Cross-Platform Evaluation of Mobile App Hardening

Benjamin Adolphi

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2012

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Cross-Platform Evaluation of Mobile App Hardening

Benjamin Adolphi

2012/07/01

Abstract

In recent years, mobile devices have become more and more popular. In 2011, for the first time, more smart-phones than PCs were sold. This has also been realized by malware authors who have made great progress in developing malware for mobile platforms. At the same time, vendors of security solutions are limited by the restrictive security architecture of the mobile platforms. This master thesis will investigate in what way, these security solutions are limited and to what degree, the restrictive security measures apply to malware. One potential way for a security solution to overcome the restrictions of the platforms is not to protect the whole platform, but only processes that deal with sensitive information. This is referred to as *app hardening* and will be investigated concerning its applicability in a mobile environment.

Acknowledgments

This master thesis would not have been possible without the support of many individuals. First, I would like to thank my supervisor Prof. Hanno Langweg for pushing me to do the best work possible. I would also like to thank my colleagues at Promon and especially my external supervisor Tom Lysemose for many productive discussions and for providing me with insight in Promon's app hardening solution. Also many thanks to my opponent, fellow student and neighbor Svein Engen for his good feedback and many fruitful discussions. Last but not least, I would like to thank all my family and friends for keeping me motivated to finish my work.

Benjamin Adolphi, 1st of July 2012

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we will provide a short introduction to the topics covered in this thesis. We will describe problems we will investigate, explain the motivation behind the thesis and present the research questions we will answer. In the end, we will state claimed contributions and present the structure of the remainder of the thesis.

1.1 Topic covered by Project

In an attempt to improve the security of their platforms, vendors of mobile platforms have introduced more restrictive access to system resources. This presents a problem for protection mechanism of traditional desktop platforms, which try to protect the entire system and therefore require access to a wide range of system resources.

The goal of this master thesis is to investigate another kind of protection mechanism called app hardening with respect to the deployment on mobile platforms. App hardening is a protection mechanism that does not try to prevent a system from being infected. It operates under the assumption that the system is already infected. Its goal is to enable a user to securely deal with sensitive information, even if the system is infected. This is achieved by limiting the protection to the program that deals with the sensitive information. This is done for example by monitoring the program and terminating it, if it behaves in an unexpected way, e.g. because malware is attempting an infection. A big advantage of app hardening is that it can detect malware without knowing of its existence, because it does not look for specific kinds of malware. App hardening approaches look for behavior of applications that is not expected on an uninfected system. Because of that, it is therefore able to detect previously unknown malware. Another aspect of app hardening is the binding between the security module and the application. This means that the application to be protected should only launch when the security module is present (Tom Lysemose Hansen, Promon AS, personal communication, February 10th, 2012).

1.2 Keywords

The keywords presented below were taken from the ACM Taxonomy [1]:

Security and Privacy Protection, Invasive Software, Access Controls, Security Kernels, Operating Systems, Mobile Environments

1.3 Problem Description

In an attempt to increase security for their users, many popular mobile platforms restrict the capabilities of applications by use of so called sandboxes. The idea is to make it more difficult for malware authors to create malware with enough capabilities to be useful. With malware becoming more and more advanced on mobile platforms, the need for more sophisticated anti-malware tools equally increases. However, by trying to restrict what malware can do, mobile platform vendors equally restrict those anti-malware tools. The problem here is that the software enforcing the sandbox will always contain bugs, so there will always be a way for a clever malware author to circumvent it. This causes the problem that while malware authors are only limited by their creativity to circumvent the sandboxes, serious anti-malware tools should play by the rules.

Traditional anti-malware tools need system-wide access to resources like the filesystem or the network interface in case of an anti-virus program or a firewall, respectively. This is, however, not provided to programs on all mobile platforms. In order to get a better understanding of these restrictions, the master thesis will investigate to what extent malware and anti-malware tools are restricted.

Apart from mobile platform vendors changing their policies, one way to deal with these restrictions is not to protect the entire platform, but only critical applications like e.g. online-banking. This approach is referred to as app hardening. One example of an already existing app hardening implementation comes from the company Promon AS[1], which has implemented app hardening on the Windows desktop platform. This master thesis shall investigate if the app hardening approach of Promon can also work on mobile platforms.

1.4 Justification, Motivation and Benefits

Mobile devices have become very popular in the recent years. This can be seen from the fact that in 2011, the number of smart-phones sold worldwide for the first time surpassed the number of PCs sold [2]. Vendors sold a total number of 472 million smart-phones world wide in 2011 and estimations are talking about 982 million smart-phones being sold in 2015 [3].

With mobile devices becoming more and more popular and at the same time increasing their functionality and power, malware authors have realized the potential of mobile platforms. This can be seen from the drastic increase of malware on these platforms. The first mobile platform malware was detected in 2004 on the Symbian OS platform [4]. In the beginning of 2012, only 8 years later, the number of known malware families on mobile platforms has gone up to more than 700 with more than 25% of these being discovered in 2011 only [5].

Apart from the threats, that can be found on desktop platforms, for example when online bank-

[1] `http://www.promon.no`

ing or electronic voting are used, mobile platforms have access to some unique interfaces that are of special interest to criminals. An example is the connection to the GSM/UMTS network, which can be misused by malware to directly create revenues for criminals by dialing premium-rate telephone numbers. GPS is another interesting interface which enables criminals to acquire detailed information about their victims' whereabouts.

The facts stated above show that there exists a great need to protect private and professional users of mobile devices from being infected by malware. This master thesis will try to improve the situation by investigating if app hardening approaches can be implemented on mobile platforms and if this will result in increased security for its users.

1.5 Research Questions

In order to investigate the above stated problems, the thesis will answer the following research questions:

1. What are the limitations of malware running on mobile platforms compared to desktop platforms?
2. What are the limitations of traditional security solutions running on mobile platforms?
3. Can app hardening solutions increase the security of mobile platforms and what are their limitations?

1.6 Claimed Contributions

In a review of current mobile malware in the wild, the authors of [6] state that application isolation on current platforms has to be strengthened to prevent malware from stealing sensitive information like user credentials and that this is a promising area of security research. Based on that, the contributions of the master thesis are to investigate in what way, this isolation in form of app hardening can be implemented on a modern mobile platform, and whether it has advantages over other security solutions. We present an overview of threats that an application could be exposed to on mobile and desktop platforms and investigate to what degree, these threats are mitigated by the individual platforms. The result is a list of unmitigated threats that require other mitigation mechanisms. We then investigated existing third party security solutions to determine if they are successful in dealing with the unmitigated threats and how they are limited by the security architecture of the platforms. Finally, we investigate methods for app hardening to deal with remaining unmitigated threats to show that app hardening can be applied on mobile platforms to increase security of applications.

1.7 Structure of Thesis

The remainder of the thesis is structured as follows: Section 2 describes the methods, we will be using to answer the three research questions. Section 3 will describe related work and the state of the art related to the research questions. In Section 4, we will answer the first research question about the possibilities and limitations of malware compared to desktop platforms. The second research question about traditional security solutions on mobile platforms will be answered in Section 5. The last research question about app hardening on mobile platforms will be answered in Section 6. Future work will be discussed in Section 7 and finally, a conclusion and summary of the results will be given in Section 8.

2 Choice of Methods

This master thesis will use a qualitative research methodology [7, p. 135] to gain in-depth knowledge about mobile platforms and app hardening to answer the research questions. This will be performed in form of a case study research design [7, p. 137], where multiple platforms and the app hardening protection mechanism are investigated and evaluated.

In order to perform the case study, we have to pick the subjects of the study. All the research questions deal with mobile and desktop platforms. We have decided to investigate three mobile and one desktop platform. The choice of the mobile platforms are Android, iOS and Windows Phone 7. The reason for choosing Android and iOS lies in their popularity. In first quarter of 2012, both platforms had a combined market share of more than 80% [8]. This will make sure, that the results of this thesis are applicable to a wide range of currently used mobile devices. The third mobile platform, Windows Phone 7, is not as popular. It was chosen for two reasons: One reason is that it has not been investigated by previous research and the other reason is that it is similar to the Windows desktop platform in that it is developed by the same manufacturer (Microsoft). Choosing the mobile and desktop platform from the same manufacturer will make it easier to see the differences in mobile and desktop platforms. Windows was chosen as the desktop platform because it is still the most popular desktop platform and also because Promon's app hardening solution is currently only implemented on Windows, so it can be considered a good reference platform to compare the mobile platforms to.

We will only give an overview on how we answered the research questions here. A detailed description can be found in Section 4, 5 and 6.

The first research question is about the difference in limitations of malware on mobile platforms compared to desktop platforms. In order to answer this question, we decided to use threat modeling based on [9] to find threats that could be manifested by malware on mobile and desktop platforms. We will then investigate these threats on each of the platforms to determine how likely it is that the threat can be manifested. We end up with a list of threats on each platform that are currently not or only partially mitigated by the selective platforms and therefore need to be dealt with by other means.

The second research question investigates the limitations of traditional security solutions like anti-virus software and firewalls on mobile platforms. This is achieved by first finding relevant security solution categories that can increase the security of a platform. We then try to find samples for these solutions for analysis by searching the official application distribution platforms

and other third party application distribution platforms, if available. To determine the functionality and level of protection, these solutions offer, reverse engineering is used. The result of this analysis will be an overview of how widespread security solutions on each of these platforms are and how sophisticated they have become. We will then use this to see if the security solutions are able to mitigate the threats found in the first research question.

After the first two research questions have been answered, we will have an understanding of the degree of which mobile platforms need protection mechanisms and how well traditional security solutions can be implemented to fulfill these needs. We then know which threat could be dealt with using app hardening. In order to investigate how app hardening could be implemented on mobile platforms and whether it could take care of the remaining threats, we will investigate possible implementations of app hardening approaches on the mobile platforms.

3 State of the Art

This chapter will give an overview of the state of the art related to the three research questions. First, an overview of the investigated platforms is given. Then, the protection mechanisms of the different platforms are described. After that, we discuss related work concerning malware on mobile platforms and third party security solutions. Finally, we present related work on app hardening.

3.1 Overview of investigated Platforms

This section will give an introduction of the platforms that are to be investigated in this thesis, namely Android, iOS, Windows Phone 7 and Windows 7. It will furthermore discuss the architecture of each of these platforms, how applications are developed and how they are deployed.

3.1.1 Introduction

In this section, an overview of the platforms is presented. Apart from a quick look at the history, some details about licensing and supported hardware platforms is given.

Android

Android is a mobile platform that is developed by the Open Handset Alliance. The Open Handset Alliance (OHA) was founded in November 2007 by a group of IT companies headed by Google [10, p. 17]. At the same time, the OHA announced their first project: Android.The operating system kernel is an adopted version of Linux, which is publicly available under the GPL 2 license. Most of the other Android components are open source as well and are distributed under an Apache license. Only some special applications like the Maps application and the Google Play Store are closed source [11]. Android currently runs on a multitude of Smartphones, Tablet PCs and also some Netbooks which run on ARM or x86 processors [12, p. 10] made by many different manufacturers. At the time of writing, the current stable Android version is 4.0.3, called *Ice Cream Sandwich*.

iOS

iOS is a mobile platform developed by Apple for mobile devices. It was first introduced in June 2007 as operating system for the iPhone [13]. iOS is based on Apple's desktop operating system Mac OS X. It shares the same basic operating system Darwin with iOS, which includes some basic libraries and the XNU kernel, which is a hybrid kernel based on a Mach kernel and some components from FreeBSD and NetBSD [12, p. 8]. Apart from Darwin, which is licensed under the Apple Public Source License, iOS is closed source. It currently runs only on devices developed by Apple which includes the iPhone, iPad and iPod which all run on ARM processors [12, p. 10]. At the time of writing, the current stable version is iOS 5.1.1.

Windows Phone 7

Windows Phone 7 is Microsoft's platform for mobile devices. It is based on previous mobile operating systems from Microsoft starting with Pocket PC 2000, which was released in 2000. A later incarnation was Windows Mobile, which was first released in 2003 and it's latest version Windows Mobile 6.5 was released in 2009. These operating systems were mostly targeting business customers, but failed to address the needs of private customers as Android and iOS did. To make up for that shortcoming, Microsoft released a newly designed and renamed version of Windows Mobile under the name Windows Phone 7 in October 2010. As with the previous operating systems, Windows Phone 7 runs a Windows Compact Edition (CE) kernel [12, p. 1ff.]. Like iOS, Windows Phone 7 is closed source. It was developed to run on ARM processors of devices from different manufacturers and is currently only deployed on smartphones. Windows Phone 7's latest version is 7.5, which is called *Mango*.

Windows 7

Windows 7 is a desktop operating system created by Microsoft, which was released in October 2009. It is based on a long line of Windows versions starting with Windows 1.0 in 1985. It is based on a closed source Windows NT kernel and currently only runs on x86 processors on PCs and some tablet PCs [14].

3.1.2 Architecture

To get a better understanding on how the different platforms work internally, this section presents an overview of the different operating system architectures.

Android

The Android architecture is depicted in Figure 1 (adopted from [12, p. 9]). At its core, it currently runs a Linux 2.6 kernel, that is adopted to run on mobile hardware, by inclusion of necessary drivers for devices like keypads and cameras. It also includes some other extensions like the Binder driver which enables interprocess communication in Android [15] and some hooks for the Android security framework. On top of the kernel are system libraries that provide commonly used functionality like a relational database (*SQLite*), encryption (*OpenSSL*), HTML rendering (*Webkit*) and a standard C library (*libc*) among others. These libraries are all written in native C/C++ [16]. Android applications are however primarily written in the Java programming language. To execute these applications, Android contains a Java Virtual Machine (JVM) called Dalvik. Dalvik runs applications in the Dalvik EXecutable (DEX) format, which is a format different and incompatible to traditional Java byte code implementations [17] and is optimized to run on mobile platforms. On top of that, Android provides an Application Framework that offers developers all the necessary functionality to create an Android application. This Framework is written in Java and provides functionality for example for creating a user interface (*WindowManager*) or accessing the device's location (*LocationManager*) [10, p. 28]. At the highest layer of the Android architecture are the applications that are developed in Java having access to all the functionality provided by the lower layers. This includes applications that are shipped with Android like the *Contacts* or *Phone* applications.

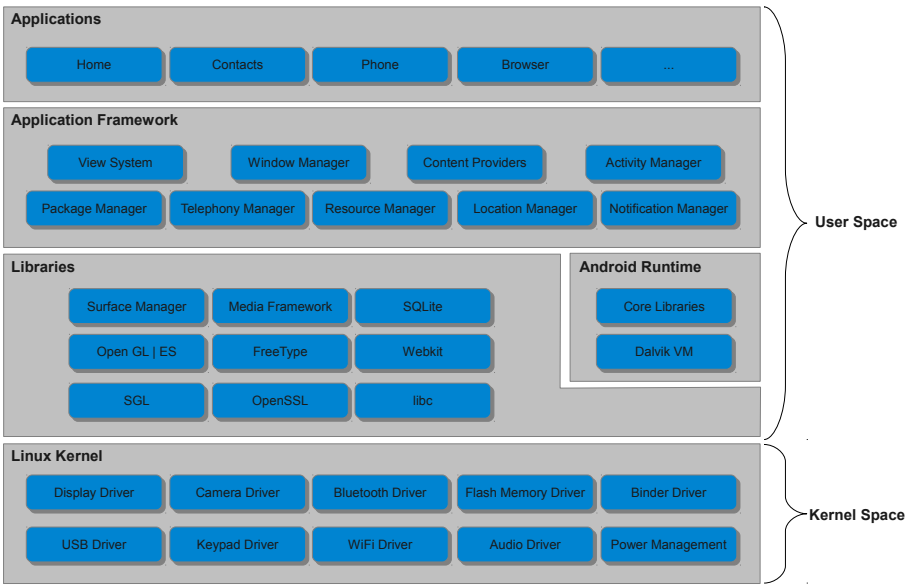

Figure 1: The architecture of Android

iOS

An overview of the iOS architecture is given in Figure 2 (based on [18]). The basis of iOS is a XNU kernel that provides basic hardware abstraction and other operating system services like virtual memory, threads, networking, etc. Direct access to the kernel is restricted to only a few libraries and frameworks and is not allowed from normal applications. To access kernel features like POSIX threads or BSD sockets, developers can use the *LibSystem* library. The lowest layer in user space is the *Core OS* layer. This layer provides low level functionality like accessing external accessories (*External Accessory Framework*), accelerated math and signal processing computations (*Accelerate Framework*) and cryptography (*Security Framework*). On top of the Core OS layer builds the *Core Services* layer which provides basic services that common applications use. It includes frameworks to work with network protocols (*CFNetwork*), the device's location (*Core Location*), access to phone based information (*Core Telephony*) and others. The *Media* layer contains frameworks for graphics, audio and video. This includes frameworks for hardware-accelerated graphics rendering (*OpenGL ES*), reading and writing images (*Image I/O*), handling various audio formats (*Core Audio*) and working with video content (*AV Foundation*). Most applications are mainly using the frameworks of the *Cocoa Touch* layer which provides basic building blocks for applications. Developers are however not restricted in case they want to use lower level frameworks. The most important framework of the Cocoa touch layer is the *UIKit* framework which provides functionality to create event driven applications like user interface controls, multitasking, handling input events and push notifications [18].

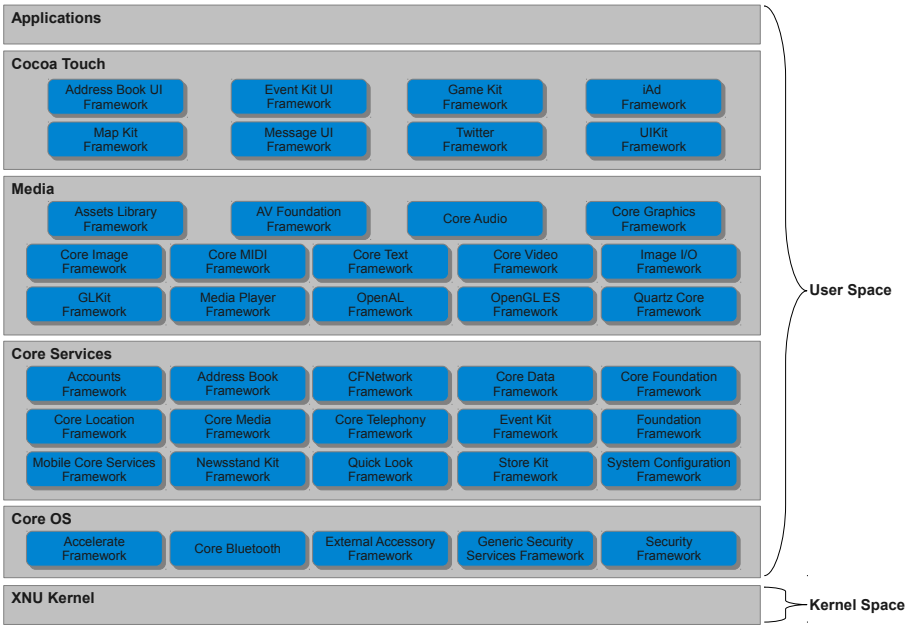

Figure 2: The architecture of iOS

Windows Phone 7

Figure 3 (based on [12, p. 5f.]) shows the architecture of Windows Phone 7. The interface to the hardware and other basic operating system functionality is provided by a Windows CE kernel in version 6. On the user space side, Windows Phone 7 only allows applications to be written using their .NET Framework which means that Microsoft does not allow native applications for example using the Win32 API as in earlier Windows Mobile versions. The .NET Framework provides applications with a runtime environment, the *common language runtime (CLR)*, which executes applications written in a language supported by .NET. The Framework also provides developers with a rich class library for common tasks like creating user interfaces, network communication, threading, etc. On top of that, applications can either be developed based on Silverlight or XNA. Using the .NET framework directly is not possible. Silverlight is an extension to .NET which is specialized to provide a rich user interface as well as extended multimedia animation features. XNA is another extension to .NET which provides advanced game development features and was originally developed for Microsoft's Xbox gaming platform.

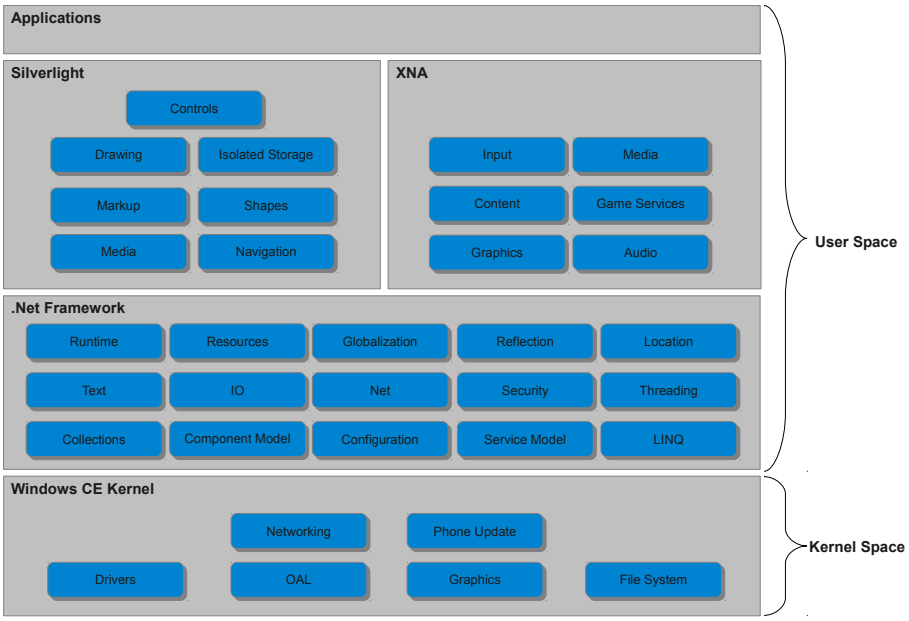

Figure 3: The architecture of Windows Phone 7

Windows 7

The architecture of Windows 7 is shown on Figure 4 (based on [19] and [20, p. 50ff.]). At its core, Windows runs an NT kernel in version 6.1. The kernel only provides basic operating system functionality like scheduling, I/O services and management of resources. Other functionality in kernel space like device drivers, networking, filesystems and event handling and other user interface operations (*Win32k*) are implemented as modules that can be dynamically loaded into

the kernel in case they are needed [19]. On the user space side, the *Ntdll* library provides applications with access to system calls in kernel mode. It furthermore contains functionality to load dynamic libraries [20, p. 57f.]. On top of the Ntdll builds the *Win32 API*. The Win32 API is a large collection of DLLs that provide applications with basic functionality such as access to kernel interfaces (*Kernel32*), user interfaces (*User32* and *GDI32*) and many more [19]. Applications are then built on top of the Win32 API. They can either access the Win32 API directly or they can have an intermediate runtime environment like *CLR* and *JRE* to execute .NET and Java applications [20, p. 3]. Apart from normal applications, Windows 7 also offers so called *Services*. They implement basic system functionality outside the kernel space. Examples of such services are *Smss* which is responsible for starting Windows and the Window subsystem (*Csrss*) [20, p. 74ff.].

