Privacy Issues with the Android Market
RP1 Project Paper

Bastiaan Wissingh Thorben Krüger

January 28, 2011

Contents

1 Introduction and Motivation

In the growing market of the so called smart-phones, Google's open source Android platform in recent years has established a firm foothold. As an operating system, Android primarily targets commercial mobile hardware with (comparably) limited processing power, such as smart phones and (recently) tablets.

The user of such a device is often given the opportunity to install additional (third party) mobile applications, (colloquially abbreviated to "apps"), which one commonly finds, installs and manages through a preinstalled catalogue application.

Although there exist several different implementations of such an application, the most ubiquitous on the Android platform is probably the "Android Market" distributed by Google on almost all Android-branded mobile devices. When a third party decides to publish an Android application with the goal of exposing it to as many users as possible, the most common approach is to aim for publication in the "Android Market", which does not require much more than paying a one-time fee and agreeing to a number of terms and conditions.

As the sole authority over the Android Market, Google reserves the right to remotely remove any application that has been installed via this channel, should it turn out to violate any of the terms of service agreements. Indeed, functionality to achieve this has been implemented in the Android Market application and already made use of to remove a proof of concept "malicious" application by security researcher Jon Oberheide[7][6].

This questionable "feature" of the Android Market to support the remote-controlled removal of an application gives rise to speculations about undisclosed additional functionality with even higher impact on the integrity of a system. However, there has been little further research into this topic and its possible implications.

This project aims to establish verified facts pertaining to this topic before investigating remedies for any issues found.

At the beginning of this project, the following questions were established as worth investating:

- *What exactly is the Android Market?*
- *How does the Android Market work, with respect to application installation/removal procedures*
- *Are these procedures vulnerable for malicious attacks? And, if so, for what kind of attacks?*

2 Background Research

In the following paragraphs, a brief look into Google's Android Market system with regard to the above-posed questions is provided, according to publicly available information like the documentation on the Android website from Google and already published papers regarding the Android Market.

2.1 Android Market: Informal Definition

The Android Market is a solution from Google for an online software store to host free and paid applications for the Android platform. The Android Market was announced on 28 August 2008 and was made available to users on 22 October 2008. The Android Market can be compared to online software stores for other smart phone platforms like the App Store, developed by Apple for their iOS platform.

The applications published on the Android Market platform can be browsed and downloaded through a special application on an Android device, called `Market`. Browsing is also possible through a common web browser by accessing the platform's website, which essentially exposes the same information as is also accessible from the Market application itself. However only the latter offers the functionality for downloading and istalling an application on the actual device. An entirely web-based download of apps is not possible. Recently[1] [9][8], support has been added into the Market website for triggering application installation on a user-registered device without requiring any interaction with the device itself.

The Market application is proprietary and comes pre installed on most factory installed Android devices.

2.2 Android Market: Functionality

As mentioned in the previous paragraph, the Android Market is used as a way to distribute applications for the Android platform. The applications on the Android platform are distributed and installed through so called `.apk` files, which stands for Android Application Package file. These self-contained `.apk` files contain all components needed for a certain application, like the application's code (`.dex` files), the resources, the assets, and the manifest file [13].

On the Android Operating System there are two services available for handling the `.apk` files:

- A so called Package Installer application[23], which is a higher level application that interacts with the Package Manager service to install applications. This *PackageInstaller.apk* package can be located in the directory */system/app/*.
- A so called Package Manager service[22], which is a standard feature of the Android operating system used for managing packages and can be directly used by issuing the command *pm* which is located in the directory */system/bin/*.

[1]February 2011, so these developments could not be taken into account for this project any more. [9][8] are still highly relevant to the topics we discuss here, so are recommended reading to put our research in current perspective.

The Package Installer service can be used when `.apk` files are directly installed, for example from and SD-Card, while the Package Manager service is used when applications are installed through the Android Market application or through the Android Debug Bridge tool of the Android SDK.

2.2.1 The Installation Procedure

On Android there are basically three different methods of installing Applications (or `.apk`) files[24]:

- The first method is by using the Market application on an Android device, which interacts directly with Package Manager.
- The second method is by installing `.apk` files directly from the device memory or SD-Card with the Package Installer service.
- The third method is by using the `adb` (Android Debug Bridge) tool from the Android SDK, which interacts directly with the Package Manager.

