NTNU – Trondheim
Norwegian University of
Science and Technology

An Application Security Assessment of Popular Free Android Applications

Christian Håland

Master of Science in Computer Science
Submission date: October 2013
Supervisor: Guttorm Sindre, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

An Application Security Assessment of Popular Free Android Applications

Author: Christian Håland

Supervisor: Guttorm Sindre

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering
Norwegian University of Science and Technology

October 2, 2013

Sammendrag

Antallet Android applikasjoner som finnes på Google Play har nå overskredet 1 million og markedet har over 1,5 milliarder nedlastinger i måneden. Den lave inngangskostnaden i app markedet har ført til at mange uerfarne utviklere publiserer applikasjoner som ikke er sikre nok. Vi analyserer og tester 20 Android applikasjoner ved hjelp av en egenutviklet testmetodologi basert på en generisk testmetodologi laget av av OWASPs mobilprosjekt. Vi ser etter vanlige sårbarheter og feil ved å gjøre koderevisjon på dekompilerte applikasjoner og utfører statisk og dynamisk analyse av applikasjonene.

Vi evaluerer også eksisterende sårbarhetsklassifiseringer for å finne ut hvilke av disse som egner seg for å klassifisere sårbarheter i mobile applikasjoner. Formålet med klassifiseringene er å utdanne utviklere og profesjonelle innen sikkerhet om sårbarhetene vi finner og kunne diskutere tiltak for å unngå disse sårbarhetskategoriene i fremtidige applikasjoner.

Til slutt diskuterer vi hvilke konsekvenser sårbarhetene potensielt kan ha for sluttbrukere og i hvilke angrepsscenarioer en angriper kan benytte seg av disse.

Abstract

The number of applications for the Android platform found on Google Play is now over 1 million and there are over 1.5 billion downloads each month . With a low cost of entry the Android platform attracts developers many of which do not have the necessary competence or experience to develop secure applications. We assess 20 Android applications using a custom testing methodology based on the OWASP Mobile Project and look for common vulnerabilities. We decompile applications using Dare [14] and review the code manually as well as using static and dynamic analysis to look for vulnerabilities.

We also evaluate existing vulnerability classifications and argue which ones are most fitting for to apply to Android vulnerabilities for educational and research purposes. We then categorize our findings according to OWASP Mobile Top 10 and present mitigation strategies for each category as a whole. Finally, we argue the implications of the vulnerabilities to end-users.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The number of people owning mobile devices has exploded in recent years causing a hunger for quality services and applications for these devices. Google brags that over a million new Android phones are activated every day and that they have now reached a total of 900 million activated Android devices [5]. Gartner reports that Android has a market share of 74% of the total smartphone market [27] and is gaining market shares in the tablet market as well. The growing number of Android devices has led to a massive market for innovative and well-designed applications. Developers are rushing to fulfill the market void and with a low entrance cost, (25$ to publish applications on Google Play) the quality of applications suffer. Less experienced developers that are unfamiliar with the underlying platform design often make bad design decisions or write vulnerable code. To manage and reduce the number of vulnerabilities it is therefore important to identify and classify the most common vulnerabilities and reason about how we can improve applications and avoid similar mistakes in the future.

Contrary to Apple, Google does not manually verify applications before they are published to market. Instead applications are screened for malware using automated tools [34], but these tools will not reveal all vulnerabilities that can be exploited by an attacker, or by a malicious application that have made it onto the device. With mobile devices containing increasing amounts of personal and sensitive information there is a need for better understanding the risks a user is exposed to by installing flawed applications. Flawed applications also involve risks for the businesses or independent developers deploying these applications. If personal data stored by a business are stolen due to a vulnerable application this usually leads to unwanted media attention and distrust from customers. But it can also lead to direct economic loss caused by downtime of critical services or money stolen from online bank accounts. This is bad for businesses and may also have a negative impact on users if for instance their passwords are lost and an attacker can gain access to other services using the same credentials.

With a large share of the market, Android attracts unwanted attention from malware creators and other malicious users. The number of malware applications detected on the Android platform is reported to be increasing [33], but we have yet to see the explosion of malware predicted. We are however seeing increasingly complex and clever ways in which malware code disguises itself [54].

Smartphones are small but highly sophisticated computers and can match stationary computers from a few years back in terms of processing power and memory. With the growing popularity of the Bring-Your-Own-Device (BYOD) paradigm, many mobile devices are connected to corporate networks [40]. It is therefore in every company's interest that employees' mobile devices are not under the control of an adversary who may have malicious intents. Modern mobile devices have cameras and microphones and are almost constantly connected to the Internet, making them perfect tools for stealing information or eavesdropping on confidential conversations. To prevent malicious users from taking control of devices through vulnerable mobile applications we must examine them for vulnerabilities to learn how to build them more securely. Still, with BYOD becoming more popular we cannot solely rely on the security measures implemented on mobile devices to keep them from leaking private or corporate information. We should also keep in mind the defense-in-depth principle and not only install technological measures at device and network level, but also educate users about security to improve awareness of threats.

1.2 Objective

There has been a lot of attention given to the security of the Android system's lower levels and internals [11, 23, 25, 32, 39]. Less attention has been given to application security, but there is some work in this area [18, 22]. Enck et al. [18] did a horizontal study of 1,100 applications on which they performed static analysis to discover vulnerabilities. Fahl et al. [22] did a study of 13,500 applications with regards to SSL use and user awareness of data transport encryption.

In this work we choose a smaller set of applications and do a more thorough investigation using both static and dynamic analysis with less focus on automated vulnerability scanning. We analyze applications in order to find patterns of vulnerabilities so that we can automate the search for these in the future. We use existing automated tools where possible and complement these with manual examination and testing.

The objective of this work is to answer the following questions:

- RQ1: How can we effectively identify vulnerable application code in real-world Android applications?
- RQ2: What are the most common vulnerabilities or flaws in Android applications?
- RQ3: Which vulnerability classifications exist, and are they suited to classify Android application vulnerabilities?

- RQ4: How can vulnerabilities in Android application code be exploited by malware or malicious users?
- RQ5: What are the implications of vulnerable application code to end-users?

By answering these questions we hope to contribute to fewer exploitable applications in the future, making it harder for malicious entities to take advantage of unknowing end-users. By classifying vulnerabilities we hope it will be easier to reason about how each class of vulnerabilities can be prevented on a more general basis rather than having to patch single vulnerabilities separately as they are discovered.

1.3 Limitations

As computer technology is maturing, developer tools, programming languages and applications have been put to the test and been placed under scrutiny by many researchers and developers. This raises the bar for discovering original ways that attackers are able to misuse devices, systems and applications. Experience, skills and endurance is required to find exploitable flaws, all of which can only be gained by practice. Limited experience with Android and security assessments in the field is therefore an obstacle that has to be overcome by studying the work of others and imitating techniques they have found to be effective. At the same time it is important to try to look at applications from a fresh perspective in order to discover original ways in which to exploit them.

1.4 Outline

Chapter 2 gives background information on the Android system and Android's security model. We also present malware and root exploits that have been observed in the wild on the Android platform. The chapter is concluded by looking at the application security work that has been done on Android in the past.

Chapter 3 presents potential research methodologies and which ones were chosen. The process of the security assessment that we perform is explained and the techniques we use are presented. The chapter concludes by discussing the ethics of security assessments.

Results are presented and discussed in chapter 4. The conclusion of the work is drawn in chapter 5 and future work suggested.

Chapter 2

State of the art

This chapter provides background on the Android system and its design with regards to security. It explains what constraints are imposed on an application, how an application's data is protected and how applications interact. The information in this chapter provides a basis of knowledge that is useful when performing security assessments of Android applications.

This chapter also details some vulnerabilities and attacks on the Android system itself which have been discovered by security researchers and Android enthusiasts. These vulnerabilities are helpful to get an understanding of the flaws that have made Android exploitable in the past. To discover new vulnerabilities it is crucial to have an understanding of the system and which parts of an application are more likely to be vulnerable.

To be able to thoroughly test Android applications one needs a solid base of knowledge on the Android security model and how the Android system is built. Experience with Linux is of great help in exploring and understanding the system because of the way Android employs the Linux kernel at the base of its architecture.

The reason we look at malware in this section is that malware is being used to take control over systems and steal personal information, often by exploiting flaws in installed applications or the Android system itself. Therefore it is useful to know how these malware applications operate, what information they try to steal, and how they have been used in the past. Having an understanding of existing malware will help us look for vulnerabilities in applications that can potentially be exploited.

Malware detection systems are interesting because they they aim to find maliciously behaving applications. We wish to find misbehaving applications as well, even though the applications are not malware per se, they can show some of the same characteristics such as sending a phone's unique identifier [1] over the network. By studying how malware detection systems find these flaws we can replicate their methods.

[1] The International Mobile Equipment Identity (IMEI) is used to block stolen phones from the network in the US. IMEIs therefore have value on the black market because they can be used to re-enable stolen phones.

2.1 Android security model

Android's security model is designed to be multi-layered and flexible. At the bottom layer Android leverages the Linux kernel's security mechanisms to create an application sandbox for each application. At the Android system level, permissions are used as a mechanism to declare and limit an application's access to the system API. Enck et al. give an overview of the pitfalls and complexities of Android security in [20].

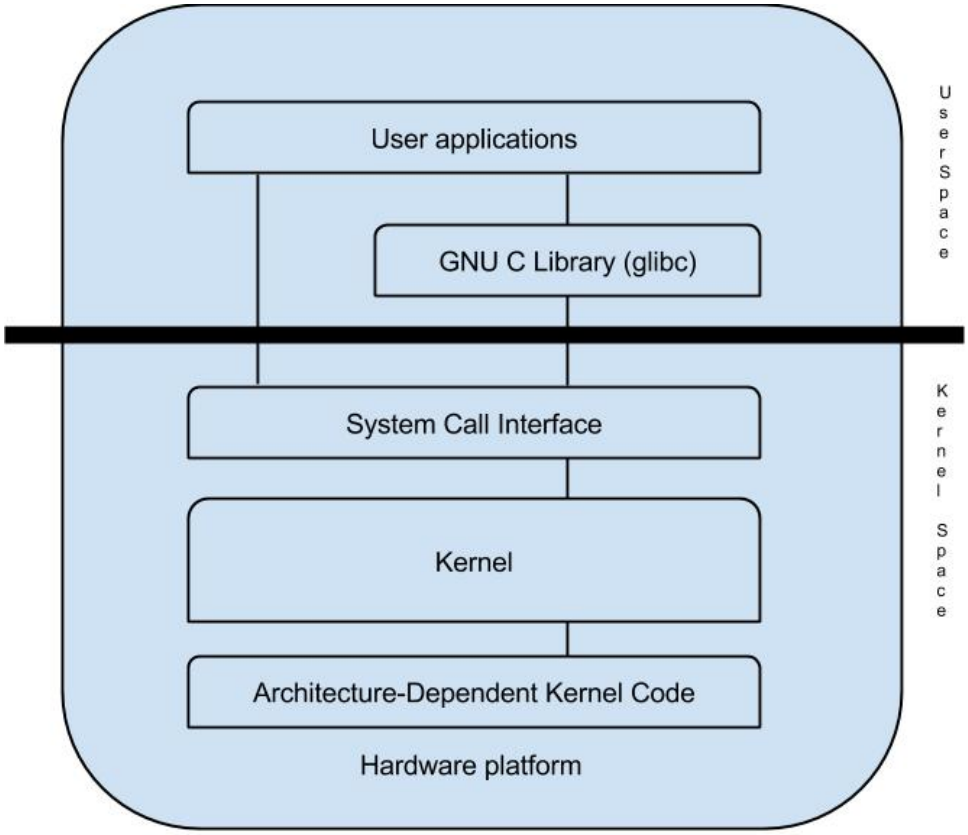

Figure 2.1: The figure gives a high-level view of the linux architecture. Applications are run in user space and are presented with an API to the Android System.

2.1.1 Sandboxing

On the Android system, each application is run in its own environment called its sandbox. The sandbox abstraction entails that the app runs in its own process, is given its own part of the file system and that it has limited access to system hardware. The reasoning behind using sandboxing is to limit the privileges of potentially malicious applications that have somehow made it onto the device in accordance with the principle of least privilege [7]. On traditional operating systems like Windows and Linux, programs are

limited to the privileges of the logged-in user. It is advisable to create user accounts with less privileges than the administrator and have the system prompt for privilege escalation when it is needed to perform some task. This way malicious applications will not have full administrator access to the system. Android, being an operating system built on the Linux kernel have designed its system security to take further advantage of the principle of least privilege by creating more fine-grained environments. Environments are no longer separated by which user is logged in, instead each application is assigned its own user id and started with its own environment (or sandbox) isolated from other applications [44].

2.1.2 Permissions

The permission model is one of Android's basic security mechanisms. Applications declare the permissions that they require in their manifest file, the main XML configuration file of the application. When a user installs an application the permissions that the application requests are shown to the user along with a description of each permission. The user must then choose to accept or reject the installation of the application based on the permissions requested. The user cannot reject individual permissions, she has to reject the whole application if she objects to any one of them.

Permissions are used as a mandatory access control mechanism for controlling interprocess communication. Applications on Android can announce that they are capable of certain features and subscribe to receive requests for these features from other applications. A simple example is an application that wants to use the smartphone's camera. That application would have to specify the camera permission in the manifest and when the application wants to take a picture, an intent object is broadcast to all applications that have announced their capability of providing camera functionality. The user is then presented with the choice of which camera application to use to take the picture. Android predefines permissions for the most common resources like camera, WIFI, GPS location and many more, but the system is also flexible and lets developers define their own permissions for functionality that their application offers to other applications. This way the permission system is an extendable and flexible model. There are however critical questions that have been asked about the permission model. Are the permissions granular enough? Is it wise to let the users decide which applications they want to trust?

At the time of writing the Android documentation lists 141 different permissions. Each permission is given one of four protection levels. The "normal" protection level is given to permissions that have little or no potential negative impact on the user. The "dangerous" permission level indicates permissions that can have severe negative impact on the user and should be accepted with caution. Then there is "signature" level which is a permission level that is only granted to a requesting application if it was signed with the same certicate as the application that declared the permission. The last level is the "systemOrSignature" level which is similar to the "signature" level but is only for use in special situations and should not be used by developers.