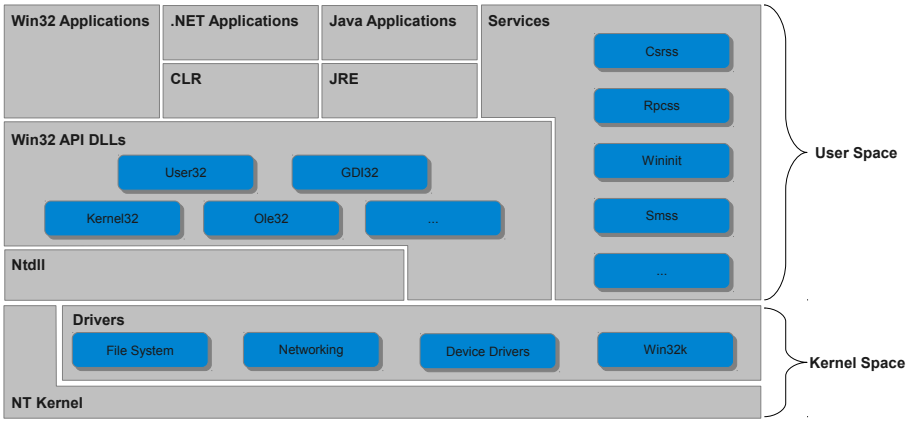

Figure 4: The architecture of Windows 7

3.1.3 Application Development

This section discusses the different ways how applications are developed for each of the platforms.

Android

To develop applications for Android, the OHA offers a Software Development Kit (SDK) which includes debugging and testing tools, documentation, examples, source code, an emulator, the application framework and other helpful tools to develop Android applications. The officially supported integrated development environment (IDE) for Android is Eclipse and the SDK includes an Eclipse plugin for easy integration[10, p. 33]. Supported development platforms are Windows, Mac OS and Linux [21, p. 17]. Debugging Android applications is done through the Android Debug Bridge (ADB) which is a daemon, that can be run on a device or inside an emulator.

Apart from writing the whole application in Java, Android provides the so called *Native Development Kit (NDK)*. Using the NDK, developers are able to develop parts of their application in native code as a library that can then be called from Java using the *Java Native Interface (JNI)* [16]. The reason for that is mostly to increase performance. These libraries are loaded into the context of the same Dalvik process, that the Java application that calls the native code runs in [22].

As mentioned before, Android applications are developed in Java and then compiled into a special byte code (DEX) that Dalvik understands. This DEX code together with other resources that are needed by the application is put into an archive which has an *apk* extension. These files can then be used to deploy the application on a device [22]. Each of these files contains a manifest file that provides meta information about the application [23].

Android applications are not developed as a traditional monolithic application. Instead they consists of instances of four kinds of components [23]:

- *Activities* are used to display a user interface.
- *Services* are used to perform processing in the background.
- *Content providers* are used to store and retrieve data.
- *Broadcast receivers* are used to receive broadcast messages.

These components are declared in the application's manifest file. They can be public or private depending on whether they can be called by other applications or not and can declare that specific permissions are required to access them (see Section 3.2.2).

Inter-component communication (ICC) in Android is performed using so called *intents*. Intents are an asynchronous message exchange mechanism that can be performed between two components of the same process or two components of a different process, they are therefore also an inter-process communication (IPC) mechanism. Intents can be explicit and implicit. Explicit intents are sent to a specific component while implicit intents are sent to the messaging system which then determines the best suited component to launch [24]. Implicit intents make the system very flexible, an example would be a device which has multiple applications installed to display a map. By sending an implicit intent, no predefined application is launched, instead the user is able to choose between installed applications.

The process of an intent causing a component to execute code is called an *action* in Android [23] and the kind of action to be performed is specified by the *action string*, that is included in the intent. Other information that can be included in an intent is additional data, a category and a field for extra information for the specific intent [24]. Also, intents can be directed to a single component or broadcasted to a group of receivers. In order to be able to receive an

intent, a component has to declare that it is willing to process it. This is done in the manifest file of the application by so called *intent filters*. An example for the use of intent filters is that Android applications do not have a *main()* method that traditional applications have. Instead, an application has to define an intent filter for receiving a *MAIN* intent when the application is launched.

The ICC mechanism presented above only works between activities, services and broadcast receivers. In order to access and manipulate data of a content provider, data is addressed by use of a special URI that each content provider has to declare. This URI is used to deal with data using an SQL like interface.

iOS

iOS applications are typically developed using the iOS Software Development Kit (SDK) which includes an IDE called Xcode, a simulator, examples and documentation. The SDK is only available for developers using the Mac OS X platform [25, p. 1ff.].

Applications are developed in Objective-C 2.0, which is an object oriented extension of the C programming language. This means that in iOS, applications are written in native code without an intermediate runtime environment. iOS includes many Frameworks. Apple distinguishes between public and private Frameworks. While public frameworks are documented and meant to be used by third party developers, private frameworks are only to be used by Apple since they might change rapidly and might be security critical [25, p. 10]. An example for such a private framework is the *IOKit* framework which provides access to low level hardware interfaces. Third party applications will not be allowed in the App Store if those private frameworks are used [25, p. 34], [26].

Applications are packaged in application bundles which are simple ZIP files with an *ipa* extension and contain the executable and other resources used by the application.

iOS offers two ways to share data between applications: Using custom URL schemes and using custom file type handlers. Custom URL schemes are used to launch an application whenever a URL with a specific URL prefix is opened [27]. Custom file type handlers are used to launch an application when a file of a specific type is accessed. These communication methods are declared in the manifest file of the application called *Info.plist* which provides meta information about the application [25, p. 303].

Windows Phone 7

To develop applications for Windows Phone 7, Microsoft offers the *Windows Phone Developer Tools*. This includes the Visual Studio IDE, the .NET, Silverlight and XNA Frameworks, the Win-

dows Phone emulator, samples and development tools. Microsoft offers these tools only for Windows Vista and Windows 7, which means that developing for Windows Phone 7 is not possible on older Windows versions or even other platforms like Mac OS or Linux. Using the Windows Phone emulator furthermore requires the developer to possess a graphics card that at least supports DirectX 10 [12, p. 16].

As mentioned before, there are two ways to develop applications for Windows Phone 7. Traditional event driven applications that include a 2D interface are developed using the Silverlight Framework. While games and graphic intensive applications that need GPU acceleration are developed using XNA. Applications are written in either C# or Visual Basic. Native programming languages are not supported.

Finished applications are packaged in a *Silverlight Application Package*, which is a ZIP archive with the *xap* extension that contains .NET assemblies and other resources needed by the application.

As of now, Windows Phone 7 does not offer a way to perform interprocess communication or other ways to exchange data between applications on the device. The only exception are so called *choosers* which provide access to a limited amount of shared phone resources like the picture library. All other interprocess communication has to be done using cloud- or web-services [12, p. 61].

Windows 7

Windows 7 does not put any restrictions on they way applications can be developed. Therefore, a wealth of possibilities exists for developers to create applications and describing all of them would be out of scope of this thesis. However, Microsoft promotes developing applications using their .NET framework as in Windows Phone 7.

3.1.4 Application Distribution

This section will discuss the way, applications are distributed on the different platforms. All the mobile platforms use a central distribution platform as the only or at least the only official way to install applications. A comparison on how these distribution platforms restrict malware from being installed is presented in [28, 29, 30]. Platforms differ in multiple ways: Some only allow installation from a vendor repository while others allow their users to install software from third party repositories over which the vendor has no control. In order to make sure, that applications comply with the security requirements, many vendors require analysis of the applications before they are allowed in the distribution platforms [31]. Vendors usually do not report their analysis procedures, but [29] presents a list of possible checks. This analysis can be done in a static and in a dynamic way and can be carried out automatically or by a human.

Because of the number of applications available at the distribution platforms, manual analysis of each application would be impossible. Symbian for example deals with this problem by only analyzing applications manually if they request dangerous privileges [6], but research is focused on automatic analysis. In a static analysis, the application's binary is analyzed, for example by searching for calls to suspicious functions [13]. Dynamic analyses observe the application while it is being executed. An example is to observe the resource consumption of the application [13] or to trace privacy violations during runtime [17]. More sophisticated methods have been proposed. A static analysis method is proposed in [32], where the authors propose a method to analyze the data flows in iOS binaries to detect information leaks. The authors of [31] propose a dynamic approach where they analyze information flow and execution paths of the running application.

Android

There are three ways to distribute applications for Android devices: Using the official Google Play Store, using a third party market or the memory card to transfer and install apk files on the device and using the debugger.

The Google Play Store is the official way to install applications and is administered by Google. It is a central distribution platform that offers applications that have been created by developers who have registered with Google as a developer before. Registering as a developer requires payment of a one time fee which is payed by a credit card. By using a credit card, Google achieves some degree of identifying a developer. The Google Play Store can either be accessed from the Google Play Store application on the device or via an online platform.

When an application is uploaded, it is published almost immediately on the Google Play Store. Google does not review applications prior to publication on the Play Store, but may do so afterwards and remove applications if they do not comply with the terms and conditions of the Play Store [6]. Instead of reviewing all application themselves, Google relies on the users to write reviews about the application and rate it in the Play Store. If many users complain about an application, Google will remove it from the Play Store [33]. Applications are not only removed from the Play Store, Android also contains functionality called a *kill switch* which can remove applications from devices after installations [29]. To increase the security in the Play Store, the authors of [34] propose to at least perform a static or dynamic analysis on JNI calls before publication in the Play Store to reduce the chance of applications breaching security by using dangerous native APIs.

Besides the official Play Store, users are also able to download and install applications from unofficial markets created by third parties. Another way to install applications is to download apk files on a desktop computer, transfer them onto a memory card and then install the files from the memory card onto the device. Before a user can install applications from third parties outside the Play Store, he or she has to explicitly activate an option in the Android settings allowing access to *Unknown sources* which warns the user about possible damages caused by

untrusted software [6]. The last possibility to install applications is to use the Android Debug Bridge, which is only intended for developers [22].

iOS

Deploying applications on an iOS device can be achieved in four ways: Through the App Store, through ad-hoc deployment, through the debugger and using enterprise deployment.

Normal users install applications through the App Store, a centralized distribution platform for iOS applications. In order to submit applications to the App Store, a membership in Apple's developer program is needed. Before applications are included in the App Store, they are reviewed by Apple and checked for inappropriate content, bugs in the application, usage of APIs that are private and other criteria [30]. Only when these checks are passed, applications will be included in the App Store. As on Android, Apple also has the possibility to delete unwanted applications after they have been deployed on devices [29].

During development, applications can also be installed on the devices for testing without the App Store either for debugging the application on a device directly using the debugger or by using ad-hoc distribution, which is a possibility that Apple offers to deploy applications for testing. Both these approaches require the unique ID (UDIDs) of the devices that the application should be deployed on, to be registered with Apple [25, p. 28].

The last possibility to deploy applications is using Apple's enterprise deployment program. This enables companies to deploy applications in-house. One way to distribute applications in that manner is using so called *over-the-air* deployment to as many users as desired without having to register UDIDs of devices. That way, users can install applications from a webpage without involving the AppStore or Apple.

Windows Phone 7

In Windows Phone 7, there are two ways to deploy applications: Using the Windows Phone Marketplace or to download them on a developer phone.

The official way to install applications on a Windows Phone 7 phone is to use the Windows Phone Marketplace offered by Microsoft. As in iOS and Android, the Windows Phone Marketplace is a centralized application distribution platform. In order to submit applications to the Marketplace, developers need to register an account at Microsoft's developer portal *AppHub*.

Before applications are published in the Windows Phone Marketplace, they are reviewed by Microsoft. This review usually takes five business days and checks the application for reliability, resource consumption and presence of malicious content.

Developers have a second way to install applications on their phones. They have to register their phone as a developer phone in AppHub and are then able to install and test their application directly on their phone without going through the Marketplace [12, p. 26].

Windows 7

As of now, Microsoft does not offer any kind of central distribution platform for Windows 7. Applications can be acquired from arbitrary locations, most of the time from the developer's webpage. Microsoft has therefore no influence on what software is installed and users have the freedom and responsibility to decide if they trust the developers and also if they trust the location, they acquired the application from. Different from the mobile platforms, Windows 7 therefore also has no centralized way to update applications. Users have to check the webpage of the developers individually to install updates [15, p. 64f.].

3.2 Protection Mechanisms of investigated Platforms

This section will describe protection mechanisms that the different platforms offer, which problems these mechanisms have and how they can be improved. The description will focus on protection mechanisms that are relevant for this thesis, namely sandboxing, permissions and code signing. Other protection mechanisms like encryption and memory protection will not be discussed. From the presented research on these protection mechanisms, it can be seen that most research has been done concerning Android, which is probably due to the fact that Android is by far the most open platform [35].

Mobile platforms usually do not provide full access to the system. This is due to security reasons, since it would allow bypassing the above mentioned protection mechanisms. Bugs in operating system services running with administrative privileges can however be exploited to acquire these permissions. This is used today extensively by users to perform so called jailbreaking or rooting of their device because it enables users to customize their operating system and run applications that are not sanctioned by the manufacturers. This process is usually done two ways: either the original software on the device is patched or new firmware is flashed on the device. However, [6] points out that the exploits used in the jailbreaking or rooting process also make it very easy for malware authors since they do not have to find the vulnerabilities themselves. The authors also found that for the Android devices they investigated, at least 74% of the time, an unpatched privilege escalation vulnerability enabling root access was available.

3.2.1 Sandboxing

A sandbox is defined as "an environment in which the actions of a process are restricted according to a security policy" [36, p. 444]. They are usually implemented "by adding extra security

checking mechanisms to the libraries or kernel" [36, p. 444]. The objective of using sandboxes are to implement the principle of least privilege: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job." [37]. Sandboxes are used in mobile platforms as a mandatory access control (MAC) mechanism to prevent one application from influencing other applications by process and filesystem access isolation. Furthermore, they are used to restrict the access to system resources to a minimum required for applications to run [38]. In case, an application needs elevated or extended permissions, it can leave the sandbox using permissions [29] (see Section 3.2.2).

Android

Android inherits POSIX permissions from the underlying Linux kernel [29]. The POSIX permission system assigns an unique user id to each user. Furthermore, a user can belong to multiple groups, which are identified by group ids.

This mechanism is usually used on a filesystem level. But it is also used for other purposes, e.g. as access control mechanism for system calls. Each file is assigned an owner (user id) and a group (group id). Based on that, permissions for reading, writing and executing can be assigned to the file for the owner, the group and for everyone else. Group permissions can furthermore be used to assign a group of users access to files or to system resources like devices which are represented by device files [39]. The same way, system files like configuration files and libraries are protected. These files are owned by the *root* or *system* user and therefore not manipulatable for normal applications [16]. When an application creates a new file, by default, it is only accessible to the application itself [40]. It is however possible to change these permissions for files that the application owns [41], which makes the POSIX mechanism a discretionary access control (DAC) mechanism.

This approach is however dependent on filesystems that include POSIX permissions on file metadata. This is a problem with devices that have memory card slots to use memory cards that are typically formatted with the FAT32 filesystem that does not support POSIX permissions [29]. For that reason old Android versions did not allow to install applications on external memory cards since the POSIX permission sandbox will not work there. Since Android 2.2, the sandbox on the memory card is enforced by encrypting installed applications [15].

The same mechanism is also used to separate processes of different users on a multi-user system. Android uses this mechanism in a new way to create a sandbox. First, every application is given its own user and group id [39] and is installed in its own folder which only the application has access to. That way, applications are isolated from each other. Developers who want their applications to share data directly have the possibility to request the same user and group id for multiple applications. This requires the applications to be signed by the same developer key [22] (see Section 3.2.3). Second, it is also used to control access to system resources by membership in a group (see Section 3.2.2).

The sandbox in Android is enforced at kernel level [15]. Than means that there is no difference if the application runs in a Java VM or if the code is native [41]. The Dalvik VM therefore is not used to sandbox applications as is the case of Java on desktop platforms [15].

Problems with the Android approach arise with the existence of an all powerful root user that has permission to do anything. This is required for services that need extensive access to system resources that can not all be provided by permissions. One example for Android is the *zygote* process which is used to start Java applications on Android [42]. Having services run with root privileges opens the door for attackers compromising the sandbox mechanism since applications running with root privileges can simply circumvent the sandbox. For that reason, researchers have proposed the use of Linux Security Modules (LSMs) such as SELinux and AppArmor [40, 43]. LSMs are hooks added to the Linux kernel to implement security policies independent on the actual user. More details on SELinux and its implementation on Android by the National Security Agency (NSA) can be found in [40, 42].

iOS

In iOS, sandboxing is achieved by the *Apple XNU Sandbox framework*, which is a module of the *TrustedBSD framework*, that is used in Mac OS X. It provides a sandbox, that can be configured for each process separately using policy files [44, p. 133]. The TrustedBSD framework contains hooks for system calls in the XNU kernel. Based on the policy defined for a process, system calls are either allowed or denied by the Apple XNU Sandbox framework. The policy files are based on regular expressions and can be used for example to disallow other applications access to an application's installation folder or configuration files, but also to restrict access to system resources [13]. All third party applications in iOS are running identical sandbox configurations while pre-installed applications and services can have different configurations. The effectiveness of this mechanism, however, depends on the policy files. In [13], many privacy problems due to insufficient sandboxing rules are presented. The authors for example show how a user's location can be estimated by reading unprotected settings files of applications like the weather application. There is no public documentation on how sandboxes on iOS work and can be used, but the authors of [44] have reverse engineered the most part and provide a detailed description. Even though iOS, being based on UNIX as well, also supports POSIX permissions, the sandbox does not depend on these permissions and every application runs as the same user *mobile* [33]. Compared to Android where every application is isolated by having a different user id, all iOS applications running as the same user can be compromised if a vulnerability in one of these applications is found [28]. iOS does not has the same problem with processes running as root users since the applications are still sandboxed. If a vulnerability in a process running as root is exploited, an attacker has to first break out of the sandbox to access system resources. However, not all processes are sandboxed in iOS. This can for example be seen from the first jailbreak for iOS where the process that listens for SMS messages and was not sandboxed contained a vulnerability that could be exploited to gain unconditional access to the system [33].

Windows Phone 7

Like Android and iOS, Windows Phone 7 also has a mechanism to sandbox applications. This includes the mechanism of isolated storage which completely prevents an application to access the filesystem outside the application's installation directory. In order to determine the degree of sandboxing of an application, Windows Phone 7 defines four different chambers in which applications can be executed [12]:

- The *Trusted Computing Base*, is the highest security level and has access to almost everything. This is the chamber the kernel runs in.
- Applications running in the *Elevated Rights Chamber* have access to everything except the security policy of the system.
- In the *Standard Rights Chamber*, all native applications except the ones which require special permissions are executed. Note that only applications that are included in the initial firmware can be native.
- The *Least Privilege Chamber* initially offers only the basic permissions needed to run an application. This is the chamber that third party applications run in. The permissions of this chamber are variable, which means that by using capabilities (see Section 3.2.2), an application can gain permissions equal to those of the Elevated Rights Chamber.

Microsoft does not offer any details on the exact implementation of this sandboxing mechanism and we could not find any third party documentation on it so we are not able to describe at what level the sandboxing applies and which permissions are offered to applications in which security chamber.

Windows 7

Windows 7 contains a sandboxing mechanism as well. However, the reason for sandboxing in Windows 7 is different that for mobile platforms. Sandboxing was introduced not for security reasons, but to enforce the digital rights management, meaning that processes that deal with protected digital content need to be protected from access by other processes to prevent illegal copying of digital content, such as movies. This is different from the sandboxing on mobile platforms where the goal is not to protect a process, but to protect other processes from a potentially malicious process. This can be seen as a reverse sandbox. Sandboxing in Windows 7 is achieved by so called *Protected Processes*. Protected processes can however only be created if the application is signed by a special *Windows Media Certificate*. A process that uses this security feature is then protected against access from other processes, even if the process has administrative privileges. It is, however, possible to deactivate the protection by loading a kernel driver that

manipulates the process block [20, p. 346ff.]. Apart from that, many Windows applications, e.g. browsers and PDF readers have included functionality to isolate parts of their application from each other and to restrict access to system resources.

3.2.2 Permissions

As mentioned in the previous section, the sandboxing method tries to give applications the least privilege it needs to run. But some applications need more rights than the absolute minimum, for example to communicate with the Internet or to access location information from a GPS receiver. In order to leave the sandbox, mobile platforms have included a mechanism to request additional permissions if needed. Differences in platforms lie in the time these permissions are requested and how granular they are. A comparison of different mobile platforms concerning permissions can be found in [28].

