapter 2 addressed this, and pointed out that fragmentation is a serious issue concerning the overall device security. Whereas Apple controls the value chain in iPhone production all the way from its customized hardware to what applications get green-lit, the same can not be said for Android-driven phones. This means the Android OS cannot be tailored and optimized for a specific hardware setup. Security through hardware is really about the OS utilizing everything that can be enabled by the hardware, which is difficult when the hardware elements vary, as they tend to do for Android devices.

One solution is adding hardware to a finished product, which then becomes a form of security barrier. It is presented as the first of two solutions. Solution number two capitalizes on the fact that one specific piece of hardware from one specific manufacturer happens to be in nearly all smartphones today.

Listed below are some notable advantages and disadvantages of the current situation in hardware based smartphone security.

Advantages

– Each manufacturer's freedom in choosing hardware and product materials facilitates a wide selection of options differing in quality and price, rendering vast new markets possible, thus increasing sales substantially.

Disadvantages

– There are no areas in hardware dedicated to run operations that require complete confidence and security.
– Today's hardware solution does not accommodate separating the device's operations environment into discrete domains, e.g. secure for business use and non-secure for private use.
– An Android smartphone's hardware and software are designed and produced by different people, which can make it hard for the OS to utilize the hardware elements to full effect.
– Software based AV solutions may drain system resources, even though AV companies strive to minimize the system load caused by their solutions.[31]

5.1.1 Inserting additional hardware

A micro-SD solution by Cupp Computing

Cupp Computing is in the final stages of developing a hardware based, self-contained solution based on a Micro SD card.[55] The purpose of their solution is to provide anti-malware functions, firewall, Domain Name Server (DNS) filtering and Intrusion Detection System (IDS)/Intrusion Prevention System (IPS). In addition, the technology introduces the possibility to separate a device into two domains; a private domain and an enterprise/business domain.[55]

The Cupp solution is based on a Micro SD card package with 8 Gigabyte (GB) of flash storage, 512 Megabyte (MB) of Random Access Memory (RAM), an ARM Cortex-A system on a chip (System on a Chip (SoC)) and a hardened, Linux-based OS.[55]

Notable features

Improved device performance Traditional software based security, like antivirus programs, is operating within the OS, runs on the device's CPU and takes up RAM. This is not the case for external hardware based security systems, because it provides an additional CPU and its own RAM. The result is less load, overhead and general increase in performance (relative to software based solutions) for the device.

Non-invasive Bring your own device (BYOD) High level of security can be introduced to any device with the introduction of a Micro SD card solution, such as Cupp's. Removing the card from the device will return the device to its original state.[55] This results in two different security schemes without impacting the users device, which can provide a non-invasive BYOD, and is a feature useful for enterprise and business scenarios.

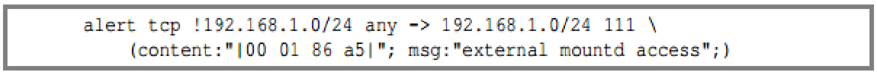
Hardware-based key management Seeds for encryption protocols can be stored on the Micro SD card, providing a separate and more secure location. This will prevent exploits in applications and restrict access the Android system's secret encryption keys.[55]

Network monitoring To perform IDS/IPS, Cupp's solution uses a version of Snort ported to work on an ARM platform. Snort is a network intrusion system capable of performing real time traffic analysis, and it can be run in three different modes:[56]

- **Sniffer mode** reads packets of the network.
- **Packet Logger mode** logs packets.
- **Network Intrusion Detection System (NIDS) mode** performs detection and analysis on network traffic.

NIDS works together with a set of defined rules. These rules are enforced on the packets that flow through the network adaptor and then decide what action to be taken. Snort rules are a way of performing detection, and they look to detect vulnerabilities without using signatures (which are normally used by AV programs, see section 3.3). Signature based detection look for known exploits, and an exploit is a method or technique to take advantage of a vulnerability. By performing vulnerability detection, Snort is capable of detecting unknown exploits. This is a huge advantage over signature-based schemes, because it can be used to detect zero-day attacks.[57] A set of freely distributed community rules are provided by Snort (and its community) and can be found on their web site[58]. A basic rule is illustrated in figure 5.1.

Application scanning Many of the more powerful techniques for detecting malware are not possible to perform on an Android device because of restrictions introduced by the operating system (see section 3.1 for more information about the Android system security). The permissions based security model prevents interaction between applications, and this limits what software based security programs can achieve. This is because they are in fact just another application, and have to operate



```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 111 \
    (content:"|00 01 86 a5|"; msg:"external mountd access";)
```

Figure 5.1: A basic Snort rule used for address negation.[56] The first part (before the parenthesis) is the header and in this case it detects any orients from outside the 192.168.1.0/10 network. Inside the parenthesis is the rule option and it sends an alert if the header gets a match.[56]

with the same restrictions that applies to all applications. The result is that software based security programs can not provide sandboxing or do analysis of application binaries. [55]

By having dedicated off-device security solutions, Cupp has the ability to quarantine incoming applications and denying them access to the host device.[55] In addition, they can then perform more advanced application security analysis, which includes scanning of the APK and examination of the internal code.[55] Figure 5.2 provides a basic illustration of the application interception and file examination.

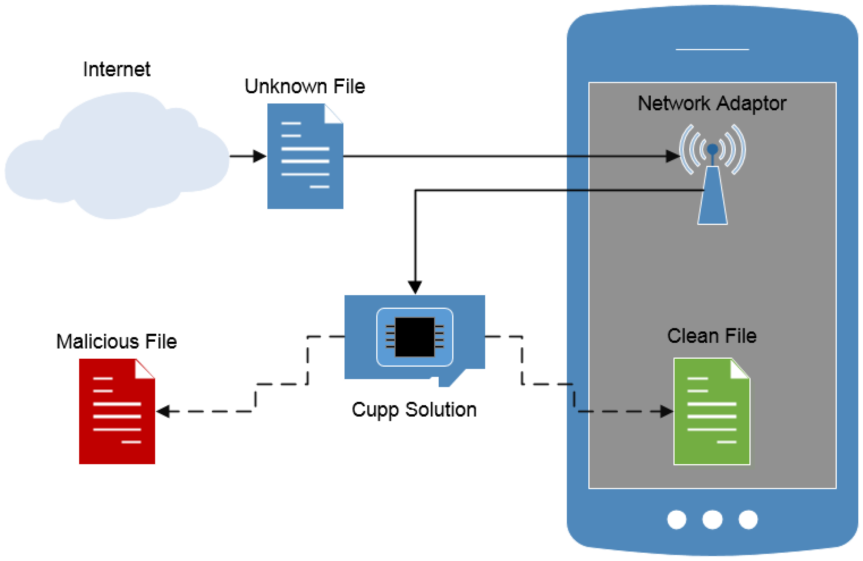


Figure 5.2: An incoming unknown file is intercepted and analysed by the Cupp solution.

Advantages and disadvantages

The solution that Cupp provides (or a similar type of attachable hardware solution) responds well to the disadvantages of the current system:

– A separated security solution will introduce an area dedicated to security operations.
– With the solution provided by Cupp, it is possible to have a non-invasive solution for separating the device.
– This solution is independent from device manufactures.
– By providing a Micro SD card with its own CPU, memory and storage space, the system load is drastically reduced.

In addition, it introduces a lot of positive new features, which were elaborated in the previous section. The most prominent of these features are off-site analysis of applications, network monitoring and the reduced system load. Additionally, this solution does not have any prerequisites and can be used with any device.

The advantages of the current system is mostly upheld, but by removing the security away from the device OS it puts more pressure on developers to get the security solution fault free and reliable. Hardware security based on the introduction of additional hardware will still be able to support multiple platforms and manufactures. Because the solution is independent from the device itself.

Probably the most significant disadvantage of the additional hardware solution is the complexity and challenges it presents in terms of usage to a regular user. In addition some other security conserves appears, among others; what happens if the Micro SD card containing the security solution gets lost or stolen? Can an attacker access and take advantage of the sensitive inform located on the card? To defended against this scenario some form of access control has to be introduced, which again results in more complications for the end user.

Cupp and Threat Model

The Cupp solution introduces a lot of new security features, some of which are not already present in the Threat Model. Application Scanning and Network Monitoring each get their own layer in the Threat Model, and the non-invasive BYOD is an improvement on the System Security layer.

5.1.2 New mechanisms in existing hardware

ARM's TrustZone technology

ARM is a British company best known for designing processing units (CPUs). Their processors have become the predominant choice among smartphone manufacturers, with a market share in 2010 in excess of 95 percent.[59] ARM develops the architecture and instruction set, but does not in fact manufacture the CPUs. Instead, it licenses the designs and instruction set architectures to third parties. Companies such as

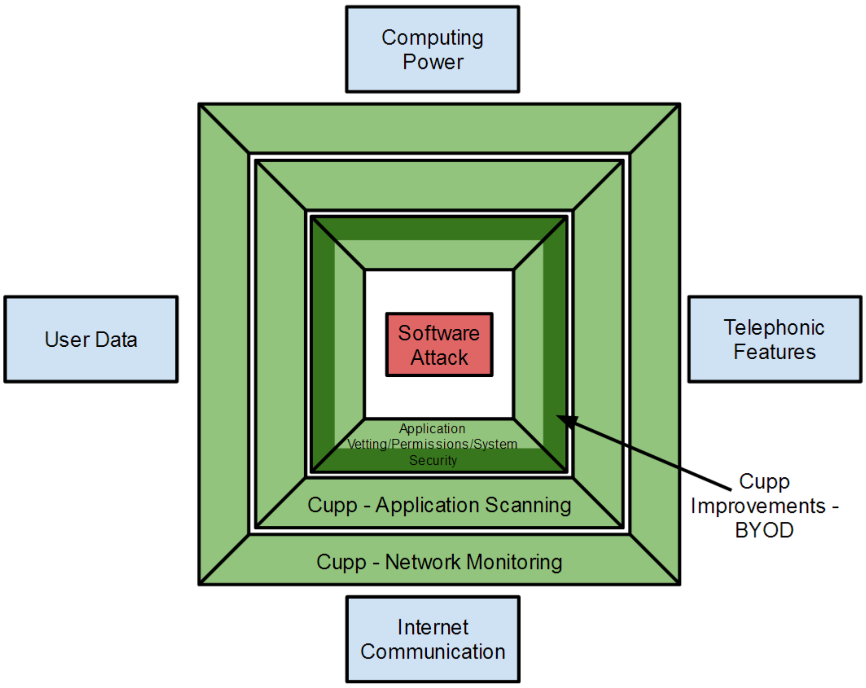


Figure 5.3: Threat Model for Cupp security solution.

