Functional and Security Testing of a Mobile Client-Server Application

Daniel Holmberg
Victor Nyberg

Supervisor: Simin Nadjm-Tehrani
Examiner: Marcus Bendtsen
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstance. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Daniel Holmberg
Victor Nyberg
Students in the 5 year Information Technology program complete a semester long software development project during their sixth semester (third year). The project is completed in mid-sized groups, and the students implement a mobile application intended to be used in a multi-actor setting, currently a search and rescue scenario. In parallel they study several topics relevant to the technical and ethical considerations in the project. The project culminates by demonstrating a working product and a written report documenting the results of the practical development process including requirements elicitation. During the final stage of the semester, students form small groups and specialise in one topic, resulting in a bachelor thesis. The current report represents the results obtained during this specialization work. Hence, the thesis should be viewed as part of a larger body of work required to pass the semester, including the conditions and requirements for a bachelor thesis.
Abstract

Today’s massive usage of smartphones has put a high demand on all application developers in the matter of security. For us to be able to keep using all existing and new applications, a process that removes significant security vulnerabilities is essential. To remove these vulnerabilities, the applications have to be tested. In this thesis, we identify six methods for functional and security testing of client-server applications running Android and Python Flask. Regarding functional testing, we implement Espresso testing and RESTful API testing. In regards to the security testing of the system, we do not only implement fuzz testing, sniffing, reverse engineering and SQL injection testing on a system developed by a student group in a parallel project, but also discover a significant security vulnerability that directly affects the integrity and reliability of this system. Out of the six identified testing techniques, reverse engineering exposed the vulnerability. In conjunction with this, we verified that the system’s functionality works as it is supposed to.
Acknowledgements

We would like to thank our supervisor Simin Nadjm-Therani for all the help and support we have received during this thesis. We would also like to acknowledge all the support and valuable feedback received from our classmates.
Contents

Abstract iv
Acknowledgements v
Contents vi
List of Figures viii
List of Tables ix

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Aim ......................................................... 2
  1.3 Research questions ...................................... 2
  1.4 Delimitations ............................................ 2
  1.5 Structure ................................................. 2

2 Background 3
  2.1 The system ............................................... 3

3 Tools for software testing 5
  3.1 Functional testing ........................................ 5
  3.2 Fuzz testing ............................................ 6
  3.3 Sniffing ................................................... 6
  3.4 Reverse engineering ..................................... 6
  3.5 RESTful API testing .................................... 8
  3.6 SQL injection ............................................ 8
  3.7 Related works .......................................... 9

4 Applying these testing tools 10
  4.1 Functional testing ....................................... 10
  4.2 Fuzz testing ............................................ 12
  4.3 Sniffing .................................................. 13
  4.4 Reverse engineering .................................... 13
  4.5 RESTful API testing .................................... 15
  4.6 SQL injection ............................................ 17

5 Results 19
  5.1 Functional testing ....................................... 19
  5.2 Fuzz testing ............................................ 19
  5.3 Sniffing .................................................. 20
  5.4 Reverse engineering .................................... 22
  5.5 RESTful API testing .................................... 23
# List of Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>System topology</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>SQLIA flow</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Login activity</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>Map related activities</td>
<td>11</td>
</tr>
<tr>
<td>4.3</td>
<td>Contacts related activities</td>
<td>12</td>
</tr>
<tr>
<td>4.4</td>
<td>SQL injection</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>Trace of a log in</td>
<td>20</td>
</tr>
<tr>
<td>5.2</td>
<td>Trace of adding a case</td>
<td>21</td>
</tr>
<tr>
<td>5.3</td>
<td>Trace of a call between two clients</td>
<td>22</td>
</tr>
<tr>
<td>5.4</td>
<td>Relevant data stored in user table before manipulation</td>
<td>22</td>
</tr>
<tr>
<td>5.5</td>
<td>Relevant data stored in user table after manipulation</td>
<td>23</td>
</tr>
<tr>
<td>5.6</td>
<td>Unauthorized access</td>
<td>23</td>
</tr>
<tr>
<td>5.7</td>
<td>API testing results</td>
<td>24</td>
</tr>
<tr>
<td>5.8</td>
<td>App crashes</td>
<td>24</td>
</tr>
<tr>
<td>C.1</td>
<td>Source code of target application shown in the JD-GUI program</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Espresso testing results ....................................................... 19
5.2 Monkey testing results ...................................................... 19
Introduction

From 2014 to 2020, the number of smartphone users in the world is expected almost to double [1]. Android plays a big part in this since it contributes to approximately 85% of the global smartphone usage [2]. Whether you believe this is a good or a bad thing, a more connected world implies that more devices are exposed to various threats.

1.1 Motivation

The smartphone is one of the most common categories of all connected devices we use on a daily basis. We use them for a considerable amount of tasks, and the area of use is expanding rapidly. So is the application market. When handling data, we do not want anyone to manipulate or access our data without our consent. Since we are so dependant on the technology, we also want it to work seamlessly. These requirements put a demand on developers to test their applications for flaws that might lead to unwanted consequences.

Today, the word security, especially information security, concerning connected devices is a supreme topic about which many people are worried. We believe that this might depend on lack of knowledge by the public. With all this in mind and the fact that being an engineer does not just come with abilities, but responsibilities, engineers nowadays really need to consider the security aspect when developing new technology and software. Would you trust a new car although it has not yet gone through extensive tests?

In this thesis, we are going to test a system that has been developed in parallel with the project that aimed to implement a proof-of-concept system for the Swedish Emergency Service used in a search and rescue scenario. This system consists of an Android application that communicates with a Python Flask server. A detailed description of the system will be presented in chapter 2.

If an application needs to store any form of information, it is most likely using a database which runs on a server. The number of ways that this server can be set up in, is quite a few. In conjunction with different programming languages there also exists a wide range of frameworks that can be used to make the server suit one’s needs.
Flask is a micro-framework based on Python which was developed in 2010 [3]. With relatively new frameworks comes a great need for testing to make sure that they do not have any significant security flaws.

1.2 Aim

The aim of this thesis is to briefly acquaint the reader with functional testing and give a more profound and comprehensive understanding of a few security testing techniques in regards to Android applications. The thesis also aims to chart which of these techniques that are most suited for determining the confidentiality, integrity, availability and the reliability of the system that is going to be tested.

1.3 Research questions

Based on the purpose of this project, we in this thesis will:

1. Investigate which security vulnerability of the developed system that affect the confidentiality, integrity, availability and the reliability of the system.
2. Identify methods for client-server applications running Android and Python Flask to illuminate the system’s security vulnerabilities.
3. Apply the identified methods to the developed system to detect security vulnerabilities.

1.4 Delimitations

As we mentioned in the aim of this report, we will limit the scope of this thesis in regards to the number of testing methods. It is not possible to cover all existing testing techniques. We will specifically not treat denial of service attacks since this topic is widely covered in another thesis similar to ours in parallel with this project.

Due to our previous knowledge of the system under test, this thesis does not perform extensive SQL injection tests. Knowing that the system utilises libraries which mitigates SQL injection attacks, the expected outcome of such an attack would result in unnecessary time spent. However, since you cannot be sure that a library behaves as it should, some basic SQL injection testing has been performed.

Worth noticing is that every command used in this study were executed on the macOS and thus has to be changed to work in a Windows environment. Also, all tests performed targeting the front-end Android application were made during the development process, and thus not on a signed application.

1.5 Structure

The rest of the paper will be structured according to the following. First, chapter 2 will cover the background relevant to this thesis and chapter 3 will include the theory. After this chapter, the method is added. This is later followed by the results and discussion. Last but not least our conclusion regarding this thesis is presented.
As mentioned in chapter [1], we are conducting another project in a larger group in parallel with this one. This other group consists of seven people, and the project goal is to develop a proof-of-concept system for a user group such as the Swedish Emergency Service.