Porter Felt et. al tested the Android system's permission checking associated with API-calls in [23] and uncovered faulty documentation where the permissions listed as required for a function call were either wrong or did not exist. Porter Felt and her colleagues also pointed out that the lacking documentation may be due to a missing centralized access control policy. A common problem is that applications request more permissions than they need. This may happen due to lacking documentation or because of lacking developer competence. Applications that overrequest permissions undermine the permission model by making it harder for users to differentiate applications based on the permissions they request. Porter Felt et al. built an automatic tool called Stowaway that analyzes an application's API calls and its use of other system components to figure out the minimal set of permissions required for the app to function. Stowaway could be helpful to developers to limit the requested permissions to only the ones needed.

2.1.3 Inter-application communication

Since applications are sandboxed and the Android system aims to achieve reuse of functionality the system needs some controlled way to provide communication between applications. Android solves this by providing what is called "Intents". Intents are message objects that can be passed to the system to communicate with other applications. An application requiring some special functionality that can be found in some other app can create an intent telling Android to look for installed applications with the needed functionality. For example an application wanting to take a picture can create an intent asking the system for a camera application. If the system finds one it will launch it, let the user take a picture and then possibly return the image data to the calling application. An application wanting to expose functionality to other applications must explicitly declare itself capable of handling certain intents by putting an entry in the manifest file. This is called exporting a component. There are four types of application components: activities, services, broadcast receivers and content providers. The first three of these can pass and receive intents.

Intents can be either explicit or implicit. Explicit intents name the application component that they are destined for and the action that is intended. For instance, in the example given above an application wanting camera functionality could explicitly name the built-in Android camera application in the intent. On the other hand, using an implicit intent the application programmer could specify only the action, type or category of the functionality needed. Android then performs a process called intent resolution to find the applications that export the described functionality and presents the user with a choice if more than one application is found.

Chin et al. performed an analysis of inter-application communication and provided a threat model in [11]. They also built a static analysis tool called ComDroid to analyze compiled application files for possible inter-application communication vulnerabilities. Many of the applications they tested contained these kinds of vulnerabilities.

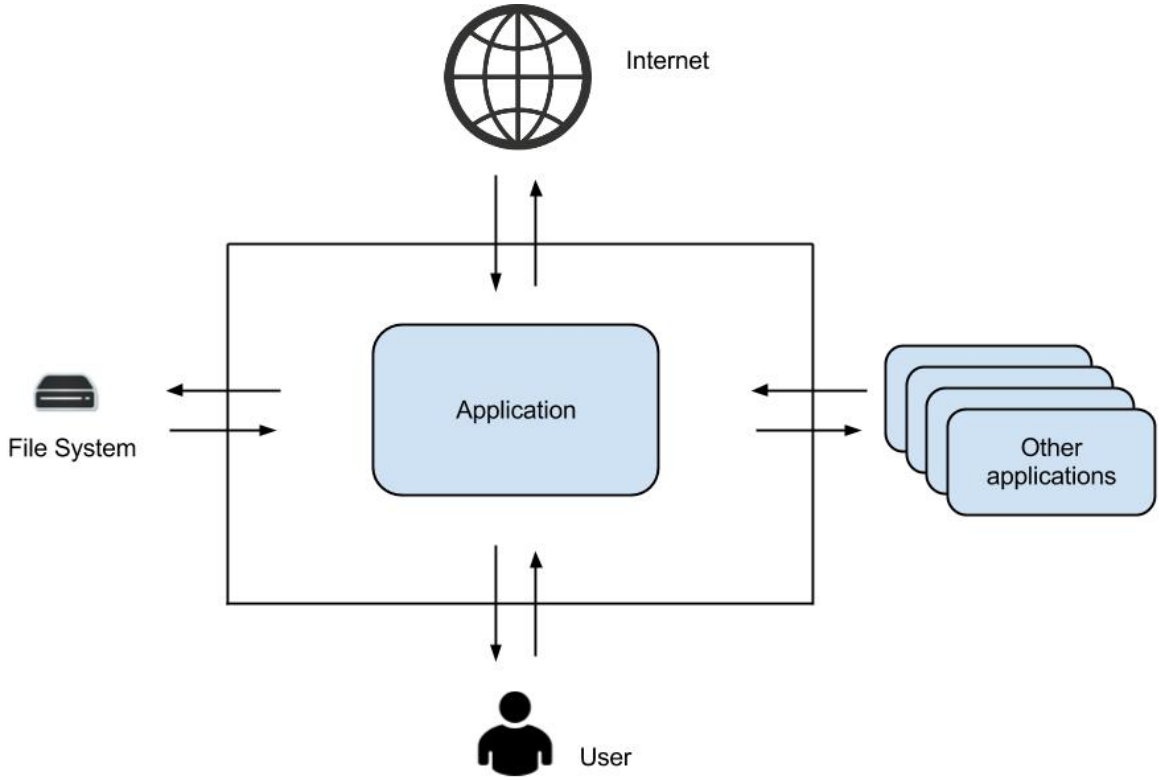

Figure 2.2: The figure shows an Android application's main input boundaries. All of these are typical attack surfaces. Applications may have several other attack surfaces including interfaces such as Bluetooth or Near-Field-Communications (NFC).

Eavesdropping, intent hijacking and injection attacks

Inter-application communication poses the classic network security challenges of confidentiality and integrity between two communicators. Faulty applications can allow malicious applications to eavesdrop on inter-application communication, hijack intents to provide malicious results, or perform denial-of-service attacks.

Intent hijacking opens up the possibility of phishing attacks on Android phones. If a vulnerable application relies on another application for, say e-wallet functionality, a malicious application may be able to intercept the intent to start the e-wallet application. This is possible because of the way the Android system treats implicit intents. When several applications subscribe to the same implicit intent the user will be presented with the choice of which application to use. The malicious application wanting to intercept the intent can disguise itself by using the same icons etc. as the authentic e-wallet application and thereby trick the user into launching the malicious e-wallet. The app can then steal the user's credit card information while the user unknowingly thinks he just payed for a service or product. The average user is likely to fall prey to this sort of phishing attack because they are not aware of this kind of threat in this "safe" environment.

Chin et al. uncovers several other similar vulnerabilities in applications that use Android's

inter-application communication system wrongly [11]. The vulnerabilities uncovered include intent spoofing and broadcast injection.

2.1.4 Application Signing

Android applications must be signed by the developer, otherwise the Android system will by default not allow them to be installed. This mechanism is in place to establish trust. Google allows applications to be self-signed. Applications that are published through Google Play will be related to the developer of the application through the developer's Google account. This connection raises the credibility of the application somewhat and gives Google better control of the origin of applications on their marketplace. Developers benefit by being able to sign multiple applications with the same certificate to establish themselves as a trustworthy source.

Google uses asymmetric cryptographic keys to sign applications. The application is signed with a private key and the developer's public key is used to verify the identity of the developer. The private key must be kept safe to keep an attacker from publishing malicious applications that look like they are coming from a recognized company or developer. Subsequent updates to applications must be signed with the same key as the original application for the system to recognize that the update is coming from the original source.

Certificate signing can also be used to provide trust bonds between applications. Applications that are signed with the same key will gain access to each others' sandbox and therefore be able to communicate freely and to share data. This gives developers the opportunity to modularize their applications. For instance, at the time of writing, Facebook has three Android applications: Facebook messenger, Facebook and Facebook pages manager. These three apps can share data, saving space and allowing users to install only the functionality they need.

Jeff Forristal at Bluebox Security presented an Android vulnerability at the 2013 Black Hat conference which lets an attacker modify an existing application's code without breaking the signature of the application. This vulnerability would let an attacker modify legitimate applications to do malicious stuff and then publish the malware as an application update without the user being able to tell that the application code's integrity was compromised [26].

2.1.5 System and Kernel Level Security

The Linux kernel is at the base of the Android system. Linux is widely deployed and has been put through all kinds of scrutiny by researchers, attackers and developers. The Linux kernel provides the Android system with discrete access control (DAC) in the form of user- and group based permissions. Each application in Android is run with a unique

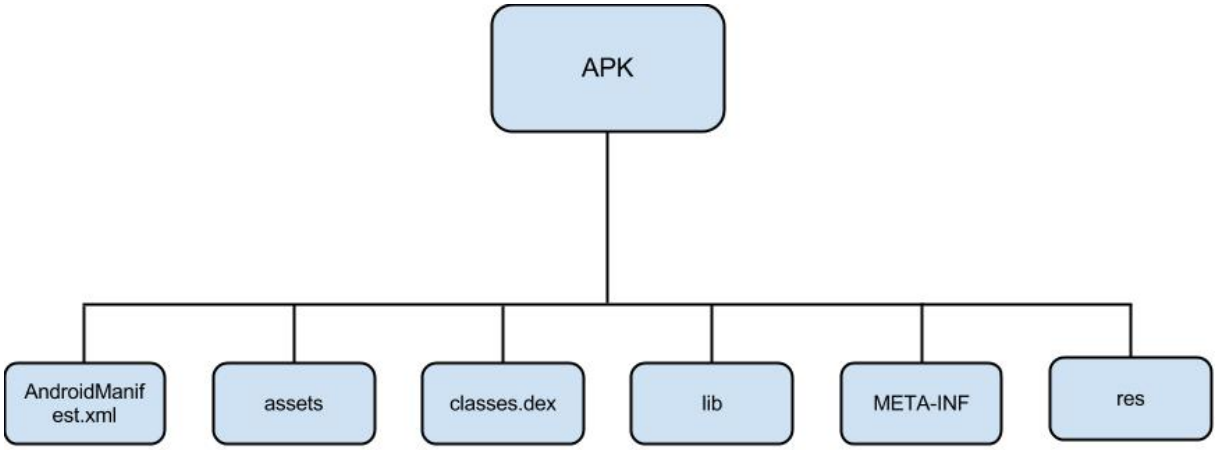

Figure 2.3: Android applications are distributed as APK files. APK files are just ZIP files containing all the application's resources, libraries, and main application code in Dalvik executable (.dex) format.

user id to protect applications from each other. These kernel-level mechanisms comprise what Android calls the Application Sandbox. Each application lives in its own sandbox and is assigned its own private disk space and cannot access other application's resources or communicate with them by default. Native applications and operating system libraries are also run within the application sandbox. This setup helps prevent standard memory corruption errors from giving an attacker complete control of the system. Instead if an attacker exploits a memory corruption error to execute code, he would still only have the permissions of the original application and could only access resources within the application's sandbox.

2.2 The Android Boot Process

To be able to look for weaknesses in the Android security model we must acquaint ourselves with the boot process of the Android operating system. The process starts at the lowest levels and initializes the device all the way up to the abstraction layer where apps are run. The security environment in which applications are run is determined by this process and is therefore useful to have a certain grip on. The boot process is portrayed in figure 2.5

When an Android device is powered up a specific boot ROM code is executed by the CPU to initialize the device hardware and locate the boot media. This step is analogous to the BIOS used in regular computers. Once the hardware is initialized the next step is to find the boot media. Android devices store this on the NAND flash. Once the boot media is found the initial boot loader is loaded into internal RAM and then executed. The boot loader can be compared to the ones used for regular computers such as the GRUB used on Linux. The boot loader has two stages: The initial program loader (IPL) and the

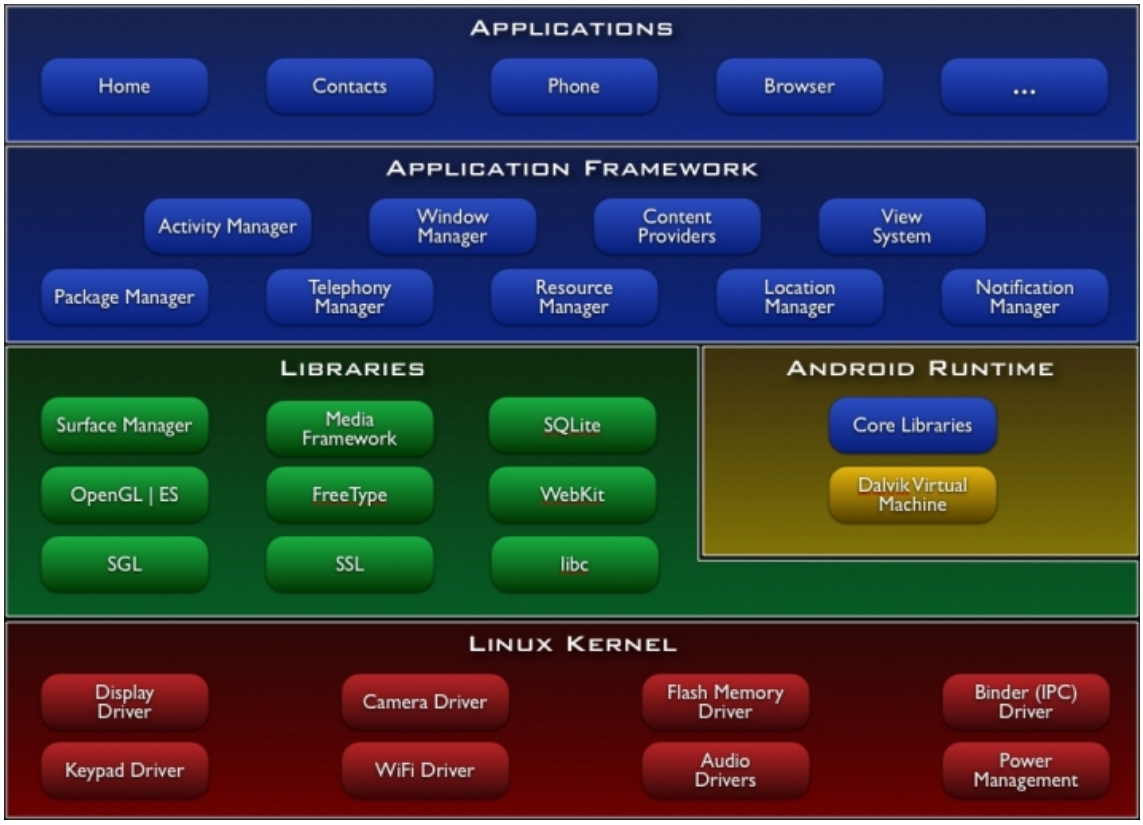

Figure 2.4: The figure shows the layered structure of the Android system and the software components it includes at each layer.

second program loader (SPL). The IPL sets up external RAM and loads the the SPL. The SPL is the final step of the boot loader and it is responsible for providing the different boot modes, such as Recovery and Fastboot. The regular boot mode loads the Android operating system, initializes file systems needed and then locates and loads the Linux kernel. Boot parameters are then loaded and execution control is passed to the Linux Kernel.

The first thing that the system does after the Linux kernel is started is to execute the init.rc file found at the root of the Android file system. This script mounts the SD-card if one exists and initializes the system directories that the system needs. This process is typical for machines running Linux and it initializes the process hierarchy, sets all the environment variables and basically just readies the system.