In the Android Operating System applications are separated from each other by the kernel through the use of sandboxes. This means that an application by default does not have direct access to shared system resources or data. To be able to share resources and data with other applications, an application must explicitly be granted with the required permissions.

Those permissions are granted to a certain application by the Package Manager service upon installation. The Package Manager service grants those permissions based on checks of the signatures of the applications declaring those permissions and/or interaction with the user (it displays the permissions of the application to the user, in order for the user to agree).[2]

Also `.apk` files need to be signed with an certificate (does not need to be a certificate authority) in order to distinguish the developers of different applications.[14]

2.2.2 The Remove Procedure

On Android there are basically two different methods of removing Applications (or `.apk`) files:

- The first method is by unstalling .apk files directly from the device memory or SD-Card with the Package Installer service by using the Manage applications function of the System Settings.
- The second method is by using the adb (Android Debug Bridge) tool from the Android SDK, which interacts directly with the Package Manager.

Beside these two methods to remove applications from the Android platform, there is also a third option, however, this option can not be used by a user of an Android device but only by Google. This third option gives Google the possibility to remotely remove applications by sending a specific command to an Android device[6].

[2]See section 6 below for further details on the permission system.

2.2.3 Cloud to Device Messaging service (C2DM)

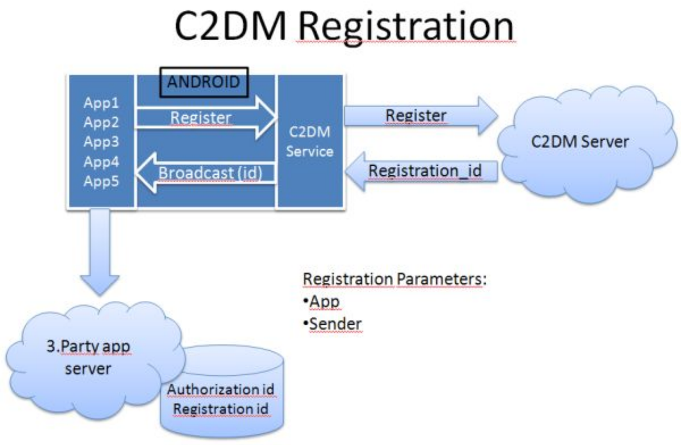
C2DM, also known as Android Cloud to Device Messaging, is a service from Google that helps developers send data from their servers to their applications on Android devices. The service provides a mechanism that servers can use to tell mobile applications to contact the server directly to fetch updated application or user data.

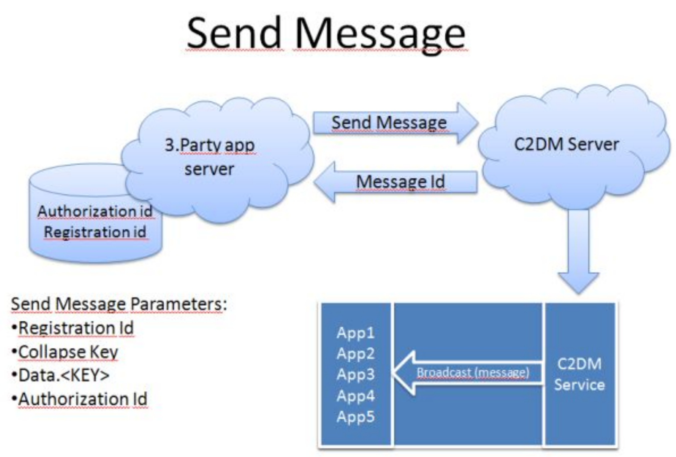
The C2DM service from Google basically exists of three parts:

- The Android device with the application that needs to receive push messages from the developer.
- The C2DM service servers from Google, which forward the messages between the developer and the application on the Android devices.
- The developer servers who push messages to the C2DM servers from Google, to contact the applications on the Android devices.

In order for the developers server to be able to send push messages to the application on the Android device, the Android device first needs to be registered with the Google C2DM service. This is because the Google C2DM servers work as sort of in the middle service. The developers servers can send a message to the Google C2DM service, which then sends the message to the Android device, in order for the Android device to directly contact the developers server for the new data.