2.1 The system

The system consists of an Android application which communicates with a back-end server to:

- Store and retrieve data about different emergency cases.
- Setup a communication channel between various employees and also with other rescue authorities, such as the police and paramedics.

The purpose of the system is to be used both on a daily basis as well as in emergency situations. Another aspect to consider is that to access the system, every user is required to be authorised.

The application is developed for the Android OS using Android Studio, and the back-end server runs on Python and utilises the Flask microframework. The server also consists of a single MySQL database which is implemented with the help of SQLAlchemy, a Python specific database toolkit [4]. There exist two identical sets of servers and databases where the second set is just a replica which will be used if something happens to the first one, that is, a backup. The topology can be visualised according to the following figure.
To understand what will be tested and why, we will now present the functionality of the app. This is later in chapter 4 followed by the course of events that will have to take place to test the system.

One of the most important functionalities of the application is that it demands a valid username and corresponding password by the user to be able to log in. Every user has a certain role within the organisation and reciprocal app permission which allows the user to interact with different functions.

Another critical functionality of the app is providing a service where employees can call each other. Both via a standard voice connection and via video. The second primary activity for this app is the map functionality. Users can put out pins on a map which symbolise different cases. These cases could, for example, be fires or car crashes. Since different users can have separate authorisation (with the use of a stored role_id-attribute) when it comes to functionality, some might be able to add pins, and some might also be able to edit and delete them. A user’s location is also shown on the map in order to facilitate distribution of resources. Thus, a user’s latitude and longitude is stored on the back-end server as lon- and lat-attributes.

These permissions also play a big part regarding the call functionality. Some users should not be able to call everyone whereas others should have the ability to call whoever they want. This requires the app to use multiple layers of validation and security checks to provide dependable and secure functionalities.

This thesis will use the system mentioned above as a tool for investigating its problem statements.
In this chapter, we aim to give an overview of software testing. The focus will lay on the security testing, but as mentioned in the introduction, we will also synoptically present the necessary background for Android client-side functional testing. The following chapter will lay the foundations of our method.

3.1 Functional testing

To fully understand the security testing scope, we must first understand the system vulnerabilities and why it is important to also test the user interface (UI) on the client-side. The primary purpose of these UI tests is to test the actual functionality of the app and possible unwanted activities it might generate. These UI tests can be categorized as functional tests.

From a user perspective, and possible attackers, the UI is what defines the Android application and is one of the most exposed parts of the whole system. The UI enables a variety of possible unwanted user inputs and activities that need to be investigated and tested by the developers. Fortunately, there exists multiple different frameworks and tools to help developers to create automated UI tests on the client side. One of the most used and most appreciated frameworks is the JUnit, with the use of the Android specific Espresso API [5].

JUnit & Espresso

JUnit is an open source framework that provides functionality to create unit tests, which in object-oriented programming is referred to testing a specific method or class. This framework makes it possible to build UI tests in Android with the Espresso API. Espresso is provided by the Android Testing Support Library and is a comprehensive tool for automated UI tests [6]. It utilises the AndroidJUnitRunner class which is a JUnit test runner that simplifies the process of running tests in three steps [7]:

1. Deploys the developed app and tests onto an emulator or device.
2. Runs tests on the emulator or device.
3. Finally reports back the results of the performed tests.
3.2 Fuzz testing

Fuzz testing, or fuzzing as it is also commonly known by, is the art of intentionally giving a program unexpected inputs to discover bugs and thereby vulnerabilities [9]. The problem with this method is that there may be an infinite number of inputs that can be tried to find errors.

The process itself might not be so complicated. Even though most fuzzing is executed automatically, the most trivial form of fuzzing can be hammering the keyboard and see what happens. It is not hard to find inputs to try. The difficulties consist of exploring and investigating all possible inputs. This is an NP-complete and thereby combinatorially intractable problem.

Monkey testing

Monkey testing is one form of fuzzing and is very easy to use. It can for example run directly on your emulator in Android Studio via the command line and simulates a specified number of pseudo-random inputs for the application. The compelling thing about this feature is that the testing environment is customisable. It can, for example, be customised to disable and enable individual input types depending on the intention of the tester [10].

3.3 Sniffing

Sniffing is one of the most common types of security vulnerability exploitation techniques and has the intent of accessing information sent between communicating parts, e.g. exploiting the communication between a client and a server. This type of connection can, for example, be sniffed by using an open source program called Wireshark.

Wireshark

Wireshark is developed for analysing network packets and tries to display packet information as detailed as possible. Thus giving the user a way of measuring the network layer activity and its content [11].

3.4 Reverse engineering

Reverse engineering is a technique with the objective of dissecting a target application to facilitate the analysing process of the application for maintenance or further development purposes. This technique can be further divided into two main parts; code reverse engineering and data reverse engineering. The goal of code reverse engineering is to understand how the application logic is implemented by analysing the source code, whereas data reverse engineering attempts to illuminate which data is stored by the application and how it can be used in a different context [12]. Reverse engineering is not a technique of altering the target application but merely an analysing procedure [13].

With the knowledge obtained from reverse engineering, the developers can then perform a so-called restructuring of the target application. The primary objective of restructuring is to modify the retrieved source code concerning unmet specifications of the original application.
3.4. Reverse engineering

without altering the actual functionality of the application [13]. This can be used in different contexts, rather than just further development, through manipulation of data stored in the system by modifying the source code.

Reverse engineering process

All Android applications are packed into a single Android Package Kit (APK) file. Thus, the process of reverse engineering an Android application is initiated by obtaining that specific APK-file. This can be done by extracting the APK directly from a device with the use of applications like APK Extractor [14] or Assistant for Android [15]. Once the APK-file has been extracted, the applications class-files (of type .java) can be accessed and dissected through the use of three open source programs:

- **apktool** [16] - Decompiles Java bytecode into Smali assembly code and binary resources into initial XML format, which enables code reviewing of the important AndroidManifest.xml and resources-folder [17].
- **dex2jar** [15] - Converts the Dalvik Executable classes.dex-file, located inside the APK, into a .jar-file containing multiple Java specific .class-files.
- **JD-GUI** [19] - Decompiles .jar-files into readable .java-files.

Each tool plays an integral part in disassembling and disclosing the source code of the target application [20].

However, it is not possible to manipulate the obtained class-files due to incomplete and inaccurate conversion when using dex2jar to convert Dalvik bytecode (written in the .dex-files) into Java source code (stored in the .class-files). Nevertheless, the mentioned JD-GUI can restrictively be used to understand the higher level implementation logic of the application. So, to successfully manipulate the source code of an Android application, one needs to manipulate the low-level assembly source code instead. As described in [17], the low-level assembly language used by the Dalvik Virtual Machine (DVM) is called Smali. Each .dex-file consists of multiple .smali-files and Smali code is a lot easier to comprehend than Dalvik bytecode.

Once the APK has been disassembled, with for example apktool, the previous signature of the application is invalid and thus needs to be re-signed. This makes it impossible to create a manipulated APK, extracted from Google Play, and later distribute it as an update on the same Google Play market. However, this is only true if the source APK has already been installed on the device, thus allowing installation of privately signed manipulated APKs [17]. This means that there cannot exist duplicate applications with different signatures on the same Android device.

There exist multiple approach of mitigating a reverse engineering process. The currently most used approach is to obfuscate the source code by altering the name of classes, methods and variables. Some also adds redundant code-snippets to make code analysing exhausting. One of the most commonly used tools for obfuscating Android source code, though it is somewhat minimal protection, is a program called ProGuard [21].