Android

Android offers two different levels of permissions: Permissions for access to system resources and permissions to access other components through inter-component communication [23]. Applications request these permissions in their manifest file [16] (see Section 3.1.3).

Access to system resources is usually based on POSIX group memberships. This membership is checked in the kernel when an application wants to access a resource. If a permission is granted to an application, the application's user is added to a specific group to enable access to the resource [45]. Inter-component communication permissions are enforced by a reference monitor that enforces mandatory access control policies [16]. Android already includes many ICC permissions for Android components, but third party components also have the possibility to declare new permissions that should be required in order for them to be accesses. ICC is only possible through this reference monitor in Java, meaning that native code can not bypass this system, it has to use the JNI to call Java code if ICC is needed [45].

Android provides hundreds of permissions and these permissions are grouped into different threat levels [45, 23]:

- *Normal* permissions are permissions that can not cause harm to the user or system, but might merely be an annoyance. They are granted to applications automatically without prompting the user.
- *Dangerous* permissions are permissions that could cause harm to the user. Such permissions include sending text messages or accessing the camera. If an application requests a dangerous permission, it will be stated to the user at install time. Only if the user accepts all permissions

that the application requests, the application can be installed. It is not possible to selectively deny requested permissions. It is also not possible to later remove permissions that were given to the application at install time [46].

- *Signature* permissions are permissions that could seriously compromise the device if misused. One such permission would be to register an input method that can get informed about user input. They are granted for applications that were signed with the same key as the application that defines the permission. In the case of permissions defined by the system, this would be the key of the entity that installed the operating system on the device (typically the device manufacturer). In case the keys match, the user will not be prompted about the permission during install time. If the keys do not match, requested permissions will simply be not granted.
- *Signature or System* permissions are similar to signature permissions, they exist only for legacy compatibility. They are granted to applications that are installed in the system image of Android or that are signed by the entity that installed the system image.

Android's permission system has been widely investigated in research and solutions to found problems have been proposed. An overview of this work is given below. Even though, most of the work on permissions is on the Android permissions system, the problems and solutions are for the most part also transferable to other platforms.

This first point of criticism is, that a user will quickly become overwhelmed with having to agree to many permissions. Also, the meaning behind some of the permissions might not be clear to everyone or it might not be clear that combinations of permissions could be dangerous. One example is that read access to the address book is only dangerous if the application has an additional permission to access the Internet. Because of these problems, most users will just agree to grant them [46]. The authors of [47] have developed a tool called *Kirin* that extracts the permissions that an application requests from its manifest file and then checks them against a security policy that defines security goals of a stakeholder. That way, also combinations of permissions that do not comply with the security policy are detected and the user is not overwhelmed with finding out if the requested permissions are dangerous. The authors of [46] have analyzed permission usage of around 1000 Android applications to determine whether the kind and amount of used permissions can be used to detect malicious applications and how the permission usage differs between payed and free applications. They found that combinations of permissions give good detection results.

Even though Android includes well over 100 permissions, the granularity of some permissions is criticized. An example is the *INTERNET* permission which regulates access to the Internet. This permission is all or nothing. Either an application has access to the Internet or not. There is no way to e.g. limit access to a specific domain or protocol. The authors of [48] describe a system they call *Dr. Android and Mr. Hide*. Their system works without having to modify the Android

base operating system. It consists of two parts *Mr. Hide* is a service that runs in the background and offers access to sensitive resources using more fine grained permissions. The counterpart is *Dr. Android* which can rewrite parts of an Android application to use *Mr. Hide* to access system resources instead of the traditional system.

The next problem with Android permissions is that they are only requested at installation time and can not be revoked later after installation. It is also not possible to deny specific permissions at install time. All permissions have to be granted for the installation to proceed. Removing permissions on a system level would of course be possible after installation. The problem with that approach is, that applications expect to have the requested permissions and will possibly crash if they suddenly do not [22]. The authors of [24] present their system *Apex*, which tries to solve the problem by extending Android to include permissions that can be changed during runtime.

The authors of [16] show, that the Android permission system is susceptible to privilege escalation attacks. This is due to transitive permission usage when one component can not access data of a second component, but is able to access the data through a third component, which does have access to the second component's data.

The ICC mechanism of Android gives developers the freedom to decide which other components are allowed to communicate with them by assigning permissions. Falsely assigned permissions or failure to assign permissions to security critical components as been analyzed extensively in [23, 27, 49]

Another vulnerability in the Android permission system is described in [22]. The vulnerability arises from the possibility of multiple applications sharing the same user and group id (see Section 3.2.1). In case applications share the same user and group id, their permissions will be combined. This can for example be used to evade the system presented in [47] where dangerous combinations of requested permissions by one application are detected. One application could for example ask for the *INTERNET* permission while the other application requests the *READ_CALENDAR* permission. While the permissions alone are not dangerous, they are if applications are able to share permissions.

iOS

iOS does not have an elaborate permission system like Android. It offers only permissions to high level functionality like the device's location or receiving push notifications. These permissions, unlike Android, are not granted during installation time, but can dynamically be granted during runtime [28]. Apart from these exceptions, each application in iOS has the same permissions through the use of the same sandboxing rules for each application. This one-size-fits-all approach to sandboxing, however, gives applications permissions that they actually might not need [33]. Every application has for example access to the Internet and the user's address book, which

inevitably causes privacy violations with some applications. Apple relies on their review process to catch applications that misuse their extensive permission, but it is unlikely they are able to find all misbehaving applications.

Another way in iOS to grant applications more permissions than the sandbox offers is by use of so called *entitlements*. An entitlement is a permission that is granted to an application independent of sandbox rules and without involving the user. These permissions can be quite dangerous. One example is the entitlement *run-unsigned-code* which allows an application to run code that is not signed. However, the provisioning profile for developer certificates has to specifically allow an entitlement to be granted and these provisioning profiles can only be created by Apple, which means that Apple can control which developer is granted which entitlements [44, p. 79].

Windows Phone 7

Windows Phone 7 includes a permission system that is similar to Android to elevate the security level of applications running in the least privilege chamber. An application includes required permissions, which are called *capabilities* by Microsoft during install time and can not be elevated during runtime. As of now, Windows Phone 7 only contains a limited set of coarse grained permissions. They include access to the device's location, the Internet and the microphone [12, p. 259].

Windows 7

On old Windows versions, it was usual for users to be logged in as an administrative user that was able to do anything he or she wanted. This approach, of course, is inherently insecure, but was necessary since many applications needed administrative privileges in order to work. To deal with that problem, Windows Vista introduced a system called *User Account Control (UAC)*, that enables users to run with normal user rights, by using filesystem and registry virtualization. Since some tasks require permissions that only an administrator has, e.g. when installing a driver, UAC also provides an easy interface to gain administrative permissions temporarily, by entering the administrator's credentials. This process is called *elevation* and also still allows running processes as the administrator user, but should only be used when it is absolutely necessary [20, p. 520ff.]. A user's permissions are saved in an access token that is assigned to every user upon login. This token is then inherited to every process that the user creates and is used to determine which actions a process is allowed to perform [20, p. 438ff.].

3.2.3 Code Signing

Platforms use code signing for different reasons. One reason is to identify the developer that created the code. Another reason is to ensure that the integrity of the application was not violated on its way from the developer to the user. A third reason is to make sure, that only applications,

that have been reviewed and approved by a manufacturer, can be installed on the platform. This section describes how the platforms use code signing and how it is implemented.

Android

In order for an application to be able to be installed on Android, it needs to be signed by the developer. This signature verification is only done during install time [16]. Furthermore, when an update is installed for an application, checks are performed to verify that the update comes from the original developer [29]. The keys used to sign the application do not need to be signed by a trusted certificate authority. It is also possible and common to use self-signed keys. This satisfies non-reputability goals since a developer can not deny having signed an application [41]. However, it does not prevent the integrity of an application to be compromised since it can simply be signed by another key. One instantiation of this problem is refereed to as *application repacking* where an attacker downloads an application, adds possibly malicious functionality and then signs it with his own key [16].

iOS

iOS applications also need to be signed. Signing is not performed by the developer, but by Apple after the application has been approved for the App Store [29]. An exception to this rule is during development where the developer can download a so called *provisioning profile*, which contains the developer certificate and enables the developer to execute applications signed with his key [25, p. 29]. Another way to sign applications without having Apple do the signing for the AppStore is by using enterprise distribution where a company is supplied with an own key to sign applications for distribution in-house. The reason for signing applications in iOS is mainly not to identify the developer, but to ensure integrity of the application when it is executed and also to ensure that only Apple signed and approved applications can be executed on iOS devices. Code signing in iOS is enforced on memory page level. When executable code is loaded into a memory page, the page can only be marked as executable if it has been marked as signed beforehand. A detailed analysis on how Apple implements code signing can be found in [44, p. 69ff.]. When an iOS device is jailbroken, code signing is disabled to allow non Apple signed programs to run. This, however, also opens the door for unsigned Malware [33].

Windows Phone 7

Windows Phone 7 takes a similar approach to code signing as iOS. Signing is done by Microsoft once the application has been approved for the Windows Phone Marketplace. The application is signed using keys that have been assigned to a developer during registration, meaning, that the developer of an application can be identified by the signature of the application. Only applications that were signed by Microsoft can then be run on a Windows Phone 7 device [50].

Windows 7

Windows 7 also supports code signing. Signatures can be applied to all kinds of code, e.g. applications, libraries or kernel drivers and is done using a certificate that has to be approved by Microsoft. However, signing is only enforced on code that runs in kernel mode. This is referred to as *Kernel Mode Code Signing (KMCS)*. Since KMCS would exclude old drivers that have been created by companies that might not even exist anymore, KMCS is currently only enforced on 64-bit versions of Windows [20, p. 246f.].

3.3 Mobile Malware

Malware is defined as "a set of instructions that cause a site's security policy to be violated" [36, p. 613]. Different kinds of malware exist. They can be separated into two categories depending on how they infect a system: Malware that tries to tick a user into installing it and malware that exploits a vulnerability in the system to install itself without user interaction. While there are a lot of different kinds of malware, we will only present two categories that are especially relevant on mobile platforms:

The first type of malware can be described as a trojan horse. "A *Trojan horse* is a program with an overt (documented or known) effect and a *covert* (undocumented or unexpected) effect" [36, p. 614]. Trojan horses try to tick a user into installing software that performs a legitimate function, but also contains hidden functionality for example to compromise a user's privacy by sending his calendar to a third party [46].

The second type of malware can be described as a computer virus. "A *computer virus* is a program that inserts itself into one or more files and then performs some (possibly null) action" [36, p. 616]. A special variant of a computer virus is the computer worm. "A *computer worm* is a program that copies itself from one computer to another" [36, p. 623]. While viruses infect files on a single system, a worm tries to spread to other systems. Early malware of this type mainly used Bluetooth as a way to spread such as the first ever mobile phone worm discovered in 2004 called *Cabir* [4]. Nowadays, since a main part of the mobile devices have access to the Internet, infection is mainly performed by exploiting vulnerabilities in TCP/IP services, for example by use of browser exploits [51]. A detailed description of how a worm can be created for the Windows Mobile platform can be found in [52, 53].

In the last few years, mobile malware has evolved from research or fun projects to malware that creates their authors financial profit. The development is analogous to that of malware on desktop computers [6]. This can be seen from the effects that malware has. An analysis of all known mobile malware between 2004 and 2008 in [34] shows that the effects of malware were mainly manipulating the device for fun, a big part of the malware had no effect at all apart from spreading. An analysis of all known malware between 2009 and 2011 in [6] paints a different

picture: Most of the malware is concerned either with gathering information about the user or making money directly by calling or sending SMS messages to premium numbers.

A detailed analysis of present and past mobile malware is beyond the scope of this thesis, but can be found for example in [4, 34, 54, 55, 56, 57, 58].

Another way to differentiate between malware is whether it uses the official mechanisms that the platform offers to a day-to-day user or whether it requires full access to the system in form of administrative privileges [59].

As seen in Section 3.2.1, all mobile platforms offer one way or another to sandbox applications that have legitimately been installed to prevent possible malware from compromising the system. But that these protection mechanisms do not protect the user completely can for example be seen from the possibility of privacy violations in iOS, as demonstrated in [13]. Nevertheless, malware is restricted when it is running with legitimate permissions, therefore, malware that uses a vulnerability in the platform to perform privilege escalation have emerged. What makes these kinds of malware easy to develop is, that as described before, the jailbreaking and rooting communities are very motivated and successful to find such privilege escalation vulnerabilities and publish exploits for them. These exploits can of course also be used by malware to escalate privileges [6].

The authors of [60] describe an effort of creating a proof of concept malware for iOS, that uses an exploit, which is used to jailbreak iOS devices. It then uses its privileges to deploy a library level root-kit to hide from the user and uses worm functionality to spread and create a bot-net controlled by a command and control server. [61] explains the process of creating various root-kits on a platform called Openmoko. According to [6], malware is already taking advantage of gaining root access, but [52] explains, that currently, no malware is using these capabilities to create a worm and/or bot-net. But since the possibility exists, this will only be a matter of time. The possibility of creating such a sophisticated malware shows, that malware on mobile devices has reached the sophistication of malware on desktop platforms.

Another interesting question is how widespread mobile malware is at the moment. In the beginning of 2012, the number of discovered malware families on mobile platforms has reached over 700 [5]. This is of course still in strong contrast to desktop platforms where 400 million malware variants have been discovered in 2011 only [62]. However, the growth of new mobile malware being discovered is increasing very fast. More than 25% of all mobile malware ever discovered was detected in 2011 only [5].

According to an analysis of the Finnish anti-virus manufacturer F-Secure, 64.8% of all mobile malware discovered in 2011 was targeting the Android platform [5]. For iOS, currently two malware variants, both from 2009, are known. Both target jailbroken phones and exploit a vulnerability when installing an SSH server on the phone after jailbreaking because jailbreaking sets

a root password that is the same on each device [5, 6]. Neither F-Secure nor we are aware of any malware that targets unmodified iOS devices and we are also not aware of any malware that targets Windows Phone 7.

3.4 Third Party Security Solutions

As we have seen above, all of the investigated platforms have security problem one way or the other. Because of that, third parties develop security solutions to improve the security of the platforms. The authors of [39] identify multiple measures to improve security of the Android platform. Some of these are specific to the Android platform or already implemented on other platforms or are out of scope of this thesis. But three of them, anti-malware tools, firewalls and intrusion-detection and prevention systems could theoretically by applied to all mobile platforms and will be discussed in this section.

It has to be noted that all of the work presented below just gives theoretical insight into implementing these security solutions on mobile platforms that most of the time require the platform to be modified before the solutions can be implemented. We were, however, unable to find any work on concrete implementations on unmodified devices. This was mainly the reason for investigating the second research question in this thesis where we look at concrete implementations that currently exist (see Section 5).

3.4.1 Virus Scanners

A virus scanner is traditionally defined as "a program that examines systems for the occurrence of known viruses." [63]. Scanning for viruses, or more generally: malware, can be done in different ways. The first scanners were simply scanning files and the memory for known malware patterns. These malware patterns are called *signatures*. This first approach was very inflexible since slightest changes in the malware could evade the scanner. More advanced methods have been developed that do not depend on an exact malware signature. One such method is a heuristic approach which extracts features from known viruses and then uses only these features as signature to detect malware more generally and can therefore, to some degree, also detect previously unknown malware. A last method for virus scanning is so called *code emulation*. Code emulation runs a program in an emulated environment and observes the actions performed by the program to determine if it is malicious. [64].

Virus scanners can perform their scanning activities in two ways: Either on-demand or on-access. On-demand scanning is performed when the user requests a scan. On-access scanners are running constantly and perform scanning when specific events in the system occur. One example of an on-access scanner are scanners that are either implemented in or hook into filesystem drivers. That way, they can perform scanning when a file is accessed [64].

Virus scanners on mobile platforms have the problem that they need a lot of resources to perform their scanning. While this is not a big problem on a desktop platform, it presents a problem in a mobile environment where processing power is limited and power consumption is an issue [28, 61]. This is why the authors of [65, 66, 67] propose to perform detection not on the phone, but in the cloud. While these problems can probably be solved by further research, many of these approaches either require a modified kernel [67] or extensive access to resources like the filesystem or kernel memory [61]. On Android, these permissions can only be acquired by possessing root access to the device [59], which is not practical for a widely deployed security solution. This causes the problem that security mechanisms, that are intended to protect the platform, turn into limitations of applications trying to improve the security of these platforms [26]. These problems could be solved by platform manufacturers giving certain application vendors more privileges than normal applications as in the case of Symbian. These higher privileged applications for example have the permission to access the whole filesystem of the device to facilitate scanning for viruses on the whole filesystem [59].

3.4.2 Firewalls

"A *firewall* is a host that mediates access to a network, allowing and disallowing certain types of access on the basis of a configured security policy" [36, p. 780]. Different kinds of firewalls exist, depending on the level on which access control is performed. "A *filtering firewall* performs access control on the basis of attributes of the packet headers, such as destination addresses, sources addresses and options" [36, p. 781]. This kind of firewall only performs access control at the network level and is not concerned with the content of the network communication at the application layer. "A *proxy* (or *application level*) *firewall* uses proxies to perform access control. A proxy firewall can base access control on the contents of packets and messages, as well as on attributes of the packet header" [36, p. 781]. This kind of firewall can also perform access control on the application layer of the network communication.

On mobile platforms, firewalls suffer from the same problem that virus scanners have. In order to perform access control on the network communication, they need the ability to influence network packets at a very low level, usually within the kernel before they are processed by higher layers. Since the ability to control network packets on such a low level has severe security implications if it is not controlled, mobile platforms do not offer normal applications access to such controls. Android for example, being based on a Linux kernel, already has a user-space interface to control packet processing in the kernel called *Netfilter* [39]. However, due to platform restrictions, this interface can only be accessed from applications on rooted devices [59].

3.4.3 Intrusion Detection and Prevention Systems

Intrusion detection and prevention systems are based on the hypothesis that "exploiting vulnerabilities requires an abnormal use of normal commands or instructions, so security violations

can be detected by looking for abnormalities" [36, p. 725]. These systems either detect when a system has been compromised or detect the compromise and then act on it. From that definition, it can be seen that the above described security solutions can also be classified as intrusion detection and prevention systems.

Intrusion detection can be done in two ways. Either anomaly based or misuse based. "*Anomaly detection* analyzes a set of characteristics of the system and compares their behavior with a set of expected values. It reports when the computed statistics do not match the expected measurements" [36, p. 727]. Anomaly detection basically looks at certain characteristics of a system and checks if they are in a different state than a predefine known to be good state. This can also be called *white-listing*: Everything that is found in the white-list is normal, everything else is an intrusion. The other way to perform intrusion detection is *misuse detection*. "*Misuse detection* determines whether a sequence of instructions being executed is known to violate the security policy being executed. If so, it reports a potential intrusion" [36, p. 733]. This approach is the complete opposite of anomaly detection. Here, the system searches for known to be bad system states and reports only when one of these states is found. This can be called a *black-list* approach: Everything that is found in the black-list is an intrusion, everything else is normal. These systems have different strengths and weaknesses. The most important of them being that anomaly detection systems might categorize normal system states, that are not in the white-list, as an intrusion. This is called a false positive and means that the white-list is incomplete. Misuse detection systems on the other hand have the problem that they might classify intrusions, that are not in the black-list, as normal. This is called false negative and means the black-list is incomplete.

Intrusion detection systems can gather data for detection at different locations and levels. "Host-based agents usually use system and application logs to obtain records of events" [36, p. 744]. On the host, data can be obtained either from the system or from a single application. The latter is called *application based* [68]. The other location, where data can be collected is the network. "Network-based agents use a variety of devices and software to monitor network traffic" [36, p. 744].

Host based intrusion detection and prevention systems have been widely investigated for mobile platforms [69, 70, 71, 72, 73, 74, 75, 76, 77]. The advantage of intrusion detection and prevention system is that, opposed to virus scanners and firewalls, they do not necessarily need access to privileged system resources or interfaces. A widely researched approach for intrusion detection on mobile devices for example is to analyze battery usage to detect malware [78, 79, 80]. The advantage here is that battery usage is usually not privileged information. Another possibility to detect malicious applications is presented in [6] where the authors try to use permissions requested by Android applications to determine whether an application is malicious or not. Permission usage of applications on Android is also no privileged information.

3.5 App Hardening

The description of app hardening in this section is based on personal communication with Tom Lysemose Hansen from Promon AS on February 10th, 2012. Promon's app hardening approach is a protection mechanism that does not protect the whole system, but it protects a single application running on a possibly hostile system. The application can either be in the form of a desktop application or in form of a web-application in which case, the browser application is protected. The threats to protect against are loss of data (e.g. by keyloggers) and data manipulation (e.g. in case of an online banking application that is manipulated to redirect money transfers). An important aspect of app hardening is the fact that it is non-intrusive. This means that by using app hardening, other security solutions and other system services are not influenced in any way.

The main security mechanism of app hardening is to use a white-list approach to gather as much information about the system and the application, that is protected, as possible to determine what the normal behavior of the program is. This information is in form of internal process space information like program flow and loaded modules as well as information about interfaces to external entities like interprocess communication, the keyboard or network communication. The solution then internally monitors the application and uses the white-list to detect when the application is behaving in an unexpected way, e.g. when a debugger is attached. In case of libraries loaded into the process, white-listing is done at load time based on library certificates and checksums. During runtime, libraries are monitored for changes, e.g. when a hook is placed inside a library. The reaction to unexpected behavior can be configured and is usually defined by the security policies of the application and service providers. This could for example be to quit the application or to block the malicious behavior. Another possibility is to restrict the process to only allow actions that are in the white list, for example in case of network communication, the hosts that the application is allowed to communicate with, can be limited. This is also determined by the security policy of the application or service provider.