Apple, Samsung, Nintendo, Nvidia and Texas Instruments are all licensees of ARM's designs, utilizing its logic in production of their own SoCs.

The ARM TrustZone technology is another of ARM's designs, but not a standalone product. It is a security extension included in most of their newer architectures. The latest smartphone models from both Apple, Samsung and Sony contain a SoC based on logic where TrustZone can be enabled[1]. Whether or not the smartphone manufacturers actually have included the feature in their models, is unknown. Apple has something it calls 'Secure enclave' embedded in the architecture of its latest SoC (Apple A7), which is very similar in operation to that being enabled by ARM TrustZone, making it likely that Apple utilizes it in some way.[60]

The TrustZone solution

The primary security objective of the architecture is to enable the construction of a programmable environment that allows the confidentiality and integrity of assets to be protected from specific attacks. ARM TrustZone is designed to provide this without adding an additional dedicated security core to a SoC. The security of the system is

[1]Apple: ARMv8-A; Samsung and Sony: ARM Cortex A9 / ARM Cortex A15

achieved by partitioning all of the SoC hardware and software resources so that they exist in one of two worlds – the Secure world for the security subsystem, and the Normal world for everything else. Hardware logic present in a TrustZone-enabled bus fabric ensures that Normal world components do not access Secure world resources, enabling construction of a strong perimeter boundary between the two. (See figure 5.4 for a simple visualisation.) The world switch is generally orthogonal to all other capabilities of the processor, thus each world can operate independently of the other while using the same core.

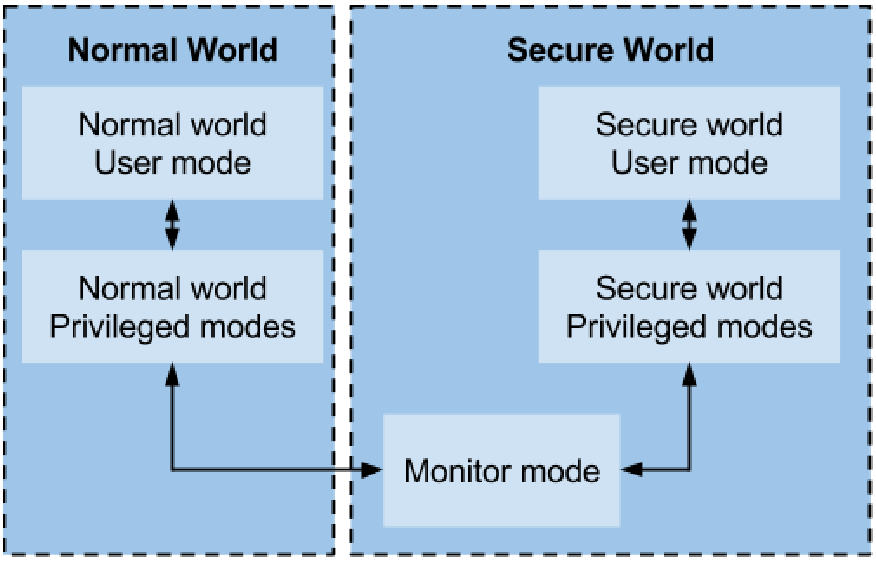


Figure 5.4: Hardware architecture of ARM TrustZone.

Entering monitor mode from the Normal world requires software to execute one of a set of dedicated instructions that may trigger entry. The monitor mode software views all such instructions as exceptions, carefully assessing each before eventually granting access to the Secure world, and is a tightly controlled mechanism. When switching world states, the monitor mode software saves the state of the current world and restores the state of the world at the location to which it switches. It then performs a return-from-exception to restart processing in the restored world. This makes the trusted and non-trusted states not only discrete in virtual memory, but also in time. A separation like this enables the application core to switch between the two worlds in a manner such that information can be prevented from leaking from the more trusted world to the less trusted world. Memory and peripherals are then made aware of which world state the core operates in, and can thus secure the communication to the kernel.

TrustZone features

Which challenges in the mobile security world does this technology seek to answer? First off, the ARM TrustZone is not providing a fixed one-size-fits-all security solution. Neither is it a plug-and-play device, something ready out of the box. It is an enabler, providing the infrastructure foundations that allow a SoC designer to choose from a range of components that can fulfill specific functions within the security environment. Implemented correctly, TrustZone ensures that sensitive data is stored, processed and protected in a trusted environment. An example is Apple's way of handling its so-called TouchID, a feature where users can use their fingerprint to unlock a screenlocked device. Apple stores fingerprint templates in the Secure world (or secure enclave, in Apple's example), and has the fingerprint scanner only operate when the Secure world state is active.[60] Likewise, other peripherals such as keypad, camera, Near Field Communication (NFC) and bluetooth communication may be secured in the same manner. In fact, any application is eligible to run in the trusted environment, as long as a trusted user interface is created. It is a matter of choosing which assets to protect.

A mobile payment application is another example. To ensure secure payment, the application can display payment information in a trusted window, having the user input a PIN on the keypad (or using a fingerprint for that matter) to accept the wireless transaction there. By securing the channels through which users feed a smartphone device with input, sniffing and stealing information by hijacking the user input becomes a lot more difficult. Below is a list of application examples.

- Secured PIN entry for user authentication in mobile payments & banking
- Enable deployment and consumption of high-value media (DRM protection)
- Separate private and business mode in BYOD
- Software license management
- Loyalty-based applications
- Access control of cloud-based documents
- Secure boot
- Certificate management

Typically, a rich operating system is run in the Normal world, and the smaller security-specialized features, like those in the list above, are run in the Secure world.

TrustZone in Android

The Android OS and ARM TrustZone in tandem is a plausible scenario. With ARM being the principal supplier of processor logic in the smartphone world, there are no evident hardware reasons to exclude the technology today. It could, however, be said to counteract the advantage mentioned in the chapter intro, that Android benefits from the multitude of hardware and smartphone manufacturers. One brand having exclusive rights to some part of the Android package is not in accordance with how the Android universe has been marketed so far.

Determining whether each of the relevant smartphone manufacturers have enabled TrustZone in their latest chipsets, is beyond the scope of this report, but the option to do just that certainly seems within their reach. Whether or not the Android OS in itself is compatible with TrustZone's mechanics, requires some deeper analysis, but is very likely and therefore assumed to be possible.

TrustZone and the Threat Model

The impact of TrustZone to the Threat Model is visualized in figure 5.5 below. The outcome is a solidified **System Security** layer, more likely to deflect attacks on user data in particular.

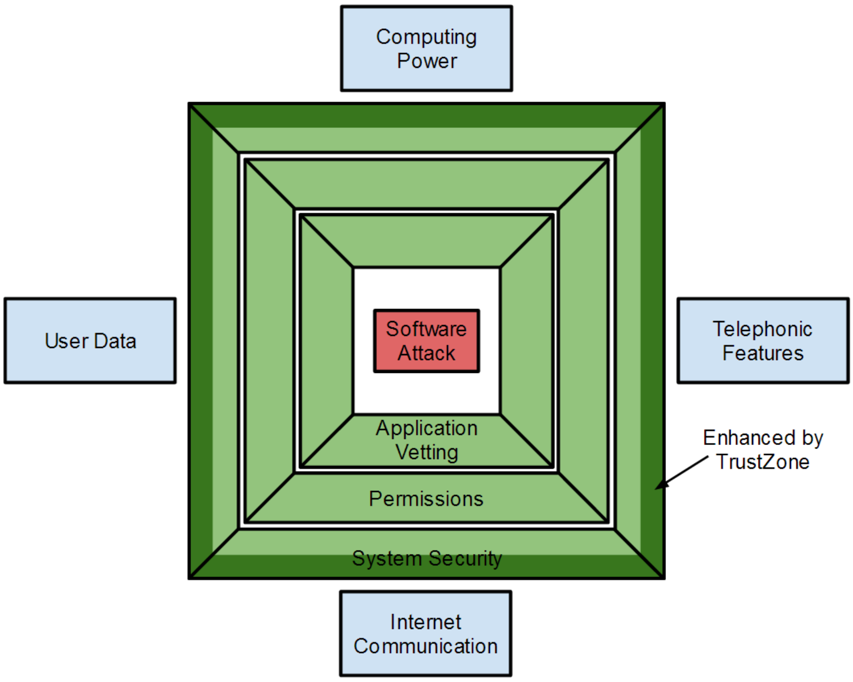


Figure 5.5: Threat Model for TrustZone security solution. The darker shade of green in the system security layer marks the expected impact of the TrustZone technology.

5.2 Advertisement solutions

Most of the security problems related to today's advertisement solution originate from the fact that the Advertisement Library is embedded into the host application (see section 4.2). The Advertisement Libraries require additional permissions and a new activity has to be added. This has both positive and negative results:

Advantages

- Because the Advertisement Library is so strongly embedded, it is hard do separate and block it. This makes it difficult to defraud the Network Provider.
- Its implementation method is fairly easy, and does not present the developer or the user with technical difficulties.