An increased need for research and studies on Android application security has induced multiple different methods and tools for detecting a repackaged Android application. Jiang and Zhou et al. [22] demonstrates a way of detecting repackaging tendencies with the use of a fuzzy hashing technique. Gonzalez and Stakhanova et al. [23] dictates a less computational heavy detection process referred to as AndroidSOO which instead leverage the layout of the important .dex-file and its string identifiers offset. All of the aforementioned meth-
ods provides a complementary way of testing an Android application for possible security vulnerabilities made by malicious repackaging intents, such as the one made in this thesis.

3.5 RESTful API testing

The developed system serves a RESTful API. This means that it uses HTTP requests to GET, POST, PUT and DELETE data [24]. For the sake of the reliability of the system, it is essential that the API behaves as it is supposed to. To ensure this, the functionality has to be tested. This can be done with any API development environment, such as Postman [25]. Postman is a tool used to send a requests to a server and will be used in our case due to prior experiences.

One can use this tool to test functionalities such as response time, the presence of content types and validation of JavaScript Object Notation (JSON) schemas. A server also needs to be tested by sending requests to it. The system holds information about several cases that maps emergencies as explained in section 2.1. What happens if we feed the system with masses of requests in short intervals of time? This could potentially cause the server to overload. This is called denial of service attacks [26], also known as DoS attacks, and is out of the scope of this thesis as mentioned in section 1.4.

3.6 SQL injection

SQL injection is a technique where the attacker inserts a mischievous string of SQL syntax into an SQL query. This mischievous string can be entered into a web page input or some other input field where the string later is sent to an SQL database. This gives the attacker an opportunity to retrieve or tamper with data inside the targeted database [27].

The course of events that take place during an SQL injection attack (SQLIA) can be visualised according to the following figure.

```
+-----------+          +----------------+          +---------------+
|            |          | SQL queries    |          |               |
|            |          | Unauthorised data |          | Server        |
|            |          | extraction      |          |               |
```

Figure 3.1: SQLIA flow

Many web applications have some input fields that are required for accessing specific data. The app sends this input to its server for validation where the input most often is a part of a query. The query that runs on the database could, for example, look like the following.

```sql
Listing 3.1: Example query
SELECT * FROM table WHERE user = 'userid' AND password = 'password';
```

where userid and password are input from the user as explained above. table and user are table and column names respectively in the database and could be named anything. This means that without arrangements, there exist countless opportunities for attackers to type in strings of data directly into the server with the purpose of achieving the desired effect.
The primary process of an SQL attack sequence is that you learn from each error message that you get and evolve your attacks accordingly. The error messages might give useful information regarding vulnerable parts of the application and the database schema.

## 3.7 Related works

The technique adopted in this thesis are a selection of many commonly known techniques when it comes to application security testing. Therefore, the methodologies of these techniques are covered in several earlier reports. Haibin Hu for example performs a widely covered SQL injection review which presents different kinds of SQLIA more in-depth. Sjöstrand and Westberg present Espresso testing among other things on a similar application as ours. The intention of that thesis was not to investigate the availability and reliability which we thought left a gap for us to explore. Their focus was on investigating client side confidentiality and integrity related vulnerabilities of their system, whereas our focus lays on the server side.

Flask’s website state that they try to provide defence against a few different security attacks but explicitly state that the attacks not mentioned are to be taken care of by the developer. The only topic Flask says they have been working on that relates to any of our security testing methods is the JSON security. This means that there are a lot of attacks that are relevant to investigate. A search for scientific articles regarding Flask security testing does not provide much information, and gives us even more room for unexplored attacks.

Batyuk and Herpich et al. perform reverse engineering of an Android application. This paper presents a course of events that are of high relevance for us in this thesis. We do, however, besides the reverse engineering, manipulate the exploited code and later build the application again which is not covered in the aforementioned article.

Fledel and Shabtai et al. identify threats to the Android framework by providing a security assessment. They explain how the Android environment is built and also conduct a code review of various Android components. Their primary focus lays on malicious code injections through different inputs and discover that an attacker can inject code via the web browser. They also state that WebKit, which is Android’s web engine, are prone to injections according to previous attack history. These injections are especially XSS injections which is proposed as future work in this thesis. On top of that, the writers point out that Android handles SQL injections well. These are all general insights for the whole Android framework whereas our work in more specialised towards a specific app. Moreover, our focus lays on investigating the app itself, not so much how it is interacting with potential vulnerabilities within the Android framework.

How developers test Android applications is a precious knowledge to determine how to evolve new testing techniques. M. Linares-Vásquez et al. considered this topic by investigating 102 contributors of open source mobile apps. Through extensive interviews and analysis of quite a few of today’s existing testing methods for Android apps, they found that developers urge for automated tests. Still, since manually written test cases with automation APIs are very fragile to changes in the GUI, many automated tests have a low usage rate. One example is the Google tool Monkey which we will use in this report. Another conclusion that the writers’ state is that code coverage when it comes to testing nowadays is not something many developers consider a good measure of test case quality. Instead, they propose something like fault detection capability and feature coverage. This is, on the contrary, something that is not covered in this thesis.
Now that we have done a review of common tools for developers to test their Android applications, it is time to determine how this will be implemented in this thesis. In this chapter, we detail how we used the tools and approaches covered in chapter 3 in the context of functional and security testing of our application.

4.1 Functional testing

As mentioned previously, UI testing has the intention to ensure that the app works as it is supposed to during user interaction. This is not equivalent to good security, but rather reflects the reliability of the app. We will, therefore, apply tests on the main functionalities of the app to collect evidence of possible security issues that come with poorly implemented functions. Below are some areas in which the Espresso testing will be applied. The full test codes are shown in Appendix A. By running the source code, Espresso simulates the corresponding human interaction for each action presented below.

Authentication testing

The authentication process of the developed application is one of the essential security features. Due to this, we tested this functionality through two tests; ValidLogInTest and InvalidLogInTest. Each test tries to log in with either valid or invalid user credentials respectively on the Login activity screen shown in figure 4.1.
4.1. Functional testing

Map functionality testing

The map functionality is the main usage of the developed application. It needs to be reliable, and thus it should not be able to fail when in use. If so, this would severely influence on the reliability and availability of the application. With this in mind, we constructed a test in which we added a new case to the map through the use of the ValidLoginTest mentioned in the section above. The map view is shown below in figure 4.2 (a) and visualises several cases on the map for an explanatory purpose.
4.2 Fuzz testing

Contacts testing
As mentioned in chapter 2, another important functionality is the access to contacts related to cases in which the logged in user is currently working on. To test this functionality, we conducted yet another Espresso-made test which would check if an ordinary voice call could be established between the user and a contact relevant to the user.

![Contacts activity](image1)

![Call activity](image2)

Figure 4.3: Contacts related activities

Because this app is developed as a mockup for the Swedish Emergency Service, the system language is Swedish. For those who do not understand this language, figure 4.1 asks for a username and a corresponding password. An example of the so-called input fields. The figure 4.2(a) shows as mentioned above a map where the different figures on the map symbolise different cases. The figure 4.2(b) is visualising, as the figure caption implies, the activity for adding cases. In the first input field, the user is asked to add a title for the new case. In the second input field, a description is needed and lastly the user is supposed to choose a category for the new case. Espresso consequently accomplishes all these app-interactions mentioned above.

4.2 Fuzz testing

Monkey testing
For this test, the Android Studio built-in support for monkey testing was used. As it can simulate a vast number of pseudo-random inputs for all parts of the application, the system is put under tremendous pressure. The following command for this feature is run in the terminal to execute the test.