In the pictures below, it will first show the registration process, after which it shows how the sending message process works.





3 Methodology

This chapter describes the technical configuration and decisions made for setting up the Android and capturing environment.

3.1 The Android Emulator

There are two different possibilities to be able to work with the Android platform, either by using a mobile device (e.g., a smart phone) with the Android OS installed, or by using an emulator to run the Android OS. For this research project, initially all work was undertaken on emulators with the Android OS, although a physical device was made available at a later stage as well.

Google released an Android Emulator[11], which is an emulator which enables a developer to run a virtual mobile device on a standard computer in order to design, debug, and test applications in an actual Android run-time environment. It is provided among a number of additional tools with the Android Software Development Kit[12] (also known as Android SDK), intended to aid the devlopment of applications for the plattform.[15]

One downside of using the emulator is that it by default only creates very minimal base-system images. Crucially, these come without the proprietary Google application collection, which would include the Market app. So in order to use the Market on the emulator, its package and the required dependencies have to be obtained through inofficial channels in order to be installed manually.

The Market application itself is provided by the `Vending.apk` package and depends on the Google Services Framework (provided by the `GoogleServicesFramework.apk`) for communication with the Google servers.

For this project installation instructions from a tutorial written by Varunkumar Nagarajan[19] were used to correctly deploy these packages on emulator instances.

Besides the Android Market and Google Services Framework, the GTalk Service Monitor (which monitors the Google Talk Service) was deemed a useful addition, since the Market app appears to make use of this Google Talk Service[21] for part of the communication with Google. To get the GTalk Service Monitor running, the `Talk.apk` and `gtalkservice.apk` from the Google application collection also had to be installed.

3.2 Traffic captures

For purposes of introspection into the encrypted data passed between Google's servers and a device running the Market application, some means for decrypting an SSL/TLS session is needed. Since SSL has been designed to prevent exactly this, there is no straight forward way to do this. However, if one manages to convince the device to trust a faked SSL certificate and establish oneself as a proxy somewhere between the server and the targeted device, one is in a position to perform a "man in the middle" (MITM) attack on any encrypted session that the device intends to establish with said server. Instead of establishing the session with the intended destination, the client unwittingly establishes a secure connection only with the proxy, which then in turn establishes a "normal" connection to said destination. Data sent from either party ends up at the proxy in the clear. The logical modus operandi for this setup is the forwarding of data from either end to the other, while logging the contents. Now, the session between proxy and client would normally not be trusted by the latter, since the faked certificate used by the former to identify itself is (most probably) not signed by any certificate authority (CA) which the client trusts. The easiest way to get the fake certificate accepted is to sign it with a self-generated CA certificate that the client has been led to trust beforehand. (In this case the attack is also known as a "*trusted* man in the middle".) In case of an Android device, this last part requires physical (i.e., USB) access to a device with enabled debugging functionality (or an equivalently emulator setup). Apart from this requirement, the rest of the setup is a comparatively straight-forward process of following a simple tutorial[17] so we omit the details here.

(Due to the relative obscurity of the processes, setting up a suitable proxy to conduct the MITM attack and collect the data made up a disproportionately large part of this research. The tools `sslsniff`[18] (specialized for this scenario), `mitm-proxy`[1] (dedicated tool, heavily customized[3] in an efford to make it usable for this research),`WebScarab`[2] (general purpose traffic analysis/manipulation toolkit that turned out to be too complex and buggy for our needs) and `stunnel` (general purpose SSL encryption wrapper) were among those evaluated for this purpose.)

The native support for the MITM use-case varies widely from tool to tool. While especially `mitm-proxy` (and to a lesser degree `sslsniff`) as a dedicated tool for this scenario is highly automated, conducting the same attack with the help of e.g., `stunnel` requires some additional work, mainly with respect to cer-

[3]in an unfortunately pretty non-reusable (i.e., "hackish") way

tificate handling and the "plumbing" of traffic[3].