Disadvantages

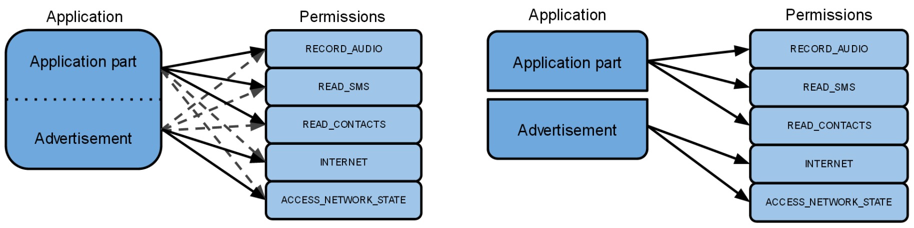
- Application and Advertisement Library share permissions and have access to the same system resources. This also includes unnecessary private information and user data, which the user might be unwilling to share with the Network Provider. In addition this information can be abused by malicious parties. See section 4.2.1 for more information.
- Because the advertisement API is run in an activity it has the ability to execute code, and this leads to the dangerous possibility of downloading additional malicious payloads at runtime. See section 4.2.2
- End users have no definitive way of knowing if an application contains advertisement.
- End users also don't know what permissions are assigned to the host application and what are meant for Advertisement Library.

The new proposal has to resolve as many negative effects of the current system as possible, and at the same time try to keep the positive. In addition it would have to retain the basic advertisement functions; displaying banners and whole screen ads, support different providers and offer the developers revenue.

In the current advertisement system there is no real way for users to protect themselves from malicious ads. One possible solution is to try and avoid free applications, but this can be complicated and expensive. This is because the user has no definitive way of knowing if an application includes advertisement, and the ad free version usually costs money. Some possible improvements to increase the security for users are presented next.

5.2.1 Permission separation within applications

A solution easily envisioned is to create different permission domains within an application.[61] That way you can have one domain for the host application which keeps all the original permissions, unaffected by the other domains (see figure 5.6). In a different domain you can have your advertisement logic, and thus limit the permissions for that domain. The methods called between the Advertisement Library and the host application will remain unchanged and run in the same virtual machine, but they will then have separate permissions.



(a) Current system. **(b)** Domain separation system.

Figure 5.6: Relationship between application and permissions in the current and the domain separation system.

This solution retains all the positive attributes of the current system, because there are minimal changes to the way the advertisement logic is included in the application. The only difference is that the Advertisement Library is defined as a subdomain in the application and will have its own set of permissions.

The solution addresses several disadvantages from the list. The separate permission domains will remove the vulnerabilities created by the shared permissions, and the same domains let the user know what permissions are assigned to the main application. The ability to execute downloaded malicious code will still be there, because the Advertisement Library is still embedded in the application and will share the same UID, but the malicious possibilities will be limited by the reduced number of available permissions. Users still wont have a definitive way of recognising advertisement in an application, because you will not be able to identify that a specific sub domain contains advertisement.

The problem with this solution is that it requires modifications to the underlying OS, and it has to be modified to support two permission domains. This is a change that has to be made by the official Android developer team, and requires more effort to deploy than if the changes could be made by the application developers. A more severe problem is that the Dalvik virtual machine (see 2.3.1) does not support separation of code within the same virtual machine.[61] This is a feature necessary to prevent different parts of the application to obstruct one another from executing correctly, which may serve as a challenging obstacle for the implementation of the domain separation.[61]

Effect on Threat Model

This solution improves the permission issue for advertisement attacks, and the result is that the Permissions edge in Threat Model 5.7 is graded darker then in the previous model (figure 4.3).

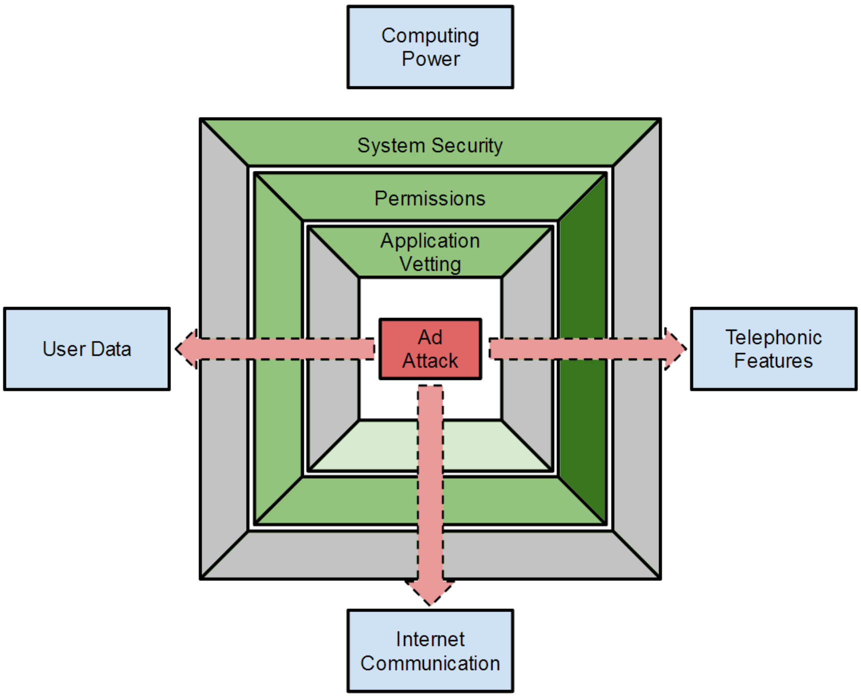


Figure 5.7: Threat Model for the Permission separation within applications solution.

5.2.2 Application separation

A different approach is to achieve the permission separation by having the Advertisement Libraries be in a separate applications [61], as applications don't share permissions (see 3.1). When a user installs a regular application, it will inform the user which Advertisement Library it requires and prompt the user to install that advertisement application (if its not already installed). Each different Advertisement Library will require one application, and all regular applications will connect with one (or more) of those applications (see figure 5.8). Both applications will have its own set of permissions, and regular applications will include a new permission; ADVERTISEMENT. This permission is used to allow applications to display advertisement, inform the user that an application contains advertisement and that it is dependant on another application to function properly. To display the ads, the system has to show two applications on the screen at the same time. To achieve this the Android system has to be modified or the solution has to include some prototype features allowing the background of an activity to become transparent[62]. In addition to displaying both applications, the advertisement applications must have the ability to register "click" events, which provides even more obstacles.[62]

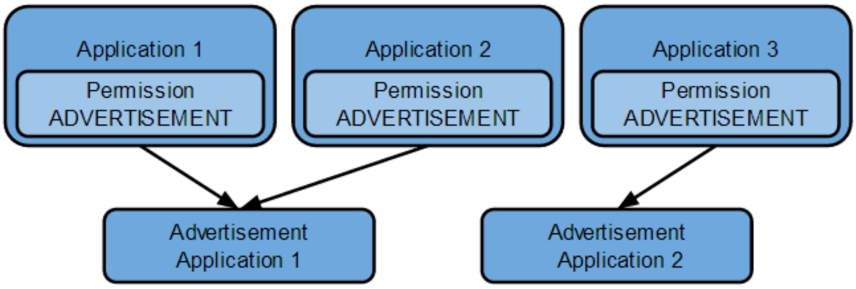


Figure 5.8: Separate Advertisement Library applications connected to different regular applications.

A problem with this solution is that both the positive attributes of the current solution are lost. With the advertisement logic placed in a different application it will be much easier to uninstall or block it, which may hurt developers that rely on advertisement revenue. In addition, the new system has a more complicated relationship between the regular application and the Advertisement Library. The result is a more complicated use for both the developer and the user.

The loss of positive features is compensated for by taking care of the negatives very well. First, by splitting the Advertisement Library and host application to two different applications, the sharing of permissions issue is resolved and they will not have the same UID. It will still be possible to download and execute malevolent payloads, but the lack of additional permissions limits the problem. Second, providing additional protection for this threat can be accomplished more easily because the advertisement logic is isolated in a separate application. End users will have a certain way of identifying advertisement in an application (through the ADVERTISEMENT permission) and what permissions that advertisement requires.

The main drawbacks with this solutions is the renunciation of the positive features of the current solution. It also requires modifications to the operating system or new application features to be able to display two applications on the screen at the same time.

Effect on Threat Model

The improvements in this solutions result in a darker graded permission edge in Threat Model 5.9 compared to the previous model (fig 4.3).

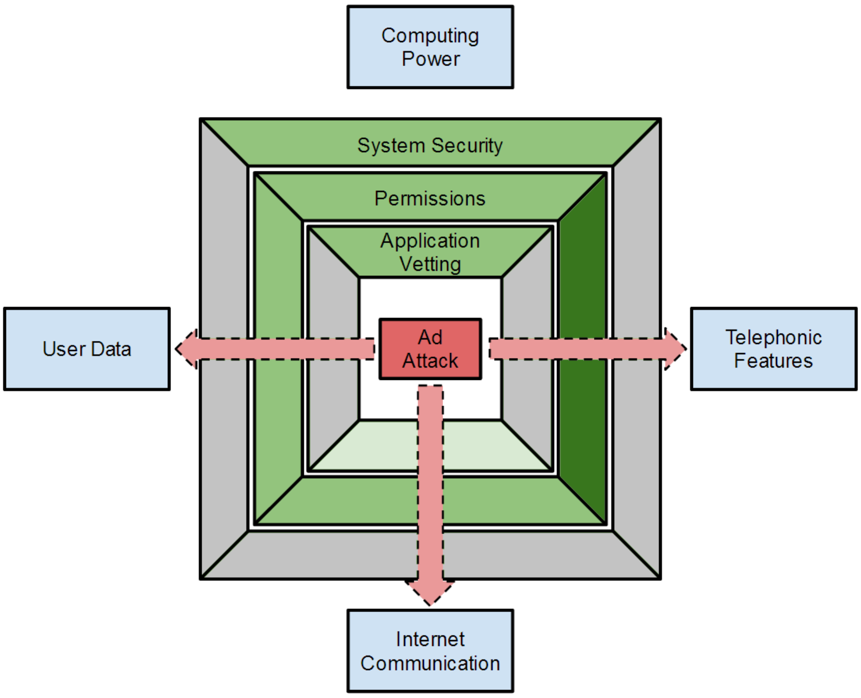


Figure 5.9: Threat Model for the Application separation solution.