Listing 4.1: Monkey testing command

adb shell monkey -p se.raddning.lekra --pct-syskeys 0 -v 5000
4.3. Sniffing

The number at the end of the string is the number of tests that were executed. This number can be set to an arbitrary number. In our case, we ran 5000 tests for each of the activities shown in figure 4.1 and figure 4.2.

Running 5000 tests on these activities means that 5000 pseudo-random inputs were executed on these activities. These inputs can be everything from typing in invalid strings as user-names to trying to add invalid cases to the map. By “invalid” we refer to something that is not in the correct format or, for example, when a mandatory field is excluded. The reason for running the specific number of 5000 tests is that we thought it seemed like quite a few inputs in conjunction with the time it took to execute them.

4.3 Sniffing

Sniffing could be performed on almost any activity within the tested application, but we have chosen to implement it on, in our opinion, the most security crucial parts. These are the Login activity, the Case activity and the Call activity.

We first started a trace in Wireshark and after that performed all the previously mentioned activities. As mentioned in section 3.3, Wireshark captures all network packets, but we are only interested in the data that is being sent between the client and the server. Therefore, we implemented the following filter to extract the relevant data.

```
ip.addr == 130.236.181.212 and ip.addr == 10.253.252.221 && ssl
```

The first IP address is our own and the second is our server’s IP address. Since the website has a dynamic IP, this means that to reproduce this test, you have to make a lookup on both your own and the website’s IP addresses. The website uses HTTPS which means that the data is sent using HTTP over SSL [33]. Thus, we can also filter by SSL. With this filter, we see all data of interest that is sent.

4.4 Reverse engineering

As mentioned in section 3.4, one of the main purposes of reverse engineering is to analyse the inner workings of a target application. From an attacker perspective, this can be applied to identify possible vulnerabilities located within the logical implementation. And as a result, determining different approaches in which a malicious attack can be performed by reconstructing the implementation of, for example, data handling inside the application.

Due to this, we will investigate the possibility of performing data changes on the back-end server by manipulating the source code on the front-end Android application. The purpose of this investigation is to identify the level of impact in which reverse engineering and reconstruction of a front-end application can have when performing malicious attacks targeting the back-end server concerning confidentiality, integrity, availability and reliability.

Worth noticing is that the target application in this investigation has not yet been provided with any method of preventing reverse engineering, such as PostGuard mentioned in section 3.4.

Server-side data manipulation

Our test consisted of reverse engineering the application and later repackaging it with reconstructed assembly code. The goal was to change the lat-, lon- and role_id-attributes of a user
on the server-side, and thereby changing the location and permission level of this user inside the app. A successful test would result in security vulnerabilities which could be further exploited with more advanced approaches for data manipulation.

To change the desired attributes of a user, we first reverse engineered the APK of the target app and later identified how the front-end app communicated to the back-end server, through a code review on the reverse engineered code. Then we manipulated the relevant .smali-files and repackaged, resigned and reinstalled it on an Android device. We performed the following steps compiled from a mixture of guides and scientific papers [34] [20] [17]:

**APK Extraction**

1. Extracted the desired APK using the app *APKExtractor*.

**Code analysis**

1. Obtained *classes.dex*-file from the APK using macOS provided *unzip*:

   ```bash
   unzip path/to/app.apk classes.dex
   ```

2. Converted the *classes.dex*-file into a *classes.jar*-file using *dex2jar*:

   ```bash
   bash path/to/dex2jar.sh path/to/obtained/classes.dex
   ```

3. Dissected the created *classes.jar*-file using *JD-GUI* (see Appendix C):

   ```java
   java -jar /path/to/executable/jd-gui.jar
   -path/to/created/classes.jar
   ```

**Code reconstruction**

1. Decompiled the APK using *apktool*:

   ```bash
   apktool d path/to/app.apk
   ```

2. Opened the created app-folder with any text editor at hand, in our case we used *Atom* [35].

3. Manipulated the relevant .smali-file by altering the *lat*, *lon* and *role_id* attributes as shown in Appendix D.

4. Repackaged to an executable APK with *apktool*:

   ```bash
   apktool b path/to/app.apk
   ```

5. Signed the modified APK by first creating a private keystore with Android SDK built-in *keytool* and then signing it with Java SDK built-in *jarsigner*:

   ```
   1 keytool -genkey -v -keystore my-release-key.keystore
   -alias alias_name -keyalg RSA -keysize 2048
   -validity 10000
   2 jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
   -keystore my-release-key.keystore path/to/app.apk
   -alias alias_name
   ```
4.5. RESTful API testing

We could now install the application on our Android device and run the app with the reconstructed source code.

To identify the necessary syntax to change, and the values needed, we identified some key methods and variables used to update a user's `lat` and `lon` attributes. We then created our Android project, containing the values and commands we wanted to use to manipulate the relevant `.smali` file. Lastly, we decompiled the built project APK with `apktool` to reveal the corresponding Smalltalk syntax.

4.5 RESTful API testing

Since one fundamental part of the system is its reliability, API testing is a necessary step. As mentioned previously in this thesis, Postman can be used for this. There exists a considerable number of different test skeletons in Postman that can be executed, but we have limited ourselves to perform nine different tests which cover all relevant API functionalities. All tests are written in JavaScript and are performed for each sent request. The skeletons need to be custom fitted for the data that are to be analysed. The source code for the following tests can be found in Appendix B.

Noteworthy is that since the following tests are functional tests and not direct security tests, they demand prior knowledge regarding user credentials and API structure. All this will be explained in the following sections.

Authorization test

The first essential step that needs to be taken is certifying that a person needs to be authorised to receive a valid response from the server when sending a request. This can be tested by sending a request to the server without attaching an authentication bearer. The bearer consists of a token which in turn is created by a hashing process of a user’s credentials. If this bearer is not present or if the token is invalid in the request to the server, it should not respond with any form of valid data, just an error message.

The URL for our main server is [https://itkand-2-1.tddd82-2018.ida.liu.se](https://itkand-2-1.tddd82-2018.ida.liu.se). To access all cases in the database on our server one need to append the prefix `/api/case` to the URL mentioned above. To test whether the server demands authorisation or not, a GET request is sent to the server using Postman. Note that to test the authorisation, one could send any request to the server. The response would either acknowledge you as authorised or give an error of which you could determine whether it demands authorisation or not. Due to our knowledge of the system, we send a GET request.

JSON schema validation

A schema is the structure of the API, so the purpose of this test is to validate that the incoming schema matches the schema that already exists [25]. Below is a data snippet showing a case. Here, `case` is an array containing all the cases, this time only the case with id = 1 is extracted.

```json
Listing 4.2: Example case
{
    "cases": [
        {
            "description": "",
            "id": 1,
            "latitude": 58.4145,
```
This JSON data follows a particular structure, and can be validated by being compared with
the response to a sent request. The request that will be sent is a GET to our URL with the
prefix /api/case/?id=1 and is the same for all these test cases. Note that depending on
the outcome of the first test regarding verification requirements of a user mentioned in section
4.5, you may or may not need to be authenticated to access this kind of information through
the URL. Even if you need to be authenticated, we still think that the following tests are
necessary since they are meant to check the functionality of the API. The authorisation test is
a more security-oriented test.

To make a JSON schema out of the code snippet shown above, we used a website called
jsonschema.net [36]. This website enables input of JSON data and a root id upon which
it then automatically creates the corresponding schema. This schema is then later validated
against the incoming JSON data extracted from the server response.

Content type header check

A content type header check is as the name implies, a check that verifies that some application
data is present in the header. It is a simple test that only takes a few rows of code to implement
as seen in Appendix 9.

