POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica Informazione e Bioingegneria

A Study of the Security Aspects of the Green Move Vehicle Sharing System

Relatore: Prof. Rossi Matteo Giovanni
Correlatore: Ing. Edoardo Vannutelli Depoli

Tesina di Laurea di:
Fang Wenpiao, matricola 780709

Anno Accademico 2013-2014

ABSTRACT

A Study of the Security Aspects of the Green Move Vehicle Sharing System

by

Fang Wenpiao, Master of Science

Politecnico di Milano, 2014

The idea of vehicle sharing system can be backdated to tens of years ago both in Europe and USA. Vehicle Sharing can provide us numerous benefits by ways of relieving the traffic jam problem, reducing the air pollution, giving back the cities more space, cutting down the cost on one's trip and so on. However, it was not until the emergence of Electric Vehicles that vehicle sharing became more and more close to us. In the meanwhile, the vehicle sharing systems are required to evolve to meet several new important requirements: such as easier interactions with the system for end users, more possibilities for both stakeholders and administrators to benefit from the systems, smoother configurations.

The Green Move Project of Politecnico di Milano is exactly devised with all these in mind: with Green Move Center one could easily reserve a vehicle and receive a digital key from it, then use it to unlock the vehicle; the stakeholders can install variety of dynamic green move applications such as advertisements in the Green e-Box; the system can be easily configured for either scooters or four wheels' vehicles.

This thesis looks inside the components of the Green Move Vehicle Sharing System (GMVSS), especially the Green e-Boxes which are installed in the vehicles and used to interact with them, analyzes the security issues inside, gives several experiments to exploit the vulnerabilities, and then proposes solutions to avoid such vulnerabilities.

Keywords: Vehicle Sharing System, Electric Vehicles, Green Move, Vulnerabilities

ACKNOWLEDGMENTS

I would like to express my gratitude to all those kind people around me, who helped me a lot to make this thesis possible for me, to only some of them, it is possible to mention here.

First of all, I would like to express my deepest sense of gratitude to my supervisor Prof. Rossi Matteo Giovanni, who offered me his continuous encouragement and advice throughout the course of this thesis, and allowed me to work in the lab, without him, this thesis would not have been completed.

I am grateful to Prof. Cugola Gianpaolo for providing me the opportunity to work on the Green Move project. I am impressed by his unsurpassed knowledge and the ability to explain a complicated problem in such an easy and understandable way.

I am thankful to Ing. Edoardo Vannutelli Depoli for providing me the source code of Green Move and a lot of technical guidances during the course of the project. He spared a lot of his time to help me to understand the architecture of the project.

My sincere thanks also goes to Daniele Rogora for his kindness, friendship and support during my writing of this thesis.

Last but not least, I would like to thank my family, for their endless support and great patience during my study here in Politecnico di Milano.

CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

The term "vehicle sharing" is now the widely accepted international term. The principle of vehicle sharing is that individuals gain the benefits of private cars without the costs and responsibilities of ownership [1]. Instead a household accesses a fleet of vehicles on an as-needed basis.Vehicle Sharing has sprung up in different parts of the world and operations are organized in many different ways in different places.

However, in order to become a real alternative to the private owned diesel or petrol cars, the existing vehicle sharing systems still need a push. The system needs to cover a wide variety of vehicles to fulfill the requirements from different types of end users [2], for example, while parents may decide to choose a 4-wheeled, 4-seats vehicle to send their kids to schools, an office employee could get a 2-seats vehicle to take him home from work. Additionally, end users are willing to use multi-functional and easy-using systems to enjoy the benefits of technological evolutions, for example, users would like the system to be smart enough to faciliate their driving, to remind them of important notifications.

The Green Move Vehicle Sharing System (GMVSS) could answer the above needs by its key component, Green e-Box, and its flexible mechanism that allows applications to be dynamically installed and removed from vehicles, to tailor in-vehicle functions to the needs of the user. The Green e-Box (GEB) is a hardware/software interface, which allows the system to interact with heterogeneous fleet of vehicles in a uniform way (described in more detail in the backgroud chapter). Those dynamically installed or removed applications are called Green Move Applicaitons or Dynmamic Applications in the system. They are bundles of java code that, packeaged into JAR files, can be sent to the GEBs, where they are executed within the Green Move Android application running on the GEB. The Dynamic Applications could provide users more convenience and better feedbacks during their trip using the system. For example, the Drive Style Dynamic Application developed by the Green Move group could offer the drivers excellent drive experience, the Advertisement Dynamic Application can provide the users interesting ads during the trip.

Since the developers of the Dynamic Applications may be be different from original members of the green move team, all Dynamic Applications should be guaranteed to run in a secure enviroment so that they could not misuse the security-sensitive data, or access the security-sensitive operations improperly. Therefore it is necessary to build specified sandboxes for the Dynamic Applications running within them.

In the Java platform, a normal way for sandboxing a Java application usually involves two steps: first we have to make sure that only the needed permissions to the untrusted code are granted in the policy file; then we need to customize a Security Manager to control all the accesses to the protected system resources carefully.

This thesis first analyzes the interactions between the Dynamic Applications and the rest part of the system, and then builds a customized Security Manager to avoid the vulnerabilities found in the system.

However, all Dynamic Applications are designed to run on the Android platform, which takes a different security model from the Java sandbox model. We are more encouraged to take advantage of the applcation sandbox model on the Android platform. Android platform adopts the user based securiy mechanism in Linux, by default, each Android application runs as a separate process in the platform. Based on such application sandbox model, I proposed a new implementation of the GEB application to avoid the vulnerabilities of the existing system described in this thesis.

The rest of this thesis is organized as follows. Chapter 2 introduces the relevant technologies, chapter 3 presents the architecture of the Green Move Vehicle Sharing System, chapter 4 presents several vulnerabilities of the system and shows how they can be exploited, then chapter 5 proposes some solutions to them, and finally there is the conclusion part in chapter 6.

CHAPTER 2

Background

As explained in Chapter 3, the Green Move system relies on a number of technologies, which are introduced in this chapter to make the document self-contained.

2.1 Introduction to Android Platform

Android is a software stack for mobile devices which includes a Linux-based operating system, middle-wares and key applications. The Linux kernel handles core system services and acts as hardware abstraction layer (HAL) between the physical layer and the Android Software Stack. Figure 2.1 shows the software layers of the Android Platform.

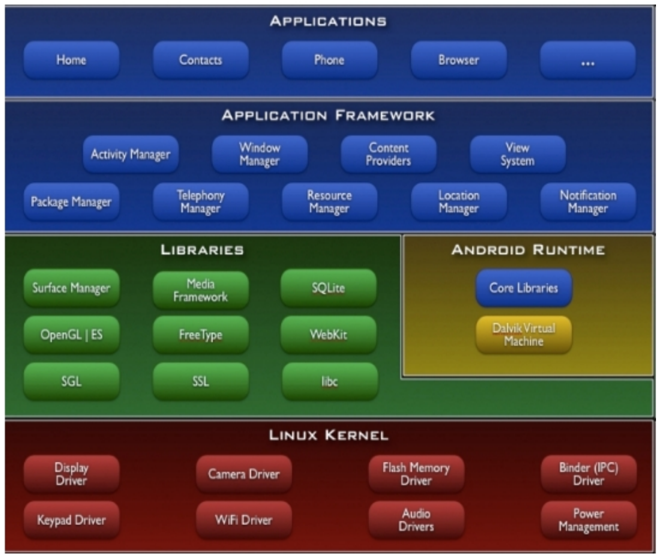


Figure 2.1: Architecture of Android Platform

Android applications are written in Java programming language, and run inside in the Google customized virtual machine called Dalvik, quite different from the normal

Java Virtual Machine (JVM). Each Android application runs within an instance the Dalvik Virtual Machine (DVM), which resides in a Linux kernel managed process, see Figure 2.2.

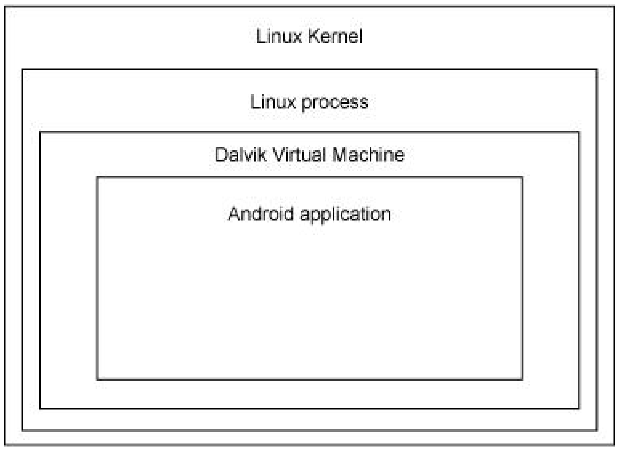


Figure 2.2: Android Dalvik Virtual Machine

Typically an android application contains one or more of the following independent components [3] [4]:

Activities

An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface.

Services

A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

Content providers

A content provider acts as an simplified database service. Its main job is to manage accesses to the persisted data, whether they are located in the local file system, a SQLite database, or any other persistent storage.

Broadcast receivers

A broadcast receiver responds to the broadcast events from the system, and the events could be: the screen is turned off, a requirement to start a service from another application and so on.

All the above components could either be configured in the same Linux process, or they could also run in separate processes.

2.2 Sandboxing Untrusted Code

Sandboxing is a technique related to executing untrusted code in a environment without letting it tamper with the surrounding code, to make sure that the untrusted code can not alter the way the surrounding program behaves. The sandbox should also prevent untrusted code from accessing sensitive resources and operations of a system without giving required permissions.

In the Java platform, a sandbox can be implemented using the Security Manager and the Permission:

```java
SecurityManager securityManager = System.getSecurityManager();
if (securityManager != null) {
  Permission permission = ...;
  securityManager.checkPermission(permission);
}
```

A permission represents an access to a system resource, like the following permission defines the access to the thesis.pdf file:

```java
Permission readPermission = new java.io.FilePermission('/home/fang/thesis.
    pdf', 'read');
```

Finally one could grant a specified permission to an untrusted program via the policy file in Java.