This mechanism is similar to an anomaly and application based intrusion detection system (see Section 3.4.3). The difference is in the goals of the systems. While intrusion detection systems usually try to detect infections of the whole system, app hardening tries to protect a single application in a hostile environment. Some research has been done regarding application based intrusion detection systems on server platforms. The authors of [68] for example describe an approach for integrating an IDS in the Apache web-server. They collect needed information at its source inside the application for subsequent intrusions detection. An approach to profile applications in a Java environment is presented in [81]. The authors then use the profiling information to sandbox programs in order to detect and prevent exploitation of the software. Another similar approach is taken in [82] where the authors perform a live analysis of the Linux kernel's behavior. They inserted hooks in the kernel to acquire the required data for searching for anomalous behavior. Even though, there has been some work done similar to this aspect of the app hardening approach, we were unable to find any approaches that are targeting mobile platforms. The result of this thesis will therefore expand previous research on application based intrusion

detection systems into the new area of mobile platforms.

Another important aspect of app hardening is to bind the application to be protected to the security module. Only if the application is protected by app hardening should it be possible to use it. In case of a web-application this is for example achieved by the security module integrating a challenge-response mechanism in the communication with the webserver and by the webserver only allowing connections from applications that successfully pass the challenge-response test. That way, the server makes sure that only browsers that are protected by app hardening are able to communicate with the web-application.

A third important aspect of app hardening is its deployment. Promon offers multiple ways to deploy app hardening for applications. App hardening can either be compiled into a program, it can be in form of a separate application, that is used to launch the program, or in form of a Java applet, that is dynamically delivered from a server and then also used to launch the application to be protected. A critical point in time for app hardening is when the security module is started inside the application that is to be protected. At this point, it is important to make sure that the process is initially in a clean state or that the process can be cleaned. The earlier the security modules is loaded, the easier it is to make sure that the process is clean. That is the reason why Promon tries to get into the process as early as possible by launching the application. After the security module has confirmed that the process is in a clean state, it then checks its own integrity and can then start to monitor the application for malicious behavior.

App hardening also tries to protect the interfaces of an application. This can for example be to prevent a keylogger from capturing keystrokes made in the application or to prevent malicious applications from taking screenshots of the application.

4 Malware on Mobile and Desktop Platforms

This chapter investigates the first research question: “What are the limitations of malware running on mobile platforms compared to desktop platforms?”. We will investigate this question by analyzing which threats malware could manifest on the platforms. The threats we are investigating are from the view of an application that is running on a platform. We limit malware capabilities in our investigations to possess permissions that are granted to normal applications and not as the system administrator, i.e. they run on the same privilege level as other applications. We impose this restriction since malware with administrative privileges will be able to deploy rootkits to hide its presence and is therefore able to deceive applications that only have normal permissions. Also, administrative privileges are only obtainable on the mobile platforms with use of an root exploit and are therefore not as easily obtainable as on desktop platforms. Another reason for this is the fact that, as we have seen in Section 3.3, currently, malware on mobile platforms using administrative privileges through root exploits is not very common.

The methodology of our threat investigation is loosely based on the book “Thread Modeling”, written by the two Microsoft employees Frank Swiderski and Window Snyder [9]. The first step is to determine which assets a generic application has. We then determine which possibilities exist to interact with these assets and could therefore result in the assets being compromised. The book refers to these interaction possibilities as *entry points*. The authors of [9] then propose to analyze for each entry point, how an asset can be compromised. However, as the authors state, finding threats is a complicated process since the completeness of the found threats is hard to proof. As described in [9, p. 42], finding all threats for a system can not be achieved by simple brainstorming. The authors therefore propose a different approach to ensure that every threat is found. By analyzing all data flows from the entry points to the assets of a system, all possibilities of compromising the assets can be found. However, for large systems like an operating system, this will be a very cumbersome process. We can not possibly perform this process for four platforms in the given time. This is also owing to the fact that all platforms except Android are proprietary and therefore need to be fully reverse engineered in order to perform an analysis. This analysis would also be very version dependent since the security model of the platforms change when new vulnerabilities are discovered and fixed.

We will therefore investigate possibilities for using entry points to compromise assets on each of the platforms without claiming to have found all possibilities. The investigations will rather be based on an example threat that we give for a subcategory of the each entry point for each of the assets. For each of the entry points, we will determine how difficult it would be for malware to compromise the assets based on our investigations. We will end up with an overview of which

entry points are most vulnerable on each of the platforms to see where the platforms require additional security in form of third party security extensions. This will also allow us to draw conclusions about the differences in threats between mobile and desktop platforms.

4.1 Assets

When we talk about assets, we are talking about the assets that an application has while running on a platform. We base our assets on the definition of Common Criteria which states that "Security-specific impairment commonly includes, but is not limited to: loss of asset confidentiality, loss of asset integrity and loss of asset availability." [83, p. 35]. These are basically the security goals of information security. We relate those impairments to an application to define our assets, which can be found in Table 1.

Asset	**Description**
Confidentiality of application data	Sensitive data that belong to an application should not leak to other applications. An example of such an asset would be a password that a user types in an application.
Integrity of application data and flow	The application data and execution flow should not be influenceable by other applications. An example of such an asset would be the decision of an access control mechanism of an application.
Availability of application services	The services offered by the application should not be disrupted by other applications. An example of such an asset would be the scanning functionality of a virus scanner.

Table 1: Identified assets

4.2 Entry Points

In order to determine which kinds of entry points an application has, we look at the execution environment, that an application runs in. This execution environment is defined by the services that the operating system of the platform offer. The authors of [84, p. 39f.] have identified common classes of such services, which will be used as a basis for defining the entry points. Table 2 shows these found entry points. In order to determine that they are complete, the APIs of Android[1], iOS[2] and Windows Phone 7[3] were reviewed to see if we have found all entry points for the relevant mobile platforms.

[1] `http://developer.android.com/reference/packages.html`
[2] `http://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Reference`
[3] `http://msdn.microsoft.com/en-us/library/ff626516(v=vs.92).aspx`

Entry Point	Description
Filesystem	Interface to deal with the filesystem. This includes creating, manipulating and deleting files and folders as well manipulating meta-information of files.
I/O operations	The interface provided by the operating system to deal with I/O devices (e.g. memory or hard-drives).
Communications	In order for applications to communicate with other applications, they are provided with communication interfaces. Communication can be with applications on the same computer, which is called interprocess communication (e.g. through shared memory). The other possibility is to communicate with applications on a remote computer (e.g. through TCP/IP).
User interface	The interface between the application and the user. The user interface is bidirectional, meaning that the user can input information to the system (e.g. by a keyboard) and that the system can output information to the user (e.g. by a screen).
Program execution	The operating system provides the possibility to launch an application and control its life-cycle (e.g. by pausing or terminating it).

Table 2: Identified entry points

4.3 Impact of Threats

When evaluating the risk that a threat poses, we need to know how severe the impact of the threat is. Our initial idea was to evaluate the impact of each of the threats in order to rank them and have a basis for which threats to investigate first. The impact of a threat is closely related to the assets. The more important the asset is, the bigger the impact, should it be compromised [9, p. 52]. Since we are investigating threats to a generic application, we can not evaluate the impact that a threat has. This is due to the fact that different applications will have different priorities when it comes to the assets. We can see this from the example of two applications: a virus scanner and an online-banking application. While from the view of a virus scanner, availability is more important than confidentiality, this will be exactly opposite from the view of an online-banking application. Integrity of the application data and flow, however, will be important to both types of applications. Because of this, we will not perform an evaluation of the impact of the various threats. This needs to be performed on a per-application basis separately and can then be combined with the results of our work which determines the likelihood that one of the threats can be manifested.

4.4 Investigation of Threats

Now that we know about the entry points and the assets, we can start our investigation of possible threats that arise from the entry points through an example threat to determine the likelihood that the threat could be used by malware to compromise the assets. We use a qualitative approach to assign a likelihood to a threat being manifested. This means that we will not assign a concrete number to the likelihood, but a category instead. Table 3 gives a description of the different likelihoods that can be assigned to the threats. These categories are our own definition, but are similar to the method proposed for *calculating attack potential* in the Common Evaluation Methodology (CEM) of Common Criteria [85, p. 361ff.].

Likelihood	**Description**
High	The threat is easy to manifest because the platform is designed to allow it.
Medium	It is possible to manifest the threat under certain restrictions.
Low	It is not possible to manifest the threat without finding a bug in the implementation of the system.

Table 3: Possible likelihoods that can be assigned to a threat

We chose our own definition since the method in CEM includes a lot of factors that are not relevant for our investigation, like access to the evaluation target and required equipment. The factors *time* and *specialist technical expertise*, that are also included in the factors proposed by the CEM to evaluate the potential of an attack, are also present in our definition. This is because takes less time and much less technical expertise to compromise a system that is compromisable by design than to compromise a system where a vulnerability in the implementation has to be found first.

For the medium likelihood, we have identified the following restrictions under which the threats could be manifested:

- The user has to be deceived into performing an unwanted action.
- The developer of an application has to make a mistake that is not prevented by the platform.
- Manifesting the threat only gives limited access to the resource.

Below, we will perform experiments on each of the platforms for each of the entry points trying to compromise the assets we have defined. The experiments are partially taken from previous research and well known methods or based on own ideas. Due to time constraints, threats for compromising the integrity and availability through the user interface entry point as well as threats for compromising the confidentiality and integrity through the program execution interface have not been investigated. This is subject to future work. Before we start with the description of the

experiments, a description of the experiment environments will be given.

For the Android experiments, we developed in *Eclipse Indigo 3.7.2* running on *Gentoo Linux* with a *Linux 3.3.0 64 bit kernel*. We developed the experiments using the Android emulators for *Android 2.2 and 4.0.3*. In order to verify that the experiments also yield the same results on an actual device, we ran the finished experiments on a *ZTE Blade* phone running *Android 2.3.3*. For the experiments that required root access, we used a rooted *Samsung Nexus S* phone running *Android 4.0.3*.

The iOS experiments were developed in *Xcode 4.2* running on *Mac OS X 10.6.8*. We developed the experiments with the *iOS 5.0 SDK* and used the *iPad 5.0 and iPhone 5.0 emulators* to test the experiments. In order to verify that the experiments yield the same results on an actual device, we ran the finished experiments on three different iOS devices: A jailbroken *iPhone 2G* running *iOS 3.1.2*, a non-jailbroken *iPhone 4S* running *iOS 5.1* and a non-jailbroken *iPad 2* running *iOS 5.0.1*. In later experiments, the iPad 2 was jailbroken using the *Absinthe 0.4* jailbreaking tool to test different behavior on jailbroken and non-jailbroken devices.

The Windows Phone 7 experiments were developed in *Microsoft Visual Studio 2010 Express for Windows Phone* from the *Windows Phone SDK 7.1* running on *Windows 7 Professional SP1 64 bit*. The experiments were run in the *Windows Phone Emulator* for *Windows Phone 7.5*. We did not run the experiments on an actual device since, as we will see below, we did not find a way to successfully compromise any asset through any of the entry points.

The Windows 7 experiments were developed in *Microsoft Visual Studio 2010 Professional* on *Windows 7 Professional SP1 64 bit* running in *Virtual Box 4.1.14* on *Gentoo Linux* with a *Linux 3.3.0 64 bit kernel*.

4.4.1 Filesystem

This section will explore how the filesystem can be used on the investigated platforms to compromise the assets of an application. An example threat for each of the assets is given below (note that we only focus on the installation of an application). Those threats will be used to investigate the likelihood for compromising each asset:

- Confidentiality: Stealing of application secrets from installation.
- Integrity: Modification of application installation.
- Availability: Deletion or corruption of installation.

Android

On Android, we started investigating where and how applications are installed in the filesystem. We collected the data below by connecting to an Android device using the Android Debug Bridge, which gives us a command shell to explore the filesystem. We found the following locations where applications are installed in Android.

Normal applications are installed in the folder */data/app/*. Installation means copying the APK file in that directory which will then be executed when the application is first launched. Table 4 shows the permissions for the folder, normal applications are installed in, as well as for the APK files. Permissions are represented as read (r), write (w) and execute (x). In case no letter is given, the permission is not set. The permissions are also divided in three groups. The first group stands for the user that owns the file, the second group stands for the group that owns the file and the last group stands for all other users.

File	Type	Owner	Group	Permissions
/data/app/	Folder	system	system	`rwx rwx --x`
/data/app/<app-name>.apk	File	system	system	`rw- r-- r--`

Table 4: Filesystem permissions for normal applications installation

As can be seen, all users are able to access the */data/app/* folder (execute permission), but are not able to list its content (read permission). The APK files themselves are readable for everyone, but only writable for the *system* user. This means that malware first has to determine the filename of the APK file it wants to manipulate. This can not be done on a filesystem level, but the Android framework provides this information through the *PackageManager* class. However an application running as a normal user can not write to the APK file, meaning, that it can not modify the installation.

Android offers the possibility to have a simple form of copy protection for applications installed from the Google Play Store. This is chosen as an option when a developer submits his application to the Google Play Store. Copy protected applications are installed in */data/app-private/* and can then only be read by the root user. However, this system has been declared as deprecated by Google[4] and we could not find any applications that use it, so we were unable to investigate it further. We do, however, not expect that a malware is able to write to APK files in this folder if it can not even read them.

System applications in Android are installed in the */system/app/* folder. Table 5 shows the permissions for this folder and APK files contained in it. The only difference to normal applications is that all users can list the contents of the folder (read permission). It is, however, not possible to write to the APK files either.

[4] `http://support.google.com/googleplay/android-developer/bin/answer.py?hl=en&answer=186113`

File	Type	Owner	Group	Permissions
/system/app/	Folder	root	root	`rwx r-x r-x`
/system/app/<app-name>.apk	File	root	root	`rw- r-- r--`

Table 5: Filesystem permissions for system applications installation

The next location, where applications can be installed is on the memory card. This option has to be granted by the developer of the application in the application's manifest file by setting the *installLocation* parameter. Applications installed on a memory card are installed in the */mnt/asec/* directory. Here, each application has its own folder that contains the APK file. Table 6 shows the permissions for these files. We see the same situation as before: normal users can not write to the APK file.

File	Type	Owner	Group	Permissions
/mnt/asec/	Folder	root	system	`rwx r-x r-x`
/mnt/asec/<app-name>/	Folder	system	root	`r-x r-x r-x`
/mnt/asec/<app-name>/pkg.apk	File	system	root	`r-x r-x r-x`

Table 6: Filesystem permissions for application installations on memory cards

If an application contains native libraries that are used via JNI, these libraries are extracted in the folder */data/data/<app-name>/lib/*. As before, we can see from Table 7 that a normal user is not able to modify these libraries.

File	Type	Owner	Group	Permissions
/data/data/	Folder	system	system	`rwx rwx --x`
/data/data/<app-name>/	Folder	app_x	app_x	`rwx r-x --x`
/data/data/<app-name>/lib/	Folder	system	system	`rwx r-x r-x`
/data/data/<app-name>/lib/<libname>.so	File	system	system	`rwx r-x r-x`

Table 7: Filesystem permissions for application libraries

Our investigation shows that reading the installation of an application is possible, but writing and deleting is prevented by filesystem permissions. We therefore assign the likelihoods found in Table 8 for compromising the assets through the filesystem entry point.

iOS

On iOS, we also looked at the installation directories for applications and their permissions. There is no standard way to access a command shell without jailbreaking the device on iOS. So in order to explore the filesystem on iOS devices, we wrote a small program that lists the contents of a directory together with some meta-information about the contents. This program was inspired by the UNIX program *ls*. It basically recursively lists the content of a given directory and prints information on each file by using information acquired from the system call *stat*. The source code of the application can be found in Appendix A.1.

Asset	Likelihood
Confidentiality of application data	High
Integrity of application data and flow	Low
Availability of application services	Low

Table 8: Likelihood of compromising assets through the filesystem on Android

Applications in iOS are installed in */private/var/mobile/Applications/<UUID>/*. The UUID is a random identifier that does not give any hints about which application it belongs to. Table 9 shows the relevant content of the */private/var/mobile/Applications/* folder. Note that the application used to run the test has *<UUDI-1>* in this case. The application with *<UUDI-2>* is another application. We can see, that on a filesystem level, the mobile user, which is used to run applications in iOS has full access to the *Application* folder. The *<UUDI-1>* folder contains all files of the <bundle-id1> application. This includes the folder *<app-name>.app/* which contains the installation of the application including the application's executable *<executable>*. As we can see, the mobile user has full access to all of these folders and files. When we, however, want to get information about *<UUDI-2>*, the system-call *stat* that we use to get information about files fails even though this folder and its sub-folders have the same permissions as *<UUDI-1>*. We get the same effect if we want to open the folder for reading or writing even though we have permissions to do it. Here we can see the effect of the iOS sandbox. While the filesystem allows full access to other application's installation directories since all applications are run as the user mobile, the sandbox prevents us from accessing it.

File	Type	Owner	Group	Permissions
Applications/	Folder	mobile	mobile	`rwx r-x r-x`
Applications/<UUID-1>/	Folder	mobile	mobile	`rwx r-x r-x`
Applications/<UUID-1>/ <app-name>.app/	Folder	mobile	mobile	`rwx r-x r-x`
Applications/<UUID-1>/ <app-name>.app/<executable>	File	mobile	mobile	`rwx r-x r-x`
Applications/<UUID-2>/	???	???	???	`??? ??? ???`

Table 9: Filesystem permissions for application installations on iOS

The author of [86] has decompiled the iOS sandbox and has come to the same conclusions about how the sandbox deals with separating access to different application's installation directories. We see this as further confirmation of our findings.

Our investigation shows that reading, writing and deleting of the installation of an application is prevented by the iOS sandbox. We therefore assign the following likelihoods to compromising the assets through the filesystem entry point:

Asset	Likelihood
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 10: Likelihood of compromising assets through the filesystem on iOS

Windows Phone 7

In Windows Phone 7, applications are only able to access the filesystem through the isolated storage (see Section 3.2.1). This means that it can only access a small part of the filesystem that contains its own data and can not see anything else outside this location. The only way we see to access the underlying filesystem is to modify the operating system using an exploit.

Our investigation shows that reading, writing and deleting of the installation of an application is prevented by the Windows Phone 7 sandbox. We therefore assign the following likelihoods to compromising the assets through the filesystem entry point:

Asset	Likelihood
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 11: Likelihood of compromising assets through the filesystem on Windows Phone 7

Windows 7

In Windows 7, applications are usually installed using an installer. These installers are executed with administrative privileges and install applications in the *C:\Program Files* folder. Applications that are installed this way are owned by the *Administrators* group. Table 12 shows the permissions for the Administrators and Users groups for the installed files. While users of the Administrator group can fully modify the installation, normal Users have only the permission to read and execute files.

Group	Full Control	Modify	Read & Execute	Read	Write
Administrators	✓	✓	✓	✓	✓
Users			✓	✓	

Table 12: Filesystem permissions for application installations in Windows 7

The above stated is however only true for applications that are installed in this way. Applications that are, for example installed by extracting a ZIP file, can be installed by a normal user and can therefore be manipulated by other applications that run with the same user permissions.

Since Windows 7 does not have a sandbox that limits access to the filesystem, accessing files is

only governed by filesystem permissions. Our investigation shows that reading of the installation of an application is possible on Windows 7. Writing and deleting of the installation depends on where the application is installed. We therefore assign the following likelihoods to compromising the assets through the filesystem entry point:

Asset	**Likelihood**
Confidentiality of application data	High
Integrity of application data and flow	Medium
Availability of application services	Medium

Table 13: Likelihood of compromising assets through the filesystem on Windows 7

4.4.2 I/O Operations

This section will explore how I/O operations can be used on the investigated platforms to compromise the assets of an application. An example for a threat for each of the assets is given below (note that we only focus on the memory of an application). Those threats will be used to investigate the likelihood for compromising each asset:

- Confidentiality: Stealing of application secrets from memory.
- Integrity: Introduce new code in running application through memory.
- Availability: Corrupt memory of application.

Android

In Android, we have identified three ways to read memory of another process. First, the Linux kernel exposes the pseudo device */dev/mem*, which is an interface to read the physical memory. However, finding the right place in physical memory requires malware to reverse the virtual memory system, which is quite difficult and very platform and version dependent. Another way to read memory from a specific process is through the *proc-filesystem*. For each process, a file under */proc/<pid>/mem* is created which gives direct access to the memory of the process. The last possibility, we found is using *ptrace*, which is a kernel interface which is usually used by debuggers.

The permissions for the above mentioned files can be found in Table 14. As we can see, only the *root* user can read the physical memory file. The process memory file can only be read by the application that the memory belongs to.