Contains string check

This test verifies that every expected string is present in the response from the server. In our
case we expect the following strings:

- Description
- Id
- Latitude
- Longitude
- Title
- Type

JSON value check

If an API is to be able to work correctly, it is crucial that it returns correct values. This could
be implemented on every string that our case contains, but we limited the amount of tests to
one since this should give a fair indication of whether the functionality is correct or not. The
value that is going to be checked is “id” and as the code snippet in listing 4.2 shows, the value
that the data is compared against is 1.
4.6 SQL injection

There are two ways of sending data to the server in this system. One is through the URL field, and one is through the input fields of the applications. It is easy to see that the API does not provide any form of query string handling in the URL, thus giving us no choice but to send the queries through the input fields of the application.

As explained in chapter 3, a complete query to access some information demands at least a table name. Since we act as external attackers without any knowledge about neither the source code nor the database schema, all table and column names are unknown. We do however know what the application does and which service it provides. This gives us the opportunity to make legit guesses regarding these names.

The app demands a username and a password. This makes us believe that there is a table in the database called "user" and a column named "password". We, therefore, try to insert the following query into the password input field of the login page of the application as shown in figure 4.4.

Listing 4.3: Inserted query

\[
\text{SELECT} \ *
\text{FROM} \ user \ \text{WHERE} \ \text{password} = \text{password OR 1' = '1;}
\]

Figure 4.4: SQL injection

The reason for trying "... or 1 '=' 1;" is that it always returns true. This is a simple technique which is commonly known. Kaur and Kour [37], for example, mention this, among many other methods, in their article. The goal is that the server should validate the password and thereby authenticate the user without typing in the correct password.
The reason for the fact that we do not try to inject the server through the username field is that the application has a constraint which allows the user only to type in non-letters. If you decide to insert a letter, nothing shows. This, however, forces the attacker to know a valid personal number of an employee. For this test, we assume that the attacker has this kind of information which is far from unbelievable since complete personal username is public information.
5 Results

In this chapter we describe the results of the tests described earlier in chapter 4.

5.1 Functional testing

Espresso testing

Table 5.1: Espresso testing results

<table>
<thead>
<tr>
<th>Tested activity</th>
<th>Related functionality</th>
<th>Test</th>
<th>Passed</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login activity</td>
<td>Authorization</td>
<td>ValidLogInTest</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>InvalidLogInTest</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Map-/Add case activity</td>
<td>Map</td>
<td>AddCaseFunctionalityTest</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Contact-/Call activity</td>
<td>Contacts</td>
<td>CallFunctionalityTest</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

The table 5.1 states all the test results from our four functional tests made with Espresso. All our tests run to completion and were successful in performing desired user interactions.

5.2 Fuzz testing

Monkey testing

It is almost impossible to visualise all the results of the monkey testing since it applied 5000 pseudo-random inputs. Here is, however, a collocation of all the tests.

Table 5.2: Monkey testing results

<table>
<thead>
<tr>
<th>Tested activity</th>
<th>Number of pseudo-random inputs</th>
<th>Number of unexpected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log in</td>
<td>5000</td>
<td>0</td>
</tr>
<tr>
<td>Main</td>
<td>5000</td>
<td>0</td>
</tr>
<tr>
<td>Adding a case</td>
<td>5000</td>
<td>0</td>
</tr>
</tbody>
</table>
As shown in table 5.2, no unexpected results were found. By “unexpected” we are referring to some action that is not allowed or that accesses some information in an invalid way.

5.3 Sniffing

Here we present the Wireshark results from all traces captured. Overall, this illustrates that no information could be obtained in clear text. Everything is encrypted.

Log in activity

The figure shows the traffic captured when logging into the system. The packet sent first between the client and the server, Client hello, is marked in deep blue and its data is shown at the bottom of the figure. As we see, the data is encrypted and makes thereby no sense.

Figure 5.1: Trace of a log in
5.3. Sniffing

Case activity

Figure 5.2: Trace of adding a case

Just as the first trace, these packets do not contain any readable data. Since we are adding a case instead of logging in this time, the data that is being sent is Application data. An arbitrary packet is marked above, and the corresponding data is shown in the bottom of the figure.
5.4. Reverse engineering

Call activity

The last trace is captured during a call from one client to another. Once again, Application data is sent between the two parties, and as expected, no information is readable.

5.4 Reverse engineering

In figure 5.4 we can see the database before the manipulated APK has been installed and executed. The row which we wanted to manipulate with false data, is the row containing personal_number = 9602301114. As shown in figure 5.5 we can see how our manipulated APK has successfully changed the stored data of the user’s lat- and lon-attributes but were unsuccessful in changing the role_id-attribute. The result of this will be discussed in chapter 6.

Figure 5.3: Trace of a call between two clients

Figure 5.4: Relevant data stored in user table before manipulation
5.5 RESTful API testing

Below, all the test results regarding the API testing are presented.

Authorization test

A GET request is sent, and the server responds with this

![Figure 5.6: Unauthorized access](image)

The body of the response also contains the message “Unauthorized Access”. This means that the server demands a valid bearer with the request.

All API test cases

Since all the API tests could be executed at once, all test results were presented at once. These are the results of the tests described in chapter 4.
5.6. SQL injection

When inserting the SQL query presented in section 4.6, the following action occurs.

The application crashes for some reason. We are going to investigate the reason behind this further in chapter 6.
Discussion

In this chapter we summarise and discuss the outcomes of our tests. We will also discuss the work in social and ethical aspects.

6.1 Results

We performed several kinds of tests on our system, and just a couple of them gave some results worth discussing in more detail. The only tests that resulted in something that could point at some vulnerability in the system were the reverse engineering, and the SQL injection trails. Regarding all other tests, it only showed that the system works as it is meant to and are thereby not going to be discussed in detail.

Functional testing

The Espresso testing worked as it was supposed to and gave no sign of bugs. Since functional testing is fundamental and perhaps the first thing that developers test during the process of creating the app, this result is not surprising since the system does not have that many different functionalities that can go wrong. As mentioned in section 5.1 we only constructed one Espresso test regarding adding a case to the map. There exists endless of different cases that could have been added if you consider all possible inputs for all input fields as shown in fig 4.2(b). One could have changed both the input strings that is typed in manually and the category selection. We did, however, not think that it was necessary to put more time into investigating a larger verity of inputs since one test covers the functionality at large.

The API testing was, however, a bit harder to have preconceptions about. These tests are also fundamental, but not as intuitive as the user interface tests. All performed API tests passed and we could of course, given a broader scope, have completed more test cases. It is hard to predict if it is probable that more test cases would have resulted in some error. Since we covered the vast majority of the API calls the system makes, we do not expect that the results would have been notably different.
6.1. Results

Security testing

Overall, many of the executed security tests did not reveal any vulnerability. Since the fuzz testing performed quite a few random inputs to our system, we thought that something unexpected would occur. So was not the case and with a accumulated amount of 15 000 pseudo-random inputs, none caused the application to behave unexpectedly. This result is quite surprising but is probably based on a well-performed exception handling during the development of the app.

The sniffing showed that the data sent between the client and the server is encrypted. Today, HTTPS is widely used and thus has gained trustworthiness. Therefore, the result was expected.

Reverse engineering

Although the front-end app did not disclose any obvious implementation mistakes made by the developers, we were still able to perform some actions which successfully changed the data stored on the designated back-end server as shown in section 5.4. This is directly related to both integrity and reliability vulnerabilities. Data manipulation in the database, in our case the change of location coordinates, could potentially affect the use of the application. If a case or a user appears to be at one location but is somewhere else, the application cannot be trusted, and the system is deemed compromised.