Initial focus lay on making the `mitm-proxy` tool usable for the purposes of this research. Without modification, this tool only sniffs HTTPS without relaying any packets that come without a proxy-header. The open-source nature of this tool allowed for a number of modifications which were made towards a more universal proxy. Although this endeavour was ultimately successful for purposes of proxying and sniffing any traffic from a common web browser, we unfortunately could not apply the result to the Android platform due to compatibility issues that ultimately were beyond our capabilities to fix.

The `stunnel` tool is intended as a very general purpose way for agnostically wrapping TCP traffic inside an SSL-encrypted tunnel. It does not offer proxying capability itself. Indeed, an `stunnel` can only provide a single (prespecified) certificate to a connecting client, ruling out any scenario other than a very targeted attack (for a single known destination). With this tool, the entire actual MITM infrastructure has to be provided by different means. Apart from a `stunnel` listening for client connections, a second `stunnel` needs to be set up to on-demand connect to the remote server. The traffic data needs to be passed between client and server-`stunnel` and also logged. (In our implementation, a major mistake was made at the logging-stage of this setup: Since the `stunnels` were locally interconnected via sockets on the localhost it was incorrectly deemed sufficient to dump all traffic from the loopback interface. For unknown reasons, these dumps turned out to only show severly garbled and incomplete data, leading to several days of vain debugging before the problem was found.)

The approach that ultimately proved to be the most fruitful involved the usage of `sslsniff`. As the name implies, the purpose of this tool matches our usecase rather well. On paper[18], the tool supports the on-the-fly creation and signing of faked certificates depending on the intended destination and using those to automatically perform a MITM-attack on a connecting client that intends to establish an SSL session with any remote host. However, after the initial evaluation of the program did not produce very promising results[4], we temporarily focused on the above mentioned (and at that time) more promising alternative approaches instead before turning back to it once more. While the automatic mode still couldn't be made work as advertised, a different attack-mode was workable. In this *targeted* mode, one provides (manually signed) certificates for a number of targets. A connecting client then gets served with the corresponding faked certificate, depending on the intended destination. For convenience, a customized script was used to semi-automatically generate the required certificates from our fake root CA, which we had arranged to be trusted by the targeted clients.

(`sslsniff` is meant to be run as a *transparent* proxy. In this mode, a client is not aware of its connections going through a proxy and therefore does not need to have support for proxy headers. Transparent proxies have the drawback that they can not easily be deployed on the same host as the clients. With almost no knowledge about such topics, we faced a rather steep learning curve, since

[4]I.e., we experienced a lot of random segmentation faults during testing.

this turned out to require setting up a specialized machine for the purposes of interception and correct routing of all traffic through said machine with some additional port forwarding/packet mangling to reroute those packets we wanted to intercept to the corresponding MITM-setup.)

3.3 APK Disassembly

The ideal approach for learning more about the possible functionality of an application would be source code introspection. However, since the application is only distributed in a pre-compiled binary format, this is not directly possible. Fortunately there exist dissassemblers specifically targeting binaries compiled for the Dalvik virtual machine (which is central to the Android system).

> “Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.
>
> Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format[. . .].
>
> The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.”[16]

The `.dex` format is a very efficient binary format of machine instructions for the Dalvik VM. Although there exists some documentation about the exact opcodes used by the machine, reverse engineering an application on this level would be rather daunting. However, luckily there is `baksmali`:

> “smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android’s Java VM implementation. The syntax [. . .] supports the full functionality of the dex format (annotations, debug info, line info, etc.).”[10]

From the reverse engineering perspective this is almost as good as it gets. `baksmali` produces a form of high-level assembly, the syntax of which is very unfamiliar from the perspective of someone used to higher-level languages. Finally, if one desires to recreate the directory structure of the disassembled code, there is `APKTool`:

> “[APKTool] is a tool for reengineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications; it makes possible to debug smali code step by step. Also it makes working with app easier because of project-like files structure and automation of some repetitive tasks like building apk, etc.
>
> Features [. . .] decoding [of] resources to nearly original form (including resources.arsc, XMLs and 9.png files) and rebuilding them.” [5]