In order to test the ptrace interface, we first created a native Android application that runs a loop

File	Type	Owner	Group	Permissions
/dev/mem	Character Device	root	root	`rw- --- ---`
/proc/<pid>/mem	File	app_x	app_x	`rw- --- ---`

Table 14: Filesystem permissions for memory files

and contains an integer variable that we will try to read. It also prints the memory address of the variable so we do not need to search for it. A malware trying to read sensitive information from another application will of course not have the luxury of having the address served on a plate like this. Finding the right location will therefore be more difficult in practice, but not impossible. We used the ptrace interface to read from that address in another native Android application as can be seen below. We basically attach to the process, which stops its execution, read from its memory and then detach again so the process continues running. This can be seen in the code below:

```
#include <sys/ptrace.h>
#include <stdio.h>
#include <errno.h>

int readInteger(int pid, void * address)
{
  // Attach to process
  if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) == -1)
  {
    printf("ptrace(PTRACE_ATTACH) failed: %i\n", errno);
    return -1;
  }

  // Wait until process is stopped
  int status;
  wait(&status);

  // Read data from process
  long result = ptrace(PTRACE_PEEKDATA, pid, (void *)address, NULL);

  if(result == -1)
    printf("ptrace(PTRACE_PEEKDATA) failed: %i\n", errno);

  // Resume process
  if(ptrace(PTRACE_DETACH, pid, NULL, NULL) == -1)
  {
    printf("ptrace(PTRACE_DETACH) failed: %i\n", errno);
    return -1;
  }

  return result;
}
```

When we ran the programs as different users, attaching failed. If the reading program was executed at the *root* user or the same user as the other application, reading was successful. In our case, we only consider applications that do not have root permissions. The only way to run as the same user as another application is if it is signed with the same key (see 3.2.1). This means that

malware can only read from the memory of another application if it has the key used to sign the targeted application, which is not very likely. The same behavior was observed when we tried to write to the memory of another application.

Our investigation shows that reading and writing memory of another application is not possible for normal applications. We therefore assign the following likelihoods to compromising the assets through the I/O operations entry point:

Asset	**Likelihood**
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 15: Likelihood of compromising assets through I/O operations on Android

iOS

Since the ptrace interface is defined by the POSIX, we also find it in the iOS XNU kernel. We performed the same experiments as on Android and also the source code for the reader is mostly the same. When we ran the reader program on an iOS device against an application that runs as the *mobile* user, it was able to attach to other processes, which is surprising since this means that malware can halt each normal application on an iOS device from within ins sandbox. The actual reading, however, failed. The same behavior was also observed for writing.

The iOS XNU kernel, being based on a Mach kernel (see Section 3.1.2) offers another way to access the memory of another process. Mach defines the concept of *ports*, which are communication channels between processes (or tasks, as they are called in Mach). Acquiring such a port is done using the system call *task_for_pid*. The port can then be used for example to read or write to the memory of another task using the functions *vm_read* and *vm_write*. However, acquiring a port to another task is subject to strict security checks. In order to check whether we can acquire a port of another task and then use it to read from the task's memory, we wrote a small test application. When we ran it, the call to *task_for_pid* failed due to sandbox restrictions. The author of [86] has an explanation: In order to be allowed to call *task_for_pid*, an application needs the *task_for_pid-allow* entitlement. This entitlement is for example used by Apple's debugger to attach to running processes. Since the entitlements for an application are controlled by Apple and normal applications are not granted any special entitlements, malware can not use *task_for_pid* from a normal application without Apple's knowledge.

Our investigation shows that reading and writing memory of another application is not possible for normal applications. We therefore assign the following likelihoods to compromising the assets through the I/O operations entry point:

Asset	Likelihood
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 16: Likelihood of compromising assets through I/O operations on iOS

Windows Phone 7

We were unable to find a way to read or write memory of another application on Windows Phone 7, which is not surprising when we consider how tight the Windows Phone 7 sandbox restricts access to the filesystem and system functions. Our investigation shows that reading and writing memory of another application is not possible for normal applications. We therefore assign the following likelihoods to compromising the assets through the I/O operations entry point:

Asset	Likelihood
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 17: Likelihood of compromising assets through I/O operations on Windows Phone 7

Windows 7

In Windows 7, Microsoft offers the API function *ReadProcessMemory* to read the memory of another process. We have implemented a test program that tries to read an integer from another application:

```
#include <windows.h>

int ReadIntegerFromProcess(unsigned int processId, LPCVOID address)
{
  HANDLE process;
  int buffer;

  // Open process
  process = OpenProcess((PROCESS_VM_OPERATION | PROCESS_VM_READ), TRUE, processId);

  if(process == 0)
    return -1;

  // Read from memory
  if(!ReadProcessMemory(process, address, &buffer, sizeof(int), NULL))
    return -1;

  // Clean up
  CloseHandle(process);

  return buffer;
}
```

When we ran the program, we were successful in reading memory from a process that is run by the same user. Reading memory from processes of other users, however, failed. We do not consider this a big problem since a normal user will run all his applications as the same user. If a process runs with administrator privileges, it can enable a special permission called *SeDebugPrivilege* which will allow access also to processes of other users [87]. Analogous to *ReadProcessMemory*, Windows 7 offers the API function *WriteProcessMemory* to write to the memory of another process. When we tried to write to another process, we got the same results as for reading memory: If the processes are run by the same user, the writing is successful, otherwise it fails.

The experiment we did, was to investigate is whether it is possible to execute code that was written to an application before. There were multiple ways we could have tried, we could for example have used hooking to execute our own code when the application calls specific functions. We have, however, decided to investigate a more flexible and less invasive way based on [88]. Windows offers an API function called *CreateRemoteThread* that creates a new thread in another process which starts at a specified address in the processes memory. Using this function and *WriteProcessMemory*, we can write code to the remote process and let it execute. This can be used to have another process load a new DLL file (using the function *LoadLibrary*) and execute a function from it in another process. An example of an implementation can be found in Appendix A.2.

Our investigation shows that reading and writing memory of another application is possible for normal applications and new code can be introduced and executed in another application. We therefore assign the following likelihoods to compromising the assets through the I/O operations entry point:

Asset	**Likelihood**
Confidentiality of application data	High
Integrity of application data and flow	High
Availability of application services	High

Table 18: Likelihood of compromising assets through I/O operations on Windows 7

4.4.3 Communications

This section will explore how the communication interfaces can be used on the investigated platforms to compromise the assets of an application. An example for a threat for each of the assets is given below (note that we only focus on IPC interfaces). Those threats will be used to investigate the likelihood for compromising each asset:

- Confidentiality: IPC discloses confidential information.

- Integrity: IPC manipulates program internal data.
- Availability: IPC is blocked.

Android

Our investigation of the communications entry point on Android is based on [27], which includes a very good and detailed descriptions of vulnerabilities in Android IPC. In order to see if these vulnerabilities could be used to manifest the above mentioned threats and would also still work on the latest Android version, we implemented exploits for the relevant vulnerabilities.

In order to intercept IPC and gather confidential information or block IPC, malware can use intent-filters to capture implicit intents (see Section 3.1.3). As mentioned before, implicit intents are messages that do not have a specified recipient, but will be sent to an application that is suited to handle them. One example for an implicit intent is for an application to launch a browser with a predefined URL. This is usually done as follows:

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.hig.no"));
startActivity(intent);
```

Here, an application simply creates an intent to view a URL and does not specify a specific application to handle the intent. Usually, this intent will be processed by the Android browser. But malware can specify its own intent filter to handle intents that request to view a URL that includes the *http* scheme. This is achieved by specifying an intent filter in the manifest file:

```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="http" />
</intent-filter>
```

The effect of this is that the user is presented with a dialog to choose which application should handle the intent. This can be seen in Figure 5.

This dialog can be used to trick the user into choosing the malware application instead of a legitimate application to handle the request and even to set the malware as default action for the intent. This will result in possibly confidential information to leak to the malicious application or in blocking IPC between legitimate applications. This problem can of course be evaded if a developer only uses explicit intents for critical information. That this is, however, a common mistake can be seen from the investigations presented in [27].

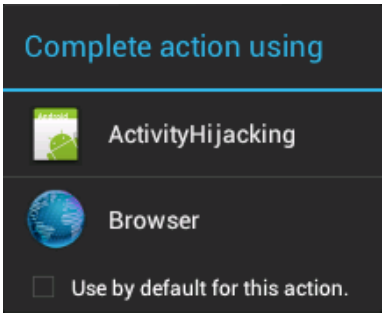

Figure 5: User dialog to choose which application should handle an intent

Intents can also be used to launch internal components of an applications and thereby possibly changing the application's state. We will demonstrate this using an example. The application *Norton Antivirus & Security 2.5.0.398* that we will investigate closer in Chapter 5 declares some of its activities with an intent filter. One example is for an activity called *callfirewall* that can be seen below:

```
<activity android:label="@string/callfirewall" android:name="com.symantec.mobilesecurity.ui.call
firewall.AppMain" android:launchMode="singleTask">
  <intent-filter>
    <action android:name="com.symantec.callfirewall.main_ui" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

On Android, activities are not exported by default, meaning that components of the same application can invoke them but components of other applications can not. However, if an application specifies an intent filter, the application is exported by default. This means that in our example, other applications can send intents to the application and this will result in the activity being launched. An implementation can be seen below:

```
Intent intent = new Intent();
intent.setAction("com.symantec.callfirewall.main_ui");
intent.addCategory("android.intent.category.DEFAULT");
startActivity(intent);
```

This will launch the *callfirewall* activity from another application even though the original application does not seem to provide a way to launch this activity, probably because the activity belongs to a payed upgrade of the application. There are two ways to protect against this problem. First, an application can explicitly declare an activity to not be exported even though an intent filter is present. Second, an application can specify a permission that an application invoking the activity needs to possess. The fact that not even a security critical application like an antivirus scanner does this shows that most developers do not do anything about this problem.

This should also not be required since the Android platform should by default set all activities to not exported. By not doing this, Android is violating the principle of *failsafe defaults*: "The default situation is lack of access, and the protection scheme identifies conditions under which access is permitted." [37].

Our investigation shows that intercepting IPC for compromising confidentiality and service availability is possible on Android if the developer of an application uses implicit intents. Compromising the integrity is possible if the receiving application does not explicitly declare its activities as not exported. We therefore assign the following likelihoods to compromising the assets through the communications entry point:

Asset	**Likelihood**
Confidentiality of application data	Medium
Integrity of application data and flow	Medium
Availability of application services	Medium

Table 19: Likelihood of compromising assets through communication on Android

iOS

As described before, iOS currently has two ways to exchange information between applications (see Section 3.1.3): using URL handlers and using file type handlers.

IPC through file type handlers is meant to exchange files between applications. For example, if a user receives a PDF document in the iOS mail application and wants to open it, he or she is presented with the left dialog in Figure 6, asking what to do with the file. If he or she selects *Open In...*, a list of available applications that can open the file is presented (see Figure 6 on the right). The list is constructed by all applications that have registered a file type handler for this filetype. This is done by setting the *CFBundleDocumentTypes* property in the manifest file of an application. Once the user clicks on the entry of an application, the application is launched and the file is made available to the application. This mechanism could obviously be misused to trick a user into sharing possibly sensitive files with a malicious application.

The other mechanism to share information between application is through URL handlers. This works as follows: when the user clicks on a link, iOS looks at the scheme (e.g. *http://*) of the URL, selects an application to open it and passes the scheme-specific-part of the URL to the application for processing. iOS already has a lot of these scheme-application mappings installed by default, for example to launch the phone application if the user clicks on a link like *tel://01234567*.

All URL schemes that have not been registered by iOS can be registered by other applications [89, p. 112]. The Facebook application for example registers the *fb://* scheme to communicate between the browser and the Facebook application. The fact that this mechanism also works from a web browser and the user is not required to physically click on a link (malware could

Figure 6: File type handlers in iOS

for example use JavaScript to trigger the URL), can already be seen as a security problem. The problem is that a malware author could potentially trigger his own malicious application just by the user visiting a webpage.

But there is another security problem to consider: What happens if two applications register the same URL scheme? According to the Apple documentation: “If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.” [89, p. 112]. This non-deterministic behavior could mean that a malicious application would randomly be able to intercept communication between two other applications by registering the same URL scheme and thereby possibly compromising the applications confidentiality or service availability.

In order to examine this behavior more closely, we performed a small experiment to determine if this behavior is really non-deterministic or if we can control it. Our test was to install the Facebook application and our own application that also registers the *fb://* URL scheme and see what happens if we click on a link with the *fb://* scheme in the browser application.

We suspected that the choice of application would be dependent on the installation order, i.e. the application that is installed last will override the URL handler of the other application. Because of that we did two tests during which we installed the applications in different orders.

In the first test, we did the following:

1. Uninstall both applications and reboot the device.
2. Install the Facebook application.
3. Clicking on the link redirects us to the Facebook application.
4. Install our application.
5. Clicking on the link redirects us to the Facebook application.

6. Reboot the device.
7. Clicking on the link redirects us to our application.

Steps 1–5 seem to indicate that it is not possible to override the URL scheme handler if it is already registered. If we then reboot the device, however, the URL scheme handler seems to have been updated. This lead us to believe that a URL scheme handler will indeed be overwritten, but only after a reboot. In order to test our thesis, we then reversed the order of installing the applications:

1. Uninstall both applications and reboot the device.
2. Install the application.
3. Clicking on the link redirects us to the our application.
4. Install the Facebook application.
5. Clicking on the link redirects us to our application.
6. Reboot the device.
7. Clicking on the link redirects us to our application.

This result refutes our thesis. It seems like our application is always overriding the URL scheme of the Facebook application. Another idea that we had is that the order is determined by the bundle identifiers of the applications that uniquely identify an application on iOS. This could for example be done alphabetically. We therefore changed the bundle identifier of our application a few times and ran the two tests again, but none of the changes had any influence on the outcome.

In order to make this attack reliable, a closer look of the Apple implementation in form of reverse engineering would be required. We did, however, not go into more details here.

As we have seen, it is possible to trigger other applications in iOS by using file type handlers and URL schemes. This can be used to inject files and data into another application. What makes this problem worse, is that it can be triggered remotely from a webpage. This is, however only a problem if the receiving applications contains a vulnerability when processing the received data or when it enables other applications to use IPC to control it. One example vulnerability was described for the Skype application, which can be triggered by a URL handler to place a call without asking the user first[5]. That way, it is possible for malware to call premium numbers for immediate profit. We verified this by creating a webpage that uses JavaScript to open a Skype link to call a phone number when the user visits the page.

Our investigation shows that intercepting IPC for compromising confidentiality and service avail-

[5] `http://broadcast.oreilly.com/2010/11/insecure-handling-of-url-schem.html`

ability is possible on iOS. Compromising the integrity is only possible if the receiving application of IPC contains a vulnerability. We therefore assign the following likelihoods to compromising the assets through the communications entry point:

Asset	Likelihood
Confidentiality of application data	High
Integrity of application data and flow	Medium
Availability of application services	High

Table 20: Likelihood of compromising assets through communication on iOS

Windows Phone 7

Windows Phone 7 does not support IPC [12, p. 61], so we could not investigate possible problems with it. We therefore assign the following likelihoods to compromising the assets through the communications entry point:

Asset	Likelihood
Confidentiality of application data	Low
Integrity of application data and flow	Low
Availability of application services	Low

Table 21: Likelihood of compromising assets through communication on Windows Phone 7

Windows 7

Windows 7 offers a lot of ways for applications to perform IPC. We will look at only one here: Windows applications that contain a graphical user interface can communicate via so called *Window Messages*, which are part of the Win32 API. Those messages can be used for example to change the user interface of another application. They can also be used to read the content of an element of the user interface. We performed a test to see if we could read the text that is typed in the *Notepad* application from our application. Our test program can be seen below:

```
BOOL readTextFromNotepad(char * text, int len)
{
  HWND handleNotepad;
  HWND handleTextfield;

  // Find handle of Notepad main window
  handleNotepad = FindWindowA("Notepad", NULL);

  if(handleNotepad != 0)
  {
    // Find handle of text field in Notepad
    handleTextfield = FindWindowExA(handleNotepad, NULL, "Edit", NULL);

    if(handleTextfield != 0)
    {
```

```
      // Read text from text field
      if(SendMessageA(handleTextfield, WM_GETTEXT, len, (LPARAM)text) == 0)
        return FALSE;
    }
  }

  return TRUE;
}
```

In order to read the text from notepad, we first need the so called *handle* of the text field in notepad. Every graphical element in a Windows that is based on the Win32 API has a handle that uniquely identifies it. We get the handle for the Notepad text field by first searching for the Notepad window and then searching for an *Edit* field (the text field) in that window. We can then send the message *WM_GETTEXT* to the field asking for its text. After that, we have successfully retrieved the text from the text field in another application. Of course, the Notepad text field s not that critical. The same approach would, however, also be possible for password fields or other sensitive windows.

We then thought about the limitations of this approach. Does this also work for example with applications that run with elevated privileges? A quick test showed that this is indeed the case. What did not work, however, was to try to read text from Windows that were on a different desktop, e.g. on the desktop of another user that was logged in.

After that, we used the same approach that was used to read text from Notepad to see if we could also write to Notepad. We used the same approach as above but this time, we sent the message *WM_SETTEXT* to the text-field. This resulted in the text in Notepad being changed. As with reading, this also worked with privileged windows but not on other desktops.

In order to block IPC in Windows 7, so called *subclassing* can be used. When a graphical Win32 element receives a message, a predefined message handling procedure is invoked that processes the incoming message. If an attacker wants to intercept these messages, he can redefine the message handling procedure for a given window and simply block unwanted messages e.g. the message for closing the window *WM_CLOSE* and forward the other messages to the original message handling procedure. This can be seen in the code below:

```
LRESULT CALLBACK evilProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  if(uMsg == WM_CLOSE)
    return 0;
  else
    return CallWindowProc(originalProc, hwnd, uMsg, wParam, lParam);
}
```

Exchanging message handling procedures for a given window handle is achieved using the Win32 API function *SetWindowLong*. The only problem is that this only works if the code that changes

the window handling procedure runs in the same process as the application that is the target of the attack. A simple way to get around this is to inject a DLL into the target process and perform the exchange from there (see Section 4.4.2 on how to inject a DLL into a target process).

Our investigation shows that IPC can be used to compromise the confidentiality and integrity of an application. IPC can also be blocked by performing subclassing. We therefore assign the following likelihoods to compromising the assets through the communications entry point:

Asset	**Likelihood**
Confidentiality of application data	High
Integrity of application data and flow	High
Availability of application services	High

Table 22: Likelihood of compromising assets through communication on Windows 7

4.4.4 User Interface

This section will explore how the user interfaces can be used on the investigated platforms to compromise the assets of an application. An example for a threat for each of the assets is given below (note that we only focus on user input). We did, however, only investigate the threat to availability. We do, however also list the threats to the other assets here for further research. Those threats will be used to investigate the likelihood for compromising each asset:

- Confidentiality: User input is intercepted.
- Integrity: User input is simulated.
- Availability: User input is blocked.

Android

On Android, the platform does not seem to offer a way to be notified about global input events. It does, however, offer the possibility to install different software keyboards. These keyboards obviously need access to the inputs the user makes. In order to test this, we implemented our own software keyboard based on the *SoftKeyboard* example in the Android SDK. We found that we are notified about all the input that the user makes on our software keyboard through the interface *KeyboardView.OnKeyboardActionListener*. We are notified about key press and key release through the callback methods *onPress* and *onRelease* so we can easily log the inputs to a file for example. In order for the keyboard to be recognized by Android, it needs the *BIND_INPUT_METHOD* permission. This causes the user to be alerted when the keyboard is selected as one of the input methods as can be seen in Figure 7.

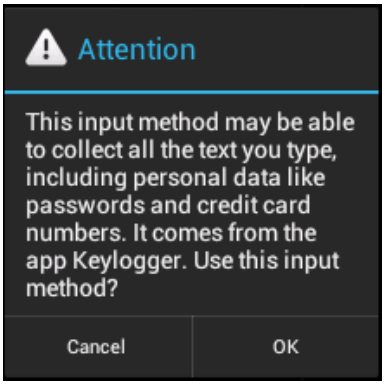

Figure 7: Dialog displayed to the user when installing a custom input method

A user will most likely get suspicious by this dialog. Especially if the application is called *Keylogger*. But if the keyboard seems to include interesting features, it might very well be that the user will accept the dialog.

Our investigation shows that the user interface can be used to compromise the confidentiality of an application on Android. This is achieved using a keylogger in case the user can be tricked into activating it. We therefore assign the following likelihood to compromising the confidentiality asset through the user interface entry point:

Asset	**Likelihood**
Confidentiality of application data	Medium

Table 23: Likelihood of compromising assets through the user interface on Android

iOS

On iOS, it is not possible to be informed about global input events without a jailbreak. It is also not possible to install different keyboards without a jailbreak. The author of [13], however, has found that on iOS, in order to improve the auto correction feature, most of the words that a user types are logged to a file in the folder */User/Library/Keyboard/*. The exact filename depends on the language. For the default English language, the file is called *dynamic-text.dat*. This file is readable to a normal application (as we have verified with our filesystem exploration tool described in Section 4.4.1). This file basically consists of a list of words. The author of [13] also has released a tool called *SpyPhone*[6] that displays the words contained in the file.

To verify that this vulnerability still exists in he newest iOS version, we ran this program on one of our daily used phones. The result was that we could see a lot of private information that was contained in messages that were written before. This file does, however not contain any

[6] `https://github.com/nst/spyphone/`

passwords, that we typed before or any numbers that we typed, only words. In order to test this closer, we first deleted this file that we call *keyboard cache*. We then tried to type some text in different applications. We observed that a lot of words will end up in this file. Simple words like *and, or, while, then* etc. were not recorded. We then wrote our own application which contained a password field and a normal text field. Passwords typed in the password field never ended up in the file, while most of the words typed in the normal text field were recorded. The text-fields have the option to deactivate auto correction. This will effectively prevent words from being recorded. Other UI elements, for example the *UIWebView*, which is used to display web pages does not offer this and input made there still ends up the file.