The next step is to launch a binary called app_process which is then renamed to Zygote. Zygote is literally defined as: “It is the initial cell formed when a new organism is produced“. And this reflects the role of Zygote. It is a “warmed-up“ process which has all the core libraries linked in. When a new application is started, the Zygote process is forked and the new process uses the libraries already loaded by the original Zygote process. This way Android does not have to copy the memory of the libraries to the new process and achieves some speedup [8].

After Zygote is started it forks a new process which becomes the system server. The tasks of the system server is to start up Android's background services such as the "Power Manager", "Package Manager" and the "Hardware Service". These background services each run in their own thread in the system server process. They provide service to the user applications at the top layer [64].

The boot process is often different for each manufacturer because they may provide custom bootloaders or have device specific init scripts.

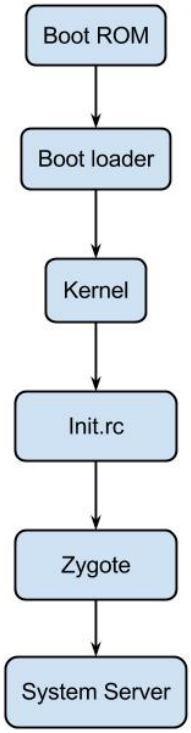

Figure 2.5: The figure shows the main application boundaries. All of these are typical attack surfaces. Applications may have several other attack surfaces including interfaces such as Bluetooth or Near-Field-Communications (NFC).

2.3 Vulnerabilities

A vulnerability is a weakness in a product that could allow an attacker to compromise the integrity, availability, or confidentiality of that product [10].

This definition given by the Microsoft Security Response Center is elaborated further by excluding by-design weaknesses such as using plain text FTP to communicate user credentials. We disagree with this point of view and include by-design weaknesses that can be relatively easily fixed in the set of vulnerabilities. For instance, weaknesses in a

product by design such as using weak cryptography to store or transfer sensitive data that could be of use to an attacker would by us be considered a vulnerability.

A vulnerability has the potential to compromise one or more of the three security properties of a software product: Confidentiality, availability or integrity. Note that we do not require that an exploit exists for a weakness to be classified as a vulnerability.

The reason we are trying to classify vulnerabilities is so that developers, security researchers and students can learn about and refer to them at a later point. By grouping vulnerabilities into classes we can more easily reason about mitigation strategies or automatic ways of discovering vulnerabilities for the class as a whole instead of for each single vulnerability. This requires the categories of the classification to be at a relatively low level to have precision, but at a high enough level for the vulnerabilities to be comparable across platforms and technologies.

2.3.1 Classifications of Vulnerabilities

There are several popular vulnerability classifications, but none that we are aware of target mobile platforms specifically. We list some of he classifications here and discuss which ones are applicable to classify Android application vulnerabilities. None of the classifications below are scientifically rigorous because they are often ambiguous and overlapping. See [37] for a more complete treatment of classifications for vulnerabilities and attacks and a more thorough explanation why these taxonomies are scientifically incomplete.

Classification by Software Development LifeCycle Phase

This classification tries to categorize vulnerabilities according to when they where introduced into the software lifecycle. At a basic level the software lifecycle is defined as three high level phases: design, implementation, and operation and maintenance [37]. One difficulty with this is that a system can be viewed at several different levels of abstraction. This means that problems at one level may belong to the implementation phase, but at another level to the design phase. This taxonomy may be useful for educating developers about how to avoid vulnerabilities in the different phases in the development life cycle. However, we aim to reason about how to mitigate classes of vulnerabilities and the variation of vulnerabilities that can be introduced in each development phase is large and makes it hard to reason about.

Classification by Genesis

This classification divides vulnerabilities into two groups by the way that they were introduced: accidentally or intentionally. This classification gives an interesting view, but it is a very rough categorization and it will sometimes be hard to objectively decide whether a

vulnerability was introduced by accident or by a malicious developer. This classification will not be very helpful to our purpose because we assume that we primarily deal with accidental vulnerabilities and aim to reason about mitigation despite underlying intention.

Classification by Location in Object Models

This classification tries to categorize vulnerabilities by assigning them to a part of an model object. For instance using the ISO Open Systems Interconnect (OSI) model we could say whether a vulnerability belongs to the network level or the transport layer of the model. This is an interesting idea, but would require us to develop an object model for Android applications which is likely to make the resulting classification harder to compare to vulnerabilities from other platforms or technologies.

Classification by Affected Technology

This classification tries to classify vulnerabilities by the underlying technology. For instance format string vulnerabilities in C would be one such category. Meta-character vulnerabilities is another example. SQL-injection vulnerabilities would be a subclass of meta-character vulnerabilities. This classification is very precise, but we find it to be a bit too low level for our purpose, because there are many vulnerabilities that are technology agnostic and therefore does not fit into this classication.

Classification by Error or Mistake

This classification categorizes by the error or mistake leading to the vulnerability. An example is the "double-free" memory management mistake. Another example is allowing the use of weak passwords. This classification is a bit loosely defined and allows us to choose the level of abstraction that best suit our purpose. OWASP Mobile Top 10 is basically based on this classification and has chosen a reasonable abstraction level.

STRIDE

STRIDE [38] is a classification of threats based on the security properties of the system that are exploited by an attacker. STRIDE is an acronym for the following high-level categories:

Spoofing identity: Identity spoofing concerns an attacker illegally logging in using another user's authentication information.

Tampering With Data: This category concerns malicious modification of data. E.g. an attacker may tamper with any data client side before sending it back to the server. Therefore the server must always validate data coming from the client.

Repudiation Users may dispute transactions if there is insufficient auditing or record-keeping of their activity. An example of repudiation is an online bank system user saying that he did not transfer money to another account. The system needs to have logging or auditing capabilities (non-repudiation controls) to prove that the user in fact did (or did not) do this. Otherwise the bank would probably have to write the money off as a loss and reinstate the user's money, even if an attacker was the one who somehow transferred the money.

Information Disclosure Disclosure of a user's private information contrary to the system's policy.

Denial of Service Denial of service attacks deny service to valid users. E.g. By making a Web server unavailable to users an attacker has performed a denial of service attack.

Elevation of Privilege This type of threat concerns an attacker gaining privileged access to a system and thereby could possibly modify or destroy the system. An example is a Linux user on a university computer system gaining root privileges and doing malicious things to the system.

OWASP Mobile Top 10 Risks

The Open Web Application Security Project has started a mobile section of the project and has created a list of the Top 10 risk categories on mobile devices in 2011. The categories are relatively high level and is a mixture of potential attack techniques (e.g. client side injection) and mistakes (e.g. Broken cryptography and improper session handling). The list is now slightly dated and obviously incomplete since it only includes the top 10, but we assume that most of the categories are still relevant [45].

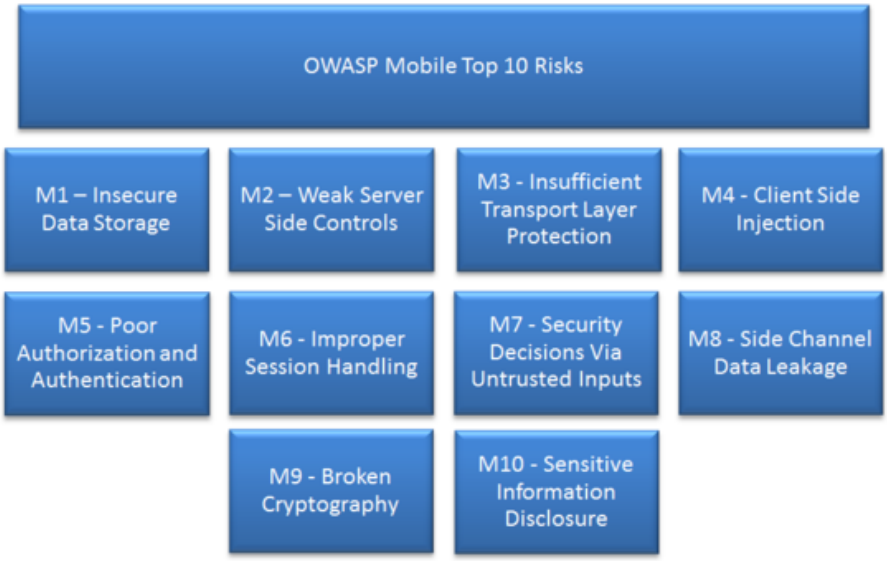

Figure 2.6: The top ten mobile risks identified by OWASP for 2011

Summary

The aforementioned classifications are some of the more popular ones in existence. Most of them are not specifically aimed at classifying Android vulnerabilities and many of them have other purposes than education. For instance STRIDE meantt to be used at Microsoft to do threat modeling when creating or auditing applications. The OWASP Mobile Top 10 classification is suitable for both security people and developers because it splits the risks up into more fine grained categories that can still contain a number of different vulnerabilities. It is also tailored for mobile risk and we therefore believe it is the most useful for educational purposes. OWASP Mobile Top 10 is not a complete classification and so we will have to extend it if we find vulnerabilities that do not fit into the existing categories, but it serves as a good starting point for classifying vulnerabilities and is basically a vulnerability classification by error / mistake.

Classification by SDLC	+ Well suited to educate developers - Not very useful for security researchers and professionals because vulnerabilities are grouped by software development phase
Classification by genesis	- A Binary classification is too coarse for our purpose - Hard to decide which category to place a vulnerability because we don't know the originator of the vulnerability
Classification by Location in Object Models	+ Possible to tune the granularity / abstraction level by creating a custom model - No standard object model for mobile / Android
Classification by Affected Technology	- Vulnerabilities are often technology agnostic
Classification by Error or Mistake	+ Basically what OWASP Mobile Top 10 does - Can lead to a large amount of categories
STRIDE	- Only 7 categories - More useful for threat modeling - A bit too coarse classification for our purpose
OWASP Mobile Top 10	+ Specifically aimed at mobile risks + Suitable abstraction level - Is not complete, only contains the top 10 risks identified

Table 2.1: Summary of the pros and cons of the evaluated classifications

2.4 Malware

We have chosen to use the following definition of malware by Roger Clarke [12]:

"Malware is software, a software component, or a feature, that comes by some means to be invoked by a device, and that on invokation has an effect that is unintended by the person responsible for the device, and potentially harmful to an interest of that or some other person"

This is a general definition of malware, but it applies to malware on Android devices as well.

Most people think about malware as something only found in traditional computers such as laptops, and are unaware that malware is becoming widespread on mobile phones as well. Recent trends show that we are likely to see a further increase in malware on Android systems in the coming years [33].

The undesirable effects of malware are among others lost user data, system resource depletion or an attacker having full control over the device. There are several categories of malware. On Android, the most seen category of malware have so far been trojans, which are applications that seemingly perform some legit function but also contains malicious code. The reason why trojans are so common is that it is the most convenient way of getting an application installed on the device and thereby being able to execute code.

Trojans are defined by their attack vector, or the way the trojan application comes to be on the device, namely that they are intentionally installed by an authorised user as the result of social engineering or otherwise convincing the user that the application is legit.

2.4.1 Categories of Malware

There is a multitude of malware and several different ways of categorizing them based on properties such as attack vectors, functionality, if they spread or not etc. The categories are usually named such that they represent the way that they behave. We introduce what we feel are the most commonly accepted categories of malware, and the ones that are most relevant to the Android platform as of the time of writing.

Trojans

Trojans are programs that disguise themselves as legitimate applications, are intentionally installed by an authorized user, but contains code that secretly executes and performs malicious activities that have undesirable effects. Trojans are non-self-replicating and often include a backdoor to allow an attacker to control the system on which the trojan was installed [62].

Viruses

Viruses are programs that can replicate and spread from one computer to another. They typically attach or inject themselves into executable files. When the legitimite file is executed the virus code is usually executed with it, causing it to spread to other hosts and files. The malicious activity of a virus varies beyond spreading and attaching themselves. They may have noticable impact on users such as modifying or corrupting files or they may just spread without doing any real harm [58]. In the latter case a virus is still using system resources and should therefore be considered malicious or at least unwanted.

Worms

Worms are programs similar to viruses in that they spread from host to host. However, worms do not attach themselves to existing programs. [59].

Rootkits

Rootkits are stealthy malware that usually conceals itself after it has been installed. It typically attacks the lower-levels of the system and may inject itself into system library files or alter parts of the operating system [60]. The rootkit may be used to hide the malicious activity of an attacker by modifying what is returned to the user by system programs. An example of this is that a rootkit can modify the output of the "top" command to not show some malicious process running on the computer.

Spyware

Spyware is software that gathers information without the user's knowledge and shares it with some unauthorized party. Spyware may differ in the way they gather information or what information the spyware author is interested in. Typically "valuable" information to an attacker is credit card information, the phone's IMEI number or personal information that can be used by so-called ransomware [2] Other information that a spyware application might be interested in is location information, WIFI passwords, contact list information, browser history and SMS-messages. A modern mobile phone contains a lot of information with storage space up to 128GB on some phones [3]. Targeted spyware applications could use a phone's hardware to do microphone recordings, take pictures using the camera's phone or steal account credentials. Most of these things would either require the attacker to gain root privileges or have the user accept all the permissions that the application

[2]Ransomware is malware that takes a device hostage, that is it renders the device useless until the victim takes some action such as paying the attacker money in exchange for a code that unlocks the device.

[3]The Samsung Galaxy S4 comes with up to 64GB internal memory and up to 64GB of external removable storage

would need at install time. The attacker could then for instance log the keystrokes of the user to obtain sensitive information and send it to a remote server using the network [61].

On the Internet there are known to be websites that trade stolen personal information in bulk and criminals wishing to perform credit card fraud or other malicious activities are willing to pay for this information. A Symantec report from 2008 based on data from underground forums and chat rooms claims that valid user e-mail addresses were worth between $0.33/MB to $40/MB [52].

2.4.2 Android Malware

Android malware is still in its cradle after only five years of the platform being available. The last few years' surge in popularity seems to have piqued malware authors' interest and we are seeing an increase in malware on the Android platform [33].

Android malware have been observed to have several different malicious intents. A survey of mobile malware in the wild was performed by Porter Felt et al. in [24]. They recognized the most common malicious activities to be in decreasing order: Collection of user information, sending of premium-rate SMS messages, credential theft, SMS spam and search engine optimization fraud. They also found that some of the malware was written for novelty or amusement and some malware would press the target for a ransom. The creativity of malware authors has evolved along with the technology that the malware targets, and we will surely see new ways that malware operate in the future.