3.3.1 The Reverse Engineering Process

1. One might start with `grep`ing the `smali` files produced by running `APKTool` for any such telltale character sequences that were found through the above described traffic analysis.
2. Analyzing the matching lines from the previous step, one might find certain regularities, such as the character sequences being defined as string constants.
3. Constructing a broader search from those regularities (e.g., search for all string constants) and inspect those matches as well

4 Results

After having briefly looked at what the Android Market is and how the installation and remove procedures work, this section will give a more in depth view of the installation and remove procedures of the Android Market and discuss whether or not these procedures are vulnerable to malicious attacks. To be able to look into the installation and remove procedures in more depth, there has been made use of intercepted network traffic from the Android OS during these procedures and of the different logging possibilities available on the Android platform which can be accessed by using the `logcat` utility in the Android SDK.

Also an interesting observation is that the Android operating system tries to keep up a connection with the GTalk Servers from Google. A soon as there is network connectivity (cellular data or WiFi), the Android operating system connects to the GTalk Servers. Research done for a previous project indicates[5] that a connection with the GTalk Servers might be prefered over a cellular data connection, but such a connection is not mandatory. Contrary to WiFi, a cellular data TCP connection can persist even if the device switches connectivity to a different GPRS/UMTS cell. Due to the extremely small amount of data being passed through this connection, one could argue that the cost impact of this is probably negilible[6].

To connect to the GTalk Servers, the Android platform makes uses of the XMPP (Extensible Messaging and Presence Protocol)[25] to connect to `mtalk.google.com` on port `5228`. This protocol is an open-standard communications protocol which is based on XML (Extensible Markup Language).

4.1 How does the Android Market work

During the research project, we tested the following versions of the Android Market application (which were available at the moment):

- Android Market version 1.8.2
- Android Market version 2.2.6

[5]Reportedly, an Android phone connected to a WiFi hotspot for purposes of traffic analysis chose GPRS for the GTalk connection, as was determinde by also doing traffic analysis on the phone itself.

[6]It might be worth investigating if things are handled differently when the device is configured for roaming.

4.1.1 The installation procedure in depth

If we take a look at the log files created by the Android OS during the installation procedure with the following command `adb logcat -v long *:V`, we can see that the Android Market does not install applications itself, but it downloads the applications and let the Package Manager service install the applications.

Android Market version 1.8.2 To test this, we installed a random application from the Android Market onto the Android Emulator. While the installation took place, we did a Man-in-the-Middle attack on the Android Emulator and kept a look at the logfile of the emulator. If we then take a closer look at the logfile, you can see that the Market application (or `Vending.apk`) downloads the `apk` file, passes it on to the Package Manager which installs the `apk` and reports back to the Market application when the package is installed.

An interesting observation is that during the installation process, the SSL/TLS Man-in-the-Middle was able to intercept traffic on the GTalk Service connection. More specific, when the installation button in the Android Market application was pressed, it triggered some service at Google to send an `INSTALL_ASSET` message through the GTalk Service connection to the Android OS which then started the download and install procedures on the Android OS.

Below is a (pretty-printed) example of the `INSTALL_ASSET` messageintercepted during the installation of the *CNN* application from the Android Market:

```
tickle_id 1295709543744
assetid -3618933983076165489
asset_name CNN
asset_type APPLICATION
asset_package com.neoapps.android.cnnnews
asset_size 223526
asset_signature 5U3_HpYkDMUGNnhgihad4ZbWxi4
asset_blob_url http://android.clients.google.com/market/download/
                            Download?assetId=-3618933983076165489
                                    &userId=13426579174145539424
                                    &deviceId=4369701806128838147
download_auth_cookie_name MarketDA
download_auth_cookie_value 00380775007263922778
direct_download_key \$AIJ80zEinxs9R0bxyNxlsKn/e/UOTrkUcA==
                     @J0:1295709543754037X
```