5.2.3 AdDroid

The third solution is a proposition presented in a paper by Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner, named "Addroid: Privilege separation for applications and advertisers in Android".[61] It suggests including the Advertisement Library in the Android system and not the advertisement Publishing Network. This system will require a new permission; ADVERTISEMENT, such that when an application declares this permission they will get access to an API that provides advertising functionalities.[61]

AdDroid consists of two parts. The advertisement API is the first of them, and it exists in the application space (within the host application). Classes and methods used for advertisement functions are provided to the developer through this API.[61] This includes which advertisement network to connect to, and a new user interface element for displaying the ad banner. The API will be the same for all applications,

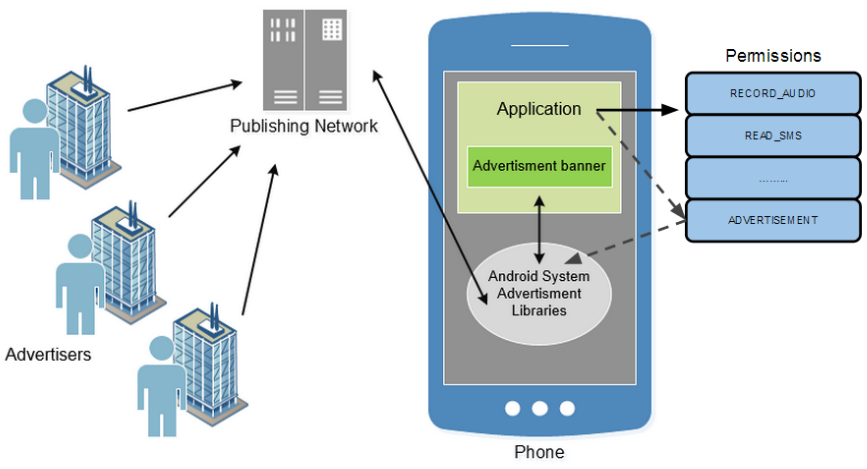


Figure 5.10: Relationship between Advertisers, Publishing Network, Advertisement Libraries and Application in AdDroid.

no matter what Publishing Network is selected by the developer.[61] Because the API is located in the application space it also shares the host application's permissions.

To avoid permission sharing between the host application and the Advertisement Library, the sensitive functions are placed in a system service instead of the API.[61] AdDroid's system service is located in the Android system services outside of the application's internal space. Access to the system service is protected using the ADVERTISEMENT permission. When an application requests a new ad to be displayed, it will make an IPC call to the system service that then performs the permission restricted operations.[61] See figure 5.11 for an illustration of the relationship between the API and the system service. Most of the advertisement logic is placed in the API, and the only functions located in the system service are:

- Contacting and fetching ads from the Advertisement Provider.
- Receiving the ads sent by the Advertisement Provider.
- Providing the application with the received ads.

This solution does very well with regards to the advantages of the current advertisement system, and even improves on the implementation method. The solution where the Android OS handles most of the advertisement logic will be a much more straightforward process for developers. Because the Advertisement Library is already included in the system many of the steps of implementing the Advertisement Library (see section 2.6.1) will not be necessary. Examples of unnecessary steps are declaring the Advertisement Library activity and implementing the code provided by the advertisement Publishing Network. Additionally, the only required change to the

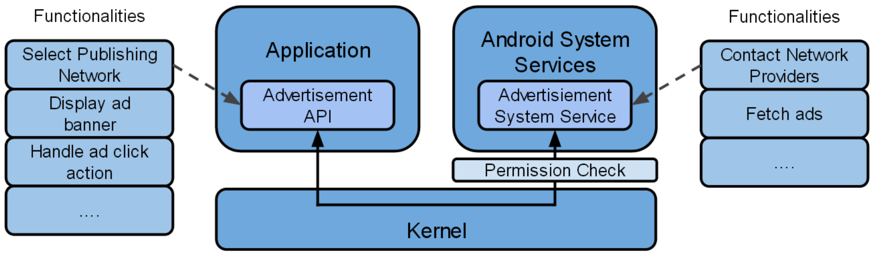


Figure 5.11: Connection between the advertisement API (located within the application) and advertisement system service (located within the Android system services domain).

manifest is the inclusion of the ADVERTISEMENT permission.

AdDroid solves all of the listed disadvantages with the current advertisement solution:

- With the permission separation in place for the Advertisement Library, advertisement Publishing Network can not request sensitive information from a user's device.
- Because the Android system provides the Advertisement Library code, malicious advertisement Publishing Networks has no way of introducing malevolent functions and vulnerabilities into it.
- The ADVERTISEMENT permission alerts the user that the application contains advertisement.
- In this solution all the permissions listed is assigned to the host application.

A major complication related to this solution is that the official Android developer team has to implement this solution in the OS, which is more trouble than if the solution could be deployed by developers. In addition, in the current advertisement solution the advertisement Publishing Network update and introduce new functionalities into their own Advertisement Library, and this happens independently of Android releases. With AdDroid, updates to the Advertisement Library has to come through a new official Android system update, which then suffers from the mentioned OS fragmentation (figure 2.2).

Effect on Threat Model

Like the previous two solutions, this also improves the **Permission** edge, resulting in a darker graded green in the Threat Model (figure 5.12) compared to the previous model (figure 4.3). Unlike the others, the AdDroid solution also improves the System Security, and thus helps prevent malicious Advertising Libraries and fetching of dynamic code-threats that target Internet Communication assets.

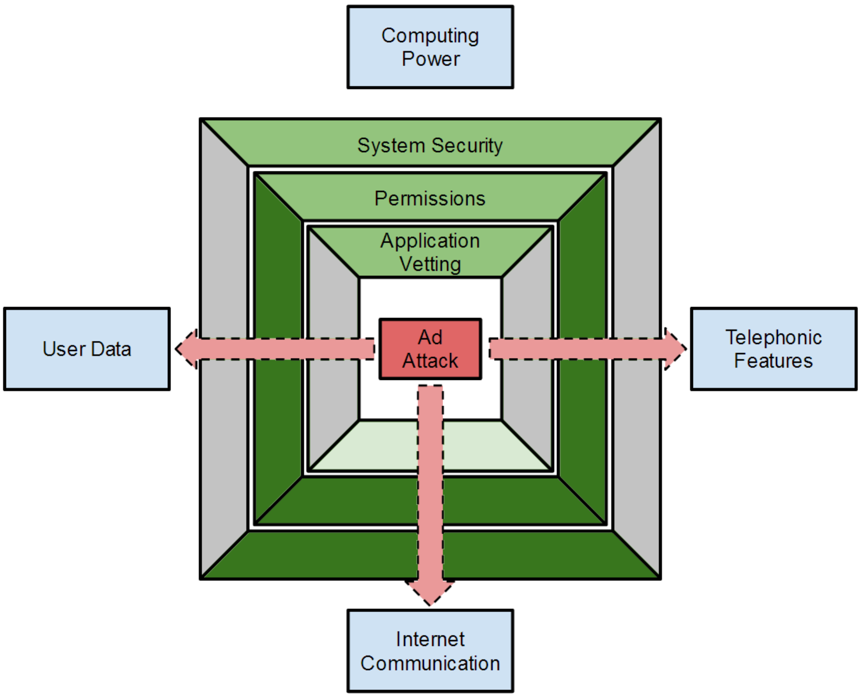


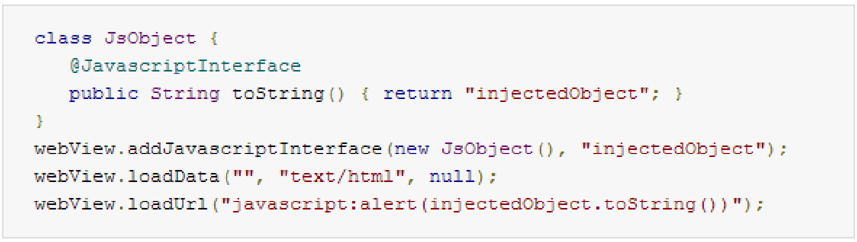
Figure 5.12: Threat Model for the AdDroid solution.

5.3 WebView solutions

The solutions here are designed to prevent WebView attacks from a malicious web page, like that seen in section 4.3.2.

5.3.1 Android system solution

With Android system API level 17 (Android 4.2 Jelly Bean), protection against WebView attacks have been added.[63] Applications targeting this API level receive the following notification: *"`WebView.addJavascriptInterface` requires explicit annotations on methods for them to be accessible from Javascript."*[63] This means that any developer has to annotate any public methods they want to access with JavaScript with "JavascriptInterface". See figure 5.13 for an example. According to Android's official web page, 32,3 % of all Android devices run on system API 17 or higher, as figure 2.2 shows.[13] For previous Android system API levels (Android 4.1 and below), all public methods are available through `webView.addJavascriptInterface`.



```
class JsObject {
   @JavascriptInterface
   public String toString() { return "injectedObject"; }
}
webView.addJavascriptInterface(new JsObject(), "injectedObject");
webView.loadData("", "text/html", null);
webView.loadUrl("javascript:alert(injectedObject.toString())");
```

Figure 5.13: Example code with an annotated method that allows `webView.addJavascriptInterface(new JsObject(), "injectedObject");"` to be executed. [64]

Devices using Android system API of 4.1 or lower equal 67,7 % of all users. These are all vulnerable to attacks exploiting WebViews, and developers should be cautious using the `addJavascriptInterface` at all. The Android developer team are aware of this problem, and a note of caution is included in the documentation for this method.[64]

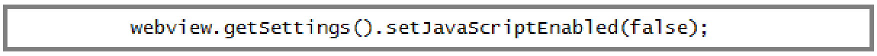
5.3.2 Alternative security measures

There are some "best practice" solutions for developers planning to include WebViews in applications available to devices not using Android OS 4.2 or later.