2.3 T-Rex Middleware

T-Rex [5] is a specially designed Complex Event Processing middleware, which could be used to generate Composite Events from a big amounts of event notifications flowing from peripheral to the T-Rex complex event processing engine. Along with T-Rex there is the TESLA language, which is used to define the rules for generating new events. So users are allowed to customize a complex rule for an appropriate case, then make an subscription from the middleware, and wait for the publication of a new event notification.

The general architecture of such event-based applications is shown in Figure 2.3. At the peripheral of the system are the sources and the sinks. The former observe primitive events and report them, while the latter receive composite event notifications and react to them. The task of identifying composite events from primitive ones is performed by the Complex Event Processing (CEP) Engine.

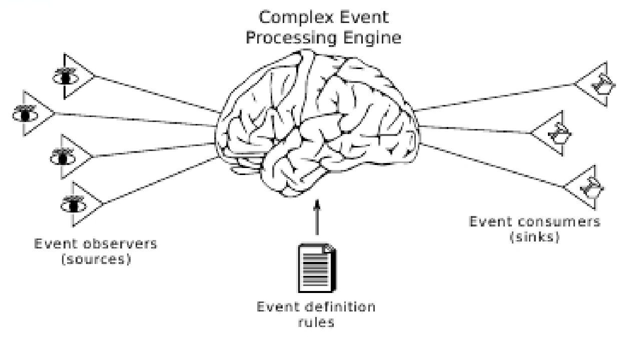


Figure 2.3: The high-level view of an CEP application

2.4 Rails, JRuby on Rails

Rails is a poluplar web application framework, which offers the developers the tools to create web applications in an intuitive and agile way. It has gained great success in the past few years, because of its unique solution to the developing of web applications [6]. The two most important principles of Rails are:

Dont Repeat Yourself (DRY) According to the DRY principle, each piece of information should live in exactly one place. If you're writing a payroll app where each Employee needs a name and a salary, you shouldn't have to define those fields in the database and your Ruby class. And indeed, in Rails, you don't – property information resides only in the database.

Convention over Configuration Also known as "sensible defaults" – when you do things the way Rails expects you to, you will not need to configure much. For example, Rails will automatically find the table for your Recipe class, provided the table is named recipes in the database.

Rails is designed to support the Model-View-Controller (MVC) pattern, and is mainly constituted by the following components:

ActiveRecord The ActiveRecord component constitutes the model part of the MVC pattern, by paring database tables with simple wrapper classes that embody the program's logic. The implementation takes advantage of Ruby language features and relieve the programmers the pain of defining model classes. It supports almost all the popular databases, such as MySQL, PostgreSQL, and so on.

ActionPack ActionPack takes care of presenting the models to users and responding to the actions they perform. It consists of three parts: ActionView and ActionController correspond to the View and Controller parts of MVC, while ActionDispatch connects a request to a controller.

The other important components are: ActionSupport which includes a lot of extensions to the Ruby core classes; ActionResource, which helps users to consume the application services in a RESTful way; ActionMailer, helps to deal with emails.

JRuby is a implementation of Ruby programming that runs on Java Virtual Machine (JVM). With JRuby on Rails, web application developers can not only fully exploit all the benefits from the Rails framework, but also take advantage of the Java Platform.

CHAPTER 3
The Green Move Architecture

As explained in chapter Introduction, the Green Move Vehicle Sharing System (GMVSS) is designed to be flexible, highly configurable and able to accommodate a wide variety of the vehicles. As shown in Figure 3.1, the GMVSS is based on three main components: the User Side Application, the Green Move Center and the Green e-Box [7] [8].

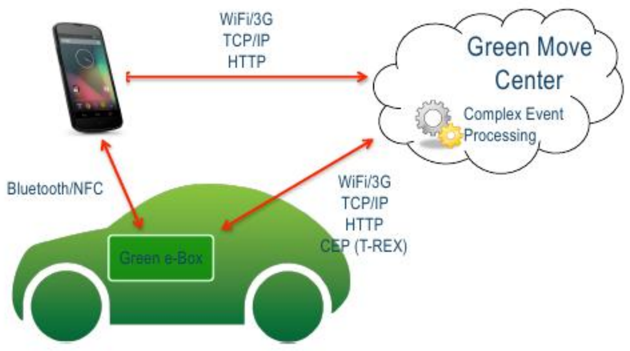


Figure 3.1: Green Move Vehicle Sharing System Components

The rest of this chapter presents the components of the GMVSS, with particular focus on the Green e-Box and the notion of dynamic applications.

3.1 User Application

User Application resides on the smartphones of end users. It is used to interact with the Green Move Center and the Green e-Box, for example it could be used to check the nearest available Electric Vehicles around the end user, make a reservation for an available vehicle and receive the Digital Key for that specified vehicle; obtain the basic information about it, and access all the other useful information about the system. All the communications between the User Application and the Green Move Center are directed through the well-known http/https protocols.

When it comes to use the vehicle for a trip, the end user can use the received Digital Key to contact the Green e-Box installed in the vehicle to unlock it via the Bluetooth or NFC channels.

3.2 Green Move Center

Green Move Center acts as the portal of the GMVSS, it is built on the JRuby on Rails framework, easy for system developers to extend and bring new features into the system. As illustrated in Figure 3.1, the Green Move Center has the mentioned T-Rex middleware running within it. System administrators are allowed to manage the vehicle information through the Green Move Center, application developers are encouraged to upload here diverse Dynamic Applications which would be installed in the Green e-Box later.

The T-Rex middleware is installed with specified rules for generating complex events for the Green e-Box, such as the event to install a dynamic application in the Green e-Box. The events could either be broadcast to all receivers (all Green e-Boxes) in the system, or only sent to a specified receiver.

3.3 Green e-Box

Green e-Box (GEB) represents the core part of the GMVSS, it makes the GMVSS a big difference from the other existing vehicle sharing systems. The GMVSS is designed to be able to deal with a heterogeneous fleet of vehicles, of different makers, so it is crucial to abstract away from the details of each vehicle. Therefore, the goal of the GEB is to provide such an abstraction layer that allows the system to interact with any GEB-equipped vehicle in an high-level way.

The GEB is composed of a low-level embedded board and a high-level Android

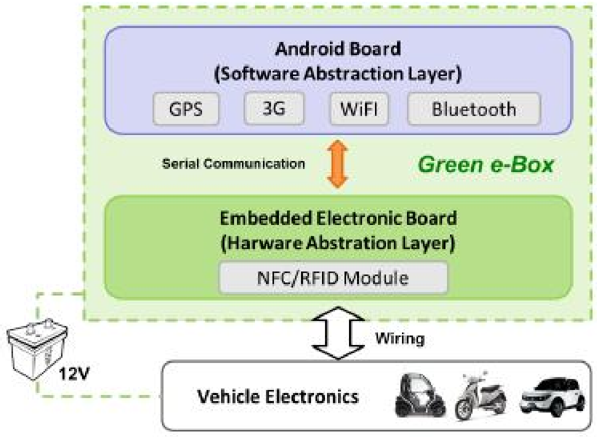


Figure 3.2: Green e-Box

board, see Figure 3.2, and it is directly connected to the 12V line of the Electric Vehicle to guarantee a non-stop service. The low-level embedded board, Embedded Electronic Board (Hardware Abstraction Layer), is wired to the vehicle electronics and is designed to abstract the vehicle-specific details; thus, it creates a virtual layer between software applications and the actual hardware, providing a general communication protocol to the high-level layers built on top of it. To achieve this, the embedded board has a CAN-bus to retrieve data directly from the vehicle ECU and several analog and digital input/output channels so that the GEB can be installed on a large variety of heterogeneous vehicles. A microcontroller handles each signal, acquiring the vehicle data at a constant rate and, since the set of available signals is strongly vehicle-dependent, it collects them into well-defined packets so that they can be easily transmitted to the high- level Android board. While the high-level Android board, Software Abstraction Layer, receives (in a vehicle-independent way) the data from the low-level board, store them in a object, instance of DataModel, and allows the Dynamic Applications residing on it to access the object easily. Applications running on the Android board can also send commands to the low-level embedded board, which further are delivered to the vehicle electronics [7].

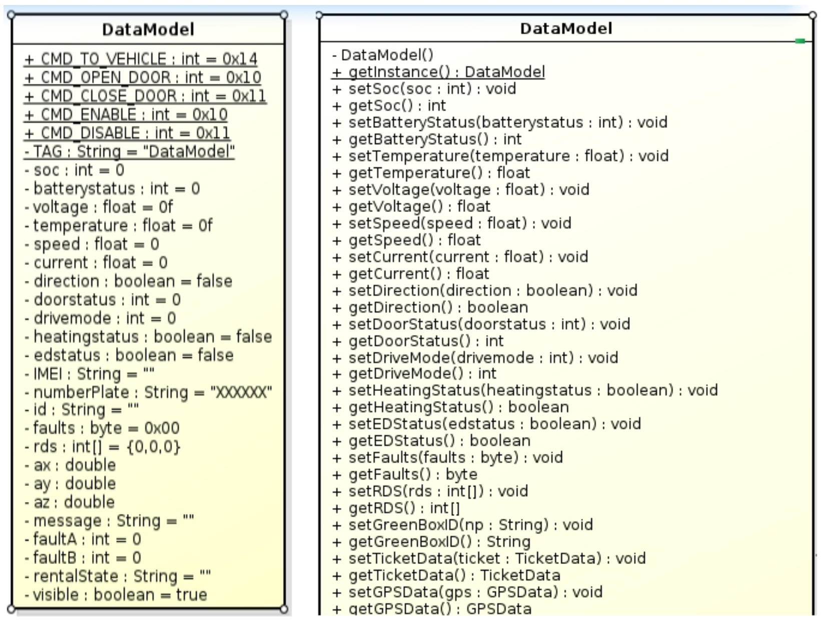
As one may notice in the figure, the Green e-Box is equipped with the NFC/RFID and Bluetooth links, thus it is possible for end users of the GMVSS to use an smartphone to contact the Green e-Box even without GPS or 3G networks.