Based on the intercepted `INSTALL_ASSET` message, we could see in the logfile of the Android system that the download is initiated with the `assetid`, which is passed to the Android Market application which starts the download of the `apk` file. This can be seen in a part of the logfile presented below:

```
[ 01-22 16:18:41.185    60:0x51 I/ActivityManager ]
Displayed activity com.android.vending/.TabbedAssetInfoActivity:
  1056 ms (total 1056 ms)

[ 01-22 16:18:48.445    60:0x101 I/ActivityManager ]
Starting activity:
  Intent { act=android.intent.action.VIEW
           cmp=com.android.vending/.AssetPermissionsSubActivity (has extras) }

[ 01-22 16:18:49.976    60:0x51 I/ActivityManager ]
Displayed activity com.android.vending/.AssetPermissionsSubActivity:
  1440 ms (total 1440 ms)

\textbf{ [ 01-22 16:18:59.445   183:0x133 D/vending  ]
[21] LocalAssetDatabase.notifyListener():
  -3618933983076165489 / DOWNLOAD_PENDING }

[ 01-22 16:19:02.345   183:0x140 D/vending  ]
[27] AssetDownloader.downloadAndInstall():
  Initiating Download for 1 applications.

[ 01-22 16:19:02.355   183:0x140 I/vending  ]
[27] DownloadManagerUtil.enqueueDownload():
  Enqueue for download com.android.vending.util.DownloadManagerUtil
                                                  \$Request@44fcdff0
```

Android Market version 2.2.6 To look into the installation procedure of this version of the Android Market application and compare it with the installation procedure of the 1.8.2 version of the Android Market application, the same procedure has been used. So a random application is installed from the Android Market onto the Android Emulator. While the installation took place, we did a Man-in-the-Middle attack on the Android Emulator and kept a look at the logfile of the emulator.

If we then take a closer look at the logfile, you can see that the activities used to install the `.apk` file are similar to the 1.8.2 version of the Android Market application. So the Market application (or `Vending.apk`) downloads the `.apk` file, passes it on to the Package Manager which installs the `.apk` and reports back to the Market application when the package is installed.

Although the installation procedure looks similar for both version when looking at the logfile of the Android Emulator, there is an interesting difference. During the installation of an application through the Android Market version 1.8.2, an `INSTALL_ASSET` message is send from an Google server through the GTalk Service connection to the Android platform. However, this does not seem to happen with the 2.2.6 version of the Android Market application. This can be checked by starting the `GTalk Service Monitor`, which is a tool that keeps track of the GTalk Service connections and shows the types of messages send and received.

4.1.2 The remove procedure in depth

So during the installation process, some service at Google is triggered to send and installation message (`INSTALL_ASSET`) to your Android device, which then starts the installation. But how does it work when a user wants to remove an application, is that then also initiated by and deinstallation message (`REMOVE_ASSET`) like John Oberheide described[7]?

If we take a look at the logfiles created by the Android OS during the remove procedure with the following command adb logcat -v long *:V, we can see that the removing of applications is not done through the Android Market (there is not even an option in the Android Market application to remove applications) but you need to take the following steps.

In the Settings application of the Android system, there is an option to manage packages. Within that package managing option, you can delete an installed application, which will cause the PackageInstaller to start the deinstallation process by opening the Package Manager, which removes all files belonging to the specific package.

So when a user removes an application, there is no deinstallation message (`REMOVE_ASSET`) involved, this means that such a message is only used when Google remotely deinstalls an application from an Android device[7].

4.2 Security Implications

The Android Market neither appears to be a particularly easy target nor does it have many obvious security flaws. However, there is the interesting case of AppBrain:

4.2.1 AppBrain

According to their website, AppBrain is a website for discovering Android applications and in addition to providing search and browse functionality, users of Android phones can download the applications they choose by simply clicking an install button on the website. AppBrain then stores the application in an application wish list and a companion native Android application then lets the user efficiently make all the desired changes on the phone.

So how does this installation process work? Does this interact with the Market application on the Android or does it use a separate service? To find out more about the process, we contacted the developers of the AppBrain concept for more information and performed a Man-in-the-Middle attack on the installation process during the use of AppBrain.

When we did a Man-in-the-Middle attack on the installation process and looked at the logfiles, we were able to intercept an `INSTALL_ASSET` message just as with the installation process of the Market application and we could also see that the same processes were called as during the installation process of the Market application.

Also according to the developers of AppBrain, the Market application from Google does not immediately initiate the download of an application once the install button is pressed, but it first sends a message to the Google servers. The Google servers then respond by sending a message which initiates the download and installation (the `INSTALL_ASSET` message mentioned before). AppBrain simulates this process by faking the initial message send from the Market application to the Google server when pressing the installation button on the website, which then starts the download and installation process on the Android Phone. To be able to use this functionality, you need to install the AppBrain application and the Fast Web Installer application, which "fake" the installation process of the Market application.