Zhou and Jiang collected and analyzed over 1200 malware samples for Android from 2010 to 2011 [66]. They found that 86% of the collected malware were repackaged versions of legitimate applications. They also found that the malware was evolving by making use of more advanced techniques such as encrypting or obfuscating payloads. Another technique they found was the use of an update component that downloaded malicious code at runtime. These new techniques make it much harder to recognize an application as malware because most malware detection tools use static analysis or signature matching. If a malicious application is downloading the payload at runtime, traditional malware detection will not work.

Malware seen in the wild is often quickly hacked together applications that include public exploit code with the original comments by the author and no modifications made. A report from September 2012 by Duo Security shows that there are big gaps in time between new publicly known vulnerabilities and patches being pushed by vendors / carriers. They found that over 50% of devices contain unpatched vulnerabilities [48].

Most mobile malware authors seem to have financial gain as their main motivation. They usually choose the path of least resistance and the simplest way to exploit users has been to send premium-rate SMS messages from which they earn money directly. This problem has been mostly solved in Android version 4.2 by introducing new controls, prompting the user when an application tries to send a premium number SMS [4]. Still, as few as

5.6% of users are running Android 4.2 at the time of writing (July 19, 2013).

Mobile phones hold a lot of personal information like contacts, e-mail and sms messages, WIFI passwords etc. Phones may also contain password managers that can give an attacker access to other services that the owner of the phone uses. Mobile banking applications are now common and even apps for home security systems that control fire alarms, sprinklers and door locks are in use. Password manager databases are likely to be encrypted but an attacker that can obtain root privileges on the device will be able to record keypresses and thereby obtain the encryption passphrase as it is entered by the user.

Sensitive information obtained by an attacker can be sold to black markets that trade stolen information en masse. For instance a phone's IMEI number is a popular target because stolen phones will have their IMEI blocked from network service by the service providers. By changing the IMEI number to a valid one the stolen phone can regain service and therefore IMEI numbers have value on the black market.

Attackers gaining access to devices containing home security systems could disable alarms before a physical break-in or activate the fire alarm just to prank the victim. These are threats that have mostly been pictured in movies but are now becoming a real possibility.

Mobile devices are also increasingly connected to other computing devices through wireless networks and bluetooth we may start to see malware that target devices that are relatively new to the network such as smart TVs.

Another common action of malware is to include the compromised host in a botnet. Botnets are collections of compromised devices that are networked and under the control of the owner of the botnet. There is typically a central authority called a Command Server that can send commands over the network to its bots to control them, steal passwords, credit card information or other personal information over time. This data, when sold, generates income for the owner of the botnet. Botnets can even be rented out to other malicious users wanting to send spam or perform a Distributed-Denial-Of-Service attack. The owner of the compromised host may not even know that her device is under the attacker's control. Many Android malwares seen in the wild include the target device in a botnet [66, 31, 50].

Payloads

The payload of a malware is basically the code that is executed once an application or a system is compromised. It represents the actions that an attacker choose to take once security of the system is penetrated.

Android malware payloads often exploit vulnerabilites to escalate the privileges of the malware. If the malware can obtain root privileges the attacker will have full control of the mobile device and be able to use the full capabilities of the device at his own will. The attack surface for root exploits are the entire Linux kernel, the Android framework

and various native utility programs, in other words, the whole system software stack. Six root exploits that have been used by attackers in Android malware was shown in [66]. Malware authors often include one or more publicly available exploits in their malware and the code is often copied from the Internet.

When vulnerabilites are publicly released there is a window of time where devices are very easy targets for malware authors until they are patched. So, because root exploits are hard to come by, attackers typically use the publicly available ones and just repackage existing applications with new publicly available root exploit code to lure users to install their malware.

2.5 Privilege Escalation Attacks

The main goal of most malware once it has made it onto the device is to escalate its privileges. Unless the user has granted the malware dangerous permissions there isn't much mischief that can achieved in the confines of the application sandbox. The ultimate goal for a malware author is to obtain root privileges on the device, but this may prove to be hard on a device that has the latest updates. However, there are other creative ways to escalate privileges with the help of other applications as described in the following sections.

2.5.1 Transitive privilege escalation

Porter Felt et al. point out the risk of permission re-delegation on Android because of the way the system allows inter-process communication through intents [25]. Android allows applications to expose functionality publicly for other applications to reuse. The system does not however check if the invoking application has the permissions required by the system to use the functionality exposed. This makes it possible for unprivileged applications to perform privileged system calls via an intermediary if the intermediary exposes functionality that basically perform these system calls without checking the invoker's permissions. Porter Felt et al. propose a solution in the form of an operating system mechanism that they call IPC Inspection. This mechanism's purpose is to prevent the exploitation of permission re-delegation by reducing the invoked application's permissions when it receives a call from a less privileged application. As a case study, Porter Felt et al. examined 872 applications on Android 2.2 for the risk of this kind of attack, and found that 37% of the applications have permissions and at least one type of public component, including 11 out 16 system applications which were at risk.

Davi and Dmietrienko et al. show that privilege escalation attacks are possible on Android applications because the sandbox security model does not protect against transitive privilege usage [15]. They perform a privilege escalation attack in practice to prove that the

vulnerability exists. Their attack has some prerequisites. For instance the Android Scripting Environment (ASE) application needs to be installed. The attack goes as follows. An application containing native code (C/C++) is installed on the system. It is assumed that the application has been authorized by the user with the INTERNET permission and the application contains a vulnerability in the native code, e.g. a "setjmp" vulnerability. The authors exploit the vulnerable application and invoke libc's system function to start a Tool Command Language (TCL) client function, connect to the ASE server and send a message to the server saying it should send 50 text messages to a premium service number. The ASE server complies because it does not check the permissions of the invoking application. This attack was possible on the Android 2.0 emulator and Android 1.6 on a Android Dev Phone 2 according to Davie and Dmietrienko et al. [15]. The attack described here does make some strong assumptions and is not straightforward because of the sophisticated technique used; return-oriented programming (ROP) without returns, but it illustrates the limitations of the Android sandbox security model.

2.5.2 Root Exploits

Root exploits are exploits that take advantage of some vulnerability in the Android system's software stack or the underlying Linux kernel to obtain root privileges on a device. Being root is useful to users who wish to customize their device by installing some custom distribution of the operating system or tweaking the device in some other way. It's also paramount to an attacker that will gain full control of the device by obtaining root privileges. By default, Android applications are restrained to their own sandbox as explained in section 2.1.1. The point of a root exploit is to break out of the sandbox temporarily by taking advantage of a vulnerable piece of code and use the obtained privileges to maintain root access permanently. Höbart and Mayrhofer built a framework that can make use of temporary root exploits to gain permanent root access on Android devices in [29]. They show four example root exploits that have been discovered by others prior to their work. I will recount one of them in section 2.5.2 to give the reader an idea of how these exploits work. There are however differences in which level of the system these vulnerabilities appear. As recently as December 2012 a bug in Samsung's processors, Exynos 4210 and 4412, was discovered [63] and exploited to gain root privileges on Android devices incorporating this hardware.

Samsung Exynos vulnerability

The Samsung Exynos vulnerability was limited to devices using the Samsung kernel source code and was named after Samsung's own chip. The attack was possible due to a character device file having public read and write permissions set. The device file was equivalent to that of /dev/mem which is an image of the main memory on the device. This allows an attacker to read or write any address in memory which makes it possible for instance to inject processes into memory. With this access there are several ways for

an attacker to gain root privileges on the device. The vulnerability was confirmed by Samsung [16].

Gaining Root by Overflowing RLIMIT_NPROC

The Linux operating system supports settings limits for each user of the system. One such limit that can be set is the maximum number of processes a user can have running simultaneously. This limit can be found by running "ulimit -u" from the command line. According to [29] there was a vulnerability in Android up to (exclusive) version 2.2 because of this limit. The Android Debug Bridge (ADB) is a command line program used to get a command shell on Android devices over USB that can be used for debugging tasks. The ADB process on the Android device is started in the context of the "SHELL" user with root permissions on Linux and then drops the programs privileges using the "setuid" command. The reason why this is exploitable is that when the maximum number of processes for a given user is reached, the setuid call will fail. In the ADB program code there were no checks to see if the setuid call returned successfully and therefore the ADB process would continue to have root privileges, giving an attacker root access to the device over ADB. This vulnerability has been fixed and a patch has been released.

2.6 Malware Detection

There has been a lot of research on malware detection on traditional computer systems in the past. Still, it has not been straightforward to adapt existing malware detection tools to the mobile platform. The difficulties have mostly come from the fact that mobile devices have less resources available to them and require programs to be power efficient. Also, traditional techniques are far from optimal as most of them rely on signature based detection, leaving a window of opportunity for attackers that distribute new or rehashed malware until new signatures are made.

There have been many different proposals on how to detect malware, but the two main categories are static and dynamic analysis. Static analysis is done by examining the source or distributed binary of software to look for patterns or behavior that can indicate malice. Static analysis is usually fast and can be effective but it's still limited because it is easy for an attacker to obfuscate the source code or apply other techniques to make it hard to analyze. For instance an attacker could encrypt a piece of code and dynamically load and decrypt it at runtime.

Signature-based detection techniques fall under the static analysis category. Signature-based detection requires security professionals to obtain samples of malware, create unique signatures for them and distribute the signatures to the database of the user's anti-malware software. This technique is limited because it is relatively easy for an attacker to change the malware by using some form of obfuscation or encryption scheme and then the anti-malware program's stored signature won't match.

Dynamic analysis on the other hand uses metrics collected from running programs to detect anomalies. Metrics such as open files, network connections and number of threads can all be analyzed to establish "normal" patterns occurring in uninfected devices. These healthy patterns can then be used to detect anomalies occurring because of malware. The dynamic analysis approach requires lots of training data to be effective and also the system must be configured to avoid too many false alarms. But contrary to static analysis, the data that is analyzed is the program's actual behavior so the data is not obscured as may be the case with source code. Other problems with dynamic analysis is how to establish a secure, but realistic environment in which to analyze potential malware and for how long one should run the analysis. Malwares may stay inactive for long periods of time just to avoid detection by dynamic analysis techniques.

The most common implementation of dynamic analysis is heuristic-based. Heuristic-based methods are based on rules that are either determined by machine learning or defined by experts.

Shabtai et al. lists 13 research projects that have protection of mobile devices as their subject in [49]. 6 out of 13 used anomaly detection and 3 out of 13 used signature-based detection. Shabtai et al. tested different classification algorithms and feature selection algorithms in their host based intrusion detection system that they call Andromaly [49]. They measured a range of operating system- and application level metrics and used these to train their system to differentiate between benign and malicious behavior. Their research occurred very early in the life of Android so they did not have have any actual malware samples to test their hypothesis on, so they had to create their own. Their research showed promising results on their limited data set, but the researchers point out that dynamic analysis by itself is no absolute solution, but should be used in companionship with signature-based detection techniques as well as firewalls. Moreover, Andromaly proved best suited for detecting longer lasting attacks that consume system resources in an abnormal fashion. Short attacks which are typical of for instance premium SMS-sending trojans are harder to detect by using a machine learning behavioral-based analysis.

In 2012 Grace et al. presented RiskRanker [28], a system for detecting malware among Android applications. It has been shown to scale to match the large amount of apps being submitted to Android markets each day. RiskRanker can also detect malware that has previously not been seen, what they call zero-day malware in their paper. RiskRanker uses an analysis method divided into two steps, first they do a static analysis of the program to look for certain malicious behavior using a set of signatures that they created that model malicious behavior. Their models include seven kinds of known root exploits. The second order of the analysis aims to detect malware that dynamically loads encrypted code. Apps may be bundled inside apps, so the analysis module for the second order analysis looks for such apps and also looks for suspicious things such as dynamic code loading, native code, programmatic access to resource directories, and code that uses Cryptographic libraries. Any of the aforementioned behaviors may indicate a malicious application and require closer inspection. A common pattern for malware is accessing resources, decrypting payload, and executing the decrypted payload which is in the form

of a native binary. RiskRanker's results were very positive. The system detected previously unknown malware samples and correctly detected many others among the 118 000 sample apps. RiskRanker was also tested against all known malware samples from the contagio mini dump [41] and the system reported 121 out of 133 samples. The samples that were not detected were amongst others spyware / grayware and phishing applications that the system did not aim to detect.

2.7 Extensions to the Android Security Model

Kirin is a security service proposed by Enck, Ontang and McDaniel [19] that aims to mitigate the risk of malware at install time. Enck et al. argue that it is often difficult for users to decide whether or not to install an application based on permissions. Kirin provides a way to automate the recognition of certain patterns of permission combinations that have been shown to be found in malware. By hooking into Android's application install process Kirin can evaluate the application's requested permissions against a collection of security rules and can then output whether the application can be considered safe or dangerous. This seems to be a good addition to the Android security model at a low cost because the checking of an application happens only at install time and therefore does not provide any run-time overhead. On the other hand install-time certification of applications is only part of the protection needed to detect and prevent malware on Android systems. Some apps actually require the permission set that is associated with certain malware and will cause false positives.

Burguera et al. proposed CrowdDroid [9], a system based on crowdsourcing that is based on users installing an app that collects data on the system calls each application make. Their idea is to crowdsource the collection of data and then centrally analyze it using a clustering algorithm to establish the difference in patterns of malicious and benign applications. This technique aims at detecting trojan application which is the prevalent sort of malware for the time being. This approach proved to be sound and provided good detection rates for both artificially constructed and real malware. Still the approach is limited to distinguish between malicious and benign applications that have the same name and version. This approach will therefore be easy to evade by attackers by providing original applications that include malware instead of just repackaging existing applications with malicious code.

2.8 Application Security

TaintDroid is an information-flow tracking system for monitoring sensitive data flows in Android devices developed by Enck et al. The tool was used to test 30 applications selected from the ones having the Internet permission of the free top 50 applications

of each category on Android Market (now Google Play). They found that half of the applications report the users' locations to remote advertising servers [17].

Enck et al. later did a horizontal study of the 1100 most popular free applications on Android Market. They used their own decompilation tool "ded" to do automatic static analysis on the recovered source code. They reported over 30 findings of varying degrees of severity. One of the most interesting findings of the paper is the number of ad and analytics libraries included in apps and that these libraries often communicate devices' IMEI over the network [18].

Transport layer encryption is important to protect sensitive user data transmitted over wireless networks. Major web sites are getting better at doing this and the web browser implements the visual cue of a padlock to ensure users that their connection is secure. Mobile applications have taken the role of the browser in many cases but the there are usually no visual cues that assure users that their connections are safe.