Why were we only successful to change the lat- and lon-attributes but not the role_id-attribute of a user? The answer to this question lays in the implementation of the communication between the app and the API under test. While analyzing the source code of the application, we realised that the app only sends data to the API endpoints /api/update and /api/case. This, in turn, meant that we were only able to change the location-attributes when the app automatically updated the user’s position to /api/update once the user had been authenticated and successfully logged in to the application. And the only time in which the application sent information to /api/case were when a user added a new case to the map-functionality. This meant that the role_id-attribute of the current user were never addressed through the communication between the front-end application and the back-end server, thus shielding it from manipulation.

With more time to investigate, it is not unbelievable that we could have managed to manipulate some other data in conjunction with user interaction. This could, for example, have been some more personal information and would thereby compromise an employees integrity even worse than just changing the location.

An attacker could also use this reverse engineering technique together with restructuring and repackaging, to distribute a manipulated version of the app to end users without their knowledge of a re-installation. Thus, allowing further data manipulation as mentioned previously, for example altering the data retrieved from the server, such as updating the user location or other vital information about contacts and cases without altering the actual functionality of the application.

As mentioned in section 5.4, developers could proactively counteract reverse engineering and repackage with the use of code obfuscation. However, developers need to consider ways in which a code obfuscation could potentially be deobfuscated. Thus, if an attacker would distribute a repackaged version of the application to end users without their knowledge, some detection method, such as AndroidSOO or DroidMOSS, should be applied as a proactive countermeasure. To adequately protect a specific algorithm (implemented within the Android application) from an attacker, that particular algorithm could instead be transferred to
6.2 Method

The hardest part of this project has no doubt been identifying what parts to test and how. Since the application is developed not only by just us but several other computer science students with quite a lot of security-oriented knowledge, we knew that finding flaws was not going to be easy. The fact that we had almost no prior experience of software testing made it even harder.

With the benefit of hindsight, we would have dug deeper into reverse engineering. We see some real potential in what could be accomplished with more time since we with no prior experience managed to tamper with data within the database.

One could also argue that with more time for research, some of the implemented tests could easily have been scoped out. This, because some libraries used as help when developing the application handles many security threats. The usage of HTTPS, for example, is meant for encryption of the communicating channels. Assuming this works as it should, the sniffing could be predicted not to give any desired results. We do however show that the implementation works as intended.

The primary focus of the description in chapter \(\text{[4]}\) is that some future reader should be able to replicate the tests and receive similar results. In this manner, we think that the chapter fulfills its purpose. However, it takes some time to get used to new programs, and if the future reader is not familiar with for example Postman, it will perhaps take some time to be able to execute the tests. Something evident for us might not come naturally for someone else. Overall, we found some interesting results and were reassured of a few security mechanisms functioning as intended.
6.3 The work in a wider context

The application that has been tested is developed as a mockup for the Swedish Emergency Service. This puts an extra demand on the app’s availability and security. The servers are to store sensitive and privileged information which could cause catastrophic consequences if the application got in the hands of an attacker. To prevent this from happening, testing is a must. Without testing, you have no evidence that the implementation works as it should. With testing at least, you have hopefully prevented some undesirable actions.

Because of the fact that this application are to store information about persons who did not give informed consent to openly share their data, there exists an apparent ethical dimension to the development. A server breach that for example revealed either personal information or information regarding existing emergency cases could decrease the credibility of the emergency service and perhaps be taken to court. Is it okay to store personal data to simplify the work at the cost of the risk of it being compromised? According to our ethical analysis performed during the early stages of the development of the application in the more extensive project group, it is okay. We made the judgment that it is worth the risk.

It may however not have been judged okay if several security mechanisms had not been implemented. The credibility of the system is also affected by whether comprehensive tests on the system were performed or not. The importance of the testing that has been carried out speaks for itself.
In this chapter, we return to the purpose and the research question given in section 1.3. We will also investigate to what extent the aim of this thesis has been achieved. Below are our research questions specified as headlines upon which we will discuss the outcomes.

As described in section 1.2, the purpose of this thesis is both to acquaint the reader with Android functional testing and give a more in-depth understanding of a few security testing techniques. We managed to both perform functional tests and dig a little deeper into some of the security testing methods.

**Investigate which security vulnerability of the developed system that affect the confidentiality, integrity, availability and the reliability of the system**

We found vulnerabilities regarding the availability of the system when performing SQL injections. It turned out to have nothing to do with SQL injections per se but revealed a functionality bug that could be removed.

The security vulnerabilities found when we reverse engineered the application are connected to the system’s integrity and reliability. Meddling with data caused the app to contain invalid information which directly affects the system’s reliability. Since the information that was changed was personal, integrity was also violated.

We did not find any confidentiality vulnerabilities because neither SQL injections nor sniffing enabled us to access any confidential information.

**Identify methods for client-server applications running Android and Python Flask in order to illuminate the system’s security vulnerabilities**

After reading several different scientific articles regarding different topics within security testing, we decided to use the following techniques to test our system:
• Espresso testing
• Fuzz testing
• Sniffing
• Reverse engineering
• RESTful API testing
• SQL injections

Overall we assess that this was a useful selection of techniques to try and identify security vulnerabilities. With the benefit of hindsight, we would have scoped out a few of the methods to dig deeper into reverse engineering. Mainly because we see potential in the topic when it comes to exploiting flaws in the application.

Of course, there exist quite a few other techniques within software testing that could have been used, but it is almost impossible to cover all of them. In regards to the scope of this thesis and the fact that we included both the client and the server side of the application, we think that our choice is relevant and reasonable.

**Apply the identified methods to the developed system in order to detect security vulnerabilities**

We managed to apply all identified methods to our developed system to an almost full extent. By ”full extent” we mean what we had planned to do. The real security vulnerabilities we were able to detect was tampering with data in the database in conjunction with reverse engineering. This is a great deal and a severe weakness of the system.

The fact that we only detected one security vulnerability could mention one of two things. Either the application overall has a high level of security, or we have failed when implementing the other testing techniques. It is hard to determine what the answer to that question is, but we are quite convinced that the tests that have been executed are executed correctly and shows a valid result.

**Future work**

In the future, it would be interesting to dig deeper into reverse engineering and experiment what can and cannot be done according to what is mentioned in section 6.1. It would also be interesting to perform an extensive investigation of SQL injections in regards to the micro framework Python Flask as well as other types of injections. XSS injections for example, which is not covered in this thesis, would be of great interest in a future work.

As mentioned in section 3.7, this thesis did not focus on fault detection capability and feature coverage. By digging deeper into this, one could perhaps produce a more general testing guide for mobile applications including concrete metrics as results.
Bibliography


[34] From APK to readable java source code in 3 easy steps. Last visited: 2018-05-17. URL: http://pof.eslack.org/2011/02/18/from-apk-to-readable-java-source-code-in-3-easy-steps/


Appendix - Espresso testing

Listing A.1: ValidLogInTest

```java
@LargeTest
@RunWith(AndroidJUnit4.class)
public class ValidLogInTest {

    private static final String VALID_USERNAME = "9602301111";
    private static final String VALID_PASSWORD = "password";

    @Rule
    public ActivityTestRule<LoginActivity> activityActivityTestRule =
        new ActivityTestRule<LoginActivity>(LoginActivity.class);

    @Before
    public void setup() {
        // Initializing Intent recorder.
        Intents.init();
    }

    @Test
    public void logInTest() {
        // Clear all input fields.
        ViewInteraction clearUsernameInputField = onView(
            withId(R.id.username));
        clearUsernameInputField.perform(clearText());

        ViewInteraction clearPasswordInputField = onView(
            withId(R.id.password));
        clearPasswordInputField.perform(clearText());

        // Typing in valid user information into text fields.
        ViewInteraction typeValidUsername = onView(
            withId(R.id.username));
        typeValidUsername.perform(typeText(VALID_USERNAME));

        ViewInteraction typeValidPassword = onView(
            withId(R.id.password));
        typeValidPassword.perform(typeText(VALID_PASSWORD));
    }
}
```
typeValidPassword.perform(typeText(VALID_PASSWORD), closeSoftKeyboard());