Disable support

When using a WebView, the developers should not enable more functions than necessary, and try to minimize some of the more dangerous ones. Three example cases are presented below:

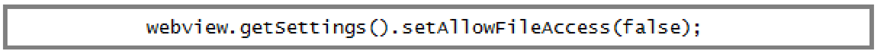
Disable Support for JavaScript If the WebView won't use JavaScript, there are no reasons for having this option enabled. The Android `WebSettings` class can be used to disable JavaScript within the WebView, and the method necessary is illustrated in figure 5.14.[65] This method is disabled by default in the system, but it is good practice to explicitly disable it.



```
webview.getSettings().setJavaScriptEnabled(false);
```

Figure 5.14: Code for disabling support for Javacript in a WebView.[66]

Disable file system access The Method `setAllowFileAccess(false)` from the `WebSettings` class can disable WebView access to the files system, as the example code below (figure 5.15) shows.[65]



```
webview.getSettings().setAllowFileAccess(false);
```

Figure 5.15: Code for disabling access to file system for a WebView.[66]

Disable JavaScript bridge The use of `addJavascriptInterface` can bind an object so that methods can be accessed from JavaScript.[64] This violates Same-Origin Policy (SOP) and should only be used if absolutely necessary, as it is the root of many WebView problems.[65]

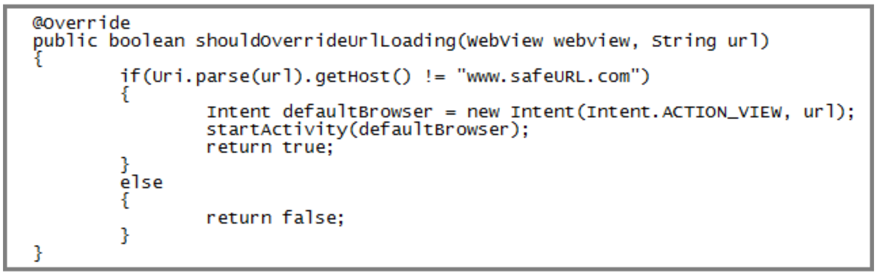
Prevent loading content from third party hosts

If a user interactively requests a resource from within a WebView, the method `shouldOverrideUrlLoading` can be used to intercept the URL which is about to load and determine if its safe to load it or not.[65] An example code is presented in figure 5.16 that examines the new URL, and if it is not a safe URL the user will be redirected and the URL will be opened in the native web browser instead of the WebView.

A problem with this solution is that the `shouldOverriedUrlLoading` does not intercept resources loaded from within an IFRAME or a "src" attribute.

Resource Inspection

To intercept resources loaded from within an IFRAME or a "src" attribute, the method `shouldInterceptRequest` from the `WebViewClient` class can be used. The



```
@Override
public boolean shouldOverrideUrlLoading(WebView webview, String url)
{
        if(Uri.parse(url).getHost() != "www.safeURL.com")
        {
                Intent defaultBrowser = new Intent(Intent.ACTION_VIEW, url);
                startActivity(defaultBrowser);
                return true;
        }
        else
        {
                return false;
        }
}
```

Figure 5.16: Code to inspect and evaluate a newly loaded URL in a WebView. If the URL is not a trusted one the native browser will launch and the new URL will be opened there.

method notices the application of a resource request, which contains the Uniform Resource Identifier (URI) of the resource. A pattern matching scheme can then be used to inspect the request. Some useful methods for the scheme are `getHost()`, `getScheme()` and `getPath()`. The method can for instance be used to detect requests for JavaScript resources.[65]

5.3.3 WebView solutions and the Threat Model

All the solutions presented here are used to improve the Android system by limiting the unnecessary accesses and functions of a WebView. These system improvements are placed in the System Security domain in the Threat Model, and the improvements are illustrated in figure 5.17.

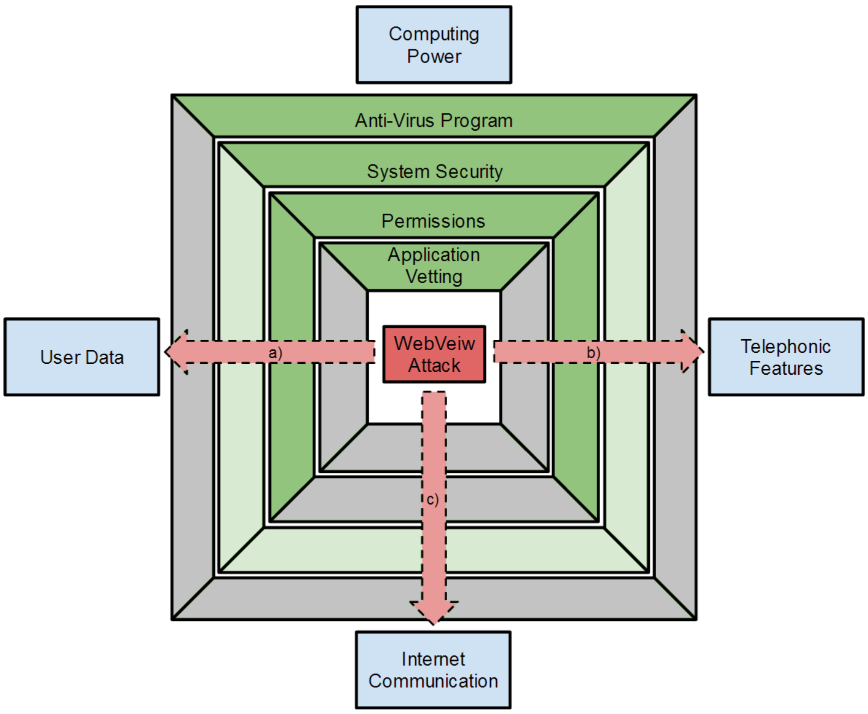


Figure 5.17: Threat Model for the new WebView solutions. The System Security layer is graded light green because the official Android solution only affects a subgroup of the users. The other security measures are not total fixes but makes the best out of a bad situation.

Chapter 6 Evaluation of security measures

In this chapter, the adequacy of both the existing and the proposed new solutions is analysed. The solutions are evaluated through the eyes of three different user groups, which are presented immediately after this chapter's introduction. Following that, the current security solutions are assessed on the grounds of their effectiveness in averting attacks for the different user groups. A quick summary then reveals where the shoe pinches in smartphone security today.

Each of the possible new solutions are then weighed in turn, measured to see what they can do to remedy today's vulnerabilities. The evaluation of the new solutions is also based on how useful they are to each of the user groups, as well as how much effort it takes to deploy and utilize.

6.1 Classifying user groups

We have created three separate user groups in order to provide the reader with more comprehensible analyses. We classify the smartphone users into three groups as follows:

A regular user is one who uses the smartphone mostly for socializing and personal recreation. He or she installs a number of popular applications, but many have limited budgets or lack the will to pay, and prefers free versions over paid if presented with a choice.

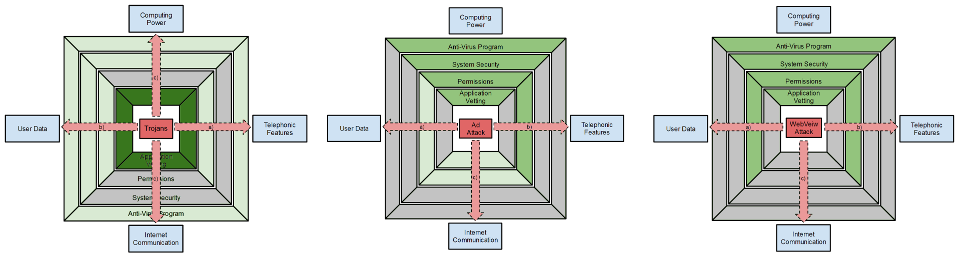
The employee or business user, is one who typically got the phone from their employer. Uses the phone to read and to write (sensitive) email, access corporate documents and store business contact information. All the same, it is also used for non-work-related pastimes, such as general Internet activity, social media and games.

High official refers to people in central positions in government or military, along with people in high-profile private corporations. Their phone is typically used sparingly, and then only for strictly necessary purposes. No applications installed other than what is used for work, and little or no usage of Internet.

6.2 Evaluating existing security solutions

As pointed out in section 4.1, the most prevailing attack form against smartphones is the Trojan. Trojan attacks have in common that they include the malicious payload as part of an application and thus need to pass the application vetting before being available for users to download (at least through official channels). The security solutions in place today handle most Trojan threats efficiently, but it is only thanks to the **application vetting** that they are effective at all. The Threat Model in figure 6.1a indicates that the **permissions**, **system security** and **AV** mechanisms all have little to no effect. With several examples of Trojan attacks bypassing the current vetting process, as seen both in section 3.2 and section 4.1.1, and no backup plans for when the vetting process fails, there is a need for improvements to the security regarding Trojans.

The situation concerning advertisement attacks is even more grave, as is illustrated clearly by figure 6.1b.



(a) Threat Model Trojans. **(b)** Threat Model Ads. **(c)** Threat Model WebView.

Figure 6.1: Threat Model for Trojan, advertisement and WebView attacks. These models were previously shown in chapter 4.

Application vetting can be said to have some effect against pushing and running code through advertisements, because to enable such behaviour, certain lines of code need to be added where the advertisement network is included. This is done prior to the application vetting and thus should be revealed by a scan. Other than that, advertisements are loaded on application run time, and thus avoid the vetting process.

Permissions have some effect on mitigating the advertisement attack threat. Since advertisements get the same permissions as the applications they run in, they are also equally limited. Malicious advertisements may probe to see where their limits are, but can do nothing to increase their privileges. If a user consequently avoids installing applications requesting unnecessary permissions, advertisement attacks can do little harm on that device. The most invasive permissions are perhaps in the Telephonic Features asset group, followed by the User Data asset groups, which is reflected in the figure above (6.1b).

System security limits attacks through advertisement the same way it does a malicious application, i.e. the sandbox stops it from interacting with other applications. The sandbox is also what's limiting **AV solutions**, stopping them from accessing other application's running processes. The AV solutions thus cannot match advertisements with known malware in their database, which renders them useless.

WebView attacks (figure 6.1c) compare to advertisement attacks in that they depend on a process within an application. **Application vetting** is easily bypassed, since all methods and functions required for an attack are accepted, as many legitimate applications use WebViews as well. A notable difference, though, is that for an advertisement attack to work, the malicious payload must be distributed through an advertisement network, running the chance of being caught and dropped long before reaching a smartphone device. A WebView attack, however, can be conducted by a totally different person than the one developing the application containing WebViews. Weaknesses in the **system security** lets an attacker make the WebView browse his or her web page that may contain a damaging payload on it. Apart from sniffing web history and user data related to the internet permission (which all applications with WebView has), **permissions** are an efficient means to mitigate this threat. There is no known way to elevate privileges through the use of WebViews, and so all restrictions as a consequence of permissions are still valid. **AV solutions** have no effect on malicious WebViews.