The mentioned DataModel class is implemented by the Singleton design pattern to guarantee there is only one instance for a running GEB application. It acts as a data center for Dynamic Applications, and there are mainly two kinds of data that may interest a Dynamic Application:

1. Data from the vehicle.

2. Location related data

The left side of the class diagram in Figure 3.3 shows the attributes in the existing DataModel class, while the right part presents the existing operations inside it.



```
DataModel
+ CMD_TO_VEHICLE : int = 0x14
+ CMD_OPEN_DOOR : int = 0x10
+ CMD_CLOSE_DOOR : int = 0x11
+ CMD_ENABLE : int = 0x10
+ CMD_DISABLE : int = 0x11
- TAG : String = "DataModel"
- soc : int = 0
- batterystatus : int = 0
- voltage : float = 0f
- temperature : float = 0f
- speed : float = 0
- current : float = 0
- direction : boolean = false
- doorstatus : int = 0
- drivemode : int = 0
- heatingstatus : boolean = false
- edstatus : boolean = false
- IMEI : String = ""
- numberPlate : String = "XXXXXX"
- id : String = ""
- faults : byte = 0x00
- rds : int[] = {0,0,0}
- ax : double
- ay : double
- az : double
- message : String = ""
- faultA : int = 0
- faultB : int = 0
- rentalState : String = ""
- visible : boolean = true
```

```
DataModel
- DataModel()
+ getInstance() : DataModel
+ setSoc(soc : int) : void
+ getSoc() : int
+ setBatteryStatus(batterystatus : int) : void
+ getBatteryStatus() : int
+ setTemperature(temperature : float) : void
+ getTemperature() : float
+ setVoltage(voltage : float) : void
+ getVoltage() : float
+ setSpeed(speed : float) : void
+ getSpeed() : float
+ setCurrent(current : float) : void
+ getCurrent() : float
+ setDirection(direction : boolean) : void
+ getDirection() : boolean
+ setDoorStatus(doorstatus : int) : void
+ getDoorStatus() : int
+ setDriveMode(drivemode : int) : void
+ getDriveMode() : int
+ setHeatingStatus(heatingstatus : boolean) : void
+ getHeatingStatus() : boolean
+ setEDStatus(edstatus : boolean) : void
+ getEDStatus() : boolean
+ setFaults(faults : byte) : void
+ getFaults() : byte
+ setRDS(rds : int[]) : void
+ getRDS() : int[]
+ setGreenBoxID(np : String) : void
+ getGreenBoxID() : String
+ setTicketData(ticket : TicketData) : void
+ getTicketData() : TicketData
+ setGPSData(gps : GPSData) : void
+ getGPSData() : GPSData
```

Figure 3.3: DataModel Class Diagram

All setXXX operations here are quite security-sensitive; they are designed to change the state of the single DataModel instance shared among all Dynamic Applications, so they need to be hidden away from Dynamic Applications, or at least should be provided with special security mechanisms.

3.4 Dynamic Applications

With the help of Android Board, it is possible to develop an Android application atop the Android platform. The GMVSS has developed the GEB application to dynamically install applications (Dynamic Applications) from the Green Move Center, thanks to the mechanisms explained later.

The GEB application relies on its Green Move Container component to hold the data from Hardware Abstraction Layer, and consume the data for management. Furthermore, the Green Move Container offers Dynamic Applications (Green Move Applications) the runtime environment, such as the T-Rex channel to communicate with Green Move Center, the GUI handler to show messages on the screen, and the mechanism to reed vehicle data and so on. It is depicted as Figure 3.4.

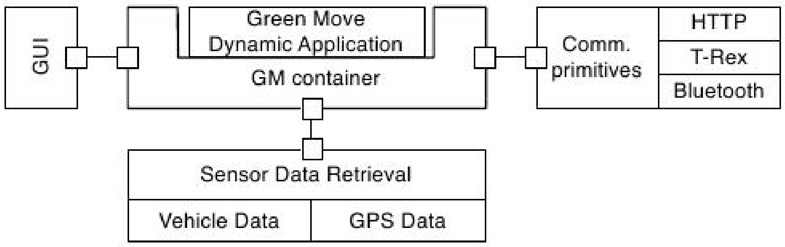


Figure 3.4: Green EBox Application

Dynamic Applications are key elements of the GMVSS, they could be loaded and unloaded automatically in the Green e-Box (to be more accurate, in the Green Move Container). Basically, they are Java applications packaged as Jar files, they could offer users meaningful services, for example, the developed Driving Style Dynamic Application could help the driver to reduce the energy consumption of the vehicle.

All Dynamic Applications rely on the tools provided by the Green Move Container to interact with the system. For example the following tools can be provided to a

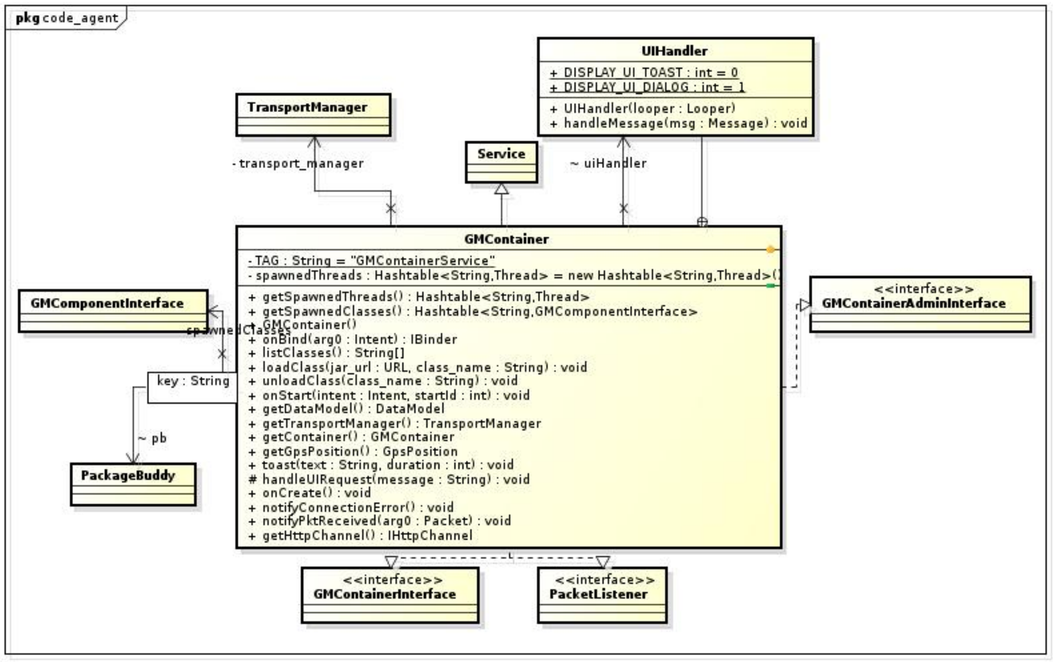

pkg code_agent
UIHandler
+ DISPLAY_UI_TOAST : int = 0
+ DISPLAY_UI_DIALOG : int = 1
+ UIHandler(looper : Looper)
+ handleMessage(msg : Message) : void
TransportManager
Service
- transport_manager
~ uiHandler
GMContainer
- TAG : String = "GMContainerService"
- spawnedThreads : Hashtable<String,Thread> = new Hashtable<String,Thread>()
+ getSpawnedThreads() : Hashtable<String,Thread>
+ getSpawnedClasses() : Hashtable<String,GMComponentInterface>
+ GMContainer()
+ onBind(arg0 : Intent) : IBinder
+ listClasses() : String[]
+ loadClass(jar_url : URL, class_name : String) : void
+ unloadClass(class_name : String) : void
+ onStart(intent : Intent, startId : int) : void
+ getDataModel() : DataModel
+ getTransportManager() : TransportManager
+ getContainer() : GMContainer
+ getGpsPosition() : GpsPosition
+ toast(text : String, duration : int) : void
# handleUIRequest(message : String) : void
+ onCreate() : void
+ notifyConnectionError() : void
+ notifyPktReceived(arg0 : Packet) : void
+ getHttpChannel() : IHttpChannel
GMComponentInterface
spawnedClasses
key : String
~ pb
PackageBuddy
<<interface>> GMContainerAdminInterface
<<interface>> GMContainerInterface
<<interface>> PacketListener

Figure 3.5: Green Move Container Class Diagram

Dynamic Application by the Green Move Container:

1. DataModel, the Green Move Container contains a reference to the DataModel instance mentioned in the above section.

2. TransportManager, used to connect to the T-Rex Server, subscribe to events of the T-Rex Server and receive the published events from it.

3. UI handler, used for showing information on the GMVSS main UI.

It is shown in Figure 3.5 that the Green Move Container extends the Service class, the Service which again is a subclass of Android Context class. Thus, it would be quite serious if one Dynamic Application has a direct reference of the Green Move Container, however, such issue exists in the system right now. The following chapter shows several ways to exploit such vulnerability.

The sequence diagrams in Figure 3.6 and Figure 3.7 help to understand the work-flow for downloading and starting a Dynamic Application in the system. Here may

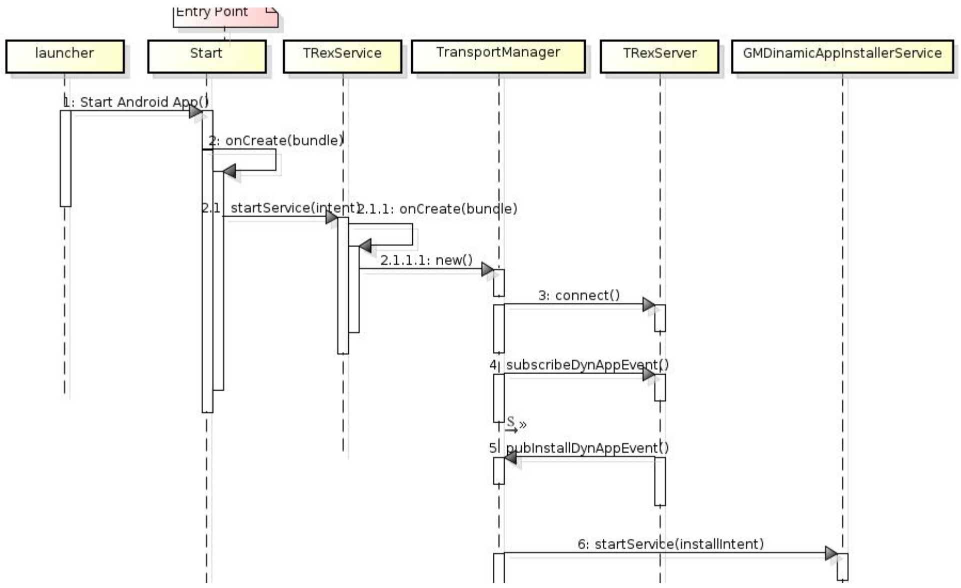


Figure 3.6: Sequence Diagram – Download and Start a Dynamic Application (1)