Fahl et al. did a study of 13,500 popular free applications from Google Play. They examined the applications for SSL vulnerabilities such as trusting all certificates, allowing all hostnames on a certificate and mixing SSL and no-SSL traffic. They found that 8.0% of applications contain SSL / TLS[4] code that is potentially vulnerable to Man-In-The-Middle-Attacks.

A survey performed in the same study showed that 50% of 754 participants could not correctly assess whether their browser session was protected by SSL/TLS or not.

[4]Transport Layer Security (TLS) is the successor of Secure Socket Layer (SSL) and they are both cryptographic protocols that aim to protect network communication from eavesdropping and tampering

Chapter 3

Research methodology

In this chapter we present our research questions again, list which research methods may potentially useful to answer our research questions and argue which ones are best suited given the time and resources at our disposal.

- RQ1: How can we effectively identify vulnerable application code in real-world Android applications?
- RQ2: What are the most common vulnerabilities or flaws in Android applications?
- RQ3: Which vulnerability classifications exist, and are they suited to classify Android application vulnerabilities?
- RQ4: How can vulnerabilities in Android application code be exploited by malware or malicious users?
- RQ5: What are the implications of vulnerable application code to end-users?

3.1 Potential Research Methodologies

In figure 3.1 we show the research methodologies that we considered using to answer each research question. We now argue which methodologies are best suited to answer each of the research questions and discard the ones that we find not to be suited because of time-constraints, lack of skills or resource limitations.

RQ1

By studying peer reviewed literature we hope to find techniques that have been previously used to discover vulnerabilities in Android applications. This is likely to yield tools that are developed by researchers, but will not necessarily help us find the best tools for the job. To find tools and techniques used in industry as well as by attackers we will use search engines and search through Internet forums.

We could contact companies doing work on Android application security to interview them about the techniques and tools that they use, however we choose not to do this because of time limitations and a lack of connections to companies doing Android application security assessments. We feel that we should be able to find tools and techniques to use from existing resources or we will make our own tools where necessary.

RQ2

By studying peer reviewed literature we look for similar work on Android application security. This will help us form a picture of which vulnerabilities are common in Android applications. By complementing this with a case study of our own where we assess popular applications we will be able to assert which vulnerabilities are most common. A problem with this approach is that we will not uncover all vulnerabilities in the applications that we assess. We will need to tailor our assessment to look for certain types of vulnerabilities and thereby possibly missing others.

A survey presented to Android developers and security professionals could provide more data on which vulnerabilities are most common in apps. One problem is to find enough participants for such a survey to provide quality data and make sure that the participants form a representative selection. Responses to the survey may not have the necessary quality based on criteria such as competence within the area, experience and willingness to contribute to the research.

RQ3

We want to find out if classifications / taxonomies for vulnerabilities sufficiently cover the vulnerabilities found in Android. To do this we could do a literature review of peer reviewed research looking for existing classifications and taxonomies for vulnerabilities and then apply them to the vulnerabilities found by answering RQ2. It is possible that some classifications are not covered in research literature but are still useful so we will extend our search beyond research literature and include Websites and Internet fora.

RQ4

To find out how vulnerabilities in Android applications can be exploited by malware we must look at what existing malware has done in the past. Some malware may be presented in research publications but we should probably extend our search to include websites presenting new malware, such as security companies selling anti-virus applications, security alert websites etc. We will use search engines in addition to research publications to find enough data to answer this question.

RQ5

The implications to users are to some degree known from vulnerabilities in applications running on traditional operating systems and web applications. We will research if there are any new implications for end-users that we are not used to seeing in traditional computing environments caused by potentially new types of vulnerabilities.

RQ1	- Literature review of peer reviewed papers - Extensive literature review including the use of Internet search engines and forums - Qualitative study: Interview security professionals about tools and techniques used to find Android vulnerabilities
RQ2	- Quantitative study: Survey Android developers and security professionals about which vulnerabilities and flaws they most often encounter - Literature review of peer reviewed papers: Find papers reporting on vulnerabilities in Android applications - Extensive literature review including the use of Internet search engines and forums - Exploratory Case study: Do a case study of Android applications to look for vulnerabilities in real applications
RQ3	- Literature review of peer reviewed papers - Extensive literature review including the use of Internet search engines and forums
RQ4	- Literature review of peer reviewed papers: Look for existing malware - Extensive literature review including the use of Internet search engines and forums - Exploratory Case study: Look at vulnerabilities found in the case study and model implications
RQ5	- Literature review of peer reviewed papers - Extensive literature review including the use if Internet search engines and forums - Exploratory Case study: Look at vulnerabilities found in case study and model implications

Table 3.1: The table shows the potential research methodologies that we could use to answer our research questions

3.2 Case Study

When considering which research methodologies are appropriate for our study we found that an exploratory case study of real applications would be interesting to collect data about which vulnerabilities are common in Android applications. Yin proposed five components of case studies in his 1994 paper [65]:

1. A study's questions,
2. Its propositions, if any,
3. Its unit of analysis,
4. The logic linking the data to the propositions, and
5. The criteria for interpreting the findings

The study's questions is our research questions presented in the start of the chapter. Due to the nature of our research questions, we do not have a proposition. An application is our unit of analysis. Point 4 is not relevant as we do not have a proposition. Our criteria for interpreting our findings is based on existing classifications of vulnerabilities

and the notion that some source code, functionality or operations of an application may be exploitable by an attacker if misused.

We base our analysis of each unit in the case study on the technical guide for security assessments presented in NIST 800-115 by the American National Institute of Standards and Technology [47]. The guide presents four main points the first of which is on creating a security assessment policy which we believe only serves as overhead for independent security researchers rather than large organizations. The remaining three points are:

- Implementing a repeatable and documented assessment methodology
- Determine the objectives of each security assessment, and tailor the approach accordingly
- Analyze findings and develop risk mitigation techniques to address weaknesses

To satisfy the first point we develop a test plan based on OWASP's recommendations. The objective of our security assessment is the same in every case: Discover vulnerabilities and flaws in the applications under scrutiny. Our testing scope is limited to application code and does not target system level code such as the network stack.

We analyze our findings in the results section and try to develop mitigation strategies for the whole categories of vulnerabilities that we find.

3.2.1 Planning

The planning phase consists of gathering information about the system (Android in our case) and the threats of interest against the assessment object which in our case is each particular Android application. In this phase we also develop a test plan which we plan to execute for each assessment object. The test plan determines the scope of the assessment. With a team including multiple people this phase would also include project management with distribution of team roles and responsibilities, but in our case we try to avoid this unnecessary overhead.

3.2.2 Choosing Applications For Testing

The applications that were chosen for testing were selected from the Top Free list on Google Play. The reasoning behind this is that these apps are used by many users and therefore any vulnerabilities would have impact on a great number of users. The list is compiled from all available categories on the market based on the number of downloads. It is localized, seemingly to each country based on the popularity of the applications in the country where the user visits the page from. We visited Google Play from Norway and our top list of free applications reflect this.

We have established certain requirements for applications to be chosen. The minimum requirement is that the application communicates over the network. This is satisfied if the application requests the INTERNET permission. Most free applications require the INTERNET permission because the revenue generated by the app is dependent on ads served over the network. We also add a criteria that the applications chosen should have a minimum of functionality. That is, simple applications such as for instance flash light applications with only a single purpose is not selected, because they are unlikely to contain serious flaws with so little functionality. Table 3.2 shows the top list at the start of the project as it is seen from Norway. Note that not all applications are on the top list anymore but were chosen while they were on the list.

	Application	**Chosen**
1	Facebook	x
2	Snapchat	x
3	Spotify	x
4	Instagram	x
5	Skype	x
6	Candy Crush Saga	x
7	Yr.no	x
8	NRK-TV	x
9	Facebook Messenger	x
10	4 Pics 1 Word	x
11	FruitNinja	x
12	Wordfeud	x
13	Subway Surfers	x
14	4 Pics 1 Wrong	x
15	Crazy Dentist	x
16	Tiny FlashLight	
17	InstaWeather	x
18	QuizBattle	x
19	180.no Mobilsøk	
20	Yr	x
21	Brightest LED Flashlight	
22	Hill Climb Racing	x
23	Vine	x

Table 3.2: Top free popular applications on Google Play at the start of the project. Many of the applications assessed are no longer popular enough to be on the top list.

3.2.3 Making a Test Plan

To create a test plan we need a framework of tests that are suitable for the Android mobile platform. The Open Web Application Security Project (OWASP) website has a subproject

called the OWASP Mobile Security Project which provides such a framework [45] in the shape of a general mobile test plan.

We adapt the general mobile test plan for the Android platform and choose the approach and tools that we use for testing the specific points in the plan. Not all aspects of OWASPs general test plan are relevant for our purpose so some points have been left out and others have been added where necessary.

Our starting perspective is black box without any of the original source code being available. This is likely to be the starting point of an attacker as well and it means that the application has to be analyzed by looking at the binary of the application to reveal the structure and actions of the applications. We later try to obtain a white box perspective by recovering the source code of the application and inspecting the resulting code.

Also, by using an Android device on which we have root permissions, it is possible to monitor the files created and stored by an application which are usually not readable except by the application itself.

The complete test plan can be found in appendix A.3. The test plan is executed for all the selected applications, but some parts may not be relevant to all applications and are left out. Other applications may require more extensive testing for some aspects if our inspections indicate it.

Information Gathering

The first step of the test plan is information gathering. For each application this involves using the application to understand its functionality and the purposes of the various functionality. Questions that we try to answer include the following: Does the application make use of any device specific hardware such as camera or microphone? Which permissions does the application request? Does the application interact with any other applications?

Static Analysis

The second step is static analysis. In this step the application is decompiled in an attempt to recover the original source code. Tools for doing this are explained in more detail in the appendix A.2. Having the source code allows us to see how functionality is implemented, what method calls are being made and how data are stored in the application. This may provide clues to which parts of an application are more likely to be vulnerable. This step is time consuming but has great potential for revealing vulnerabilities.

Dynamic Analysis

The third and final step in the test plan is dynamic analysis. Dynamic analysis involves communicating with the application through its interfaces at runtime. For instance by passing custom intents to the application, we can invoke the applications functionality directly, passing along custom parameters which we control. We use the tool Mercury (see A.2.3) for this. Another way to communicate with the application at runtime is through the network. Many applications communicate with a web service, or another device running the same application (Peer-to-peer). If the communication is not properly secured, that is data is encrypted and the integrity of the sender and receiver is checked, it may be possible to become a man-in-the-middle or even impersonate the web service. We can then control the information that the application sends or receives and thereby send unexpected data to misuse the application.

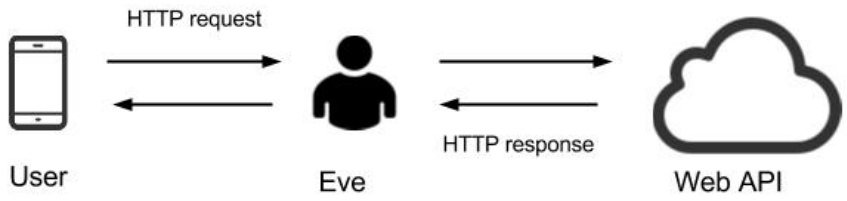

Figure 3.1: Shows how a man-in-the-middle attack can alter data in transit. This is trivial if the data is not encrypted

3.2.4 Execution

The second phase is the execution phase in which we execute our test plan in order to identify vulnerabilities and validate them. This phase also addresses the various techniques used to uncover vulnerabilities.

Network Sniffing

Network sniffing is a technique used to obtain the network traffic produced by an application on the mobile device. We run tcpdump (see A.2.5) on the device to sniff traffic generated while interacting with an application and transfer the dump of the network traffic to a computer for analysis. Wireshark (see A.2.5) is used to analyze the network traffic dump produced by tcpdump offline. What we can discover from this analyses is where

web APIs are located, what protocols are used to communicate with the web API and what data is communicated over the network.

Decompilation

Decompiling the source code using Dare (see A.2.2) we are able to recover most of the applications' source code and inspect it both manually and automatically. We inspect all the source code of each application manually in order to reveal logical flaws or hard-coded sensitive data. We also use scripts to search for terms of interest so as to not miss any important pieces of code or hardcoded secrets. This process is time consuming and admittedly error prone because it depends on the examiners skill in reading code.

File System Monitoring

Applications can store files in their own sandboxed folder. But the sandbox is accessible to anyone with root privileges, which means we can access it on a rooted test device. This means that applications should not store sensitive files unencrypted on the device. We manually pull files from an application's data directory and analyze them offline.

Log Monitoring

Android provides applications with a Log class that allows the developer to print debug information to screen. There are different levels of severity for the logging messages and each level has its own Java method. The levels are: Verbose, debug, information, warning and error. Android also provides a utility program called LogCat that is used to view or save the log at run-time. Logging output may provide sensitive or useful information to us while testing an application.

3.2.5 Testing Process

The testing process is performed iteratively over the three parts, information gathering, static analysis and dynamic analysis. Each iteration may provide information that can be used in another phase in the next iteration. The goal of the process is to get familiar with the application in order to identify attack surfaces and vulnerabilities. By understanding the data and process flow of an application we can more easily find where errors are likely to happen or where we can interfere with the process or data flow. Obtaining close knowledge about complex applications and protocols requires significant time and effort, but knowledge obtained about one application is sometimes transferable to other applications because many applications use the same popular libraries.

3.2.6 Post-execution

The last phase involves analyzing and classifying the vulnerabilities found and uncovering root causes. One of our goals is to come up with mitigation strategies for each category of vulnerabilities that we can recommend developers to apply.

3.3 Ethical Considerations

When doing security assessments of real applications, certain ethical and legal considerations must be addressed. By performing a security assessment involving testing we act as attackers in order to find and address problems before malicious users find and exploit them. When we find vulnerabilities we inform the company or developer that have created the application and give them enough information and time to fix the flaws before they are released online and in reports like this one. We do not seek to take advantage of any vulnerabilities for personal gain or to cause any harm to a company's services or infrastructure. Therefore, the testing is designed to be unintrusive and does not for instance target web services in an active manner or otherwise disrupt service for other users of the applications.

The courtesy period between when a researcher notifies a company of a security flaw in their software and when he publicly releases the details of the flaw have traditionally been recommended to be 60 days. With the changes in tempo to release schedules and the frequent discovery of zero-day exploits Google, which has been the main driver for this de-facto standard, has reconsidered and is now recommending a grace period of only 7 days. This recommendation is a step towards pushing companies to produce and push patches more quickly to users. This change of recommendation from Google is hoped to help limit the time window that attackers have to exploit vulnerabilities before they are patched [46].