Our investigation shows that the user interface can be used to compromise the confidentiality of an application on iOS by reading the words that have been saved in the keyboard cache. This will not include information like passwords or credit card numbers, but can still be used for example to extract information from confidential emails. We therefore assign the following likelihood to compromising the confidentiality asset through the user interface entry point:

Asset	**Likelihood**
Confidentiality of application data	Medium

Table 24: Likelihood of compromising assets through the user interface on iOS

Windows Phone 7

On Windows Phone 7, we could not find a way to be notified about global input events and there is no way to install a custom keyboard and input events are not logged to a publicly readable file. We therefore assign the following likelihoods to compromising the assets through the user interface entry point:

Asset	**Likelihood**
Confidentiality of application data	Low

Table 25: Likelihood of compromising assets through the user interface on Windows Phone 7

Windows 7

In Windows 7, there are multiple ways to implement a keylogger. We decided to try the following way as described for example in [90]: The Windows API function *SetWindowsHookEx* allows us to inject a DLL in running applications and have that application call a function from that DLL when a specific event occurs, for example a keyboard event. We tell the *SetWindowsHookEx* function to call the function *KeyboardEventOccured* in our DLL *Library.dll* when a keyboard event (*WH_KEYBOARD_LL*) occurs. This can be seen in the code below:

```
#include <windows.h>
#include <stdio.h>
```

```
int main(void)
{
  // Load library and get address of function
  HMODULE dll = LoadLibraryA("Library.dll");
  FARPROC proc = GetProcAddress(dll, "KeyboardEventOccured");

  // Create hook
  HHOOK hook = SetWindowsHookEx(WH_KEYBOARD_LL, (HOOKPROC) proc, dll, 0);

  if(hook == NULL)
    return -1;

  // Process messages
  while(GetMessage(0,0, 0,0))
  {}

  // Delete the hook
  UnhookWindowsHookEx(hook);

  return 0;
}
```

The implementation of the DLL basically just processes events if a key was pressed, then decodes the key that was pressed and writes it to a log file:

```
#include <windows.h>
#include <stdio.h>


BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{
  return TRUE;
}

__declspec(dllexport) LRESULT KeyboardEventOccured(int nCode, WPARAM wParam, LPARAM lParam)
{
  // MSDN: If nCode is less than zero, the hook procedure must pass the message to the CallNextHookEx
  //       function without further processing and should return the value returned by CallNextHookEx.
  if(nCode < 0)
    return CallNextHookEx(0, nCode, wParam, lParam);

  // Did we receive an event about a key being pressed?
  if(nCode == HC_ACTION)
  {
    if(wParam == WM_KEYDOWN)
    {
      KBDLLHOOKSTRUCT *hook = (KBDLLHOOKSTRUCT*)lParam;
      FILE * fp = fopen("log.txt", "a+");
      char state[256];
      int character;

      // Get and decode the key being pressed
      GetKeyboardState((PBYTE) state);
      ToAsciiEx(hook->vkCode, hook->scanCode, (PBYTE)state, (WORD*)&character, 0,
        GetKeyboardLayout(0));

      // Write the key to a file
```

```
      fputc(character, fp);
      fclose(fp);

      // Process the event normally
      return CallNextHookEx(0, nCode, wParam, lParam);
    }
  }
  return CallNextHookEx(0, nCode, wParam, lParam);
}
```

Our investigation shows that the user interface can be used to compromise the confidentiality of an application on Windows 7 by implementing a keylogger. We therefore assign the following likelihood to compromising the confidentiality asset through the user interface entry point:

Asset	**Likelihood**
Confidentiality of application data	High

Table 26: Likelihood of compromising assets through the user interface on Windows 7

4.4.5 Program Execution

This section will explore how program execution can be used on the investigated platforms to compromise the assets of an application. An example for a threat for each of the assets is given below. We did, however, only investigate the threat to availability. We do, however also list the threats to the other assets here for further research. Those threats will be used to investigate the likelihood for compromising each asset:

- Confidentiality: Information about running application is disclosed.
- Integrity: Application is launched by other application.
- Availability: Application is terminated by other application.

Android

On Android, an application can terminate other applications if it possesses the *KILL_BACKGROUND_PROCESSES* permission, which is a permission that is not considered dangerous and the user will therefore not be prompted during installation. Terminating processes is only possible when the process is currently not in the foreground. Terminating a process can for example be achieved by calling the method *killBackgroundProcesses* of the *ActivityManager* class.

In order to get around the limitation that the application may not be in the foreground, we can move the application in the background before we terminate it. This can be achieved by

calling the method *moveTaskToFront* of the *ActivityManager* class. However, this requires the *REORDER_TASKS* permission which is considered dangerous.

Our investigation shows that the program execution interface can be used to compromise the availability of an application on Android by terminating it with some limitations in case the application is in the foreground. We therefore assign the following likelihood to compromising the availability asset through the program execution entry point:

Asset	**Likelihood**
Availability of application services	Medium

Table 27: Likelihood of compromising assets through program execution on Android

iOS

As explained in Section 4.4.2, we can stop an application on iOS by using the ptrace interface. After we have attached to an application, we should be able to call ptrace with the parameter PT_KILL to terminate the application. However, our tests show that this will not just terminate the application, but it will also make the application crash in case the user tries to restart it. This can only be resolved by rebooting the device and can therefore be seen as a very effective way to compromise the availability of an application. Our first suspicion was that this is a feature that is only enabled on iOS devices that are used for development, since this is a debugging interface. So we tried the experiment on a device that was not used for development before and we got the same results. This is a problem that Apple should resolve. The ptrace interface is something that is only needed for debugging purposes and should not be enabled on all devices.

Our investigation shows that the program execution interface can be used to compromise the availability of an application on iOS by using the ptrace interface. We therefore assign the following likelihood to compromising the availability asset through the program execution entry point:

Asset	**Likelihood**
Availability of application services	High

Table 28: Likelihood of compromising assets through program execution on iOS

Windows Phone 7

On Windows Phone 7, we could not find a way to terminate another application since the Windows Phone 7 API does not include this functionality. We therefore assign the following likelihood to compromising the availability asset through the program execution entry point:

Asset	Likelihood
Availability of application services	Low

Table 29: Likelihood of compromising assets through program execution on Windows Phone 7

Windows 7

Windows 7 offers a lot of ways for applications to terminate another application. We will look at only one here: The Windows Win32 API offers the function *TerminateProcess* that can easily terminate a process when the process id is known. We have implemented a small test program that terminates another process by its process id. This program can be found below:

```
#include <windows.h>

void terminateProcess(unsigned int processId)
{
  // Open process
  HANDLE process = OpenProcess(PROCESS_TERMINATE, TRUE, processId);

  if(process == 0)
    return;

  // Terminate process
  TerminateProcess(process, 0);

  // Clean up
  CloseHandle(process);
}
```

As with reading and writing memory, terminating processes of other users is only possible if the *SeDebugPrivilege* privilege is activated (see Section 4.4.2).

Our investigation shows that this program is capable of terminating processes that have been started by the same user as the user that starts the test application. It can therefore compromise the availability of an application. We therefore assign the following likelihood to compromising the availability asset through the program execution entry point:

Asset	Likelihood
Availability of application services	High

Table 30: Likelihood of compromising assets through communication on Windows 7

4.5 Discussion

In this chapter, we have analyzed the question “What are the limitations of malware running on mobile platforms compared to desktop platforms?”. We have done so by investigating threats that

malware poses to other applications running on the same platform. The threats have been found by using an approach based on [9] where we first determined which assets and entry points an application has and then by investigating how each entry point could affect each asset resulting in a threat for the asset. We then performed experiments on each platform to determine the likelihood of manifesting each threat. We have to note that while we claim that the entry points and assets are complete, we do not claim that our experiments on the likelihood of the threats being manifested is. This is due to the fact that modeling each possible interaction between entry points and assets for four platforms that are mostly proprietary would take much more time than we had available. It is therefore possible that there is another approach to use an entry point to compromise an asset that we have not tried and that would result in a higher likelihood. The result of these experiments can be seen in Table 31. For the medium likelihoods we also give the restrictions for compromising the assets (see Section 4.4) as a footnote.

		Likelihood of threat manifestation			
Entry Point	**Asset**	**Android**	**iOS**	**WP 7**	**Win 7**
Filesystem	Confidentiality	High	Low	Low	High
	Integrity	Low	Low	Low	Medium[1]
	Availability	Low	Low	Low	Medium[1]
I/O Operations	Confidentiality	Low	Low	Low	High
	Integrity	Low	Low	Low	High
	Availability	Low	Low	Low	High
Communications	Confidentiality	Medium[2]	High	Low	High
	Integrity	Medium[2]	Medium[2]	Low	High
	Availability	Medium[2]	High	Low	High
User interface	Confidentiality	Medium[1]	Medium[3]	Low	High
	Integrity	-	-	-	-
	Availability	-	-	-	-
Program execution	Confidentiality	-	-	-	-
	Integrity	-	-	-	-
	Availability	Medium[1]	High	Low	High

Table 31: Summary of likelihood evaluation of investigated entry points and assets

As mentioned before, time constraints are the reason for not all threats being investigated. But from the acquired results, we can already see an indication of multiple things. First, on Windows 7, most of the investigated threats can be manifested without any restrictions. The remaining threats can be manifested with some restrictions. The reason for that is mostly that applications that are running as the same user are not isolated from each other, i.e. there is no sandboxing in Windows 7. This is in strong contrast to Windows Phone 7, which is also developed by Microsoft, where none of the threats could be manifested, not even when some restrictions were given. Android and iOS are somewhere between that. While there are many threats that could not be manifested on these platforms, there are still some threats that could be. We can see from this that security on mobile platforms was increased compared to traditional desktop platforms, which is

[1] The user has to be deceived into performing an unwanted action.
[2] The developer of an application has to make a mistake that is not prevented by the platform.
[3] Manifesting the threat only gives limited access to the resource.

also what we can conclude from the description of the protection mechanisms that the platforms offer in Section 3.2. We have to mention that Table 31 should not be used to determine which platform is the most secure since it is tailored towards malware attacking other applications and does therefore not say anything about other security features, for example encryption of the filesystem or memory protection.

The fact that we did not find any threats on the Windows Phone 7 platform that we could manifest is surprising. We suspect this to have two reasons: First, as we have seen in Section 3.2, Windows Phone 7 has a very strict security model that puts a lot of restrictions on developers, effectively limiting the attack surface of the platform. The other reason we see for this is that Windows Phone 7 is a relatively new platform that still lacks some basic features. While one might argue that some features are not included on purpose for security reasons, we believe that this is not necessarily the case. In order to compete with other platforms, Microsoft will have to add more capabilities for third party applications. A similar example can be seen from iOS. While the first versions of iOS did not offer the possibility to have multitasking of third party applications, allegedly to save power, this was added later to compete with the Android platform that did offer this functionality from the beginning.

Apart from Windows Phone 7, there are still threats on mobile platforms that are not mitigated by the platforms themselves. We will therefore investigate the second research question in the next chapter to determine whether traditional third party security solutions can be used to mitigate the remaining threats.

5 Security Solutions on Mobile Platforms

This chapter investigates the second research question: “What are the limitations of traditional security solutions running on mobile platforms?”. We will do so by first reviewing the market for currently available and relevant security solutions. We will then categorize these solutions for further investigation. Then we will examine samples from each of the categories on each of the mobile platforms to determine how they work and what their limitations are.

5.1 Review of currently existing Solutions

In order to get an overview of currently existing security solutions, we searched the central distribution platforms for Android, iOS and Windows Phone 7: the Google Play Store, the App Store and the Windows Phone Marketplace. We searched for general security keywords and also for well known manufacturers from the desktop security world. Used keywords can be found below:

- Protection
- Security
- Secure
- Shield
- Virus
- Malware
- Antivirus
- Firewall
- IDS
- Intrusion Detection
- Symantec
- Kaspersky
- F-Secure
- Avira
- Sophos
- AVG
- Norton
- McAfee

We found two categories of security solutions: Virus scanners and firewalls. The third category, intrusion detection systems, that we discussed in Section 3.4, could not be found.

Since this work has a limited time-frame, we did not spent time analyzing every application that we found. Instead, we chose a representative sample from each of the platforms for each of the categories. These samples were chosen by searching for “antivirus” and “firewall” in the respective stores and choosing the five most popular (determined by user ratings) applications from the search, were available. Below, we will describe which applications we chose.

5.1.1 Android

On Android, we got a lot of results for each of the categories. The most popular applications for each of these categories as of April 30th, 2012 are:

Firewalls:

- Avast Mobile Security 1.0.2129
- Droidwall 1.5.7
- Network Firewall 1.2.2
- Root Firewall 0.93
- Internet Firewall 1.10.27

Virus Scanners:

- AVG Antivirus Free 2.10.1
- Avast Mobile Security 1.0.2129
- Norton Antivirus & Security 2.5.0.398
- Antivirus Free 1.3.4
- Lookout Security & Antivirus 7.8-96e2110

5.1.2 iOS

On iOS, the situation was much more sparse. As of April 30th, 2012, we were only able to find one application with virus scanner capabilities. For firewalls, we were unable to find any application. In order to find more applications, we therefore turned to Cydia[1], which is an App Store alternative that is installed on jailbroken iOS devices and does not have any of the restrictions that the App Store has when it comes to excluding applications that do not comply with Apple policies. In Cydia, we could not find any virus scanners, but we found a firewall application. This could be an indication that virus scanners on iOS are not considered very useful yet. The found applications are listed below:

Firewalls:

- Firewall iP 2.04-1

Virus Scanners:

- VirusBarrier 1.3

5.1.3 Windows Phone 7

For Windows Phone 7, we could not find any relevant security solutions in the Windows Phone Marketplace as of April 30th, 2012. We therefore also turned to unofficial places. For Windows Phone 7, there is also an unofficial store, called *Bazzar*[2] that can be installed on unlocked phones. We also searched this store for the above mentioned keywords, but were also unable to find any relevant security solutions. We therefore could not investigate any application on Windows Phone 7.

[1] `http://cydia.saurik.com/store/`
[2] `http://wp-bazaar.com`

5.2 Investigation of Firewalls

In this section, we investigate the above mentioned firewall solutions on Android and iOS in order to determine how they work and what their limitations are.

Android

The first thing we noticed about all of the firewall solutions for Android is that all of them require a rooted device. This makes sense of course since according to the sandbox security model of Android, no normal application should be able to influence another application.

In order to investigate the applications, we first installed them on a device to explore their functionality. But in order to gain a deeper understanding of the application's inner workings, we applied reverse engineering. We used the tool *dex2jar 0.0.9.8*[3] to transform Dalvik Executables to Java bytecode and then used the tool *JD 0.6.0*[4] to decompile Java bytecode to Java code.

Our analysis of the applications shows that all of these applications rely on the packet filter *Netfilter* that is integrated in the Linux kernel. This is the reason why those firewalls do not work on each device: Phone manufacturers usually compile their own version of the Linux kernel for their device and use a custom configuration to do so, meaning that they do not necessarily include Netfilter in their kernels. Netfilter is controlled from user-space using the commandline tool *iptables*. This tool requires root privileges to control Netfilter, which is the reason why these tools require a rooted phone.

When analyzing the firewall rules used by the applications, we discovered all of the applications except *Network Firewall* are based on an open source project called *DroidWall*[5]. This is probably one of the reasons that all of the firewall solutions offer the same functionality: They allow to define for each application that has the permission to access the Internet, if it is allowed to do so using Wi-Fi and 3G networks. Distinguishing between Wi-Fi and 3G networks makes sense since this can be used to reduce costs by limiting data intensive applications on 3G networks. An example from the application DroidWall can be found in Figure 8. The application *Droidwall* additionally offers the possibility to define custom iptables scripts in order to define rules apart from that. This is, however, a feature that can only be used by advanced users with knowledge of iptables.

[3] `http://code.google.com/p/dex2jar/`
[4] `http://java.decompiler.free.fr/`
[5] `http://code.google.com/p/droidwall/`

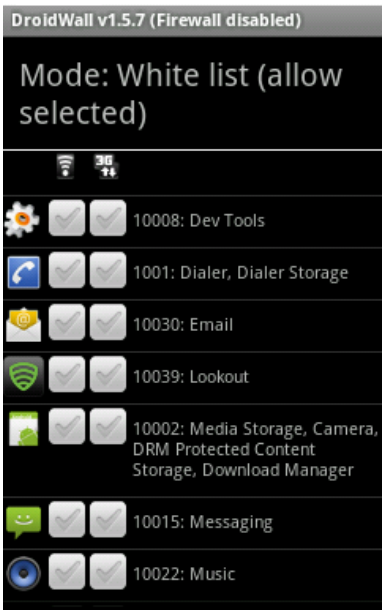

Figure 8: The Android firewall application DroidWall

iOS

The application *Firewall iP* requires a jailbroken phone. It notifies a user when an application wants to open a network connection. The user then has the choice to decide how to proceed. He or she can for example allow the connection once or define a permanent rule. This dialog can be seen in Figure 9.

In order to determine how Firewall iP works, we first took a look at the description of the application: "It hooks into the applications and will warn you if the app wants to establish a connection [...]"[6]. Unlike the Android applications, which use a packet filter in kernel-space, this application seems to implement the firewall functionality using hooking.

To determine how this is achieved, we took a closer look at the application. Apart from the actual application, Firewall iP also installs a *MobileSubstrate*[7] library called *ZFirewall.dylib*. MobileSubstrate is a hooking framework for iOS devices that have been jailbroken. It allows to hook every function call and objective-C message invocation that an application makes. It is a framework that allows developers to write extensions that hook specific functions/messages. *ZFirewall.dylib* is one of these extensions. The extensions are loaded by MobileSubstrate into running processes based on a filter that is specified in a plist file that accompanies the extension. In the case of Firewall iP, the extension is loaded into every application that links against the *UIKit* or *ImageIO* framework, meaning that the extension is loaded into every application that has a graphical user

[6] `http://yllier.net/FirewalliP/FiPDepiction.html`
[7] `http://svn.saurik.com/repos/menes/trunk/mobilesubstrate/`

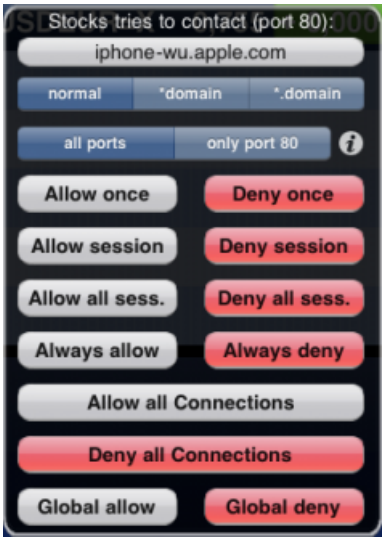

Figure 9: The iOS firewall application Firewall iP

interface since they are all based on UIKit.

Using the tool *otool* on Mac OS X, which can be used to acquire information about Mach-O binaries (the executable format used in iOS) and is also able to decompile them, we took a closer look at the MobileSubstrate extension. We found that it hooks multiple functions. Apart from some functionality to get informed about user interface changes, it hooks the two system calls *connect* and *sendto*, which are used to open socket connections and send data and can therefore effectively influence the network connectivity of an application. We found the same approach in a very early stage open source iOS firewall project called *mobile-firewall*[8]. The disassembly below, which was generated using *IDA Pro 6.1*, shows how Firewall iP performs the hooking of the system calls:

```
__text:0000F6DC                 LDR     R0, =(_connect_ptr - 0xF6E6)
__text:0000F6DE                 LDR     R2, =(_orig__connect - 0xF6E8)
__text:0000F6E0                 LDR     R1, =(__Z10my_connectiPK8sockaddrj+1 - 0xF6EC)
__text:0000F6E2                 ADD     R0, PC
__text:0000F6E4                 ADD     R2, PC
__text:0000F6E6                 LDR     R0, [R0]
__text:0000F6E8                 ADD     R1, PC
__text:0000F6EA                 BLX     _MSHookFunction
__text:0000F6EE                 LDR     R0, =(_sendto_ptr - 0xF6F8)
__text:0000F6F0                 LDR     R2, =(_orig__sendto - 0xF6FA)
__text:0000F6F2                 LDR     R1, =(__Z9my_sendtoiPKvmiPK8sockaddrj+1 - 0xF6FE)
__text:0000F6F4                 ADD     R0, PC
__text:0000F6F6                 ADD     R2, PC
__text:0000F6F8                 LDR     R0, [R0]
__text:0000F6FA                 ADD     R1, PC
__text:0000F6FC                 BLX     _MSHookFunction
```

[8] http://code.google.com/p/mobile-firewall

Another way to implement a firewall on jailbroken iOS devices – that is currently not used by any application – would be to take a similar approach as the applications on Android do. The XNU kernel of Mac OS X contains a packet filter called *ipfw*, which it inherited from FreeBSD. From user-space, this packet filter can be controlled via a command-line tool with the same name. Porting this tool to iOS and creating a graphical user interface for it would result in a solution similar to the solutions on Android. Our investigations show, that this approach can not easily be implemented since the iOS kernel currently does not include the *ipfw* functionality. As opposed to Android, compiling a new kernel is not possible since the source code for the XNU kernel used in iOS is not fully available.

5.3 Investigation of Virus Scanners

Android

To determine the functionality of the virus scanners on Android, we first installed them on a device. The first thing we checked, was *when* and *what* they scan. When it comes to *what* these scanners scan, we found that they either scan the installed applications or certain writable locations of the filesystem, first and foremost the locations where external media is mounted. The installation directories of applications can be accessed by every application as we have seen in Section 4.4.1. The same holds for reading files on the locations where external media is mounted (usually under */mnt/*). When it comes to *when* scans are performed, Android offers a lot of possibilities that are used by the investigated scanners. An overview is given in Table 32.

Trigger	**Description**
On demand	The user makes the application perform a scan.
New application is installed	Applications on Android can be notified when a new application is installed.
New medium is mounted	Applications on Android can get notified when a new external medium is mounted.
At predefined times	The user specifies a time at which scans are performed. E.g. every Monday night a 3am.
On access	Applications on Android can register for being able to handle certain filetypes. The application will then be presented as a choice for opening the file when the user opens it. This can be used to scan files before they are opened somewhere else.
Continuously	Continuously scanning is the most energy intensive method, but can e.g. detect the download of a malicious application and warn about it even before it is installed.