5 Mitigation Proposal

Since the initial suspicion of “back-door-like functionality” couldn’t be confirmed, the point of discussing mitigations is almost mute. The remote application removal feature remains the almost sole point of potential issue for some. In order to disable this functionality, the following approach might solve the issue:

Binary Android applications can be disassembled as well as reassembled back to a working version using the above described `APKTool`. This can facilitate the creation of a modified version of the Market application with the relevant functionality disabled by inserting a “premature” return instruction at the begining of the corresponding section of the disassembled code. As a result of the premature function return, the rest of the code is not executed, so a call to that subroutine remains without effect.

Since the Market application is proprietary however, the distribution of a patched binary has legal implications which one might wish to avoid. This could be achieved by developing an easy to use patch application (similar to the iPhone jailbreak toolkits) which each person with an interest in patching his own Market application can then execute. The patch application (which can’t run on an Android device itself) could make use of assembly-level patches in conjunction with a bundled distribution of the `APKTool` in order to avoid any legal issues which might remain with common binary patches.

6 Findings Beyond Scope

While researching the feasibility for using the above introduced APKTool to automatically rebuild an application for the purposes of intermediate patching, it was discovered that (due to issues with the cryptographical signing of the reassembled application) the Market application honors the (sophisticated) application permission system which is integral part of the android platform. Specifically, permission error messages were observed on the debug console of an emulator instance after a rebuilt version of the Market application was instructed to install a random app. The installation failed accordingly, since the permission system reported insufficient rights for the application to initiate this process.

This discovery strikes a significant blow to the (implicit) assumption (and premise of this project) that, as part of the core system, the Market application has additional privileges and somehow bypasses the mechanisms which were put into place for securing the system against malicious third-party apps. Indeed, it turns out that even functionality which might be deemed “critical” or “special” (such as the discussed remote application removal) requires explicit permission from the system.

Had this fact been known from the beginning, we would have "wasted" less time on trying to "open the black box" that is the Market application and instead shifted the focus of this research to the sophisticated (and thankfully open source) framework exerting equal control over all services and applications running on an Android device.

6.1 Android Permission System

The Android permission system has been subject to repeated criticism[4] from various sources. While allowing an application developer to have exceedingly fine-grained control over what kinds of permissions his application may need in order to function properly, the potential user of the application is only given two options prior to installation:

1. grant all permissions the application requests in order to be able to install and use it
2. not to install the application

The ability to selectively deny an application the permission to send (expensive) MMSes would be very welcome by many users, but apparently Google (as the major developer of the Android system) has little interest in rectifying this issue.

Some very promising scientific research has been done in this area with the development of Apex[20], an extension upon the core (and open source) parts of the android system to expose the fine grained control over permissions to the end user of an application. However, upon iquiry about the status of the project the former lead developer declared it effectively dead due to university policies and copyright issues (among others). However he declared himself willing to aid any reimplementation efford of his work with the expressed goal of releasing the results as open source for wider adoption within the android community. (In face of this remarkable proposition, further steps have been taken to hopefully ensure the revival of this work.)

7 Conclusion

Although the questions we proposed to investigate at the beginning of this project appear to have lost relevance somewhat, it maybe makes sense to recapitulate on them anyway:

The suspicion we set out to investigate, namely the presence of hidden remote-controllable functionality or even backdoors in the Android Market Application proved to be unfounded. As an answer to the question "what is the android market?" one might at least reply: "It is not (obviously) a back-door". Contrary to our expectations, the feature's implementation does not take the form of a hidden backdoor at all. Our work could certainly have been made a lot more straight forward, had we known or even suspected this from the beginning. The surprising fact that the entire application, even including "critical" functionality that is intended to be remotely enforceable is implemented without bypassing the built-in permission system only became clear in the very late stages of this project.

The answer to the third question ("Are these procedures vulnerable for malicious attacks? And, if so, for what kind of attacks?") turns out to be a rather boring "not really". (The MITM-like modus operandi of AppBrain might be worth mentioning, but there are no obvious gaping holes in the system.)