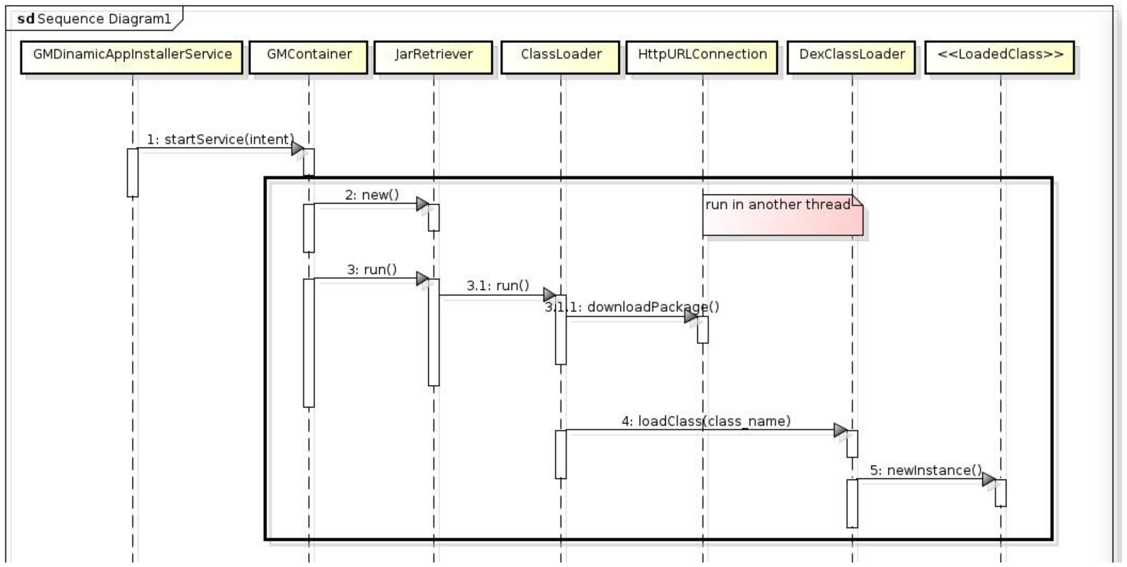


Figure 3.7: Sequence Diagram – Download and Start a Dynamic Application (2)

be a proper place to point out that all the steps involving in launching a Dynamic Application run in the same Linux process space. It is because of this, Dynamic Applications may affect each other.

As pointed out in Figure 3.7, after the package is downloaded by HttpURLConnection, the Android DexClassLoader is used to load the entry class of the package into the Dalvik Virtual Machine, then Java Reflection methods are used to initialize and start a Dynamic Application from the entry point. Although Dynamic Applications are designed to run in separate threads to secure themselves in thread level, however, the security model for an android application is based on the concept of Linux process, so the thread level security mechanism is not sufficient for a Dynamic Application.

On one hand GMVSS relies on Dynamic Applications for its powerful functionalities, on the other hand since Green Move Dynamic Applications are loaded from remote sources, which means they are kinds of untrusted code, hence they may bring the system security issues, therefore the system needs some proprietary mechanisms to guarantee they behave in the expected way. However, the existing implementation of GEB application lacks such mechanisms.

CHAPTER 4

Vulnerabilities of Green Move Vehicle Sharing System

This chapter presents several vulnerabilities and shows how they can be exploited. All the developed android applications (Dynamic Applications) are written in pure Java language here, and the experiments are done with the help of Android Emulator running on Linux 3.2.0-60-generic-pae; the tests have been carried out using Androird 4.3 Jelly Bean.

4.1 Code to Load and Start a Dynamic Application

The detailed work-flow of downloading and starting a Dynamic Application is already explained in the previous chapter, here the related source code is presented to help to understand the ways to exploit vulnerabilities in the following section.

As shown in the following code, all Dynamic Applcations must implement the **GMComponentInterface** interface, where the *public void init(GMContainerInterface container)* method is designed to be called by the GEB application program, to start the Dynamic Application. The argument **container** of the init method is designed to hold the reference of the GMContainer. The GMContainer, as explained in the previous chapter, is used to offer Dynamic Applications services like T-Rex Channel, GUI handler and so on. Finally, it is the JarRetriever object of the GEB application that really downloads and starts a Dynamic Application.

Listing 4.1: GMContainer.java

```
public class GMContainer extends Service implements GMContainerInterface,
    GMContainerAdminInterface, PacketListener
```

Notice *GMContainer* is a subclass of *Service* and also implements the *GMContainerInterface*, this is used to exploit the vulnerabilities later on.

Listing 4.2: JarRetriever.java

```

private GMContainer gMContainer;
\\ ... ... skip some other piece of code here
DexClassLoader classloader = new DexClassLoader(
  libPath, tmpDir.getAbsolutePath(),null,this.getClass().getClassLoader());
Class<?> classToLoad = classloader.loadClass(class_name);
Object myInstance  = classToLoad.newInstance();
\\ ... ... skip some other piece of code here
Class<?> partypes[] = new Class[1];
partypes[0]= Class.forName("it.polimi.greenmove.code_agent.
    GMContainerInterface");
Method init = classToLoad.getMethod("init", partypes);
init.invoke(myInstance, gMContainer);
```

The Attribute gMContainer is initialized with the reference of the GMContainer instance. And the method init is defined in GMComponentInterface that each Dynamic Application must implement.

Listing 4.3: GMComponentInterface.java

```
public interface GMComponentInterface {
  public void init(GMContainerInterface container);
  public boolean stop();
}
```

4.2 Exploit Vulnerabilities

The rest of this section presents the vulnerabilities of the dynamic applications framework that have been identified, and how they can be exploited. Chapter 5 then shows how these vulnerabilities can be fixed or, at least mitigated.

I would like to remark that, in the following, the GEB application means the Android program designed to run on the Android board of the Green e-Box. The GMContainer component is exactly contained in the GEB application.

1. It is possible for a Dynamic Application to shutdown the GEB application.

Because all Dynamic Applications run in the same Linux process space as the GEB application, thus it would be quite dangerous if one Dynamic Application stops the process. In such case, not only all the other Dynamic Applications stop working, but also the interaction between the User Application and Green eBox will be stopped, hence it is impossible for end users to unlock or lock the vehicle; moreover, the connection between Green eBox and Green Move Center will be cut too, so there is no way to monitor the status of the vehicle anymore. In fact, to stop the mentioned Linux process, one only needs to insert the instructions like following in the init method of a Dynamic Application.

```
System.exit(1);
```

2. A Dynamic Application is allowed to download and execute malicious code.

Because the GEB application needs to access Internet, so it declares its permission for Internet access in the AndroidManifest.xml manifest file, like

```
<uses-permission android:name="android.permission.INTERNET"/>
```

And due to the Android security model, such permission is granted based on the application granularity; to be more precise, it is based on the Linux process granularity, that is to say that the whole process would have the permission to access Internet. And it is exactly because of this, one could design a program that is harmful to the GEB application, such as writing out a lot of useless data only to take up the memory space, stealing sensitive data from the system and so on. Such a program could be packaged in a jar file, and downloaded by a Dynamic Application. Once the Dynamic Application gets this program, it could use the DexClassLoader to load the class (suppose the program is written in Java), and starts to run the malicious program inside.

The following piece of information is adapted from the result of executing DownloadMaliousCode.java

```
D/DownloadMaliciousCode(32127): Test Download malicious code and execute
    initialised!
D/DownloadMaliciousCode(32127): Begin to download malicious code
D/DownloadMaliciousCode(32127): Malicious code downloaded successfully!
D/DownloadMaliciousCode(32127): Begin to execute the malicious code
D/DownloadMaliciousCode(32127): Malicious code executed successfully! with
    result: HELLO WORLD FROM MALICIOUS CODE, I AM A BAD GUY
```

3. Dynamic Applications could access security-sensitive operations in the GEB application which are not supposed to be accessed, for example, it is possible for a Dynamic Application to access the unloadClass method of GMContainer class.

In the GMContainer class, there are several security-sensitive operations defined to manage the Dynamic Applications, and thus should be hidden away from Dynamic Applications. These security-sensitive operations include:

```
public String[] listClasses(); \\list all loaded dynamic applications.
public void unloadClass(String class_name); \\unload a Dynamic Application
```

However, as presented in the above piece of code, when the GEB application starts to load and run a Dynamic Application, the reference of GMContainer instance is passed to the Dynamic Application, then, although the starting point of a Dynamic Application is defined as:

```
public void init(GMContainerInterface container);
```

which means Dynamic Applications are designed to access only the APIs exposed in GMContainerInterface, still a Dynamic Application programmer could make a cast in a Dynamic Application like,

```
GMContainer gmContainer = (GMContainer) container;
```

thereafter nothing of GMContainer is hidden away from the programmer anymore.

4. There is a way for a Dynamic Application to stop services that are key components of the GEB application.

Due to the fact that GMContainer is a subclass of android Service class which again is the subclass of android Context class, it is possible to use the same trick presented in previous example to get the reference of the Context instance, and then use it to stop the services of Green Move Container:

```
Context context = (Context) container;
Log.d(TAG, "Try to Stop Trex Service! context.stopService(trexService)");
Intent trexService = new Intent(context, TRexService.class);
context.stopService(trexService);
Log.d(TAG, "Stop Trex Service Succefully!");
```

The above piece of code could stop the key service, TrexService, which is used to communicate with the T-Rex Server residing in the Green Move Center.

5. Dynamic Applications are allowed to change the data in DataModel which are supposed to be changed only by the sensors.

Logically, the security-sensitive data represented in the DataModel class should be changed only by the management part of the GEB application, and all Dynamic Applications could only read such data. In this way, it is guaranteed that all the modifications are made by the low layer sensors but not via an untrusted Dynamic Application.

However, as shown in the DataModel class diagram, once a Dynamic Application gets the reference of the only DataModel instance, which is shared among all Dynamic

Applications, by calling operations like,

```
public synchronized void setTemperature(float temperature);
public synchronized void setBatteryStatus(int batterystatus);
```

it is possible to fool all the other Dynamic Applications in the system.