Chapter 4

Results

4.1 Applications Assessed

A total of 20 applications were chosen and assessed. They are listed in table 4.1. All the applications were chosen from the Top Free list on Google Play (in Norway) during the months February to June 2013. There are various applications in the set but there is a dominance (10 out of 20 apps) of games.

Category	**# of apps**
Arcade & Action	3
Brain & Puzzle	4
Casual	2
Communication	2
Entertainment	1
Music & Audio	1
Racing	1
Social	5
Weather	1
	20

4.2 Vulnerabilities Found

Here we present the vulnerabilities that we have found by assessing 20 applications. We have classified the vulnerabilities using the categories found in the OWASP Top 10 mobile Risks where possible.

	Android permissions	Other permisssions	Total permissions
4 Pics 1 Wrong	5	1	6
4 Pics 1 Word	2	1	3
Candy Crush Saga	3	1	4
Crazy Dentist	9	0	9
Facebook	26	13	39
Facebook Messenger	25	8	33
Fruit Ninja Free	8	0	8
Hardest Game Ever 2	11	2	13
Instagram	13	2	15
InstaWeather	10	0	10
Mountain Climb Race	14	15	29
NrkTV	3	0	3
Snapchat	13	2	15
Spotify	10	0	10
Subway Surf	9	4	13
Skype	28	0	28
QuizBattle	7	3	10
Vine	9	2	11
Wordfeud	10	2	12
Yr	4	0	4

Table 4.1: The table shows the applications that were tested and the number of permissions that they request. Permissions inherent to the Android system are separated from permissions that have been added to the system by other applications.

4.2.1 Insecure Data Storage

This category contains vulnerabilities that involve sensitive information stored on the device. An Android device can store data either on the device's SD-card or on the device's internal storage. Each application has its own sandboxed folder that only the application can access. On rooted devices however, the owner of the device can access any file or folder. This means that any data stored on the device should be considered in the hands of an attacker. Therefore we consider sensitive data to be unprotected if it is stored on the device and it is not encrypted using some cryptographic scheme that is hard to break, e.g. AES.

Find 1

The 4Pics1Word app is a simple game where the user is presented with four pictures and have to guess the word that describes all of these pictures to get to the next level. The app went viral in the beginning of 2013 and was on the top of Google Play's "Most Popular Free Apps"-list in Norway for some weeks. There are thousands of levels and when a user gets stuck she can buy virtual coins to use in the game to reveal a character of the word.

There is a set of 12 characters to choose from to create the solution word and the user can use coins to remove characters from this set. Coins can be bought in different packages, with the cheapest one giving you 350 coins cost $0.99 and the most expensive package of 10000 coins being valued at $19.99.

By simply investigating the files of the application using a rooted Android device (see appendix A.1.1) it became clear that the application is storing the game's state in clear-text in JSON format. The data stored in this file include the current photo ID, current level and the number of coins owned by the user. By simply modifying this file and restarting the application it is possible to change the amount of coins owned by the user and the level that the user is on. Additionally, the application stores all the image files for all levels locally, along with the solution for each level in another JSON formatted file. This is a typical case of developers not securing the applications data appropriately, but relying on the operating systems security measures alone. On a device without root access the data would be safe due to the Android sandbox protecting the applications data directory. On a rooted device however, a user is free to modify any file of the system and the data is compromised. In this case the vulnerability could lead to a massive loss of potential revenue for the developer but does not pose any direct threat to the user.

The lack of data security in this application may have been caused by developers wanting to push the application to market as fast as possible and therefore not putting a great emphasis on securing the data of the application. This is a valid risk vs. reward decision as long as it does not compromise the user's personal information or privacy. It does however devalue the virtual currency used in the game and may result in paying users feeling swindled if others use this shady way of obtaining virtual coin.

Find 2

4Bilder1Feil is a very similar game that has made it into the Most Popular Free Apps list. In this game the task is to select the odd one out among four pictures. As a reward the player receives diamonds. The diamonds can be used to restock the five "lives" that the player starts with. Additional diamonds can be bought with real money at a cost of $1.29 for 5000 diamonds.

4Bilder1Feil has exactly the same problem as 4Pics1Word. The game stores its game state in clear text on the device, relying on Android's sandbox to keep it safe. With a rooted device, it is trivial to access and change this file. This allows a malicious user to change the number of diamonds to any number of his choosing making it possible to obtain free diamonds.

Find 3

Hardest Game Ever 2, is a game application that was recently very popular on Google Play peaking at over 1.1 million active users monthly [6]. This application integrates with Facebook to let users compare their score to their friends' score, invite friends to play the game etc. To do this the application needs to create a Facebook Application and then integrates the mobile application using the Android Facebook SDK. When the

application has authenticated with Facebook it receives an access token. The following excerpt is taken from the Facebook developer documentation and it explains what the access token received after making an authentication call is for:

> "This call will return an app access token which can be used in place of a user access token to make API calls as noted above. Again, for security, app access token should never be hard-coded into client-side code, doing so would give everyone who loaded your webpage or decompiled your app full access to your app secret, and therefore the ability to modify your app."

Looking at the consequences of losing an access token makes it clear that storing the access token in a clear text XML file is a vulnerability. Further investigation shows that this access token is invalid because the Facebook application that the access token is for has been deleted and the game has a new Facebook application with a different ID. This may be because the application has already been exploited using this vulnerability, but we have found no proof to support this speculation.

Find 4

Wordfeud Free is a game application that imitates the game of Scrabble. It connects pairs of players over the Internet to let them play together. Wordfeud requires players to log in using either their Facebook account or by using an email account and a password. We found that the Wordfeud Free Application is storing passwords in clear text in an XML file in the application's folder on the device's internal storage. This makes it possible for an attacker with root access to obtain the user's credentials which is often reused at other web services.

4.2.2 Weak Server Side Controls

Find 5

By using Dare to decompile Wordfeud Free we found that the application fetches profile images from a web API. By investigating this API we found that all profile images were accessible by providing the ID of the user in the URL. The user IDs appear to be sequential, allowing an attacker to easily collect the profile pictures of all Wordfeud users. Wordfeud lets users connect to Facebook or log in with their Facebook account and then let users use their profile picture from Facebook. This an example of an information leak that compromises the privacy of users. This is not a serious vulnerability, but a flaw nonetheless and may indicate that more authorization failures exist in the web API although we could not find any.

Find 6

QuizBattle is a two-player quiz game played on a virtual quadratic board with the players earning points and moving forward when they answer a question correctly. The game ends when one of the players reach the center of the board. Users must have an account to

play the game. The application asks for the email address of the user in the first screen and checks whether the address is already registered. To check if the email address is already registered the application makes an HTTP request to the server with the address as a parameter. The server replies with a JSON object with a boolean "success" parameter indicating whether the address is registered. There is also an analogous API call for checking usernames that are registered. The web API does not check whether the call is coming from the mobile application, this means anyone can enumerate both email addresses and usernames of the users of the game by writing a small script. This may not be directly exploitable but makes it a lot easier for attackers wanting to steal QuizBattle accounts.

4.2.3 Insufficient Transport Layer Protection

Find 7

As mentioned in Find 6, QuizBattle requires users to register for an account to use the game. They must submit a username and a password that they can use to access their ongoing games from other devices. By inspecting the network traffic of the application we found that the username and password are being sent to the server, unencrypted over HTTP with no transport layer encryption. This is clearly vulnerable to MITM attacks and makes it trivial for an attacker to steal the user's credentials and potentially also access other web services that the user employs. The response to the HTTP request containing the username and password is an authenticated session token that can be used to access the rest of the web API.

Find 8

Instagram is an application used to take pictures and share them with other users of the application. The application lets you establish relationships to other users by following them which means the pictures that they publish will become available to you in your feed. Users can also comment and like the pictures of other users.

By capturing our test device's network traffic while using the Instagram application we were able to inspect the incoming and outgoing traffic. The packet capture shows that Instagram establishes an HTTPS connection with TLS version 1 encryption. The authentication that we performed by logging in is likely done over this encrypted connection. Later however, the packet capture shows that Instagram is communicating with the Web API located at http://instagram.com/api/v1/. The connection to this API is not encrypted and so we are able to see the content of the packages sent. Most of these API calls need to authenticate the user to yield a valid response. To do this, each HTTP request contains a session token that is unique to the user's logged in session, proving that user is authenticated. The session token is stored as a cookie on the mobile device. Each of these session cookies are valid for three months and it is all an attacker needs to log into the account of the victim. Sending session cookies in clear text makes it very easy for an attacker to highjack sessions by sniffing the traffic of the victim. Because mobile devices move

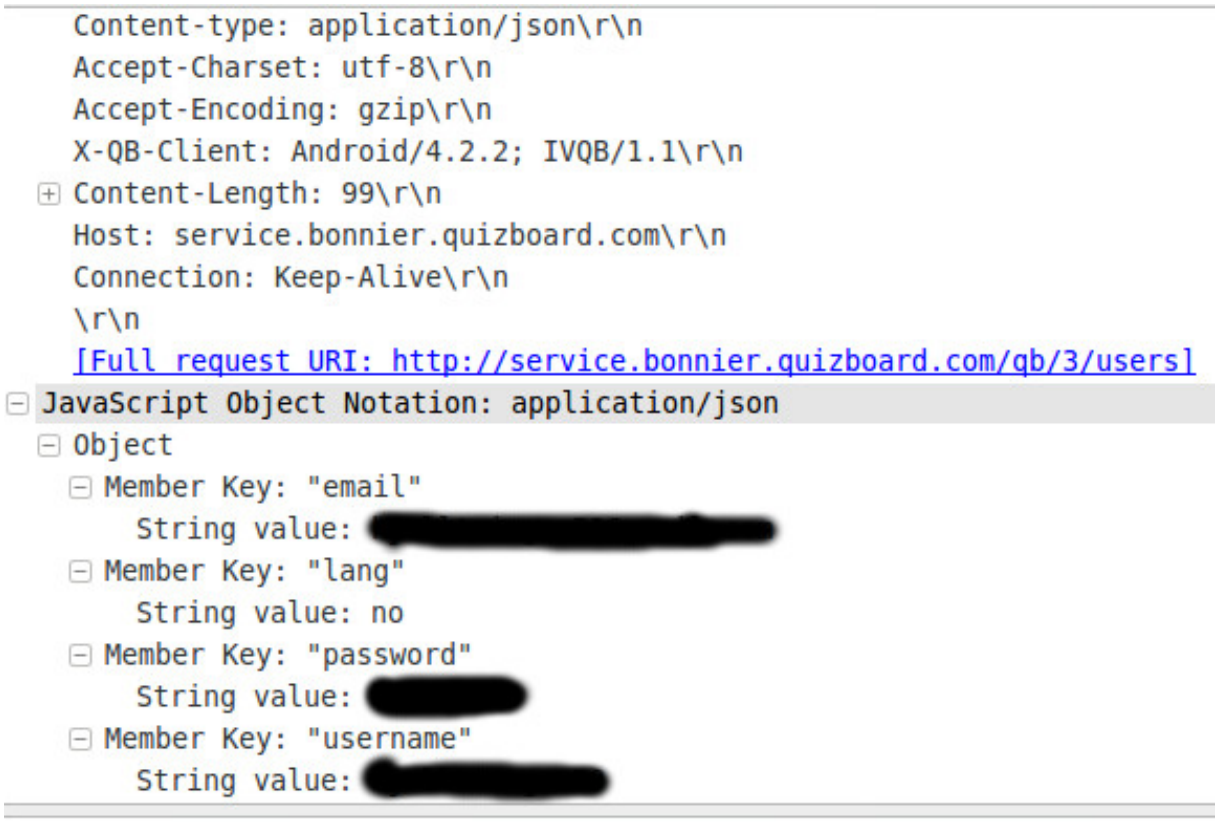

Figure 4.1: The figure shows an HTTP request coming from the QuizBattle application. The request contains JSON data including the username and password of the user in clear text. We have obfuscated the usernames and passwords here to not reveal any user credentials

around a lot and are connected to many different networks, many of them untrustable, they are very vulnerable to this form of attack.

4.2.4 Sensitive Information Disclosure

Find 9

Mobclix is a mobile ad provider and is included as an ad library included in 3 of the applications that was assessed. Examinations of the applications' web traffic show that the IMEI of the mobile device is included as a parameter of web requests to the Mobclix servers. Below we show an HTTP request made by the Fruit Ninja Free (com.halfbrick.fruitninjafree) application which includes the Mobclix library:

`http://ads.mobclix.com/?p=android&i=46397573-C748-4CD5-BFCD-DB612D04A52`
`&s=300x250&rt=mcnative&rtv=0&av=1.6.2.10&`

u=********

```
&andid=5e5149c4f06c78cc&v=3.1.1&ct=wifi&dm=HTC+Desire&
hwdm=bravo&sv=4.1.1&ua=Mozilla%2F5.0+%28Linux%3B+U%3B+Android+4.1.1
%3B+en-us%3B+HTC+Desire+Build%2FJRO03H%29+AppleWebKit%2F534.30+%28
KHTML%2C+like+Gecko%29+Version%2F4.0+Mobile+Safari%2F534.30
&o=1&ap=0&l=en_US
```

The parameter named “u“ contains the IMEI number of the sending devices in every request to Mobclix and is seemingly used to identify the client device. We have obfuscated the IMEI number here to avoid revealing it to the reader. As mentioned in chapter 2 the IMEI is sensitive information due to it being unique for every mobile device and its possible use in reactivating stolen phones that have been blocked from the cellular network.

4.2.5 Aggressive Ad Networks

The way that free applications are often made profitable is by presenting the user with ads while using the application. When we inspecting the source code of the assessed applications we found that they often include more than one ad library. This find was also documented by Enck et al. in [18]. One of the applications, Mountain Climb Race (com.awesomecargames.mountainclimbrace), also made use of on-device push notifications to show the user ads. One of the ad networks integrated in Mountain Climb Race is the Apperhand network that is reported by Symantec as malware with risk level 1 (low risk) [51, 53]. This form of ad presentation is quite intrusive and is presented in the notification bar as push notifications where the user is used to seeing trusted notifications from the system about things such as new text messages or updated applications. The mobile security company Lookout also wrote a blog post about Apperhand saying it is not malware but an ad network pushing the limits of privacy [35] Mountain Climb Race implements two other "push notification ad networks", namely SendDroid and Airpush. The problem with using push notifications for ads is that they occupy a space where users are not used to seeing ads. Therefore unknowing users may think the ads, which sometimes display messages such as “Your phone is infected by a virus“ as a an advertisement for an anti-virus program, are actually legitimate system messages.