// Clicking the sign in button.
ViewInteraction pressLoginButton = onView(withId(R.id.sign_in_button));
pressLoginButton.perform(click());

// Checking that the ProgressBar is displayed.
ViewInteraction checkProgressbarVisability = onView(
    withId(R.id.login_progress));
checkProgressbarVisability.check(matches(isDisplayed()));

// Added a sleep statement to match the communication with the server.
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

// Check if MainActivity is displayed.
intended(hasComponent(MainActivity.class.getName()));

@Before
public void setUp() {
    // Setting up Intent recorder.
    Intents.init();
}

@After
public void tearDown() {
    // Tearing down Intent recorder.
    Intents.release();
}

Listing A.2: InvalidLogInTest

@LargeTest
@RunWith(AndroidJUnit4.class)
public class InvalidLogInTest {

    private static final String INVALID_USERNAME = "0123456789";
    private static final String INVALID_PASSWORD = "falsePw";

    @Rule
    public ActivityTestRule<LoginActivity> activityActivityTestRule = new ActivityTestRule<LoginActivity>(LoginActivity.class);

    @Test
    public void logIn() {
        // Clear all input fields.
        ViewInteraction clearUsernameInputField = onView(
            withId(R.id.username));
        clearUsernameInputField.perform(clearText());

        ViewInteraction clearPasswordInputField = onView(
            withId(R.id.password));
        clearPasswordInputField.perform(clearText());

        // Typing in invalid user information into text fields.
        ViewInteraction typeValidUsername = onView(
            withId(R.id.username));
        typeValidUsername.perform(typeText(INVALID_USERNAME));

        ViewInteraction typeValidPassword = onView(
            withId(R.id.password));
        typeValidPassword.perform(typeText(INVALID_PASSWORD), closeSoftKeyboard());
    }
}
// Clicking the sign in button.
ViewInteraction pressLoginButton = onView(withId(R.id.sign_in_button));
pressLoginButton.perform(click());

// Checking that the Progressbar is displayed.
ViewInteraction checkProgressbarVisibility = onView(withId(R.id.login_progress));
checkProgressbarVisibility.check(matches(isDisplayed()));

// Added a sleep statement to match the communication with the server.
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

// Check if the Login form is still being displayed.
ViewInteraction checkLoginFormVisibility = onView(withId(R.id.login_form));
checkLoginFormVisibility.check(matches(isDisplayed()));

Listing A.3: CallFunctionalityTest

@LargeTest
@RunWith(AndroidJUnit4.class)
public class CallFunctionalityTest {
    ValidLogInTest validLogInTest = new ValidLogInTest();

    @Rule
    public ActivityTestRule<LoginActivity> mActivityTestRule = new ActivityTestRule<>(LoginActivity.class);

    @Before
    public void setup() {
        validLogInTest.setup();
    }

    @Test
    public void callActivityTest() {
        validLogInTest.logInTest();

        ViewInteraction openNavigationDrawer = onView(allOf(withContentDescription("Open navigation drawer"),
            childAtPosition(allOf(withId(R.id.toolbar), childAtPosition(withId(R.id.appbar), 0)),
                1),
            isDisplayed()));
        openNavigationDrawer.perform(click());

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
ViewInteraction pressContactsMenuItem = onView(
    allOf(childAtPosition(
        allOf(withId(R.id.design_navigation_view),
            childAtPosition(
                withId(R.id.navigation_view),
                0),
                2),
            isDisplayed())));
pressContactsMenuItem.perform(click());

// Added a sleep statement to match the communication with the server.
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

ViewInteraction pressContactCallButton = onView(
    allOf(withId(R.id.call_button),
        childAtPosition(
            withClassName(is("android.widget.LinearLayout")),
            0),
        isDisplayed()));
pressContactCallButton.perform(click());

// Added a sleep statement to match the communication with the server.
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

/**** SUCCESSFULLY ESTABLISHED A CALL CONNECTION ****/

ViewInteraction pressHangUpButton = onView(
    allOf(withId(R.id.hangup_button), withText("Lägg på"),
        childAtPosition(
            withClassName(is("android.widget.LinearLayout")),
            0),
        isDisplayed()));
pressHangUpButton.perform(click());

private static Matcher<View> childAtPosition(
    final Matcher<View> parentMatcher, final int position) {
    return new TypeSafeMatcher<View>() {
        @Override
        public void describeTo(Description description) {
            description.appendText("Child at position " + position + " in parent");
            parentMatcher.describeTo(description);
        }

        @Override
        public boolean matchesSafely(View view) {
            ViewParent parent = view.getParent();
            return parent instanceof ViewGroup && parentMatcher.matches(parent)
        }
    };

    public void describeTo(Description description) {
        description.appendText("Child at position " + position + " in parent");
        parentMatcher.describeTo(description);
    }

    public boolean matchesSafely(View view) {
        ViewParent parent = view.getParent();
        return parent instanceof ViewGroup && parentMatcher.matches(parent)
    }
}
Listing A.4: AddCaseFunctionalityTest

```java
@LargeTest
@RunWith(AndroidJUnit4.class)
public class AddCaseFunctionalityTest {

    ValidLogInTest validLogInTest = new ValidLogInTest();

    @Rule
    public ActivityTestRule<LoginActivity> mActivityTestRule = new ActivityTestRule<>(LoginActivity.class);

    @Before
    public void setup() {
        validLogInTest.setup();
    }

    @Test
    public void addCaseFunctionalityTest() {
        validLogInTest.logInTest();

        // Clicking the FloatingActionButton to add a new case.
        ViewInteraction floatingActionButton = onView(withId(R.id.markerButton));
        floatingActionButton.perform(click());

        // Added a sleep statement to match the communication with the server.
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Typing in a new case information.
        ViewInteraction typeNewCaseTitle = onView(ocr�
            childAtPosition(ocr�
                withClassName(is("android.widget.LinearLayout"))
                -> t""),
            1),
            isDisplayed()));
        typeNewCaseTitle.perform(replaceText("Fire"), closeSoftKeyboard());

        ViewInteraction typeNewCaseDesc = onView(ocr�
            childAtPosition(ocr�
                withClassName(is("android.widget.LinearLayout"))
                -> t""),
            1),
            isDisplayed()));
        typeNewCaseDesc.perform(replaceText("Fire description"), closeSoftKeyboard());
    }
}
```
typeNewCaseDesc.perform(replaceText("An apartment is on fire"),
                   closeSoftKeyboard());

// Choosing the new case type.
ViewInteraction chooseNewCaseType = onView(   
  allOf(withId(R.id.map_marker_type),
    childAtPosition( 
      withClassName(is("android.widget.LinearLayout")),
      1),
    isDisplayed()));
chooseNewCaseType.perform(click());

DataInteraction selectFireType = onData(anything())
  .inAdapterView(childAtPosition( 
    withClassName(is("android.widget.PopupWindow$PopupBackgroundView")),
    0))
  .atPosition(1);
selectFireType.perform(click());