CHAPTER 5
Solutions to Avoid the Vulnerabilities

The existence of the first vulnerability described in the previous chapter is simply due to the reason that both Dynamic Applications and the the other parts of the GEB application all run in the same process space. Therefore, to avoid it, no Dynamic applications should be allowed to execute instructions like System.exit, or each of them should run in its own process space.

In a Java Virtual Machine, it is possible to prevent an application from running security-sensitive operations like System.exit by putting a Java sandbox around it. While to make an application run in another process, the Linux system call *fork* could come to help, unfortunately, it is discouraged by the Android world, hence it is not considered here.

The second vulnerability metioned in chapter 4 is because Dynamic Applications are given too many permissions; again this could be avoided by a sandbox. In fact, the Java platform allows application designers to grant permissions to some specified hosts and ports for the Internet Access.

As to the third vulnerability, it is quite important to expose Dynamic applications only the needed APIs. One also has to pay attention to the effects of polymorphism, which could allow the methods of an object to be exposed unexpectedly.

The fourth one is similar to the third one, and it could be solved in the same way.

Lastly, for the fifth, the DataModel object should be split into two or more parts, and provides Dynamic Applications the read-only parts. In another way, a Dynamic Application could only have a copy of the DataModel object, thus it is allowed to change the data itself but not to affect the other Dynamic Applications in the system.

The following sections first explores the Java sandbox way to prevent a Dynamic Application from stopping the whole application process, then explain why it is not applied in the project, after that, the Android application sandbox and Android IPC mechanisms are introduced, and then the new architecture of the GEB application, based on application sandbox and Binder IPC, is presented.

5.1 Java Sandbox

The Java sandbox strongly relies on the Java Security Model, therefore a brief introduction to Java 2 Security Model is presented here.

5.1.1 Java 2 Security Model

The Java 2 security model is based on controlling the operations that a class can perform when it is loaded into JVM. The Bytecode Verifier, Class Loader and Security Manager are three key components of the security model [9].

Bytecode Verifier

Before executing in a JVM, a Java source program must be compiled into platform-independent Java byte code. The bytecode verifier is then used to check such byte code; it is meant to prevent the byte code from violating specified safety rules. The byte code can be from anywhere, it is unknown whether it is compiled by trusty compiler or not, so practically the Java runtime simply subjects it to series of tests by bytecode verifier.

Mainly the bytecode verifier is used to ensure that:

1. the form of compiled code is correct
2. internal stacks will not overflow or underflow
3. no illegal conversions will happen
4. the types of parameters are correct when there is method call

5. all class member accesses do not violate the scope (private, protected, public etc.)

Class Loader

All Java objects belong to classes; the class loader is used to load classes, be it local or remote, into the running environment. Classes loaded by different class loaders have different name spaces, thus name spaces could be used to separate classes into groups, and by placing special rules on the group, class loaders are enabled to prevent sensitive system resources from being accessed by untrusted code [10].

Security Manager

Security manager is a key component of the Java security model. If it is enabled by a Java application, the security manager is used to check whether a security-sensitive operation should be accepted or not based on the permissions.

Figure 5.1 illustrates the positions where these three components are placed in the Java 2 platform.

5.1.2 Java Sandbox Based on Java 2 Security Model

The Java 2 Platform is designed with the sandbox model inside it to run remote untrusted code in a safe way. It uses the three components mentioned before: bytecode verifier, class loader and security manager. Together with the help from elements like permissions, protection domains and security policies, it is possible to build a Java sandbox for a program to run inside.

Permissions

A permission defines the set of operations that could be performed on a collection of resources. A Java class is associated with a special set permissions based on some policies, when it is loaded into runtime environment.

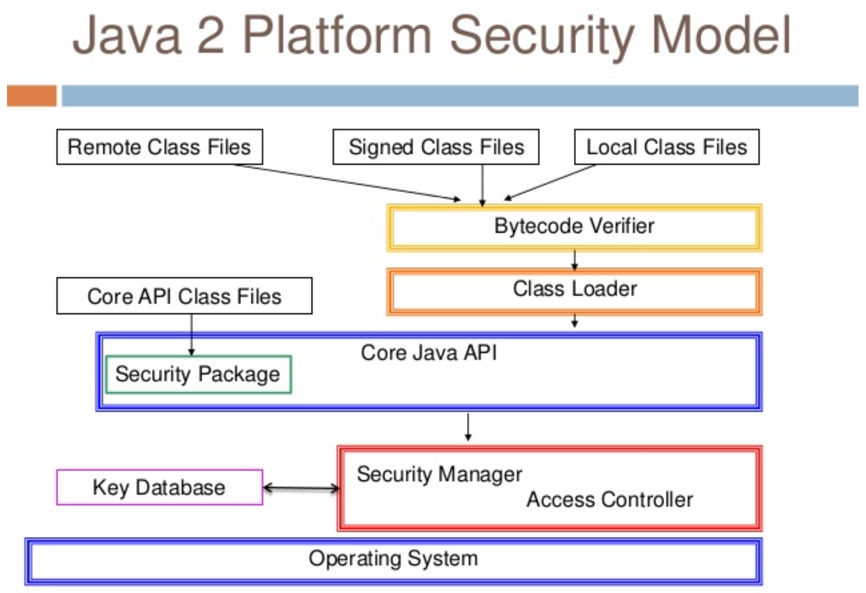


Figure 5.1: Java 2 Platform Security Model

In Java 2 platform, there are many kinds of permissions that are used to secure the system, such as: java.io.FilePermission is used to declare the access to the filesystem, the permission of RuntimePermission(”exitVM”) type would be checked to decide if the program has the privilege to execute System.exit(code) instruction, and the SocketPermission is put there to control Internet access.

Protection Domains and Security Policies

A protection domain is a grouping of a code source and permissions; that is, a protection domain represents all the permissions that are granted to a particular code source.

A security policy is the facility to specify which permissions should apply to which code sources, it is encapsulated by the Policy class *java.security.Policy*.

For example the following policy object defined in a policy file creates a protection domain for classes residing in http://it.polimi.gmc/foo.jar with the read permission for files in directory ${user.dir} and its subdirectories.

```
grant CodeBase http://it.polimi.gmc/foo.jar {
  permission java.io.FilePermission ${user.dir}/-,  read;
};
```

Figure 5.2 illustrates the Java 2 sandbox model in the Java 2 platform.

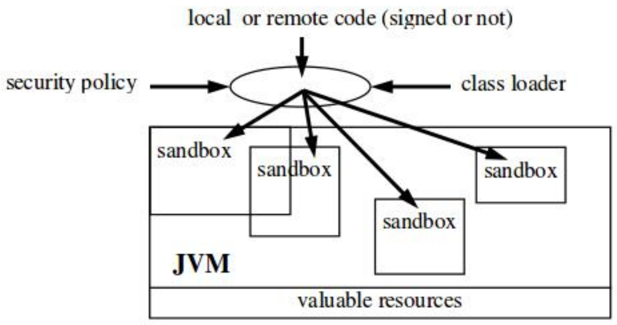


Figure 5.2: Java 2 Sandbox

5.1.3 Using Java Sandbox to Avoid System.exit

Before showing the solution for preventing the System.exit call, let us take a brief look at what happens when the System.exit gets called.

Listing 5.1: System.java

```java
public final class System {
 \\ ...
  public static void exit(int status){
    Runtime.getRuntime().exit(status);
  }
 \\ ...
}
```

Listing 5.2: Runtime.java

```java
public class Runtime {
  \\ ...
  public void exit(int tatus) {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
      security.checkExit(status)
    }
    Shutdown.exit(status);
  }
  \\...
}
```

Listing 5.3: SecurityManager.java

```
public class SecurityManager {
    \\ ...
    public void checkExit(int status) {
      checkPermission(new RuntimePermission("exitVM."+status));
    }
    public voi checkPermission(Permission perm) {
      java.security.AccessController.checkPermission(perm);
    }
    \\ ...
}
```

As shown in the above pieces of code, all that System.exit does is to call the exit(int) method of the Runtime class. Then the checkExit(int) method of the SecurityManager is called to create a RuntimePermission(exitVM) permission, and passed it to the checkPermission(perm) method. Finally, the AccessController is delegated to check if the calling class has been granted the RuntimePermission(exitVM) permission; if the permission has been granted, the checkPermission method of the AccessController will return silently, otherwise it will throw out a SecurityException. Thus, the RuntimePermission(exitVM) permission should not be granted to classes that are forbidden to call System.exit(int). However, such a permission is granted to all code loaded from the application class path by default, this is described in the Javadoc of RuntimePermission:

This allows an attacker to mount a denial-of-service attack by automatically forcing the virtual machine to halt. Note: The "exitVM." permission is automatically* *granted to all code loaded from the application class path, thus enabling applications to terminate themselves. Also, the "exitVM" permission is equivalent to "exitVM.*".*

Therefore, a new customized SecurityManager should be brought into the runtime environment to avoid System.exit(int) call. The following NoExitSecurityManager class gives an example of such SecurityManager.

Listing 5.4: NoExitSecurityManager.java

```
public class NoExitSecurityManager extends SecurityManager {
  @Override
  public void checkPermission(Permission perm) {
    if(perm.getName().contains("exitVM")) {
        throw new SecurityException("exit vm is not allowed");
    }
    super.checkPermission(perm);
  }
}
```

5.1.4 Why Java Sandbox is not Suitable for an Android Application

As mentioned in the previous section, by customizing a Security Manager like the NoExitSecurityManager, it is possible to prevent the GEB application halting problem. In the same way, by building a Java sandbox with granting SocketPermission to a certain trusted hosts, it is possible avoid the problem of downloading and executing malicious code inside the GEB application.

However, the Android Dalvik virtual machine is different from standard Java virtual machine, and the implementations of core Java libraries are modified a little bit for Android applications, although the interfaces are almost the same. Thus, the above methods may work quite well in a normal Java application, but it is not suitable for an Android application. It is impossible to build a Java sandbox using elements like SecurityManager, Permissions mentioned before, without modifying the core java libraries of Android Dalvik Virtual Machine.

For example, the SecurityManager is described in the following way in the Android documentation:

Legacy security code; do not use. Security managers do not provide a secure environment for executing untrusted code. Untrusted code cannot be safely isolated within the Dalvik VM.