Table 32: Different triggers of virus scanners on Android

Another interesting characteristic of virus scanners on mobile devices is the question *where* they perform scanning for viruses. All the virus scanners, that we investigated seem to perform the scanning on the device itself. Another possibility would be to perform scanning in the cloud to

reduce battery consumption (see Section 3.4.1). We searched the Google Play Store for virus scanners that scan for malware in the cloud. At least two applications can be found that advertise with scanning in the cloud: *Bitdefender Mobile Security* and *SecureBrain Antivirus*. We did, however, not investigate these solutions any further.

The last thing, we were interested in, was *what* these scanners scan for. In order to determine this, we first relied on reverse engineering of the respective applications. As with the firewalls on Android, we used *dex2jar* and *JD* to decompile Java code. Since some of the scanners also use native code, we also used an ARM version of the tool *objdump 2.20.1.20100303* from the Android NDK to acquire information about these libraries and decompile them.

We first looked at the application *Antivirus Free* since it is by far the smallest application. This application contains a SQLite 3 database that contains entries for known viruses and is periodically updated via the Internet. At the time of analysis, this database contained 64 entries. Each entry consists of a package name, a name and a description. The package name seems to be encoded in some way. If we look at the way, these entries are loaded by the application from the database file, we find the class *com.zrgiu.antivirus.DBManager*, which encrypts the package name using the class *com.zrgiu.utils.AesEncrypt*.

The AesEncrypt class basically first generates a key based on the first parameter (which as we have seen in the *DBManager* class), is a key that is stored in the application preferences. The exact details of this generation method will not be discussed here, but can be found in the original application. Using this key, the package name is encrypted with AES and then encoded using Base64. This can be seen from the reconstructed source code below:

```
public static String decrypt(String paramString1, String paramString2)
{
  try
  {
    Key localKey = generateKey(paramString1);
    Cipher localCipher = Cipher.getInstance("AES");
    localCipher.init(2, localKey);
    str = new String(localCipher.doFinal(Base64.decode(paramString2)));
    return str;
  }
  catch (Exception localException)
  {
  while (true)
    String str = null;
  }
}
```

When the application scans, it compares the package name of the application it currently scans with all the entries in the database. Obviously, this mechanism is not very strong. Only comparing package names makes it very easy for malware authors to evade detection. The only thing a malware author needs to do is change the package name of the application. All the application's

functionality can remain the same.

The next application, we investigated, was *AVG Antivirus Free*. This application stores all detection settings in an XML file called *av.xml* and gives insight into what and how scanning is performed. Two interesting settings can be found: *package_key* and *media_key*. We suspect these settings to be a blacklist of malicious package names and filenames. However, these values seem to be encoded somehow. A check on where these values are read brings us to the class *com.antivirus.core.EngineSettings*. Here, a decryption method is provided to read the values from the XML file. We do, however, have some problems to reverse engineer this method because there are errors in the disassembly.

We therefore took another approach to decrypt the values: We disassembled the original application using the tool *apktool 1.4.3*[9] in a so called *smali* format. Since this tool is also able to reassemble modified smali files, we were able to inject some code into the original application that prints the deciphered values directly from memory. That way, we were able to obtain the plain text version of the above mentioned settings.

Our suspicion was correct. The setting *package_key* contains a list of currently 316 package names that we can relate to known Android malware. The setting *media_key* contains a list of currently 42 filenames. Almost all of these files are *.exe* or *.bat* files, which are used on Windows, but not on Android.

We then searched for uses of these settings in the application. The setting *package_key* is used in the class *com.antivirus.core.scanners.i*. Here we can see, that some string matching is made on installed package names. The setting *media_key* is used in the class *com.antivirus.core.scanners.h*. From here, it is used as a parameter to a JNI function in the class *com.antivirus.core.scanners.FileScannerJniWrapper* which calls a native library called *libFileScanner.so* with the list of files. We decompiled the library using *objdump* and discovered that it simply scans the filesystem for the files in the blacklist.

This is application only slightly more advanced than Antivirus Free. It not just compares package names. It also scans the filesystem for suspicious files, however, only for some Windows malware. We also found a setting called *hueristic_active*:

```
public static boolean getHuristicActiveKey()
{
  return a.getBoolean("hueristic_active", false);
}
```

This setting is used nowhere in the application, but could be a hint that AVG intends to implement some sort of heuristic detection. However, since they even have problems to spell heuristic, this

[9] http://code.google.com/p/android-apktool/

could still take some time.

The next application, that we investigated, was *Avast Mobile Security*. The APK file of the application contains another APK file called *avast-android-vps-release.apk*. This file contains some more DEX classes and four files, which we suspect to be virus definition files: *db_dex.dat*, *db_dex.nam*, *db_elfa.dat* and *db_elfa.nam*. The files are in a binary format that we do not understand, so we had to investigate further. The class *com.avast.android.mobilesecurity.engine.internal.VpsInterface* loads the additional classes during runtime and then calls methods in the class *com.avast.android.mobilesecurity.vps.Interface*. This class reads the above mentioned definition files files and passes them as initialization data to a native library called *libavast-vps.so*, which also comes with the bundled APK file.

When the application performs a scan, it can scan installed applications and the memory card. It enumerates the APK files of all non-system applications as well as all files from the memory card. It then passes the contents of all files from the class *com.avast.android.mobilesecurity.vps.Interface* to function *preScanJni* in the native library for initializing a scan.

In the native library, the application first determines the type of file that was passed from Java. It distinguishes between four file types: DEX files, ZIP files, ELF files and the EICAR file, which is a test file that is supposed to be detected by every virus scanner. The type is then passed back to Java. If the file type is a ZIP archive (APK files are also ZIP files), it then extracts all DEX and ELF (the file format used for native libraries and executables) files and passes their content to the native *preScanJni* function as well.

After that, the Java code calls the native function *scanJni* with the content of each file and its type. It will also call this function for APK files together with other information about the application like the package name. If the function was called with a package name, it then first compares the package name against a black list of four package names that are hard coded in the library. The list of package names can be seen below:

- com.ANTIVIRUS.TESTFILE
- com.mj.utils.MJReceiver
- com.mj.iCalendar.SmsReceiver
- com.RZStudio.iMine.SmsReceiver

If the package names match, malware was detected. Otherwise it checks the type of the file. If the file type is EICAR, the function then calls *scan_eicar*, which does a basic memory compare with the content of the EICAR test file and alerts in case they match. If the file type is DEX or ELF,

it calls *scan_dex* or *scan_elfa* respectively. This behavior can be seen in our manual reconstruction of the source code of the *scan* function called from *scanJni* in Appendix A.3.

Because of time constraints, we did not investigate these functions further, but we suspect that *scan_dex* and *scan_elfa* use the definition files passed to the library earlier to scan for malware in the DEX and ELF format.

From what we have seen, *Avast Mobile Security* is much more advanced than the previously investigated applications in that it compares actual file content and not just filenames or package names and also extracts archives. It also only scans for files that are relevant to the platform and not for example for Windows malware.

We stopped our investigation here since reverse engineering the previous applications was very time consuming. A quick look at the other applications *Norton Antivirus & Security* and *Lookout Security & Antivirus*, however, makes us believe, that they use similar mechanisms as *Avast Mobile Security* since they both include files that we suspect to be malware definition files. Future work could be to determine, if these applications include methods that we did not find in the investigated applications.

To summarize, we found the following criteria, of *how* virus scanners on Android currently scan for viruses:

- Compare package names with black list.
- Compare file names with black list.
- Compare contents of executable files with malware definition files.

iOS

On iOS, we investigated the application *VirusBarrier*[10]. It is advertised as a virus scanner. A closer look, however, shows that its functionality is pretty limited. Unlike on Android, scans can not be performed on access or at regular intervals. The application also only supports scanning of files that are transfered between applications using the iOS file type handlers (see Section 3.1.3). An example can be seen in Figure 10. Transferring files happens for example, when the user receives a file via Email and wants to open it in another application. The reason for this limitation lies in the iOS sandbox. As described in Section 4.4.1, an application's access to the filesystem is very limited. This is also the reason why the application does not support scanning of installed applications: it does not have permissions to read them. It will also not be able to be informed

[10] `http://itunes.apple.com/app/virusbarrier/id436111378`

about a new application being installed since iOS does not provide a notification mechanism for newly installed applications. Since VirusBarrier can not scan on the device, it offers the possibility to scan files on remote servers e.g. using Dropbox or FTP.

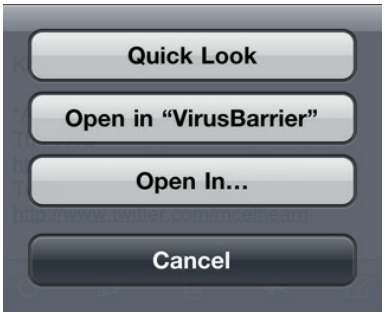

Figure 10: The iOS virus scanner VirusBarrier

Apart from that, the application does not scan for threats for iOS, but only for Mac, Windows and Unix malware. This is something that could be improved since there are actually weaknesses in iOS that have been exploited in the past like the *Malformed CFF Vulnerability (CVE-2010-1797)*, which was a vulnerability in the processing of fonts in PDF documents and was used to jailbreak devices by simply visiting a web page. Since there are vulnerabilities that could potentially threaten iOS users, e.g. by sending files via email or spreading them via Dropbox, it would be useful for the application to scan for files trying to exploit these vulnerabilities as well.

5.4 Discussion

In this chapter, we have investigated the question “What are the limitations of traditional security solutions running on mobile platforms?”. We did so by analyzing the state of the industry of current security solutions on Android and iOS, which is something that, as far as we know, has never been done in research before. We could not investigate the state of the industry for Windows Phone 7 since there are no applications available as of now. We found that there are basically two kinds of security solutions that are implemented for mobile platforms: firewalls and virus scanners. A bit surprising is the fact that there are no implementations of IDS’ on mobile platforms even though they are widely researched (see Section 3.4.3).

Firewalls are used to control incoming and outgoing network traffic of a system. On desktop platforms, firewalls are usually used to control access to the system and to prevent leakage of data from the system. On mobile platforms, firewalls can additionally be used to restrict applications that use the network intensively, when the device is using the mobile network, to save costs.

Android and Windows Phone offer the possibility to restrict the network usage of applications using permissions. The problem is, that this can only be decided during the installation. Nonetheless, it is a first step towards a application based firewall system that is integrated directly into

the security model of the platforms. However, these solutions still lack the sophistication of firewalls that we know from desktop platforms. It is only possible to allow the application to use the network or not. More fine grained control like only allowing communication with a specific host or using a specific protocol are not possible. Because of that, we think that firewall solutions can offer the user additional security on top of the security provided by the platform.

Our investigation showed that some firewall solutions exist on Android and iOS, but not on Windows Phone 7. However, all of the found applications require the device to be rooted/jailbroken. This is due to the fact that the two used methods to implement a firewall, using a kernel-space packet filter controlled from a user-space application and hooking system calls for network communication in all applications are prevented by the security models of the platforms. Since we can not expect average users to root/jailbreak their device and due to the fact that this process has to deactivate some security features of the platforms, the state of firewalls on mobile platforms is currently quite poor. This situation can only change if the platform manufacturers make fundamental changes to their security architecture. One such change could be to grant a small selection of trusted applications more extensive permissions than other applications. This could for example be done using a mechanism like iOS entitlements (see Section 3.2.2). However, this would most likely make these applications a target for privilege escalation attacks.

As we have seen in Section 3.3, malware is currently only common on the Android platform. One possible reason for this situation could be that on Android, as opposed to iOS and Windows Phone 7, applications can be installed from third parties, possibly without being checked for malware. The malware situation on mobile platforms is reflected in the situation of virus scanners. On Windows Phone 7, virus scanners are currently non-existent and the only virus scanner that we found on iOS is very limited and scans for viruses targeting desktop platforms. We see the reason for this situation not only in the fact that malware on these platforms is practically non-existent. Another problem is that theses platforms are sandboxing applications to a degree that offers very little possibility to implement a virus scanner. As long as this does not change or as long as the platforms do not offer higher privileges to security solutions, we do not expect this situation to change.

The situation is very different on Android. There is already a lot of malware and the sandbox allows broader access, e.g. to the filesystem. Because of that, many virus scanners for Android exist. The virus scanners, that we investigated, show that there are a lot of implementation possibilities. The biggest difference is in how they detect potential malware. While some scanners use only very basic protection in from of finding malware by their package name, others seem to work towards sophistication of virus scanners of desktop platforms with detecting malware by signatures. While comparing package names is a detection method that is not found on desktop platforms since package names is an Android specific feature, comparing contents of executable files or simply comparing filenames can also be found on antivirus software on desktop platforms. A detection mechanism that is found on desktop antivirus software is heuristic detection, which does not detect malware by its signature, but by searching for suspicious behavior [64]. Our

investigation shows that this is not implemented yet in any of the investigated applications, but at least AVG seems to be working on it. Also advanced scanning methods in the cloud where resources are not as limited as on mobile devices seem to be a good solution that is already implemented on Android.

Compared to the capabilities of desktop virus scanners as described in Section 3.4.1, both on-demand and on-access scanning is also present in scanners for Android. We did not see any virus scanners on Android that scan the memory of processes though. The reason for that most likely being that a sandboxed application has no access to the memory of other processes as we have seen in the experiment in Section 4.4.2. Also in contrast to desktop virus scanners, we did not see any virus scanners on Android that perform code emulation. The reason for that most likely being resource consumption. This approach could, however, be performed by a cloud scanning service without these restrictions.

After we have determined how security solutions are restricted on mobile platforms, we need to discuss whether they can protect against the threats that we have found in our first research question (see Section 4.5). Firewalls are only preventing access to services on devices from the outside and restrict applications from communicating with the outside world. The threats that we have analyzed in Section 4 are only about malware attacking other applications on the same platform. Firewalls can be considered a security improvement, as we have explained before, but they can not protect applications from malware running on the same platform since they only protect the interface between the device and the network but do not influence what happens on the platform itself. The only way, firewalls can offer protection in our context is by blocking malware from sending confidential information that was previously gathered from another application. But this is only after the sensitive information has already been extracted from an application.

Virus scanners offer more protection in our context. They can in theory detect malware before it can manifest a threat. However, all the virus scanners that we investigated rely on signatures of known malware. This means that threats can still be manifested by malware even if a virus scanner is installed as long as there is no signature. We confirmed this by performing all our experiments that have a medium or high likelihood on an Android device that had all five investigated virus scanners installed. None of our experiments did alarm the user about malicious behavior. With heuristic detection, which at least AVG is working on, this might change, however, in the future when virus scanners on mobile platforms are actually looking at the behavior of an application. For now, virus scanners can also not offer much protection against the unmitigated threats from our first research question. Because of that, we will look at our third research question in the next chapter to determine if app hardening can help protect against these threats that are neither mitigated by the platform nor the traditional security solutions.

6 App Hardening on Mobile Platforms

This chapter investigates the third research question: “Can app hardening solutions increase the security of mobile platforms and what are their limitations?”. We will do so by investigating if and how applications on mobile platforms can protect themselves against their environment. Since we did not find any threats that concern the Windows Phone 7 platform, meaning that further protection is not necessary, we only investigate Android and iOS in this section.

Because the results of this investigation were very interesting to Promon, they asked for the details of this chapter to be kept confidential, so they could use it in a new product they are developing. We therefore can only give a short and not too detailed summary and discussion here.

Based on the description of app hardening in Section 3.5, we first investigated how a security module can make sure that the environment it runs in is initially clean. This has been investigated in form of jailbreak and rooting detection as well as detecting whether the application, the security module protects against, has not been modified in form of repackaging. We were able to implement those detection mechanisms basically without any restrictions from the platforms.

When it comes to monitoring the program that the security module protects while it is being executed, restrictions on iOS had an impact on how flexible we were when choosing the best method for hooking. But we found a way to implement hooking in on both Android and iOS in basically the same way without being restricted by neither Android nor iOS.

After the basis for app hardening has been established, we could then look at the concrete threats that we found in Section 4. We looked at securing the interfaces of the application to be protected. In our case, we looked at how to prevent the threat of intercepting user input, which was found to be a medium threat on Android and iOS in Section 4.4.4. On Android, the problem was with installing malicious keyboards which we found a way to detect. On iOS, the problem was that it maintains a keyboard cache file where user input is written to. We found a way to protect against this as well, but can not go into details here. Another threat that we found while investigating the first research question about possible capabilities of malware on mobile platforms was the confidentiality of the installation files of an application (see Section 4.4.1). We found a way to mitigate this threat as well.

With respect to the open threats on Android and iOS presented in Table 31, we therefore were able to use app hardening approaches to mitigate some of the threats. There are, however, two

kinds of threats still remaining. The first kind of threat is with IPC. Those problems could probably be resolved using encryption, but were not further investigated in this thesis. The other problem was with malware being able to terminate other applications. One way to mitigate this problem would be to implement a watchdog solution where two applications monitor each other and restart the other application in case it is terminated. Investigating this method would also be subject for further work.

All in all, we can conclude that app hardening is well suited for deployment on mobile platforms since it is not limited to the degree that traditional third party security solutions are. The presented approaches show the potential of app hardening to make applications running on mobile platforms more secure. This makes it an interesting subject for further study in the future.

7 Future Work

While working with this thesis, we found multiple possibilities to extend our work beyond the available time for the thesis. This chapter will summarize the future work that we found.

When we were investigating the example threats in the first research question, we had to leave out some investigations due to time constraints. These investigations should be performed in future work to get an even better picture of the differences between the investigated platforms and which threats need to be mitigated since they are not dealt with by the respective platforms.

As described before, we were only investigating one example threat for each combination of entry points and assets. In order to find all threats that could arise from an entry point being used to compromise an asset, a data flow analysis should be performed as proposed in [9]. For each of the identified threats, experiments to determine the likelihood of compromising the assets should then be performed to get a complete picture of all threats on each of the platforms that need to be mitigated.

Also, since the mobile platforms are quite young, they are still undergoing major changes in their security architecture. This is especially true for Windows Phone 7, where we suspect Microsoft to loosen the restriction on the platform to better compete with other platforms. Because of that, the experiments from the first research question should be repeated at a later time to determine changes to threats.

During the investigation of the second research question, we had to limit the number of virus scanners we could investigate. Future work could be to investigate more antivirus scanners to find functionality that we did not find in the selected scanners. One such functionality that is worth investigating is cloud scanning that was advertised by some antivirus scanners but we did not investigate further.

When it comes to app hardening, we have made the first steps towards implementing it on Android and iOS. However, there is still a lot to investigate. What we did not investigate yet is how an application can be bound to the app hardening module that tries to protect it. We also did not investigate secure deployment of app hardening solutions yet. Other possible investigations would be to find ways to protect against the remaining threats that are not mitigated by the platforms, the third party security solutions and the app hardening approaches we have investigated.

8 Conclusion

In this master thesis, we have investigated app hardening on mobile platforms. This was done by answering three research questions. In the first research question, we have analyzed the difference in limitations that malware on a mobile platform has compared to desktop platform when it comes to compromising other applications that are run on the platform. We performed this investigation by determining threats by mapping each entry point that an application has against each asset that an application has. We then performed experiments on each of the platforms to determine likelihood of a threat being manifested by malware. By doing so, we used previous research to find ways to compromise the assets of an application, but we also found new methods that we have not seen in research before. Our results show that the difference between desktop and mobile platforms can be clearly seen. While there are only some high and medium likelihood threats on Android and iOS and only low likelihood threats on Windows Phone 7, the situation on Windows 7 is quite different since all investigated threats can either be manifested without any restrictions or with some preconditions. Since there were still threats on the mobile platforms that were not mitigated by the platforms, we investigated the second research question.

To answer the second research question, we investigated the capabilities and restrictions of traditional third party security solutions. We performed this analysis by investigating the current state of the industry. This was done by checking the various application distribution platforms for the different mobile platforms for third party security solutions. We found that there are two main categories of security solutions currently available: firewalls and antivirus scanners. To understand how these solutions work and how they are limited, we performed a deep analysis using reverse engineering. Our result is that firewalls can only be found on jailbroken iOS devices and rooted Android devices because the approaches used by the firewalls will only work if the application is not restricted by a sandbox. Currently, there are no firewalls available on Windows Phone 7. Antivirus scanners are mostly found on the Android platform. We found that there are different solutions that have been implemented that greatly differ in sophistication. While some scanners only compare the package names of applications, others actually compare file content to find malware. Also cloud scanning can be found in some applications and we have found hints that one virus scanner manufacturer is working on heuristic detection on malware. We have also explained that firewalls will not help against malware trying to compromise other applications since they are only concerned with the network communication with the outside world and are not concerned with what happens on the platform. Also, we have tested whether any of the antivirus scanners will actual detect when a threat from our experiments that we have performed in the first research question will be detected by the antivirus scanners. The result is that none of the virus scanners detected the threats, which is most likely because the scanners are all based

on signatures.

App hardening on mobile platforms was investigated to answer the third research question. The goal of these investigations was to determine how app hardening could be implemented on mobile platforms and if there are any restrictions that would limit the effectiveness of app hardening. We performed our investigation by first looking into checking the integrity of the platform in form of rooting and jailbreak detection as well as checking that the application that is to be protected has not been repackaged and is therefore in an initial clean state. We then investigated possibilities to implement program monitoring in form of hooking and found a way to implement hooking on Android and iOS in spite of the strong memory protection that iOS uses. After we have investigated the basis of app hardening, we investigated solutions to address the threats that we found during the investigation of our first research question and that are not addresses by third party security solutions as we have seen from our second research question. This was in form of protecting the installation of the application from leaking sensitive information to third parties by downloading security critical code during runtime and executing it. Another threat that we successfully address is the leakage of information from the user interface on Android and iOS. Our results show that the app hardening approach has an advantage over traditional security solutions on mobile platforms in that it is not as strongly limited by the security architecture of the platforms. Also, we found that the investigated app hardening methods are successful in increasing the security of the Android and iOS platforms. We therefore consider app hardening on mobile platforms an interesting subject for further studies.