This does not mean that the application is without issue however. Many users rightfully even regard the remote-deletion capability with suspicion and an uneasy feeling. While this "feature" "merely" impacts the integrity of one's mobile device, the privacy implications are thankfully limited.

Final remark: As outlined in section 6 our very "forensic" approach to analyzing the application as a black box first might have been not entirely applicable for this task.

A Division of Workload

- Week 1
  - Initial research - Both
  - Project proposal - Both
- Week 2
  - Full background research
    Practical information - Both
    Related research - Bastiaan
  - Practical analysis of the application
    MITM Sniffing - Both
    App Analysis - Thorben
  - Start on paper - Bastiaan
- Week 3
  - Theoretical analysis of the gained data, w.r.t., privacy issues
    Captured network traffic - Both
    Decompiled program code - Thorben
  - Develop proposal for mitigations - Thorben
  - Continue working on final paper - Bastiaan
- Week 4
  - Additional practical research[7] - Thorben
  - Finalize both paper and presentation - Both
- Week 5
  - Presentation - Both

[7]Due to unexpected discoveries

References

[1] http://preview.tinyurl.com/6ozvel.

[2] http://preview.tinyurl.com/2f3vpo.

[3] http://preview.tinyurl.com/6kqp55h.

[4] http://preview.tinyurl.com/377v7jh.

[5] Brut.all. android-apktool. online documentation, February 2011. http://preview.tinyurl.com/yfs27jl.

[6] Rich Cannings. Exercising our remote application removal feature. blog post, June 2010. http://preview.tinyurl.com/2dofgzs.

[7] (crve). Google uses remote delete to remove android apps from smartphones - update. only news post, June 2010. http://preview.tinyurl.com/5top56a.

[8] (crve). Android market poses remote installation risk - update. only news post, February 2011. http://preview.tinyurl.com/5sfajvs.

[9] (crve). Google launches android market on the web. only news post, February 2011. http://preview.tinyurl.com/48wfh8k.

[10] Jesus Freke. smali. online documentation, February 2011. http://preview.tinyurl.com/mx4y6d.

[11] Google. Android emulator — android developers. appendix, January 2011. http://preview.tinyurl.com/ahq8pk.

[12] Google. Android sdk — android developers. appendix, January 2011. http://preview.tinyurl.com/dhcpvy.

[13] Google. Glossary — android developers. appendix, January 2011. http://preview.tinyurl.com/6k3ph9o.

[14] Google. Security and permissions — android developers. appendix, January 2011. http://preview.tinyurl.com/y9bwnlj.

[15] Google. Tools overview — android developers. appendix, January 2011. http://preview.tinyurl.com/ykqfjy2.

[16] Google. What is android? — android developers. online documentation, February 2011. http://preview.tinyurl.com/d9gzek.

[17] LarsTobiasSkjongBorsting. Importrootcert - cacert wiki, 2010. http://preview.tinyurl.com/26kv382.

[18] Moxie Marlinspike. Moxie Marlinspike ¿¿ software ¿¿ sslsniff, 2009. http://preview.tinyurl.com/nhfjw3.

[19] Varunkumar Nagarajan. Varun's scratchpad: [how to] install android market on emulator. blogpost, November 2010. http://preview.tinyurl.com/23m42n5.

[20] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 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*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM. `http://preview.tinyurl.com/65ysctn`.

[21] Jon Oberheide. A peek inside the gtalkservice connection. blog post, June 2010. `http://preview.tinyurl.com/2dg588o`.

[22] The Android Open Source Project. Packagemanager.java. source code, / 2006. `http://preview.tinyurl.com/5w8stt5`.

[23] The Android Open Source Project. Packageinstalleractivity.java. source code, / 2007. `http://preview.tinyurl.com/6en5hjk`.

[24] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google android: A state-of-the-art review of security mechanisms. paper, / 2009. `http://preview.tinyurl.com/6adnxrp`.

[25] Wikipedia. Extensible messaging and presence protocol - wikipedia, the free encyclopedia, 2011. `http://preview.tinyurl.com/yr5zah`.