And the SocketPermission is described as:

Legacy security code; do not use.

And if one checks the source code of core Java libraries in Android platfrom, it could be found that the Java2 security model is not implemnted there, as a consequence, a reimplementation of parts of the core Java libraries which are related to Java sandbox must be considered. It is not a simple task, not to mention the performance issues. Therefore, a new model for the sandbox is required for the GEB application.

5.2 Android Application Sandbox

5.2.1 Android Security Model

Android is a Linux kernel based platform with Dalvik virtual machine embedded inside to support the Java programming language, and has its own security mechanisms tuned especially for mobile applications. Android combines Linux OS features like multi-users, multi-processes, and file permissions for its new security model – application sandbox [11].

Unlike the desktop environments Linux operating systems, where all applications of a user run as the same User Identifier (UID), the Android applications are installed and run as different UIDs, and individually isolated from each other. Each Android application could run in a separate process space with distinct permissions. By default, applications can not interact with each other and they have limited accesses to the underlying resources.

Figure 5.3 illustrates the Application Sandbox Model for Android platform.

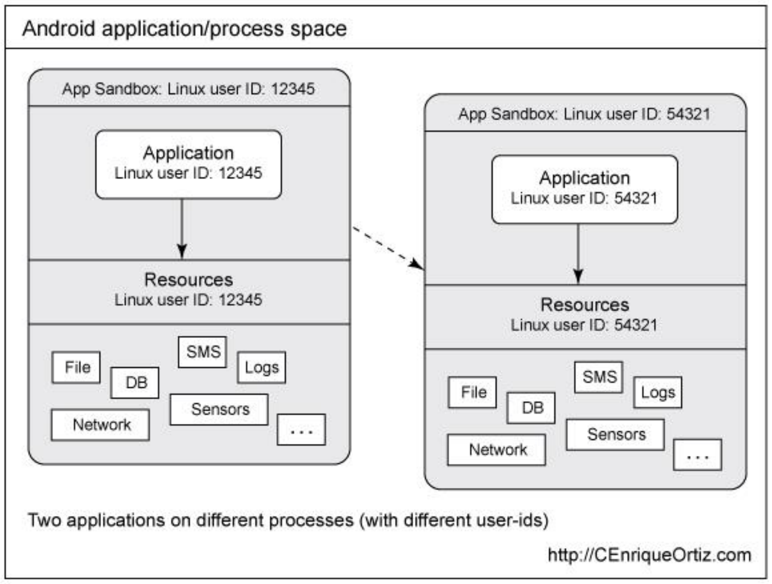


Figure 5.3: Android Application Sandbox

With the Android platform, developers can not only build applications to use the resources in an efficient way, but also protect the platform by minimizing the consequences of bugs and malicious software. The process isolation obviates the need for a complicated policy configuration file for the application sandbox. The AndroidManifest.xml file of an Android application gives application developers opportunities to grant specified permissions to an application. Such permissions are used to check whether an application has the rights to use the Internet to access the outside world, use the Bluetooth to connect to another device, use the GPS to indicate the current location, use the camera to take a photo, and so on.

All permissions of an Android application must be declared before it is installed, then a user could decide whether to accept the installation or no by the required permissions. Application developers are not allowed to grant permissions dynamically, all the granted permissions must be declared in the AndroidManifest.xml file statically.

Here is the configuration used in the GEB application, which allows the application to use Bluetooth Channel, access the Internet, read the phone state and so on.

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

As mentioned in the Background chapter, a typical Android application usually contains components like: Activities, Services, Content Providers and Broadcast Receivers, most of the time these components are designed to share the same Linux process space.

Despite applications are designed to run in a separate Linux process space by default, still there is a way to configure two or more applications to run in the same process space on Android platform, as long as they are declared to use the same process name and signed by the same user. Furthermore, Android allows Activities, Services, Content Providers and Broadcast Receivers to run in different processes even though they are from the same application.

Since different components of an application could be configured to run in different processes, there is a need for mechanisms that allow them to communicate with each other. Android gets these done in its inter-process communication (IPC) part indeed.

5.2.2 Android IPC

Inter-process communication (IPC) is a mechanism for the exchange of signals and data across multiple process. It could be used to enable computational speedup, information sharing, data isolation. Linux provides a variety of mechanisms for IPC:

Signals

A process can send signals to processes with the same UID and GID or in the same process group. It is an old IPC mechanism.

Pipes

Pipes are unidirectional bytestreams that connect the standard output from one process to the standard input of another process.

Sockets

Sockets could be used for bidirectional communications. Two processes could communicate with bytestreams by opening a socket.

Message Queues

Message queues are used for asynchronous communications. A process can send a message to a message queue for another process to consume.

Semaphores

A semaphore is shared variable that can be read and written by many processes.

Shared Memory

A location in system memory mapped into virtual address spaces of two or more processes, each process is free to access the shared memory.

All the above IPC mechanisms are commonly used in the desktop or normal server environments applications. However, Android is designed specially as a platform for mobile applications, where applications and system services run in separate processes. The number of inter-process communcations here is much greater than that of a desktop or a normal server platform, therefore, the Binder IPC mechanism is brought into Android to avoid the overhead of the above traditional IPC mechanisms.

The Binder IPC mechanism relies heavily on the Binder driver, which is implemented in the Linux kernel space. The Binder driver can be used by the standard Linux system call ioctl() to offer a efficient way for communications between processes, including exchange data between processes, remotely call a method on another process. Figure 5.4 shows how binder IPC is used for two processes.

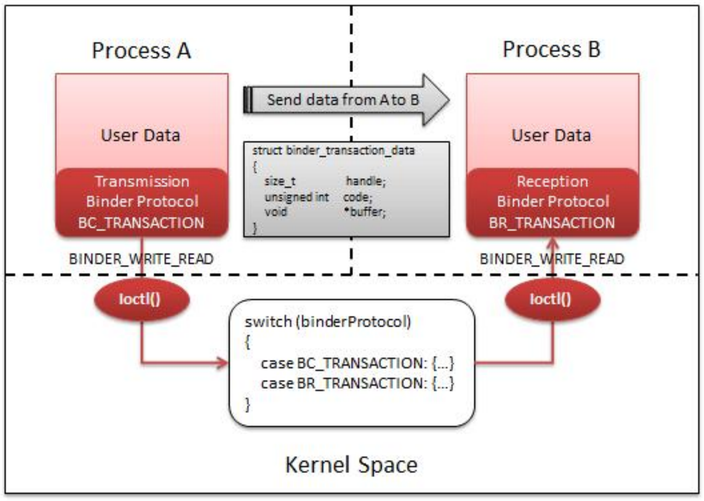

Process A
Process B
Send data from A to B
User Data
Transmission Binder Protocol BC_TRANSACTION
User Data
Reception Binder Protocol BR_TRANSACTION
struct binder_transaction_data { size_t handle; unsigned int code; void *buffer; }
BINDER_WRITE_READ
BINDER_WRITE_READ
Ioctl()
Ioctl()
switch (binderProtocol) { case BC_TRANSACTION: {...} case BR_TRANSACTION: {...} }
Kernel Space

Figure 5.4: Android Binder IPC

The Binder has the following main features:

1. With its built-in reference-counting of object references and death-notification mechanism, the Binder is quite suitable for "hostile" environments. For example, if an application process goes away, all of its windows should be removed. This is made easy by the binder's death-notification facility, which allows a process to get a callback when another process hosting a binder object goes away.

2. When a binder service is no longer referenced by any client, its owner is automatically notified to dispose of it.

3. Methods on remote objects can be invoked as if they were local – the thread appears to jump to the other process.
4. The interface of a binder object can be auto-generated by the simple Android Interface Definition Language (AIDL).
5. Building support for un/marshalling common data types.
6. Simplified invocation model via atuo-generated proxies and stubs

To summarize, because Green Move Dynamic Applications could be considered as untrusted codes, therefore, each Dynamic Application should be required to run inside a sandbox. In a standard Java virtual machine environment, it is quite common to build a sandbox using elements like SecurityManager, Permissions and so on, however, in an Android platform one needs re-implement parts of the core Java libraries in order to have a similar sandbox. Consequently, the Android application sandbox model which takes advantage of the Linux user-based protection, together with the Binder IPC, are brought into the GEB application to avoid the vulnerabilities mentioned before.

5.3 Application Sandbox for Dynamic Applicaitons

This section simply describes how to use the application sandbox model and Android permission mechanisms to customize a sandbox model for Dynamic Applications, and how the found vulnerabilities can be avoided in this model. Then the following section is given to explain all these in more detail.

In the application sandbox model, each Dynamic Application is designed to run in a separate sandboxed process. Initially, the sandboxed Dynamic Application, by default, is granted only the permission to read and write its own internal storage. Therefore, it is impossible for a Dynamic Application to download malicious code directly from the Internet (this requires android.permission.INTERNET Permission).

Also, the GEB application and the Dynamic Applications do not run in the same process space anymore. The GEB application is designed to provide all the needed remote services to a Dynamic Application, like the HTTP Channel and T-Rex Channel. All these remote services are implemented through the AIDL mechanism provided in Android platform, and the Binder IPC mechanism is used by Dynamic Applications to make a remote procedure call to the remote services.

Since a Dynamic Application runs in a seperate process, therefore, in case it maliciously executes instructions like System.exit(int), it could only stop its own process, but not the GEB application process or the other sandboxed Dynamic Application processes. And due to fact that the GEB application and a Dynamic Application are separated in different processes, a Dynamic Application now is impossible to obtain the reference of the GMContainer, thus it is impossible to access the security-sensitive operations defined in GMContainer anymore. Similarly, because Dynamic Applications could only contact the GEB application by remote calls (they are not given the reference of the GEB application's Context instance anymore), there is no way for them to stop the key services, like the mentioned TRexService, in the GEB application.

Lastly, a Dynamic Application could only remotely ask for a copy of the original DataModel object. The original DataModel object is kept in the Dynamic Application to receive the modifications of the data from the sensors and the vehicle; is never referenced by any Dynamic Application directly. Hence, even a Dynamic Application tries to modify the received copy of the DataModel object, neither the other Dynamic Applications, nor the GEB application would be affected.

5.4 New Architecture of the GEB Application

This section presents a new architecture that is designed to avoid the vulnerabilities. I would like to point out here that, althogh the new architecture has been designed, it has not been fully implemented yet. The implementation of the its Http

Channel Service in the following subsection is given as an example to illustrate the design in more detail.