Because of the new findings that were made during the investigation of the third research question, Promon has requested that these findings will not be published since they want to develop a product based on the research. We therefore could not publish the results even though the investigation of this research question includes the most new contributions. A paper called “Firewalls und Virenscanner auf Mobilen Plattformen” about the results of our second research question has already been accepted at a security conference for German speaking countries called D-A-CH Security 2012[1] and will be presented in September. A second more detailed paper about the results of our second research question called “Security Add-Ons for Mobile Platforms” has been submitted to the NordSec 2012[2] conference, which will be held in Sweden in October. A third paper called “Malicious Repackaging of Mobile Apps” was submitted to the ACSAC 2012[3] conference. This paper is about repackaging on all the investigated platforms and contains experiments that we originally did for Section 4 but we later realized that they do not fit well in the thesis.

[1] http://www.syssec.at/dachsecurity2012/
[2] http://www.bth.se/com/nordsec2012.nsf/
[3] http://www.acsac.org/

Bibliography

[1] Association for Computing Machinery. 2011. ACM Taxonomy. http://www.computer.org/portal/web/publications/acmtaxonomy, Retrieved on: 08.12.2011.

[2] Financial Times. 2011. Smartphone shipments surpass PCs. http://www.ft.com/intl/cms/s/2/d96e3bd8-33ca-11e0-b1ed-00144feabdc0.html, Retrieved on: 08.12.2011.

[3] IDC. 2011. Worldwide Smartphone Market Expected to Grow 55% in 2011 and Approach Shipments of One Billion in 2015. http://www.idc.com/getdoc.jsp?containerId=prUS22871611, Retrieved on: 08.12.2011.

[4] Hypponen, M. 2006. Malware goes mobile. *Scientific American*, 295(5), 70–77.

[5] F-Secure Labs. 2012. Mobile Threat Report Q1 2012. http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf, Retrieved on: 23.05.2012.

[6] Felt, A., Finifter, M., Chin, E., Hanna, S., & Wagner, D. 2011. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 3–14. ACM.

[7] Leedy, P. D. & Ormrod, J. E. 2010. *Practical Research: Planning and Design (Ninth Edition)*. Pearson.

[8] IDC. 2012. Android- and iOS-Powered Smartphones Expand Their Share of the Market in the First Quarter, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS23503312, Retrieved on: 26.05.2012.

[9] Swiderski, F. & Snyder, W. 2004. *Threat Modeling*. Microsoft Press.

[10] Conder, S. & Darcey, L. 2010. *Android Wireless Application Development*. Addison-Wesley Professional.

[11] Android Open Source Project. 2012. Frequently Asked Questions. http://source.android.com/faqs.html, Retrieved on: 24.01.2012.

[12] Zhou, Z., Zhu, R., Zheng, P., & Yang, B. 2011. *Windows Phone 7 Programming for Android and iOS Developers*. Wrox.

[13] Seriot, N. 2010. iPhone Privacy. In *Black Hat DC 2010*. Black Hat.

[14] Zhang, S., Wang, L., Zhang, R., & Guo, Q. 2010. Exploratory study on memory analysis of Windows 7 operating system. In *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 6. IEEE.

[15] Klement, S. Sicherheitsaspekte der Google Android Plattform. Master's thesis, Universität Bremen, 2011.

[16] Davi, L., Dmitrienko, A., Sadeghi, A., & Winandy, M. 2011. Privilege escalation attacks on Android. *Information Security*, 2011, 346–360.

[17] Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., & Sheth, A. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association.

[18] Apple. iOS Technology Overview. Technical report, Apple Inc., 2010.

[19] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., & Hunt, G. C. 2011. Rethinking the library OS from the top down. *SIGPLAN Notices*, 46, 291–304.

[20] Russinovich, M., Solomon, D., & Ionescu, A. 2009. *Windows Internals Fifth Edition*. Microsoft Press.

[21] Dwivedi, H., Clark, C., & Thiel, D. 2010. *Mobile Application Security*. McGraw-Hill Education.

[22] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., & Dolev, S. 2009. Google Android: A state-of-the-art review of security mechanisms. In *CoRR, abs/0912.5101*.

[23] Enck, W., Ongtang, M., & McDaniel, P. 2009. Understanding Android Security. *IEEE Security & Privacy*, 7(1), 50–57.

[24] Nauman, M., Khan, S., & Zhang, X. 2010. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 328–332. ACM.

[25] Sadun, E. 2009. *The iPhone Developer's Cookbook – Building Applications with the iPhone SDK*. Addison-Wesley Professional.

[26] Schröder, M.-S., Junge, F., & Heer, J. 2011. Security through Sandboxing? - Towards more secure smartphone platforms. In *IJCAI Workshop on Intelligent Security (SecArt)*, 74–75.

[27] Chin, E., Felt, A., Greenwood, K., & Wagner, D. 2011. Analyzing inter-application communication in Android. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services*.

[28] Oberheide, J. & Jahanian, F. 2010. When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, 43–48. ACM.

[29] Barrera, D. & Van Oorschot, P. 2010. Secure software installation on smartphones. *IEEE Security & Privacy*, 9(3)(99), 42–48.

[30] Anderson, J., Bonneau, J., & Stajano, F. 2010. Inglorious installers: security in the application marketplace. In *Proceedings of the 9th Workshop on the Economics of Information Security*.

[31] Gilbert, P., Chun, B., Cox, L., & Jung, J. 2011. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, 21–26. ACM.

[32] Egele, M., Kruegel, C., Kirda, E., & Vigna, G. 2011. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*.

[33] Miller, C. 2011. Mobile Attacks and Defense. *IEEE Security & Privacy*, 9, 68–70.

[34] Schmidt, A., Schmidt, H., Batyuk, L., Clausen, J., Camtepe, S., Albayrak, S., & Yildizli, C. 2009. Smartphone malware evolution revisited: Android next target? In *4th International Conference on Malicious and Unwanted Software (MALWARE)*.

[35] Anvaari, M. & Jansen, S. 2010. Evaluating architectural openness in mobile software platforms. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, 85–92. ACM.

[36] Bishop, M. December 2002. *Computer Security: Art and Science*. Addison-Wesley.

[37] Saltzer, J. & Schroeder, M. 1975. The protection of information in computer systems. *Proceedings of the IEEE*, 63, 1278–1308.

[38] Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., & Larus, J. 2006. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*. ACM.

[39] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., & Glezer, C. 2010. Google Android: A comprehensive security assessment. *IEEE Security & Privacy*, 8(2), 35–44.

[40] Shabtai, A., Fledel, Y., & Elovici, Y. 2010. Securing Android-powered mobile devices using SELinux. *IEEE Security & Privacy*, 8, 36–44.

[41] Höbarth, S. & Mayrhofer, R. 2011. A framework for on-device privilege escalation exploit execution on Android. In *Proceedings fo the 3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Device Use*.

[42] Smalley, S. 2011. The Case for SE Android. In *Linux Security Summit*. National Security Agency.

[43] Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B., & Jaeger, T. 2008. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*. ACM.

[44] Miller, C., Blazakis, D., DaiZovi, D., Esser, S., Iozzo, V., & Weinmann, R. 2012. *iOS Hacker's Handbook*. John Wiley & Sons, Inc.

[45] Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*.

[46] Felt, A., Greenwood, K., & Wagner, D. 2011. The effectiveness of application permissions. In *Proceedings of the USENIX Conference on Web Application Development*.

[47] Enck, W., Ongtang, M., & McDaniel, P. Mitigating Android Software Misuse Before It Happens. Technical Report NAS-TR-0094-2008, Network and Security Research Ctr., Dept. Computer Science and Eng., Pennsylvania State Univ., 2008.

[48] Jeon, J., Micinski, K., Vaughan, J., Reddy, N., Zhu, Y., Foster, J., & Millstein, T. 2011. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *Draft*.

[49] Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. 2011. A study of android application security. In *USENIX Security*.

[50] Microsoft. Application Certification Requirements for Windows Phone. Technical report, Microsoft Corporation, 2012. `http://msdn.microsoft.com/en-us/library/hh184843(v=vs.92).aspx`, Retrieved on: 14.02.2012.

[51] Becher, M., Freiling, F., Hoffmann, J., Holz, T., Uellenbeck, S., & Wolf, C. 2011. Mobile security catching up? Revealing the nuts and bolts of the security of mobile devices. In *IEEE Symposium on Security and Privacy (SP)*. IEEE.

[52] Becher, M., Freiling, F., & Leider, B. 2007. On the effort to create smartphone worms in windows mobile. In *Information Assurance and Security Workshop, 2007. IAW'07. IEEE SMC*, 199–206. IEEE.

[53] Leidner, B. M. Voraussetzungen für die Entwicklung von Malware unter Windows Mobile 5. Master's thesis, RWTH Aachen University, 2007.

[54] Dagon, D., Martin, T., & Starner, T. 2004. Mobile phones as computing devices: the viruses are coming! *IEEE Pervasive Computing*, 3, 11–15.

[55] Leavitt, N. 2005. Mobile phones: The next frontier for hackers? *Computer*, 38, 20–23.

[56] Töyssy, S. & Helenius, M. 2006. About malicious software in smartphones. *Journal in Computer Virology*, 2, 109–119.

[57] Lawton, G. 2008. Is it finally time to worry about mobile malware? *Computer*, 41, 12–14.

[58] Ho, Y. & Heng, S. 2009. Mobile and ubiquitous malware. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, 559–563. ACM.

[59] Vidas, T., Votipka, D., & Christin, N. 2011. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, 10–10. USENIX Association.

[60] Damopoulos, D., Kambourakis, G., & Gritzalis, S. 2011. iSAM: An iPhone stealth airborne malware. In *Proceedings of the 26th IFIP TC-11 International Information Security Conference*, 17–28. Springer.

[61] Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., & Iftode, L. 2010. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, 49–54. ACM.

[62] Symantec. 2012. Internet Security Threat Report Volume 17. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf`, Retrieved on: 18.06.2012.

[63] Cohen, F. B. 1994. *A short Course on Computer Viruses*. John Wiley & Sons, Inc.

[64] Szor, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

[65] Oberheide, J., Cooke, E., & Jahanian, F. 2008. Cloudav: N-version antivirus in the network cloud. In *Proceedings of the 17th conference on Security symposium*, 91–106. USENIX Association.

[66] Oberheide, J., Veeraraghavan, K., Cooke, E., J., F., & F., J. 2008. Virtualized in-cloud security services for mobile devices. In *Proceedings of MobiVirt 2008*, 31–35.

[67] Portokalidis, G., Homburg, P., Anagnostakis, K., & Bos, H. 2010. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 347–356. ACM.

[68] Almgren, M. & Lindqvist, U. 2001. Application-integrated data collection for security monitoring. In *Recent Advances in Intrusion Detection*, 22–36. Springer.

[69] Venugopal, D., Hu, G., & Roman, N. 2006. Intelligent virus detection on mobile devices. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*. ACM.

[70] Chow, G. & Jones, A. 2008. A Framework for Anomaly Detection in OKL4-Linux Based Smartphones. In *Australian Information Security Management Conference*.

[71] Yan, Q., Li, Y., Li, T., & Deng, R. 2009. Insights into Malware Detection and Prevention on Mobile Phones. In *Security Technology*, Ślęzak, D., Kim, T.-h., Fang, W.-C., & Arnett, K. P., eds, volume 58, 242–249. Springer Berlin Heidelberg.

[72] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. 2011. "Andromaly": A Behavioral Malware Detection Framework for Android Devices. *Journal of Intelligent Information Systems*, 2011, 1–30.

[73] Bose, A., Hu, X., Shin, K., & Park, T. 2008. Behavioral detection of malware on mobile handsets. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, 225–238. ACM.

[74] Cheng, J., Wong, S., Yang, H., & Lu, S. 2007. Smartsiren: virus detection and alert for smartphones. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, 258–271. ACM.

[75] Miettinen, M., Halonen, P., & Hatonen, K. 2006. Host-based intrusion detection for advanced mobile devices. In *AINA 2006. 20th International Conference on Advanced Information Networking and Applications*, volume 2, 72–76. IEEE.

[76] Venugopal, D. & Hu, G. 2008. Efficient signature based malware detection on mobile devices. *Mobile Information Systems*, 4(1), 33–49.

[77] Schmidt, A., Peters, F., Lamour, F., Scheel, C., Çamtepe, S., & Albayrak, S. 2009. Monitoring smartphones for anomaly detection. *Mobile Networks and Applications*, 14(1), 92–106.

[78] Jacoby, G. & Davis, N. 2004. Battery-based intrusion detection. In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 4, 2250–2255. IEEE.

[79] Kim, H., Smith, J., & Shin, K. 2008. Detecting energy-greedy anomalies and mobile malware variants. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*.

[80] Liu, L., Yan, G., Zhang, X., & Chen, S. 2009. Virusmeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, 244–264. Springer.

[81] Inoue, H. & Forrest, S. 2002. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the 2002 workshop on New security paradigms*, 52–60. ACM.

[82] Elbaum, S. & Munson, J. 1999. Intrusion detection through dynamic software measurement. In *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring-Volume 1*. USENIX Association.

[83] International IT Security Evaluation Community. 2006. Common Methodology for Information Technology Security Evaluation Part 1: Introduction and general model Version 3.1. `www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R1.pdf`, Retrieved on: 20.06.2012.

[84] Silberschatz, A., Calvin, P. B., & Gagn, G. 2005. *Operating System Concepts 7th Edition*. John Wiley & Sons, Inc.

[85] International IT Security Evaluation Community. 1999. Common Methodology for Information Technology Security Evaluation Part 2: Evaluation Methodology Version 1.0. `http://www.commoncriteriaportal.org/files/ccfiles/cemv10.pdf`, Retrieved on: 20.06.2012.

[86] Zovi, D. 2011. Apple iOS 4 Security Evaluation. Black Hat USA 2011. `https://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf`, Retrieved on: 14.02.2012.

[87] Kumar, E. 2010. User-mode memory scanning on 32-bit & 64-bit windows. *Journal in Computer Virology*, 6, 123–141.

[88] Richter, J. May 1994. Load Your 32-bit DLL into Another Process's Address Space Using INJLIB. *Microsoft Systems Journal-US Edition*, 13–40.

[89] Apple. 2012. iOS App Programming Guide. `http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/Introduction/Introduction.html`, Retrieved on: 22.05.2012.

[90] Vishnani, K., Pais, A. R., & Mohandas, R. 2011. An In-Depth Analysis of the Epitome of Online Stealth: Keyloggers; and Their Countermeasures. In *Advances in Computing and Communications*, Abraham, A., Mauri, J. L., Buford, J. F., Suzuki, J., & Thampi, S. M., eds, volume 192 of *Communications in Computer and Information Science*, 10–19. Springer Berlin Heidelberg.

A Source Code

A.1 Filesystem Exploration in iOS

```
void findFiles(NSString *name)
{
  // Open directory
  DIR *dir = opendir([name UTF8String]);

  if(dir == NULL)
    return;

  // Loop through all files
  struct dirent *dp;
  while ((dp=readdir(dir)) != NULL)
  {
    // Ignore current and parent directory
    if(strcmp(dp->d_name, ".") == 0 || strcmp(dp->d_name, "..") == 0)
      continue;

    NSString * fullNameString = [name stringByAppendingString:
    [NSString stringWithFormat:@"/%s", dp->d_name]];

    const char * fullName = [fullNameString UTF8String];
    struct stat s;

    // Did stat fail?
    if(stat(fullName, &s) != 0)
    {
      NSLog(@"? ? ? ????????? %s", fullName);
      findFiles(fullNameString);
      continue;
    }

    // Recursively seach through subdirectories
    if(S_ISDIR(s.st_mode))
        findFiles(fullNameString);

    NSString * info = @"";

    // Add filetype
    if(S_ISREG(s.st_mode))
      info = [info stringByAppendingString: @"f"];
    else if(S_ISDIR(s.st_mode))
      info = [info stringByAppendingString: @"d"];
    else if(S_ISLNK(s.st_mode))
      info = [info stringByAppendingString: @"l"];
    else if(S_ISSOCK(s.st_mode))
      info = [info stringByAppendingString: @"s"];
    else
      info = [info stringByAppendingString: @"?"];
```

```
    // Add owner and group
    NSString *user = [NSString stringWithFormat:@"%d", s.st_uid];
    NSString *group = [NSString stringWithFormat:@"%d", s.st_gid];
    info = [info stringByAppendingString: [NSString stringWithFormat:@" %s", user]];
    info = [info stringByAppendingString: [NSString stringWithFormat:@" %s", group]];

    // Add user permissions
    info = [info stringByAppendingString: ((s.st_mode & S_IRUSR) != 0) ? @" r": @" -"];
    info = [info stringByAppendingString: ((s.st_mode & S_IWUSR) != 0) ? @"w": @"-"];
    info = [info stringByAppendingString: ((s.st_mode & S_IXUSR) != 0) ? @"x": @"-"];

    // Add group permissions
    info = [info stringByAppendingString: ((s.st_mode & S_IRGRP) != 0) ? @" r": @" -"];
    info = [info stringByAppendingString: ((s.st_mode & S_IWGRP) != 0) ? @"w": @"-"];
    info = [info stringByAppendingString: ((s.st_mode & S_IXGRP) != 0) ? @"x": @"-"];

    // Add others permissions
    info = [info stringByAppendingString: ((s.st_mode & S_IROTH) != 0) ? @" r": @" -"];
    info = [info stringByAppendingString: ((s.st_mode & S_IWOTH) != 0) ? @"w": @"-"];
    info = [info stringByAppendingString: ((s.st_mode & S_IXOTH) != 0) ? @"x": @"-"];

    // Add filename
    info = [info stringByAppendingString: [NSString stringWithFormat:@" %s", fullName]];

    // Output result
    NSLog(@"%@", info);
  }

  closedir(dir);
}
```

A.2 Inject and execute Code in a Windows 7 Process

```
#include <stdio.h>
#include <windows.h>

BOOL InjectAndExecute(unsigned int processId, char * libraryPath, char * functionName)
{
  HANDLE process;
  HMODULE module;
  FARPROC function;
  LPVOID memory_address;
  HANDLE thread;

  // Open process
  process = OpenProcess((PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION |
    PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_QUERY_INFORMATION), TRUE, processId);

  if(process == 0)
    return FALSE;

  // Load module in our process so we can find the address of the procedure next
  module = LoadLibraryA("kernel32.dll");

  if(module == 0)
    return FALSE;

  // Get address of load library function
  function = GetProcAddress(module, "LoadLibraryA");
```

```
  if(function == 0)
    return FALSE;

  // Allocate memory for parameter to load library function
  memory_address = VirtualAllocEx(process, NULL, strlen(libraryPath) + 1, MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE);

  if(memory_address == 0)
    return FALSE;

  // Write parameter to remote process
  if(!WriteProcessMemory(process, memory_address, libraryPath, strlen(libraryPath) + 1, NULL))
    return FALSE;

  // Create thread to load library
  thread = CreateRemoteThread(process, 0, 0, (LPTHREAD_START_ROUTINE) function, memory_address, 0, 0);

  if(thread == 0)
    return FALSE;

  // Wait for thread to finish
  if(WaitForSingleObject(thread, INFINITE) == WAIT_FAILED)
    return FALSE;

  // Load module in our process so we can find the address of the procedure next
  module = LoadLibraryA(libraryPath);

  if(module == 0)
    return FALSE;

  // Get address of function
  function = GetProcAddress(module, functionName);

  if(function == 0)
    return FALSE;

  // Create thread to call function
  thread = CreateRemoteThread(process, 0, 0, (LPTHREAD_START_ROUTINE) function, 0, 0, 0);

  if(thread == 0)
    return FALSE;

  // Clean up
  CloseHandle(process);
  VirtualFreeEx(process, memory_address, 0, MEM_RELEASE);

  return TRUE;
}
```

A.3 Reconstructed Scan Function of Avast Mobile Security

```
enum
{
  DEX_FILE  = 1,
  ZIP_FILE  = 2,
  ELF_FILE  = 4
};
```

```
scan_result_structure * scan(int handle, unsigned char * data, int dataLen, application_info const * appInfo,
char const * packageName)
{
  scan_result_structure * scanResult = malloc(sizeof(scan_result_structure));

  if(scanResult == NULL)
    return scanResult;

  ScanContext * context = GetContext(handle);

  if(context == NULL)
  {
    char * name = malloc(16);

    scanResult->name = name;

    if(name == NULL)
      return scanResult;

    memset(name, 0, 16);

    memcpy(name, "invalid_context", 16);

    return scanResult;
  }

  if(appInfo != NULL)
    print_debug("cls = %s", appInfo->className);

  if(packageName != NULL)
  {
    if(check_pkg_name(scanResult, packageName) != 0)
      return scanResult;
  }

  if(dataLen < 0)
    return scanResult;

  if(context->type == ZIP_FILE)
  {
    char * name = malloc(7);

    scanResult->name = name;

    if(name == 0)
      return scanResult;

    memcpy(name, "unpack", 7);

    return scanResult;
  }

  if(context->type == ELF_FILE)
  {
    scan_elf(context, data, dataLen, scanResult);
    return scanResult;
  }

  if(context->type == DEX_FILE)
  {
    scan_dex(context, data, dataLen, scanResult);
    return scanResult;
```

```
  }

  if(scanEICAR(data, dataLen) != 0)
  {
    char * name = malloc(6);

    scanResult->name = name;

    if(name == NULL)
      return scanResult;

    memcpy(name, "Eicar", 6);

    return scanResult;
  }

  return scanResult;
}
```