Drive-by downloads can be completely thwarted by not allowing for downloads from unofficial sources, which is an option the user controls in the system setting, placing the mechanism under either **permissions** or **system security**.

6.2.1 Impact for user groups

The regular user is affected most by Trojan attacks, as these are the ones best suited to obtain anything of value from a normal user. Most attacks on smartphones have monetization as their ultimate goal, and there are but a very few ways to earn a profit directly from an attack. Given that the average user has little to no information stored on their device of value to anyone else, malware that collects user data does nothing to enrich an attacker. Sending SMSs to premium rate services is the only way awarding near instantaneous reward, while mining cryptocurrencies and ransomware can get the attacker paid after some time. All of these require specific permissions to employ, and only Trojan attacks allow the attacker to choose what permissions the malware requests. That does not mean advertisement or WebView attacks cannot be fruitful, but both depend on the underlying application requesting the necessary permissions.

This user group is the least likely group to install additional security mechanisms by own initiative, and therefore depends solely on what is in place out of the box. The most important feature is thus arguably the application vetting process, which if it worked perfectly, would provide an environment entirely free of malware to those users only using the official Google Play Store.

Business users differ from the regular user on several levels. First, information stored on the device has a good chance of being valuable to an attacker. Access to user data assets such as email, calendar and local files is handled by the use of permissions. Second, business users are likely to have certain guidelines as to what applications they are allowed to download and install, which probably discourages applications requesting permission to send/receive SMS and email. Also, they are most surely only permitted to use the official application store.

As a result, Trojan attacks are less likely to affect this user group than the regular users. However, business users are prone targets for information theft, which means that they are more susceptible to be the specifically intended target of an attack. Disorderliness from an employee can thus prove disastrous for the enterprise if an attacker is just waiting for an opportune moment. A cautious employee sticking to the policy enforced by the employer, on the other hand, should have few worries. Apart from phishing attempts through email or SMS, there seem to be few alarming software threats to business users.

High officials are by definition careful users of their smartphones. With no casual or general use of Internet and most likely a secret number and secret email address, phishing attacks are out of the picture. Trojan attacks are equally unlikely, as a high official downloads next to no applications, and if they do, it is surely a tested and popular application from a known developer account. Advertisement attacks are improbable for the same reasons as in the business user case. Overall, high officials seem a very difficult group to target, as all their actions on the device are likely to be thought through thoroughly beforehand. Plus, complementary security mechanisms may be installed on their devices, obstructing possible attacks even further.

6.3 Evaluating new security solutions

6.3.1 Hardware solutions

Both hardware solutions suggested in chapter 5 are proprietary to parties not involved in manufacturing the smartphone devices, which may impede or deter the implementation of these features to some extent. The micro SD card solution developed by Cupp is the first of the two to be assessed here, with ARM's TrustZone solution next.

Evaluating additional hardware solutions

The product developed by Cupp, which is an additional hardware solution, is capable of providing more security measures by inserting a SD card into the device. Improved performance, non-invasive BYOD, hardware-based key management, network monitoring and application scanning are the notable features introduced by the Cupp solution (see section 5.1.1).

This solution differs from other hardware solutions, because it requires a fair share of interaction from the user. It is not a solution already implemented by the manufacturers, but instead the user has to purchase and insert it into their device and configure it in order for it to work properly. In addition, when the solution is not in use, the user has to take care of the SD card and make sure it's not lost.

Regular users don't have the same need for more security solutions like business and high official users, and thus don't benefit as much when using this solution. Also, this solution is limiting the usability of the device and introduces a lot of extra complications, so this might not be the best fit for regular users.

Business users are the optimal target group for this solution. The ability to introduce more security features to just about any device with the introduction of an SD card fits perfectly into a business environment. In a business where the employees use a lot of different devices, and security policies are hard to enforce, the IT-security department can configure and issue an additional hardware solution (like Cupp's SD card) to all devices to ensure an elevated level of security. In addition, when employees don't need the restrictive security features (like when they are out of the workplace or at home), they can just take out the secure SD card and the device is back to default.

High official users require a high level of security, as they handle the most sensitive information of all the user groups. A solution of additional hardware can help in this situation, because of the security improvements they present, but the fact that they are removable is a downside in this case. Permanent solutions are a better fit, because there are no instances where a high official would benefit from downgrading the security on their device.

Evaluating ARM's TrustZone solution

A design that places the sensitive resources in the Secure world and implements robust software running on the secure processor cores, can protect assets against many possible attacks, including those which are normally difficult to secure. Having a designated time and space in hardware in which to perform security sensitive tasks, is something that is lacking in today's Android smartphones. The solution designed by ARM seems to be able to neutralize some of the disadvantages mentioned in section 5.1. It renders timing attacks performed by other applications more difficult, since these applications are no longer run at the same time, or in the same phone state, as the secured process it seeks to attack. Also, those with BYOD devices may benefit from the possibility of separating the runtime environment into two discrete domains, which should make the switch between business and private mode more effortless.

Section 5.1.2 mentions some examples of how the use of keypad and other peripherals can be secured. There is a list in the same section naming enhancements a Secure World concept can make possible, but in that lies perhaps the biggest weakness too. It all depends on whether manufacturers include the security extension in their hardware setup in the first place and then how well developers manage to make use of the concept. While the potential for improved security is great, the disadvantages listed in section 5.1 state among other things that when different institutions are responsible for designing and making the hardware, the OS and eventually the applications, no one is really in control of the process from start to finish. This, in turn, means that no one has the authority to ensure correct implementation throughout the process and whether it ultimately functions as intended. The ARM TrustZone solution does nothing to improve this aspect. It may be even argued that it contributes to making it worse, spreading the responsibility of implementing a single security measure across both hardware, OS and application developers.

Impact for the different user groups.
In principle, the increased security that stems from using a dedicated hardware partition for sensitive operations would be equal for each user group. Neither user group needs to re-adjust their behaviour for the proposed security extension to function, but then again, **business users** and **high officials** are much more likely than **regular users** to store and handle sensitive data of value to an attacker. The added protection can thus be said to benefit business users and high officials more.

6.3.2 Advertisement

Permission separation within applications The solution presented in section 5.2.1 is probably the one that requires the least amount of change from an application developer's point of view. It also handles the permission problem with advertisements without major changes to any existing applications. The problem is the very complicated task of implementing it, and that it necessitates that the Android developer team does the implementation. Because of that it will have to be made available through a system update, and based on the fragmentation of devices (see figure 2.2) it will take a lot of time before all users can experience this possible solution.

Application separation This is a much more complicated solution (presented in section 5.2.2) the previous one, and it requires more work and knowhow by the developers to use compared with the current solution. Unlike the other two solutions, this requires only minimal implementation by the Android developer team. It requires one additional permission and some prototype features to display the ads. Two major problems with this solution is that it is very complicated to use for both developers and users (they need to install separate Advertisement Libraries applications). The other drawback is that the Advertisement Library is located in a separate application. By not having it as tightly bound as the current solution, it is easy to envision scenarios where users uninstall or block the Advertisement application, and by that stop ads from appearing in their applications.

AdDroid This solution (presented in section 5.2.3) is very developer friendly, as both the development itself and how it is implemented into an application require little action from the developer. Unlike the previous two solutions this is the only one that offers an answer to the malicious advertisement problem, and is done by letting the Android system supply the Advertisement Library. This way the Android development team has to implement it, and it will only become available through a system update. A problem with the control of the Advertisement Library in the hands of the Android developer team is that advertisement Publishing Network won't be able to create their own Advertisement Library or make innovative updates to it. As said, any changes to the Advertisement Library has to come with a system update, which will leave out a great deal of users, because of the issue with fragmentation (see figure 2.2 and section 2.1).

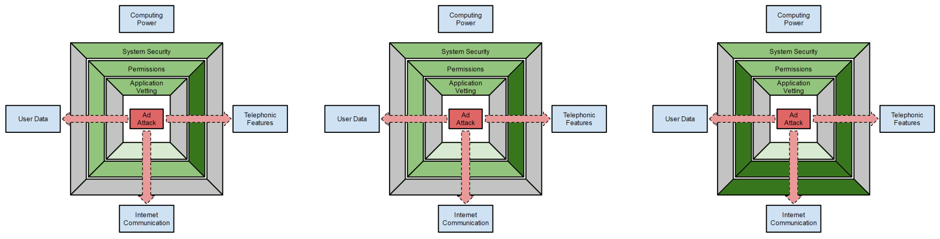
A problem related to all three solutions presented here is that they don't replace the current solution, they just offer an alternative. Even if one of the solutions were to be deployed, malicious parties could still use the current solution with all its vulnerabilities. To minimize this dilemma the new solution has to be very easy to use for developers, so they don't get tempted to go back and use the old solution.

Comparison

Easiest to use The AdDroid and Permission separation within applications solutions are best suited to handle the problem related to not replacing the current version. This is because they are both very easy for the developer to use, with AdDroid being the easiest.

Easiest to deploy Application separation is the most effortless to deploy, as it requires the least amount of help from the Android developer team.

Best security features By looking at the Threat Model (see figure 6.2) it is easy to see that AdDroid offers the best security features, it handles the permission problem and, as the only one, the malicious Advertisement Library problem.



(a) Permission separation. **(b)** Application separation. **(c)** AdDroid.

Figure 6.2: Advertisement Threat Model comparison. These models were previously shown in chapter 4.

Looking at the three different comparison categories, AdDroid comes out as the most advantageous solution. The great security features and ease of use are the main upsides, and they outweigh the negatives, especially the fact that deployment needs to be done by the Android developer team and the loss of possible innovative solutions.