4.2.6 Applications With No Discovered Vulnerabilities

In many of the applications we were not able to uncover any vulnerabilities. This does not mean that they do not contain vulnerabilities, but that we were not able to reveal any with our efforts. Because the applications that we tested are, or were, the most popular free Android applications they have probably been tested thoroughly by security professionals and developers before being released.

4.3 Mitigation Strategies

4.3.1 Insecure Data Storage

The storage provided by the Android system is sandboxed and only allows the application owning the sandbox to access the files within it. This model works for stock devices that have not been rooted and in theory makes the device a safe place to store data. The problem is that this sandboxed storage is on the client side, and out of the application developer's control. This is basically analogous to the Web's client - server architecture. The difference is that the client is now an Android application instead of a web browser and developers sometimes forget the lessons learned in application security about not trusting the client [56]. As long as the application is run on a device under the control of the user the data stored on the device must be considered accessible. Therefore no sensitive data should be stored on the device unless it is encrypted and the key is inaccessible to an attacker. Find 1 and 2 concerns games where the game state is stored on the device and can easily be tampered with by an attacker. There are at least two ways to handle this problem:

1. Encrypt the game state files and use native code to access it.
2. Store the game state on a web server and communicate client actions to the server using a web API.

Many of the game applications we assessed seems to be using the first approach. While this is not a complete solution to the problem, (the encryption key is still recoverable but requires reverse engineering skills) it raises the bar for tampering with the data. The second solution depends on verifying the transactions coming from the client on the server side and keeping track of game state on the server. This solution requires more resources, but if done right is more secure than the former alternative.

4.3.2 Weak Server Side Controls

Weak server side controls is not a problem with the mobile application per se, but the issue still affects users of mobile applications. Protecting the information of a web service API is a matter of defining who should be able to access what, and then enforce authentication, and check the authorization of users trying to access information on each request. This should be standard procedure but the lack of explicit security policies and automated testing makes bugs of this sort likely to avoid detection.

Another thing that a web service may wish to do is to verify that the requests are actually coming from the official mobile client unless the web API is intended to be open to the public.

4.3.3 Insufficient Transport Layer Protection

Mobile devices are especially prone to Man-In-The-Middle attacks (MITMA) because they connect to a lot of wireless networks, often in venues such as cafees, airports or even airplanes. These networks are not always trustworthy and even if they are, an attacker may actively fool the device to connect through a malicious host or access point. This means that encrypting traffic using SSL / TLS is a very important tool to protect sensitive information such as credentials, personal information, credit card numbers etc and should be implemented everywhere if possible. SSL / TLS relies on certificates to verify the authenticity of the server to which a connection is being established. Android comes with a list of trusted certificate authorities that is used to validate any SSL certificate that is signed by a root certificate coming from one of these certificate authorities. The problem with this is that if a certificate authority is compromised, which has happened in the past, all applications that rely on these certificate authorities become vulnerable to a potential MITMA. What developers could do in the case of mobile applications is use a technique called SSL pinning [36]. Since an application usually knows in advance to which servers it will connect, it can lower its exposure by bundling the certificate of the server, to which it is securely connecting, with the application and check the validity of the certificate presented by the server against the certificate stored locally on the device. By doing this we are sure that the server to which we are connecting is authentic and not a malicious user. This scheme relies on keeping the certificate of our service safe, but it limits the attack surface for an attacker.

4.3.4 Sensitive Information Disclosure

Find 9 showed that the ad network Mobclix sent the IMEI number of the device in clear text as part of the URL string. Seeing as this can be considered sensitive information the ad networks should either encrypt the IMEI before sending it to their servers or just use a fully encrypted transport protocol such as SSL / TSL.

Chapter 5

Discussion and Conclusion

We assessed popular free Android applications and tested them using a test plan that was refined from the OWASP Mobile Project's platform agnostic test plan and classified the vulnerabilities that we found according to OWASP Mobile Top 10 from 2011. We did not find any vulnerabilities that didn't fit into the existing categories in the OWASP Mobile Top 10.

Our first reseach question (RQ1) asked how we could effectively test Android applications. To answer this question we gathered techniques from existing well-researched domains such as Web application security and applied them to applications made for the Android platform. The test methodology is not original but it was necessary to put together existing methods and customize them for Android application testing. Our testing revealed 9 vulnerabilities. The methodology developed should be useful to security professionals and researches that wishes to test Android applications but are not quite sure how to go about it. There are now functioning tools to decompile applications with high percentage of success which makes source code inspection and static analysis possible without having the original source code. There are some tools (e.g. Mercury) for interacting with Android applications dynamically, but there are room for improvement in the tools for testing inter-application communication. Inter-application communication is an important interface to test as it provides an attack surface that can be exploited by malicious applications that have made it on to the device without going via the Internet.

For testing the web interface of an application there are already many existing web proxies and tools that are very good at this (e.g. Burp) because the principles of testing are similar to web application testing.

We believe that our testing methodology provides reasonably good coverage of an Android application's attack surfaces and the that tester would benefit from employing it. However, the tester will have to make a decision about where to focus to focus his energy given a specific application. This decision should be based on the information gathering phase in which the tester will get familiar with the functionality and application interfaces. For instance, the manifest file will reveal if there are many exported components.

If there are, then perhaps the tester should put an extra emphasis on dynamic analysis. The fact that we only made finds in 4 out of 10 categories either indicate that there are some elements missing in our testing methodology or that our sample did not contain any vulnerabilities in the remaining 6 categories.

Our testing methodology does have some drawbacks. First, it is time consuming to decompile applications using the Dare decompilation tool. Depending on the size of the application and the hardware available it could take over 10 hours to decompile an application, but as this does not require human supervision and can easily be automated we do not consider it a major obstacle. Second, the most time-consuming part of the methodology is the source code inspection. Depending on the size and complexity of the application it will take many hours to conduct a proper examination. An experienced tester could probably cut down on the time spent on this but it would still take up the largest share of the application assessment. This step is normally combined with static analysis to identify the most interesting parts of the application to focus on and to extract hard-coded secrets or useful information.

RQ2 asked which vulnerabilities are the most common in Android application. We made 9 finds in the applications that were tested. This shows that mobile applications does indeed contain vulnerabilities as expected. The most common vulnerability we found was insecure data storage (4 finds) which is the number one risk in OWASP's Mobile Top 10. We can see from the summary in figure 5.1 that the vulnerabilities that we found coincide with the rank of the risks in OWASP Mobile Top 10. Still, it is not possible to conclude if this is coincidental or not based on the small number of vulnerabilities that we found and the relatively small sample of applications that we tested. However, it can be seen as an indication of the correctness of OWASP's Mobile Top 10.

Category	**Finds**
Insecure data storage (M1)	4
Weak server side controls (M2)	2
Insufficient transport layer protection (M3)	2
Sensitive information disclosure (M10)	1

Table 5.1: Summary of the classes of vulnerabilities found

We emailed the developers about the vulnerabilities found in the applications in order for them to be aware of them and hopefully fix them. Some of the developers responded, and were positive about the interest taken in their application and were working to fix the problem. Others did not respond which may mean that they are not available, the email did not reach them or they do not care to fix these problems. There are of course differences in the companies behind these applications ranging from one developer to teams of hundreds of developers working to improve applications. Applications developed by a one man has very limited time and resources to develop and improve the application compared to big companies, which means security is likely to suffer in independent applications. Still, applications that are developed by large teams are often big and complex which are factors that can lead to vulnerabilities.

RQ3 asked which vulnerability classifications exist, and if they are suitable for classifying Android application vulnerabilities. To answer this question we collected several vulnerability classifications and evaluated their value in classifying vulnerabilities in Android applications. Based on our evaluation we decided to use the OWASP Mobile Top 10 classification which is basically a "classification by error or mistake". The OWASP Mobile Top 10 factors in risk, and is at an abstraction level that makes it easier to reason about mitigation strategies for the categories as a whole. All of our finds fit into the categories of the OWASP Top 10, but this does not mean that the classification is complete. There may be vulnerabilities that do not fit into the categories that we have not seen in our testing.

Having a good vulnerability classification with predefined categories to lean on is positive for testing as it provides a framework for the vulnerabilities to look for. The classification framework makes it easier for testers to know which vulnerabilities to look for, discuss variants of vulnerabilities with other testers and understanding why vulnerabilities arise.

RQ4 and RQ5 focused on how malware can exploit vulnerabilities in Android applications and how this will affect users. Of the vulnerabilities that we found there are some that can be exploited by malware. For instance, find 4 shows that Wordfeud stores credentials in clear text (M1) in the application's sandboxed file system which mean that they can easily be stolen by malware on rooted devices. This gives the attacker control of the Wordfeud game account of a user which isn't very serious in and of itself, but may serve as a platform for further exploiting a user's other Internet accounts. People often re-use the same passwords and so an attacker may be able to break into other accounts that the user have.

Insufficient transport layer protection (M3) isn't directly exploitable by malware that resides on the devices but could allow an eavesdropping attacker to highjack sessions. This is possible in the Instagram application because it communicates session tokens over an insecure communication channel. With a highjacked session an attacker can pose as the victim, spread misinformation, and steal or delete information. With high profile people such as politicians employing these applications this vulnerability becomes quite attractive to exploit for an attacker and may have severe impact in real life such as seen when the Assosiated Press' Twitter account was hacked and falsely reported bombings of the white house impacting stock exchanges [43].

5.1 Future work

We have tested a relatively small sample of applications and it would be interesting to do the same security assessment as we have done on a larger set of applications to see if the results coincide with the OWASP Mobile Top 10 list.

Our review of existing vulnerability classifications shows that there are no taxonomies that are a perfect match for Android vulnerabilities. Existing classifications are useful to some degree, but the fit depends on the purpose of the classification. Creating a new

taxonomy for vulnerabilities found on Android would be an interesting project.

In our research we only came across one tool for testing inter-application communication (Mercury). This tool is somewhat limited and tries to do a lot of things. It would be useful to develop a fuzzing tool specifically for testing inter-application communication on Android. This would be useful to reveal vectors that installed Android malware could leverage to exploit vulnerable applications on the same device.

Appendix A

Appendix

A.1 Devices

To conduct this research and testing we have primarily been using two devices in addition to the emulator provided by Android. One of the devices is rooted and a custom ROM is installed. It is useful to have a rooted device to work with as an attacker is likely to try and obtain root on the targeted device and which will provide a larger attack surface. Also, with many users rooting their devices to perform customization of user interfaces etc. it is important to study the extra risk they are exposed to.

A.1.1 HTC Bravo (Desire)

This device was already rooted at the start of this project but I installed an updated custom ROM to make the device more similar to most devices in use today. The custom ROM I chose is CyanogenMod [2] which markets itself as "an aftermarket firmware for a number of cell phones based on the open-source Android operating system". The reason I chose this was previous experience with it being stable and that seems to be one of the most popular custom ROMs on the Internet at the moment. The version used was CyanogenMod 10 which is based on Android 4.1.1 codenamed Jellybean and was installed in February 2013. The Linux kernel version installed is 2.6.38.

A.1.2 Samsung Galaxy S3 (GT-I9300)

The second device was a Samsung galaxy S3 with the stock OS installed and no root access. The device has version 4.1.2 of the Android operating system which is the latest update from Samsung at the time of writing. The kernel version installed is 3.0.31.

A.2 Tools

The tools used for this research are all freely available on the Internet and many of them are open source projects. Links to the tools are provided in the references as a convenience to readers who wish to perform their own research on Android Applications.

A.2.1 Apk Downloader

To analyze applications from Google's Play store we need to download the Application package file for each application. The way that Google Play works is that it requires a Google account which is also activated on the Android device. To download apps you can use the Play Store Android application to browse apps on the device, or you can log into your Google account in a web browser and browse apps on Google Play's website. If you decide to download an app you click install and choose which (if you have more than one) Android device you wish to install the application on. Google does the rest and the app is downloaded and installed on the device.

As a researcher wanting to analyze applications it is cumbersome to download and install applications to a device and then retrieve them to a regular computer for further analysis. Apk Downloader is a Chrome web browser extension that solves this problem by letting you download applications from Google Play without actually installing them on a device. The extension requires a Google account to work and valid device ID number. When it is configured correctly it will show a icon in the browser's address bar when an application's detail page is visited [1].

A.2.2 Decompilation Tools

When we have downloaded the APK file of an application we want to take a look at what's inside. How is the application coded. Does it contain any code that is vulnerable? ApkTool is an automatic tool that reverse engineers APK files and reproduces the resources of the application to the originals and the code in a format called SMALI. This tool can also rebuild an application after changes have been made to the reverse engineered format [3].

Dare is a tool created by researchers at Penn State. It converts Dalvik executables (.dex files) to readable Java source code. The techniques used are described in [42] and presented as a tool called ded. Dare is an improved version of ded. It first converts the dex file into class files and then uses a publicly available Java decompiler (Soot, at the time of writing) to recover the java source files [14].

A.2.3 Mercury

Mercury is an interactive console that lets us perform dynamic analysis of applications. The tool can do many things that is best described in the documentation, but it is invaluable in the search for vulnerabilities, unprotected resources and modules in applications [21].

A.2.4 Android Debug Bridge

The Android Debug Bridge (ADB) is a command-line tool who's purpose is to allow developers to communicate with a device or an emulator. The tool supports pulling and pushing of files, installing applications and starting activities. It gives you a shell for interacting with the Android system and is very useful for retrieving files, looking at debug output and getting familiar with the system. The tool is a part of the Android SDK and can be downloaded from Android's developer website [57].

A.2.5 Network Monitoring Tools

Wireshark is program that allows us to view a capture file from tcpdump for instance and easily filter the traffic based on protocol, source, destination or packet size etc. Wireshark makes it a lot easier to analyze the traffic and lets us view the packet content and headers if packets are not encrypted. [13]

Tcpdump is a command-line utility with that we use to capture all packets leaving and arriving the specified network interface card. We run this program on the mobile device as an easy way to capture all network traffic originating from and arriving to the application under scrutiny [55].

A.2.6 Various Other Tools

In the run of the assessment we have found the need for other tools as well such as "grep" for basic search for patterns in source code and "file" to reveal the format of files found on the device. The use of these tools are motivated in part by the book on Android Forensics by Andrew Hoog [30].

A.3 Test Plan

Phase 1: Information Gathering

What functionality does the application provide?