Obviously, we need a sandbox for each Dynamic Application, however, due to the reason mentioned before, the Java sandbox model is not suitable to be applied here. Therefore the architecture relies on the Android application sandbox to guarantee each Dynamic Application runs in a non-harmful way. In the mean while, since each sandboxed Dynamic Application runs in a separate process, the Binder IPC discussed above would be used for communications between the core part of the GEB application and Dynamic Applications.

The original GEB application is now divided into two applications, the Server side GEB application, and the Client side GEB application, it is depicted as Figure 5.5.

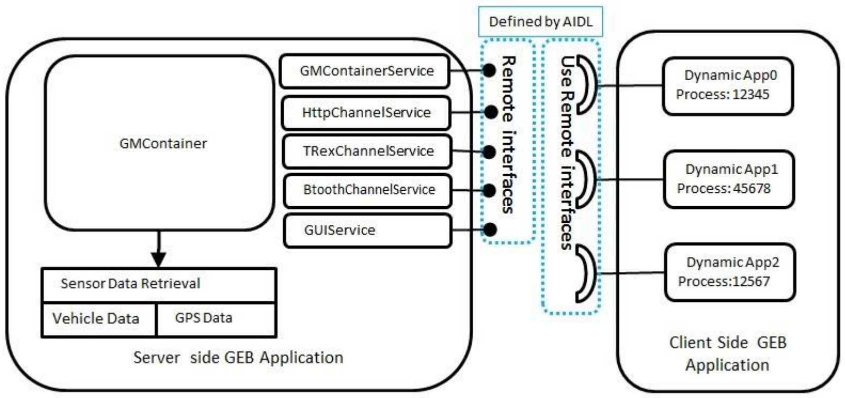


Figure 5.5: New Architecture for Green e-Box Application

Besides the basic functionalities in the original GEB application, the Server side GEB application keeps also the following remote services in addition:

GMContainerService

A Dynamic Application could call this service remotely to retrieve the data repesented in DataModel, and also the basic Green e-Box information such as the serial number, the version number, etc.

GUIService

It is used to remotely show messages from a Dynamic Application on the GUI of the Server side GEB application.

HTTPChannelService

A Dynamic Application can not access Internet directly, it has to delegate its requests to this remote service.

TRexChannelServcie

Remotely handle TrexChannel requests for Dynamic Applicaitons.

BluetoothChannelService

Remotely handle BluetoothChannel requests for Dynamic Applicaitons

The Server side is given all the required permissions, like permission to access Internet, permission to Access the Bluetooth and so on.

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

While in the Client side, none of such permissions are granted, they have to delegate all the requests that require the mentioned permissions to the server side remote services. Furthermore, each Dynamic Application contained in the client side is configured to run in a separate process. The following piece of configuration is extracted from the client side configuration file.

```
<service
  android:name="it.polimi.dynapp.service.GMDynamicAppService0"
  android:enabled="true"
  android:exported="true"
  android:permission="it.polimi.greenmove.SANDBOXED_DYNAMIC_APP_PERM"
  android:process="sandboxed.dynamic.application0">
  <intent-filter>
      <action android:name="it.polimi.dynapp.MANAGEMENT" />
  </intent-filter>
</service>

<service
  android:name="it.polimi.dynapp.service.GMDynamicAppService1"
  android:enabled="true"
  android:exported="true"
  android:permission="it.polimi.greenmove.SANDBOXED_DYNAMIC_APP_PERM"
  android:process="sandboxed.dynamic.application1">
  <intent-filter>
      <action android:name="it.polimi.dynapp.MANAGEMENT" />
  </intent-filter>
</service>

<service
  android:name="it.polimi.dynapp.service.GMDynamicAppService2"
  android:enabled="true"
  android:exported="true"
  android:permission="it.polimi.greenmove.SANDBOXED_DYNAMIC_APP_PERM"
  android:process="sandboxed.dynamic.application2">
  <intent-filter>
      <action android:name="it.polimi.dynapp.MANAGEMENT" />
  </intent-filter>
</service>
```

Since the sandboxed Dynamic Applcation is only granted the default permission to access the Android internal storage, if it needs more services from the server side, the Binder IPC mechanism would be applied for them. Figure 5.6 presents the delegation model of permissions.

Finally, based on the new architecture, the work-flow to download and start a Dynamic Application is like Figure 5.7:

The following section illustrates how the HttpChannelService is implemented, and how the sandboxed Dynamic Application interacts with the HttpChannelService.

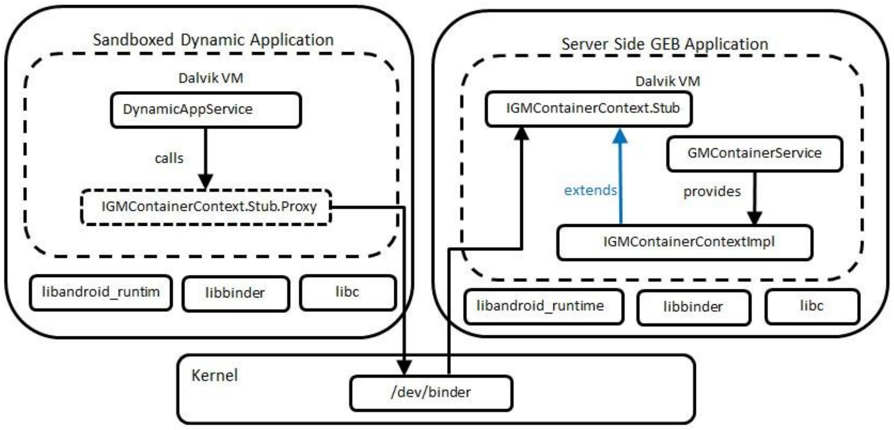


Figure 5.6: Delegation of Permissions

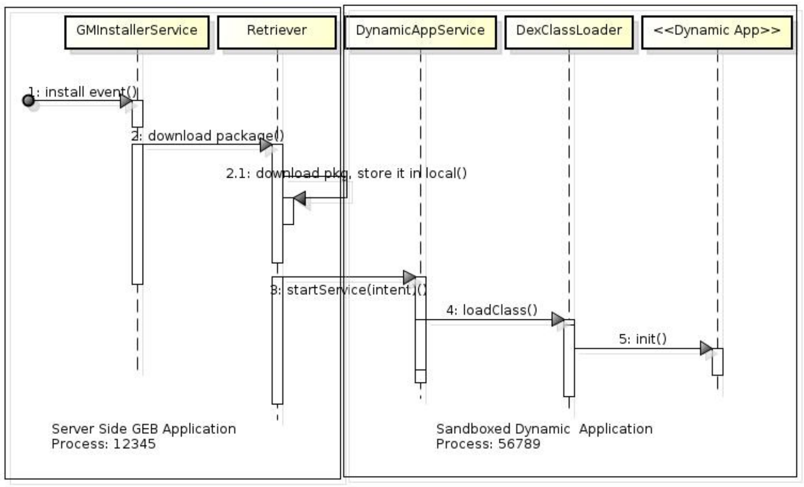


Figure 5.7: New Sequence to Download and Start A Dynamic Application

5.4.1 HTTP Channel Using the Applicaton Sandbox

A Dynamic Applciation could only delegate the http request to the the HttpChannelService residing in the server side GEB application. Furthermore, it is quite important to make sure that they can not access all the urls as they desire to, and this is implemented in the HttpSecurityManager with a white list of urls inside. The Proxy design pattern is applied here to make the HttpSecurityManager to add new policies for controlling the accesses more easily, see Figure 5.8 and Figure 5.9.

The HttpChannelProxy is then used by the remote HttpChannelService to implement the interface exposed by HttpChannelApi, which is auto-generated by Android AIDL, the interface is then exposed to a remote Dynamic Application by the Stub (auto-generated by AIDL) class and the Binder mechanism.

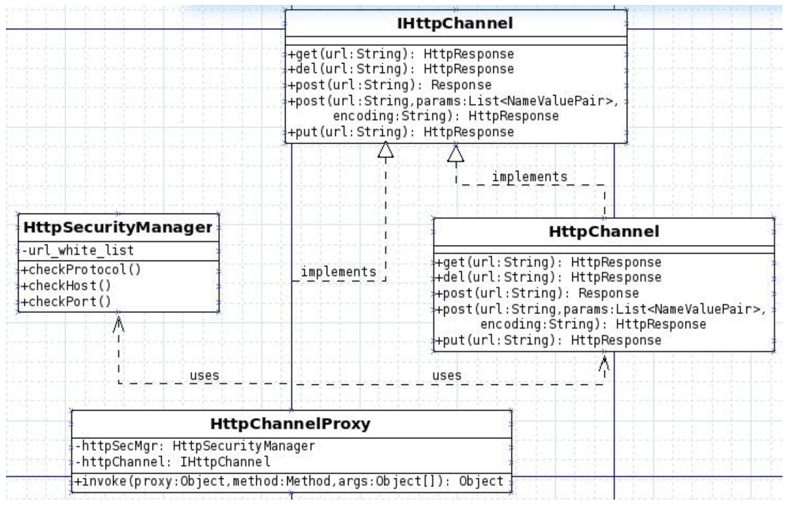

IHttpChannel
+get(url:String): HttpResponse
+del(url:String): HttpResponse
+post(url:String): Response
+post(url:String,params:List<NameValuePair>, encoding:String): HttpResponse
+put(url:String): HttpResponse
implements
HttpSecurityManager
-url_white_list
+checkProtocol()
+checkHost()
+checkPort()
implements
HttpChannel
+get(url:String): HttpResponse
+del(url:String): HttpResponse
+post(url:String): Response
+post(url:String,params:List<NameValuePair>, encoding:String): HttpResponse
+put(url:String): HttpResponse
uses
uses
HttpChannelProxy
-httpSecMgr: HttpSecurityManager
-httpChannel: IHttpChannel
+invoke(proxy:Object,method:Method,args:Object[]): Object

Figure 5.8: Fine Grained Http Channel Control (1)

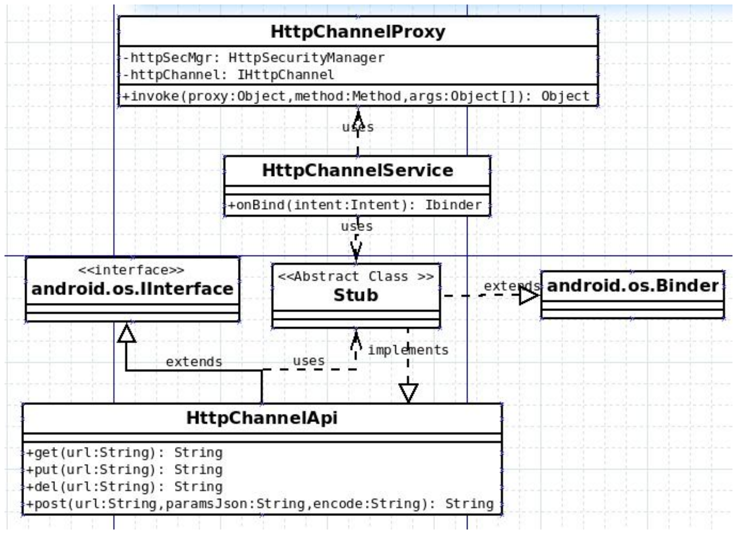


Figure 5.9: Fine Grained Http Channel Control (2)

The Delegation Model of the HttpChannelService could be depicted as Figure 5.10.