User groups

Regular users Regular users have more applications installed than the other user groups, and often look for free versions of them, resulting in many applications with advertisement on their device. On the other hand, the assets advertisement attacks target are not threats severe to regular users. The result is that advertisement solutions are welcome, but not incredibly important.

Business users Business users have less advertisement applications then regular user, but the possible impact of an attack is more ominous. As a result, the solution will effect fewer applications, but for the applications it will effect, the impact will be greater.

High official users This user group generally does not have many applications with advertisement, so the solution will have minor to no effect.

6.3.3 WebView

Android system solution

The Android system solution is very effective at stopping attacks from web pages (see section 4.3.2), by limiting the functions available for the `addJavascriptInterface` method. This solution requires more coding by developers than previously, as they have to annotate all methods they want to use with `addJavascriptInterface`. As a result, developing applications using this method becomes more complicated, but only by a very little margin, which is a compromise worth taking as the security benefits are significant. However, a concern is that the solution is deployed through system updates, and because of that, not all users can take advantage of this new feature. In fact only 32.3 % (see figure 2.2) of Android devices used Android 4.2 or higher (the version where this solution was introduced), and as described in section 2.1 it will take a long time before all devices reach this Android system version. In addition, this solution only protects against one of the WebView threats (attacks from web pages), but threats like attacks from applications (see section 4.3.3) are still troublesome.

Other security measures

Roughly two thirds of Android devices still run OS versions older than 4.2 (see figure 2.2) and there are few solutions to prevent attacks abusing WebViews for this group of users. However, there are some implementation measures that developers can make use of (see section 5.3). The first is to disable all functions and features that are not needed in the application being developed. The others are different ways to detect third party web pages' attempt to exploit the WebView. These security measures are considered good practise and limit the effectiveness of attacks from web pages, but they are somewhat complicated to implement and are not new solutions per se. They only limit, not prevent or stop, attacks from web pages, but for devices not running Android 4.2 or higher, they are the best security solutions available.

6.3.4 User groups and WebView attacks

It is important to notice that all the solutions mentioned here are intended for developers and not users. In fact, there is very little users can do to protect themselves from threats related to WebViews. If users want to protect themselves from WebView threats, they should be careful when using applications with WebView. Not opening web pages that are not intended for the WebView is recommended practice, and use the default web browser instead.

Another problem for users and applications using WebViews, is that WebViews are difficult to recognise. A lot of the popular applications use WebViews (*Facebook* is an example), but there are no warnings or notices when an application does use them. For that reason, it is hard for users to avoid applications utilizing WebView.

The downside with this solution is the extra complications for developers. Users don't have do adjust their behaviour in any way, and so there are no clear downsides for them, not counting the fact that the solutions themselves are not that effective and don't cover all threats. Thus, all user groups are secured equally well from these solutions. Nevertheless, regular users are more likely to be affected by WebView attacks and probably benefit more than business and high official users.

6.4 Comparison of solutions

To compare the solutions and find a way to rank them, this list of important criteria where used:

- How much (we believe) it improves the current solution
- Possible complications related to deployment of solution
- How large the need for a new security solution to the governing problem is.
- Are there other solutions already trying to accomplish the same
- How it affects the different user groups

The new hardware solutions. The proposed solution utilizing already existing hardware partitions the computational power into two discrete worlds. Having a Secure World (for handling sensitive data) completely separate from the Normal World (handling all other operations) has great potential. It can let the end user choose which operations and what data he or she would like added protection for, which is done by picking the applications that guard the desired assets. However, this means it is up to the developers to include the security feature in their applications, i.e. ultimately laying the responsibility for improved security with the developers. The situation is comparable for solutions built on adding hardware, where it is up to the user to install the additional security. Neither is a favorable scenario, as security measures in place by default is preferable, though in these particular cases, it may not be decisive to the suggested solutions' usefulness. Business users are most susceptible to attacks against their user data assets, but at the same time likely to seek out solutions like those mentioned here. Thus, the hardware solutions suggested in this paper may be able to catch on in the markets where their potential to improve security is greatest.

Advertisement By implementing the recommended advertisement solution, the most alarming threats related to in-application advertisement is prevented. This can be observed by comparing the Threat Models in figure 4.3 and 5.8. Because the recommended solution has to be deployed through an Android system update, it will take some time before it is going effective in all devices. All user groups profit from this solution, with regular and business users profiting the most.

WebView The Android system solution and advice to developers for best practice help limit, but not prevent, attacks through the use of WebView. Threat Model 4.9 and 5.17 illustrate the effect of this new solution, as well as the security status of the current situation. Deployment time is long for the solution from the Android developer team, because it is published through an Android system update. On the other hand, the best practice advice can be used right away. WebViews are used in many popular applications, so security improvements are going to affect all user groups.

Solution ranking

The solutions are ranked as follows:

Advertisement By examining the arguments presented above, we have come to the conclusion that the solutions related to in-application advertisement are the best. The decisive argument is the huge security improvement seen in the Threat Model (see figure 4.3 and 5.8).

Hardware solutions The two solutions presented in the hardware solution section are second best. They both offer great new security features illustrated in the Threat Models 5.3 and 5.5, but the additional effort needed by the user (additional hardware) and developers (existing hardware) to make them work properly, hamper these solutions somewhat.

WebView Least impressive are the WebView solutions. This is due to the minor improvements it makes on the Threat Model (figure 4.9 and 5.17) and because they don't address all the threats presented in section 4.3.

Chapter 7

Conclusion

Throughout the report, various security measures for smartphones have been assessed in light of the most predominant attacks. The Android platform, which has been the primary area of focus, has several security mechanisms in place, both those embedded in the OS architecture and those applied outside of the smartphone device itself. Without the existing security solutions, the different attacks presented in chapter 4 would bother smartphone users to a much larger extent than they currently do.

Nonetheless, the number of registered attacks have increased greatly over the recent years. Assessing each feature of the total security package in place today has provided an understanding of where the shoe pinches. The selection of proposed new solutions in this thesis reflect what we have identified as the weakest points in smartphone security today.

The evaluation of the suggested new security mechanisms led to a ranking of them, as seen in chapter 6, and repeated here.

1. New way of handling advertisements in applications
2. Adding hardware dedicated to security tasks
3. Partitioning existing hardware for handling sensitive data securely
4. Improving the handling of WebViews

The ranking criteria weighted most were degree of impact, the amount of improvement made according to our Threat Model and how it affected the different user groups. Refer to section 6.4 for the full justification of the above ranking.

Overall, our take is that the best new advertisement solution *should* be implemented in a future version of the Android OS, as it is the solution that would make most difference to users. Both hardware solutions seem very convenient for those using their device in business settings, due to the increased protection of sensitive data, but are not regarded as strictly necessary for regular users. WebViews need an overhaul, but sadly, the solutions we found and have suggested don't do enough to mitigate the threats that exist. Thus, it did not earn a higher ranking nor our recommendation as a next step in improving smartphone security.

While the new solutions evaluated in this thesis provide many answers, it is lacking in the most problematical area. As mentioned previously, Trojans amount to more than 90 % of all software threats to smartphones, but none of the above solutions completely solve this issue.

Future works

As the thesis work has been void of any lab work or experiments, they are lacking in data to back up the assessments of the new security solution's effectiveness in mitigating attacks. Thus, experiments would be a logical next step to explore.

Additionally, the increasing trend that businesses provide their employees with smartphones, adds to the need for solutions and protocols regarding personal and business use. When a single device is used for both scenarios, the user will obtain the vulnerabilities of both regular and business users. Some of the measures mentioned in this thesis seek to answer parts of this challenge, but more attention to the issue is needed.

References

[1] eMarketer. Smartphone users worldwide will total 1.75 billion in 2014. Available from: http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536. Accessed May 19, 2014.

[2] Pew Research Internet Project Aaron Smith. Smartphone ownership 2013. Available from: http://www.pewinternet.org/2013/06/05/smartphone-ownership-2013/, June 2013. Accessed March 20, 2014.

[3] F-Secure Labs. Mobile threat report q3 2013. Technical report, 2013. Available from: http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf.

[4] Juniper Networks. *Juniper Networks Third Annual Threats Report*, 2013.

[5] SophosLabs Vanja Svajcer. Sophos mobile security threat report. Technical report, 2014. Available from: http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.pdf.

[6] F-Secure Labs. Threat report h2 2013. Technical report, 2013. Available from: http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf.

[7] The New York Times. I, robot: The man behind the google phone. Available from: http://www.nytimes.com/2007/11/04/technology/04google.html?_r=3&hp=&pagewanted=all&, November 2007. Accessed October 30, 2013.

[8] BloombergBusinessweek. Google buys android for its mobile arsenal. Available from: http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal, August 2005. Accessed October 30, 2013.

[9] Phonearena. T-mobile g1. Available from: http://www.phonearena.com/phones/T-Mobile-G1_id3097. Accessed October 30, 2013.

[10] Gartner. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. Available from: http://www.gartner.com/newsroom/id/2665715. Accessed June 10, 2014.

[11] Mixpanel. Android manufacturer destribution. Available from: https://mixpanel.com/trends/#report/android_manufacturer. Accessed April 25, 2014.

[12] Apple. ios distribution. Available from: https://developer.apple.com/support/appstore/. Accessed April 30, 2014.

[13] Android developers, dashboards. Available from: http://developer.android.com/about/dashboards/index.html. Accessed April 3, 2014.

[14] Steve Mansfield-Devine. Android architecture: attacking the weak points. *Network Security Magazine*, October 2012.

[15] Jason Rouse Neil Bergman, Mike Stanfield and Joel Scambray. *Hacking Exposed Mobile*. McGraw Hill, 2013.

[16] Ron Amadeo. Google's iron grip on android: Controlling open source by any means necessary. Available from: http://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/, October 2013. Accessed March 25, 2014.

[17] Android. Android security overview. Available from: https://source.android.com/devices/tech/security/index.html. Accessed May 6, 2014.

[18] Android Developers. Android activity. Available from: http://developer.android.com/reference/android/app/Activity.html. Accessed March 21, 2014.

[19] Android Developers. Android content providers. Available from: http://developer.android.com/guide/topics/providers/content-providers.html. Accessed March 21, 2014.