To map this we manually navigate through the running application to understand the basic functionality and workflow of the application. This can be performed on a real device or within a simulator/emulator. Perform all registration necessary to test the application

Which permissions does the application request?

- Recover the manifest file
- If app is not installed, use APKTool and decompile the application to recover the manifest file (faster than dare)
  - java -jar apktool.jar d example.apk
- If the application is installed, use Mercury to extract the manifest file
  - Connect to phone using mercury
  - run app.package.manifest com.example.packagename > manifest.xml

Through which network interfaces does the application communicate?

During testing only the WIFI interface is enabled on the testing device.

What networking protocols are in use? Are secure protocols used where needed? Can they be switched with insecure protocols?

Sniff the network traffic of the application using TCPDump on the device and store it on the SD card for easy retrieval. Make sure all other applications are killed before starting the packet capture. Then run the following command on the device via an ADB shell:

```
tcpdump -s 65535 -w $1 'tcp' > /sdcard/my_packet_capture.pcap
```

While the above command is running, interact with the application. Finally pull the file containing the packets captured using adb's pull command:

```
adb pull /sdcard/my_packet_capture.pcap
```

What data is communicated over the network?

To see what data has been communicated over the network by the application we use Wireshark to open our .pcap file from the previous step.

```
wireshark my_packet_capture.pcap &
```

Now we can inspect each packet that the device has sent and received during our capture. We are most interested in HTTP traffic at this point because it is the protocol typically used by Android applications and because HTTPS packets are encrypted and can't be inspected without the key. What we are looking for in this step is clear text sensitive information communicated over the network and URLs of web APIs that will be tested later on.

Does the application check where the data is coming from? Does it verify SSL certificates?

By using the Burpsuite Proxy we try to become a man-in-the-middle to read the encrypted network stream. This requires that the application accepts the certificate that Burpsuite presents. This means that if the application checks that the certificate offered to it from the server is valid, then this approach will not work.

Which hardware does the application use? NFC, Bluetooth, GPS, camera, microphone or other sensors? This question can be answered by inspecting the manifest file of the application because all the hardware mentioned above requires permissions to use and applications often include a <uses-feature> xml tag. [1]

Does the application export functionality?

This can often be revealed through the manifest file where exported components should be declared explicitly like this:

<receiver android:name="com.example.BillingReceiver" android:exported="true">

Components can also declare intent filters in code which implicitly makes the component public. The latter way of exporting components can only be detected by studying the source code of the application which we do in the static analysis phase.

Does the application communicate with a web service?

If so, which technologies are used server-side? This is revealed through our packet capture and inspection of the traffic captured. Sometimes we can determine server side technology

[1] https://developer.android.com/guide/topics/manifest/uses-feature-element.html

from URLs or from error pages presented by the web service. Hosting providers can often be revealed through the URLs as well (Amazon S3, for instance) or by using the Whois tool.

Does the application have a social integration or use a 3rd party Single Sign-on API? (oAuth, facebook, google, twitter etc)

This is usually evident by using the application and looking for share to facebook, twitter etc. Also if the application allows logging in with credentials from services like Facebook, Google and Twitter it is evident by using the application. This will also be detectable when looking at the source code in the static analysis phase.

Does the application employ payment services? (Paypal, Google wallet etc)

Again, this can be discovered by using the application and looking for mobile commerce functionality. Typically games use payment services to allow users to buy in-game content.

Are there exposed web resources in the applications web API?

We look at the network capture and determine if there are any resources in the API that should be protected but aren't.

Phase 2: Static Analysis

Is the application debuggable?

If an application is debuggable can be determined by looking at the manifest file. There is a property for "debuggable" in the <application> xml tag.

What 3rd party libraries and frameworks are in use? Is the application built using a cross-platform framework?

Looking at the source code we can determine which libraries and frameworks are included or used in to build the application. This information can be used to look for known vulnerabilities in public exploit databases such as CVE (http://cve.mitre.org/) or exploitDB (http://www.exploit-db.com/).

Cross platform frameworks like Phonegap (http://phonegap.com/) and Corona (http://www.coronalabs.com/ and Adobe AIR can be used to create applications that compile to several platforms. This might be detectable in the source code.

Does the application check for rooted devices? How?

One possible way for an application to check for a rooted device is to check the result of:

```
Runtime.exec("su")
```

There are probably other ways to do this, but this should be evident during code inspection.

Does the application overrequest permissions?

Try to match permissions requested with actual functionality to identify permission overrequesting. Common sense is often enough to determine if an application is overrequesting permissions at least in the extreme cases. If the overrequesting is a matter of one or two permissions that are requested but not needed, then it's harder to manually detect this.

Are there hard-coded secrets within the application source code?

API keys, credentials, proprietary business logic etc. is what we are looking for here. Manual inspection of the source code and automatic search for a compiled list of keywords such as: "username", "password" and so on is done to find such hard-coded secrets.

How does the application handle multiple failed attempts?

For each application that have authentication, try logging in with faulty credentials and observe how the application handles this. Failing to take some sort of measure against a bruteforce attack on passwords and usernames is a definite vulnerability.

Are failed attempts logged?

This question is hard to answer unless the application gives some sort of user feedback indicating that the login attempts have been logged or that the account is temporarily deactivated after a certain number of attempts.

Are there different roles within the application?

This can typically be observed by using the application and looking for an Administrator login or some other privilege escalating functionality. Different roles should also be evident in the source code if existent.

Review file permissions of files created at run-time

By inspecting the */data/data/some.application.package* folder on the rooted device we can inspect the permissions of the files that an application creates. Also, by searching for `MODE_WORLD_WRITABLE` in the source code we will find if an application writes publicly readable files.

Can licensing checks be defeated locally?

Since we are looking at free applications it is possible that some applications that also have a paid version allows the user to upgrade the current application. This means there may be vulnerabilities that can be exploited to obtain the licensed version without paying for it. By looking at the way such a license upgrade is done in the source code, we can reveal such flaws.

Phase 3: Dynamic Analysis

Intent Fuzzing

- Identify exported activities, services and broadcast receivers.
- Look for unprotected components where intents can be injected.

Network Packet Tampering

- Using Burpsuite Proxy, monitor the live packets leaving and entering the device associated with the application under examination.
- Tamper with parameters to reveal logical flaws or find parameters that are not validated by the client application
- Look for URL parameters or data that can be altered to change the behavior or results of the network communication. For instance, change the game data that a game receives from the server.

A.4 Vulnerability disclosure

Find 1

An email was sent to games@lotum.de on June 29 explaining the vulnerability. The email address is listed on Google Play as the developers' email. No response to the email was received.

Find 2

An email was sent to info@appoh.nl on September 26 explaining the vulnerability. There has been no response at the time of writing.

Find 3

No email was sent regarding this find because the vulnerability is not exploitable any longer.

Find 4 & 5

An email was sent to support@wordfeud.no on June 26 explaining the vulnerabilities. No response was received.

Find 6 & 7

An email was sent to quizbattle@quizboard.com on August 6 explaining the vulnerability. A response was given on August 8 expressing gratitude for our interest in their game and saying that they were working to fix the problem.

Find 8

An email was sent to android-support@instagram.com explaining the vulnerability on September 26. There has been no response at the time of writing.

Bibliography

[1] Apkdownloader. http://apps.evozi.com/apk-downloader/ accessed 19.05.2013.

[2] A custom aftermarket firmware distribution. http://www.cyanogenmod.org/ accessed 31.05.2013.

[3] Android apk tool. https://code.google.com/p/android-apktool/ accessed 19.05.2013, 2010.

[4] Android. Security enhancements in android 4.2. http://source.android.com/devices/tech/security/enhancements.html accessed 19.07.2013.

[5] Android.com. Android developers. http://www.android.com/ accessed 05.06.2013, June 2013.

[6] appmtr.com. Hardest game ever 2 usage since launch. http://www.appmtr.com/facebook/app/142720442570612-hardest-game-ever-2/ accessed 30.07.2013.

[7] M.A. Bishop. *Introduction to computer security*. Addison-Wesley, 2005.

[8] Colt's Blog. Android os - processes and the zygote! http://coltf.blogspot.no/p/android-os-processes-and-zygote.html accessed 07.08.2013.

[9] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[10] Microsoft Security Response Center. Definition of a security vulnerability. http://technet.microsoft.com/en-us/library/cc751383.aspx accessed 20.07.2013.

[11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[12] Roger Clarke. Categories of malware. http://www.rogerclarke.com/II/MalCat-0909.html accessed 06.03.2013, 2009.

[13] Gerald Combs. Wireshark network tool. https://www.wireshark.org/ accessed 06.07.2013.

[14] Somesh Jha Damien Octeau, Patrick McDaniel. Dare: Dalvik retargeting. http://siis.cse.psu.edu/dare/index.html accessed 24.06.2013.

[15] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. *Information Security*, pages 346–360, 2011.

[16] Brian Donohue. Samsung acknowledges exynos root exploit. https://threatpost.com/samsung-acknowledges-exynos-root-exploit-122012/ accessed 30.05.2013, 2012.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[18] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX security symposium*, volume 2011, 2011.

[19] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[20] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *Security & Privacy, IEEE*, 7(1):50–57, 2009.

[21] Tyrone Erasmus. The Heavy Metal That Poisoned The Droid. Technical report, MWR InfoSecurity, 03 2012.

[22] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[23] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[24] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[25] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, volume 18, pages 19–31, 2011.

[26] Jeff Forristal. Uncovering android master key that makes 99% of devices vulnerable. http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/ accessed 13.08.2013.

[27] Gartner. Gartner says asia/pacific led worldwide mobile phone sales to growth in first quarter of 2013. http://www.gartner.com/newsroom/id/2482816 accessed 05.06.2013, June 2013.

[28] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[29] Sebastian Höbarth and Rene Mayrhofer. A framework for on-device privilege escalation exploit execution on android. *Proceedings of IWSSI/SPMU*, 2011.

[30] Andrew Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Access Online via Elsevier, 2011.

[31] Xuxian Jiang. Security alert: New android malware – golddream – found in alternative app markets. http://www.cs.ncsu.edu/faculty/jiang/GoldDream/ accessed 01.05.2013, 2011.

[32] Naresh Kumar and Muhammad Ehtsham Ul Haq. Penetration testing of android-based smartphones. 2011.

[33] ESET Latin America's Lab. Trends for 2013: Astounding growth for mobile malware. Technical report, ESET, 12 2012.

[34] Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.no/2012/02/android-and-security.html accessed 12.06.2013, February 2012.

[35] Lookout. Lookout's take on the 'apperhand' sdk (aka 'android.counterclank'). https://blog.lookout.com/blog/2012/01/27/lookout%E2%80%99s-take-on-the-%E2%80%98apperhand%E2%80%99-sdk-aka-android-counterclank/ accessed 11.07.2013.

[36] Moxie Marlinspike. Your app shouldn't suffer ssl's problems. http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/ accessed 08.08.2013.

[37] Pascal Meunier. Classes of vulnerabilities and attacks. *Wiley Handbook of Science and Technology for Homeland Security*, 2008.

[38] Microsoft. The stride threat model. http://msdn.microsoft.com/en-US/library/ee823878(v=CS.20).aspx accessed 03.08.2013.

[39] Charlie Miller. Exploring the nfc attack surface, 2012.

[40] Keith W Miller, Jeffrey Voas, and George F Hurlburt. Byod: Security and privacy considerations. *IT Professional*, 14(5):53–55, 2012.

[41] Contagio mobile malware mini dump. http://contagiominidump.blogspot.no/ accessed 17.04.2013, 2013.

[42] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[43] Julianne Pepitone. Ap hack proves twitter has a serious cybersecurity problem. http://money.cnn.com/2013/04/23/technology/security/ap-twitter-hacked/index.html accessed 04.07.2013.

[44] Android Open Source Project. Android security overview. http://source.android.com/tech/security/ accessed 04.02.2013, 2012.

[45] OWASP Mobile Security Project. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project accessed 25.05.2013, 2010.

[46] Anne Saita. Google advocates 7-day deadline to publicize critical vulnerabilities. http://threatpost.com/google-advocates-7-day-deadline-to-publicize-critical-vulnerabiliti accessed 30.05.2013, May 2013.

[47] Karen Scarfone, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. Technical guide to information security testing and assessment. *NIST Special Publication*, 800:115, 2008.

[48] Duo Security. Early results from x-ray: Over 50% of android devices are vulnerable. https://blog.duosecurity.com/2012/09/early-results-from-x-ray-over-50-of-android-devices-are-vulnerable/ accessed 12.06.2013, September 2012.

[49] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[50] Tim Strazzere. Security alert: Malware found targeting custom roms (jsmshider). https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/ accessed 01.05.2013, 2011.

[51] Symantec. Android.counterclank. http://www.symantec.com/security_response/writeup.jsp?docid=2012-012709-4046-99 accessed 11.07.2013.

[52] Symantec. Symantec report on the underground economy. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf accessed 18.07.2013.

[53] Irfan Asrar Symantec. Android.counterclank found in official market. http://www.symantec.com/connect/blogs/androidcounterclank-found-official-android-market accessed 11.07.2013.

[54] Roman Unuchek. The most sophisticated android trojan. http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan accessed 14.06.2013, June 2013.

[55] Craig Leres Van Jacobson and Steven McCanne. tcpdump. http://www.tcpdump.org/ accessed 06.07.2013.

[56] John Viega, Tadayoshi Kohno, and Bruce Potter. Trust (and mistrust) in secure applications. *Communications of the ACM*, 44(2):31–36, 2001.

[57] Android Developer Website. Android developers. https://developer.android.com/ accessed 31.05.2013.

[58] Wikipedia. Computer virus. https://en.wikipedia.org/wiki/Trojan_horse_(computing) accessed 19.07.2013.

[59] Wikipedia. Computer worm. https://en.wikipedia.org/wiki/Computer_worm accessed 19.07.2013.

[60] Wikipedia. Rootkit. https://en.wikipedia.org/wiki/Rootkit accessed 19.07.2013.

[61] Wikipedia. Spyware. https://en.wikipedia.org/wiki/Spyware accessed 03.08.2013.

[62] Wikipedia. Trojan horse. https://en.wikipedia.org/wiki/Trojan_horse_(computing) accessed 19.07.2013.

[63] XDADevelopers. Root exploit on exynos. http://forum.xda-developers.com/showthread.php?p=35469999 accessed 30.05.2013, 2012.

[64] XDIN. The system server in android. http://www.androidenea.com/2009/07/system-server-in-android.html accessed 07.08.2013.

[65] Robert K Yin. Applications of case study research (applied social research methods series volume 34), 1993.

[66] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95 –109, may 2012.