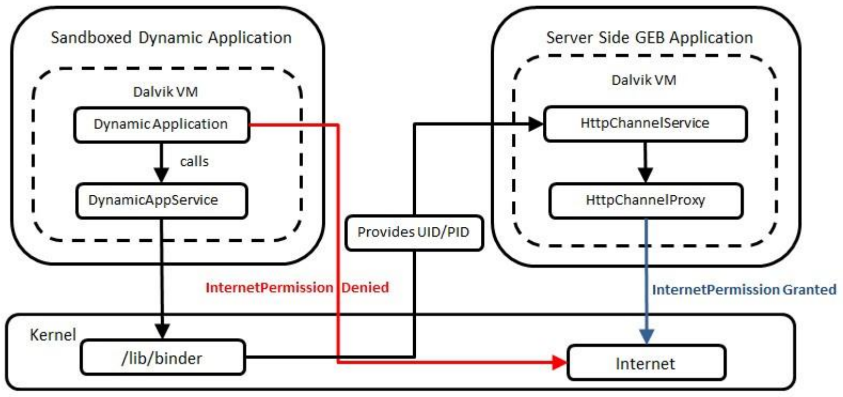


Figure 5.10: Http Channel Permission Delegation Model

5.4.2 Experiments on HTTP Channel

This section presents two Dynamic Applications to test the security aspects of the Internet Access.

The first Dynamic Application tries to access the Internet directly in the sandbox, and we can see from the result, it is denied because of the lack of permission.

Listing 5.5: Source Code of the Dynamic Application that Accesses Internet Directly

```
public class HttpChannelServiceTest implements GMComponentInterface {
    private static final String TAG = "HttpChannelServiceTest";
    private GMContainerInterface context;
    @Override
    public void init(GMContainerInterface context) {
        this.context = context;
        Log.d(TAG, "Test Case -- HttpChannelServiceTest");

        try {
            this.testHttpDirectAccess("http://www.google.com");

        } catch (Exception e) {
            Log.e(TAG, e.getMessage());
        }
    }

    @Override
    public boolean stop() {
        return false;
    }

    private void testHttpDirectAccess(String strUrl) throws IOException {
        Log.d(TAG, "Test Case -- testHttpDirectAccess");

        HttpURLConnection conn = null;

        try {
            URL url = new URL(strUrl);
            conn = (HttpURLConnection) url.openConnection();
            conn.setDoInput(true);

            Log.d(TAG, "Test Case -- testHttpDirectAccess result: " + conn.
                getResponseMessage());
        }catch (IOException e) {
            Log.e(TAG, e.getMessage());
            throw e;
        }

    }
}
```

Here is the result from the adb console for the above Dynamic Application:

Listing 5.6: Result for the Direct Internet Access: Permission Denied

```
D/GMDynamicAppService(19472): GMDynamicAppService started
D/GMDynamicAppService(19472): receive jar url: /data/data/com.polimi.greenMove
    .moveGroup/files/oo.jar
D/GMDynamicAppService(19472): receive clsName: it.polimi.greenmove.security.
    HttpChannelServiceTest
D/DynamicAppStarter(19472): Dynamic Application loaded
D/HttpChannelServiceTest(19472): Test Case -- HttpChannelServiceTest
D/HttpChannelServiceTest(19472): Test Case -- testHttpDirectAccess
E/HttpChannelServiceTest(19472): Permission denied (missing INTERNET
    permission?)
```

The second Dynamic Application tries to access the Internet via the HttpChannelService metioned before, the key part of the code is listed as following:

Listing 5.7: Dynamic Application Delegates Internet Access to HttpChannelService

```
private void testHttpRemoteAccess(String strUrl) {
  Log.d(TAG, "Test Case -- testHttpRemoteAccess");
  IHttpChannel channel = this.context.getHttpChannel();
  if (channel != null) {
    String res = channel.get(strUrl).getResponseMessage();
    Log.d(TAG, "HttpChannelServiceTest#testHttpRemoteAccess--response code: "
        + res);
  } else {
    Log.d(TAG, "HttpChannelServiceTest#testHttpRemoteAccess--null channel ");
  }
}
```

The HttpChannelService responses to the Dynamic Application, and sends it back the result of the Internet access successfully:

Listing 5.8: HttpChannelService Sends Back the Result of Internet Access Successfully

```
D/GMDynamicAppService(19321): GMDynamicAppService started
D/GMDynamicAppService(19321): receive jar url:
  /data/data/com.polimi.greenMove.moveGroup/files/oo.jar
D/GMDynamicAppService(19321): receive clsName:
  it.polimi.greenmove.security.HttpChannelServiceTest
D/DynamicAppStarter(19321): Dynamic Application loaded
D/HttpChannelServiceTest(19321): Test Case -- HttpChannelServiceTest
D/HttpChannelServiceTest(19321): Test Case -- testHttpRemoteAccess
W/ContextImpl(19321): Implicit intents with startService are not safe:
  Intent { act=it.polimi.service.HttpChannelService } android.content.
      ContextWrapper.bindService:517
  it.polimi.gebapp.GEBAppContextWrapper.getHttpChannl:66
  it.polimi.greenmove.GMAppContext.getHttpChannel:44
I/GEBAppContext(19321): HttpChannel Service connection established
D/HttpChannel(19184): HttpChannel Constructor
D/dalvikvm(19184): GC_FOR_ALLOC freed 8780K, 26% free 26141K/34960K,
  paused 11ms, total 12ms
D/dalvikvm(19321): GC_FOR_ALLOC freed 225K, 2% free 17021K/17292K,
  paused 10ms, total 10ms
D/dalvikvm(19321): GC_FOR_ALLOC freed 213K, 3% free 17126K/17548K,
  paused 8ms, total 8ms
D/HttpChannelServiceTest(19321): HttpChannelServiceTest#testHttpRemoteAccess
  -- response code: OK
D/DynamicAppStarter(19321): Dynmic Application started
```

CHAPTER 6
Conclusions

This work has laid the foundations to make the GM system more secure, by identifying some key vulnerabilities and suggesting countermeasures. However, some aspects have not been tackled, and could be the object of future works on this topic:

1. An automatic analysis tool for malicious code used to check the Dynamic Application programs before they are uploaded to the Green Move Center [12].

2. Encrypt the output data of each Dynamic Application.

3. Build a mechanism to manage the running Dynamic Applications, for example to stop a Dynamic Application once it is discovered to behave maliciously.

4. Secure the T-Rex Channel, right now the T-Rex Channel is based on the plain sockets, it is possible for one to use the Man-in-the-middle attack. The commands sent in the T-Rex channel are quite security-sensitive, like one could send a command to unload Dynamic Applications for all the Green e-Boxes in the system.

5. Common security issues with a Android application, like the Intent-Hijack.

REFERENCES

[1] S. Shaheen, D. Sperling, and C. Wagner, "Carsharing in Europe and North America:Past, Present, and Future," *Transportation Quarterly*, vol. 52, no. 3, pp. 35–52, 1998.

[2] A. G. Bianchessi, G. Cugola, S. Formentin, A. Morzenti, C. Ongini, E. Panigati, M. Rossi, S. M. Savaresi, L. T. Fabio A. Schreiber, and E. G. V. Depoli, "Green Move: a platform for highly configurable, heterogeneous electric vehicle sharing."

[3] M. Gargenta, *Learning Android.* OReilly Media, Inc., 2011.

[4] "Application Fundamentals," http://developer.android.com/guide/components/fundamentals.html, [Online; accessed March 12, 2014].

[5] G. Cugola and A. Margara, "Complex event processing with T-REX," *Journals of Systems and Software*, vol. 85, no. 8, pp. 1709–1728, 2012.

[6] C. O. Nutter, N. Sieger, and T. Enebo, *Using JRuby: Bringing Ruby to Java.* Pragmatic Programmers LLC, 2011.

[7] A. G. Bianchessi, C. Ongini, S. Rotond, M. Tanelli, M. Rossi, G. Cugola, and S. M. Savaresi, "A Flexible Architecture for Managing Vehicle Sharing Systems," *IEEE Embedded Systems Letters*, vol. 5, no. 3, pp. 30–33, 2013.

[8] G. Alli, L. Baresi, A. Bianchessi, G. Cugola, A. Margara, A. Morzenti, C. Ongini, E. Panigati, M. Rossi, S. Rotondi, S. Savaresi, F. A. Schreiber, A. Sivieri, L. Tanca, and E. V. Depoli, "Green Move: towards next generation sustainable

smartphone-based vehicle sharing," *Sustainable Internet and ICT for Sustainability (SustainIT), 2012*, pp. 1–5, 2012.

[9] L. Gong, *Inside Java 2 Platform Security (2nd Edition).* Addison-Wesley, 2003.

[10] G. McGraw and E. Felten, *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley & Sons, 1997.

[11] "Security and Permissions, Android Developers," http://developer.android.com/guide/topics/security/security.html, [Online; accessed March 12, 2014].

[12] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox System for Suspicious Software Detection," *5th International Conference on Malicious and Unwanted Softwar*, 2010.