[20] Android Developers. Android services. Available from: http://developer.android.com/guide/components/services.html. Accessed March 21, 2014.

[21] Android Developers. Android intent. Available from: http://developer.android.com/reference/android/content/Intent.html. Accessed March 21, 2014.

[22] Android Developers. Android intent-filters. Available from: http://developer.android.com/guide/components/intents-filters.html. Accessed March 21, 2014.

[23] Android. Android using internal storage. Available from: http://developer.android.com/guide/topics/data/data-storage.html#filesInternal. Accessed March 21, 2014.

[24] Xuxian Jiang Michael Grace, Wu Zhou and Ahmad-Reza Sadeghi. *Unsafe Exposure Analysis of Mobile In-App Advertisements*. In *WISEC '12 Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[25] Google. Google mobile ads sdk getting started. Available from: https://developers.google.com/mobile-ads-sdk/docs/. Accessed April 1, 2014.

[26] Android. meta-data. Available from: http://developer.android.com/guide/topics/manifest/meta-data-element.html. Accessed April 3, 2014.

[27] Google. Google mobile ads sdk banners i. Available from: https://developers.google.com/mobile-ads-sdk/docs/admob/fundamentals. Accessed April 1, 2014.

[28] Android Developers. Manifest.permissions. Available from: http://developer.android.com/reference/android/Manifest.permission.html. Accessed May 6, 2014.

[29] Android Hiroshi Lockheimer. Android and security. Available from: http://googlemobile.blogspot.no/2012/02/android-and-security.html, February 2012. Accessed April 2, 2014.

[30] Nicholas J. Percoco and Sean Schulte. Adventures in bouncerland - failure of automated malware detection within mobile application markets. Trustwave SpiderLabs, Trustwave Holdings, Inc, February 2012. Available from: http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf.

[31] AV-Comparatives. Mobile security review. Technical report, August 2013. Available from: http://www.av-comparatives.org/wp-content/uploads/2013/08/avc_mob_201308_en.pdf.

[32] IETF. Internet security glossary. Technical report, May 2000. Available from: http://tools.ietf.org/html/rfc2828.

[33] Fernando Ruiz. 'fakeinstaller' leads the attack on android phones. Available from: http://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones. Accessed November 12, 2013.

[34] Lookout Mobile Security. *State of Mobile Security*, 2012. Available from: https://www.lookout.com/static/ee_images/lookout-state-of-mobile-security-2012.pdf.

[35] McAfee. What is a "drive-by" download? Available from: http://blogs.mcafee.com/consumer/drive-by-download. Accessed November 29, 2013.

[36] Lookout. Security alert: Hacked websites serve suspicious android apps (notcompatible). Available from: https://blog.lookout.com/blog/2012/05/02/security-alert-hacked-websites-serve-suspicious-android-apps-noncompatible/. Accessed November 29, 2013.

[37] Tim Strazzere. Ggtracker technical tear down. Technical report, Lookout Mobile Security, June 2011. Available from: https://blog.lookout.com/wp-content/uploads/2011/06/GGTracker-Teardown_Lookout-Mobile-Security.pdf.

[38] Carlos A. Castillo. Android malware past, present, and future. Technical report, McAfee Mobile Security Working Group, 2011. Available from: http://www.mcafee.com/us/resources/white-papers/wp-android-malware-past-present-future.pdf.

[39] Lookout Mobile Security. *Lookout Mobile Threat Report*, 2011. Available from: https://www.lookout.com/static/ee_images/lookout-mobile-threat-report-2011.pdf.

[40] Dr Giles Hogben, Dr Marnix Dekker. *Smartphone Security.* Technical report, ENISA, 2010. Available from: http://www.enisa.europa.eu/activities/identity-and-trust/risks-and-data-breaches/smartphones-information-security-risks-opportunities-and-recommendations-for-users.

[41] Symantec. Androidos.tapsnake: Watching your every move. Available from: http://www.symantec.com/connect/blogs/androidostapsnake-watching-your-every-move, August 2010. Accessed October 13, 2013.

[42] Symantec. Android.tapsnake. Available from: http://www.symantec.com/security_response/writeup.jsp?docid=2010-081214-2657-99, August 2010. Accessed October 13, 2013.

[43] Lookout. It's no game—tap snake is a spy app for the phone. Available from: https://blog.lookout.com/blog/2010/08/17/it%E2%80%99s-no-game%E2%80%94tap-snake-is-a-spy-app-for-the-phone/, August 2010. Accessed October 13, 2013.

[44] G Data SecurityLabs. Mobile malware report - half-year report july-december 2013. Technical report, 2014. Available from: https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/GData_MobileMWR_H2_2013_EN.pdf.

[45] DSLReports.com. What is a botnet trojan? Available from: http://www.dslreports.com/faq/14158, April 2009. Accessed April 9, 2014.

[46] Lookout. Security alert: Droiddream malware found in official android market. Available from: https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/, March 2011. Accessed October 13, 2013.

[47] Lookout. Technical analysis droiddream malware. Available from: https://blog.lookout.com/droiddream/, March 2011. Accessed October 13, 2013.

[48] G Data SecurityLabs. Android malware goes "to the moon!". Available from: https://blog.gdatasoftware.com/blog/article/android-malware-goes-to-the-moon.html, February 2014. Accessed April 8, 2014.

[49] AVG. Android/dgen plankton a. Available from: http://www.avgthreatlabs.com/virus-and-malware-information/info/android-dgen-plankton-a/. Accessed April 8, 2014.

[50] Android Developers. Dexclassloader. Available from: http://developer.android.com/reference/dalvik/system/DexClassLoader.html. Accessed April 8, 2014.

[51] Lookout. The bearer of badnews. Available from: https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/, April 2013. Accessed October 14, 2013.

[52] Android Developers. Webview. Available from: http://developer.android.com/reference/android/webkit/WebView.html. Accessed December 1, 2013.

[53] Wenliang Du Yifei Wang Tongbo Luo, Hao Hao and Heng Yin. *Attacks on WebView in the Android System.* In *ACSAC '11 Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[54] Android Developers. Webviewclient. Available from: http://developer.android.com/reference/android/webkit/WebViewClient.html. Accessed April 26, 2014.

[55] CUPP Computing. The micro sd format hardware security form cupp computing. Technical report, May 2014. Version 0.9.

[56] Snort. Snort users manual. Technical report, April 2014.

[57] Snort. What is a snort-rule. Available from: https://github.com/vrtadmin/snort-faq/blob/master/Rules/What-is-a-Snort-rule.md. Accessed May 13, 2014.

[58] Snort. Download snort rules. Available from: http://www.snort.org/snort-rules#community. Accessed May 13, 2014.

[59] Timothy Prickett Morgan v/The Register. Arm holdings eager for pc and server expansion. Available from: http://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers/, February 2011. Accessed May 12, 2014.

[60] Apple. ios security. Technical report, February 2014. Available from: http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf.

[61] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. Technical report, EECS Department, University of California, Berkeley, May 2013. Available from: http://www.cs.berkeley.edu/~pearce/papers/addroid_asiaccs_2012.pdf.

[62] Yuliy Pisetsky Anhei Shu Dan S. Wallach Michael Dietz, Shashi Shekhar. Quire: Lightweight provenance for smart phone operating systems. Technical report, February 2011. Available from: https://www.usenix.org/legacy/event/sec11/tech/full_papers/Dietz7-26-11.pdf.

[63] Android Developers. Build.version_codes. Available from: http://developer.android.com/reference/android/os/Build.VERSION_CODES.html#JELLY_BEAN. Accessed May 22, 2014.

[64] Android Developers. Webview. Available from: http://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object,java.lang.String). Accessed May 22, 2014.

[65] MWR Labs. Adventures with android webviews. Available from: https://labs.mwrinfosecurity.com/blog/2012/04/23/adventures-with-android-webviews/. Accessed May 31, 2014.

[66] Android Developers. Websettings. Available from: http://developer.android.com/reference/android/webkit/WebSettings.html. Accessed May 31, 2014.

[67] Google Admob. Google mobile ads sdk. Available from: https://developers.google.com/mobile-ads-sdk/docs/admob/fundamentals#play. Accessed Mars 17, 2014.

Appendix A

AdMob Example Activity

Below is an example of an activity that has to be included in an application to create and display a banner advertisement.[67]

```java
package com.google.example.gms.ads.banner;

import com.google.android.gms.ads.AdRequest;
import com.google.android.gms.ads.AdSize;
import com.google.android.gms.ads.AdView;

import android.app.Activity;
import android.os.Bundle;
import android.widget.LinearLayout;

/**
 * A simple {@link Activity} that embeds an AdView.
 */
public class BannerSample extends Activity {
  /** The view to show the ad. */
  private AdView adView;

  /* Your ad unit id. Replace with your actual ad unit id. */
  private static final String AD_UNIT_ID = "INSERT_YOUR_AD_UNIT_ID_HERE";

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // Create an ad.
    adView = new AdView(this);
    adView.setAdSize(AdSize.BANNER);
    adView.setAdUnitId(AD_UNIT_ID);

    // Add the AdView to the view hierarchy. The view will have no size
    // until the ad is loaded.
    LinearLayout layout = (LinearLayout) findViewById(R.id.linearLayout);
    layout.addView(adView);

    // Create an ad request. Check logcat output for the hashed device ID to
    // get test ads on a physical device.
    AdRequest adRequest = new AdRequest.Builder()
        .addTestDevice(AdRequest.DEVICE_ID_EMULATOR)
        .addTestDevice("INSERT_YOUR_HASHED_DEVICE_ID_HERE")
        .build();

    // Start loading the ad in the background.
    adView.loadAd(adRequest);
  }

  @Override
  public void onResume() {
    super.onResume();
    if (adView != null) {
      adView.resume();
    }
  }

  @Override
  public void onPause() {
    if (adView != null) {
      adView.pause();
    }
    super.onPause();
  }

  /** Called before the activity is destroyed. */
  @Override
  public void onDestroy() {
    // Destroy the AdView.
    if (adView != null) {
      adView.destroy();
    }
    super.onDestroy();
  }
}
```