// Clicking 'Done' button
ViewInteraction pressAddCaseToMapButton = onView(   
  allOf(withId(R.id.register_map_marker_button), withText("Lägg till händelse"),
    childAtPosition( 
      withClassName(is("android.widget.LinearLayout")),
      1),
    isDisplayed()));
pressAddCaseToMapButton.perform(click());

private static Matcher<View> childAtPosition(   
  final Matcher<View> parentMatcher, final int position) {
  return new TypeSafeMatcher<View>() {
    @Override
    public void describeTo(Description description) {
      description.appendText("Child at position "+ position + " in parent "+ parentMatcher.describeTo(description));
    }
    @Override
    public boolean matchesSafely(View view) {
      ViewParent parent = view.getParent();
      return parent instanceof ViewGroup && parentMatcher.matches(parent) && view.equals(((ViewGroup) parent).getChildAt(position));
    }
  };
}
Appendix - RESTful API testing

Listing B.1: API testing code

```javascript
// Schema validation
var schema = {
  "$id": "https://itkand-2-1.tddd82-2018.ida.liu.se/api/case/?id=1",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "cases": {
      "$id": "/properties/cases",
      "type": "array",
      "items": {
        "$id": "/properties/cases/items",
        "type": "object",
        "properties": {
          "description": {
            "$id": "/properties/cases/items/properties/description",
            "type": "string",
            "title": "The Description Schema",
            "default": "",
            "examples": []
          },
          "id": {
            "$id": "/properties/cases/items/properties/id",
            "type": "integer",
            "title": "The Id Schema",
            "default": 0,
            "examples": []
          },
          "latitude": {
            "$id": "/properties/cases/items/properties/latitude",
            "type": "number",
            "title": "The Latitude Schema",
```
"default": 0,
"examples": [
  58.41450119018555
],
"Longitude": {
  "$id": "/properties/cases/items/properties/longitude",
  "type": "number",
  "title": "The Longitude Schema",
  "default": 0,
  "examples": [
    15.600199699401855
  ]
},
"Title": {
  "$id": "/properties/cases/items/properties/title",
  "type": "string",
  "title": "The Title Schema",
  "default": "",
  "examples": [
    "Sverige"
  ]
},
"Type": {
  "$id": "/properties/cases/items/properties/type",
  "type": "string",
  "title": "The Type Schema",
  "default": "",
  "examples": [
    "Eld"
  ]
}];

var jsonData = pm.response.json();

pm.test('Schema is valid', function() {
  pm.expect(tv4.validate(jsonData, schema)).to.be.true;
});

pm.test("Content-Type is present", function () {
  pm.response.to.have.header("Content-Type");
});

pm.test("Description is present", function () {
  pm.expect(pm.response.text()).to.include("description");
});

pm.test("ID is present", function () {
  pm.expect(pm.response.text()).to.include("id");
});

pm.test("Latitude is present", function () {
  pm.expect(pm.response.text()).to.include("latitude");
});

pm.test("Longitude is present", function () {
  pm.expect(pm.response.text()).to.include("longitude");
});
pm.test("Title is present", function () {
    pm.expect(pm.response.text()).to.include("title");
});

pm.test("Type is present", function () {
    pm.expect(pm.response.text()).to.include("type");
});

// Response body: JSON value check

var jsonData = pm.response.json();

pm.test("ID should be " + jsonData.cases[0].id, function () {
    pm.expect(jsonData.cases[0].id).to.eql(1);
});
Figure C.1: Source code of target application shown in the JD-GUI program
In figure C.1, we can see how a successful reverse engineering process described in 4.4 is disclosing the source code of the target application.
Listing D.1: Initial source code

630 .method public doCompleteUpdate()V
631 .locals 6
632
633 .line 89
634 sget-object v0, Lse/raddning/lekra/Manager/MapManager;->TAG:Ljava/lang/String;
635
636 const-string v1, "Updating...
637
638 invoke-static {v0, v1},
Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
639
640 .line 90
641 move-object v0, p0
642
643 .line 91
644 .local v0, "mMapManager":Lse/raddning/lekra/Manager/MapManager;
645 new-instance v1, Lorg/json/JSONObject;
646
647 invoke-direct (v1), Lorg/json/JSONObject;-><init>()V
648
649 .line 93
650 .local v1, "postData":Lorg/json/JSONObject;
651 const-string v2, "battery_level"
652
653 invoke-virtual (v1, v2),
Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
654
655 .line 95
656 const-string v2, "lat"
657
658 invoke-virtual (v1, v2),
Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
659
660 .line 96
661 const-string v2, "lon"
invoke-virtual {v1, v2},
    \ Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;

    .line 98
    try_start_0
    new-instance v2, Lorg/json/JSONObject;
    invoke-direct (v2), Lorg/json/JSONObject;-><init>()V
    .line 100
    .local v2, "pos":Lorg/json/JSONObject;
    const-string v3, "lat"

    iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocation:
        \ ion:android/location/Location;
        if-nez v4, :cond_0

    sget-object v4, Lorg/json/JSONObject;->NULL:Ljava/lang/Object;
    goto :goto_0

    :cond_0
    iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocation:
        \ ion:android/location/Location;
        invoke-virtual {v4}, Landroid/location/Location;->getLatitude()D
        move-result-wide v4
        invoke-static {v4, v5}, Ljava/lang/Double;->valueOf(D)Ljava/lang/Double;
        move-result-object v4
        :goto_0

    invoke-virtual {v2, v3, v4}, Lorg/json/JSONObject;->put(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/Object;

    .line 101
    const-string v3, "lon"

    iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocation:
        \ ion:android/location/Location;
        if-nez v4, :cond_1

    sget-object v4, Lorg/json/JSONObject;->NULL:Ljava/lang/Object;
    goto :goto_1

    :cond_1
    iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocation:
        \ ion:android/location/Location;
        invoke-virtual {v4}, Landroid/location/Location;->getLongitude()D
        move-result-wide v4
        invoke-static {v4, v5}, Ljava/lang/Double;->valueOf(D)Ljava/lang/Double;
        move-result-object v4
        :goto_1

    invoke-virtual {v2, v3, v4}, Lorg/json/JSONObject;->put(Ljava/lang/String;Ljava/lang/Object;);
.line 103
const-string v3, "battery_level"

iget-object v4, p0,
    \ Lse/raddning/lekra/Manager/MapManager;->app:Lse/raddning/lekra/LekraApp;
invoke-virtual {v4}, Lse/raddning/lekra/Manager/MapManager;->getBatteryPercentage()F
move-result v4
const/high16 v5, 0x42c80000  # 100.0f
mul-float/2addr v4, v5
invoke-static (v4), Ljava/lang/Math;->round(F)I
move-result v4
invoke-static (v4), Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v4
invoke-virtual {v1, v3, v4}, Lorg/json/JSONObject;->putOpt(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;

[line 104
const-string v3, "pos"

invoke-virtual {v1, v3, v2}, Lorg/json/JSONObject;->putOpt(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;
:Try_end_0
.end local v2    # "pos":Lorg/json/JSONObject;
goto goto_2

[line 108
:catch_0
move-exception v2
local v2, "e":Lorg/json/JSONException;
invoke-virtual {v2}, Lorg/json/JSONException;->printStackTrace()V
.end local v2    # "e":Lorg/json/JSONException;
goto_2

[line 109
new-instance v4, Lse/raddning/lekra/Manager/MapManager$2;
invoke-direct {v4, p0, v0}, Lse/raddning/lekra/Manager/MapManager$2;-><Init>(Lse/raddning/lekra/Manager/MapManager;Lse/raddning/lekra/Manager/RequestManager;)V
invoke-virtual {v2, v1, v3, v4},
    \ Lse/raddning/lekra/Manager/RequestManager;->doPostRequest(Lorg/json/JSONObject;)V
call LJava/lang/String,Lse/raddning/lekra/Utilities/RequestCallback;
.return-void
.end method
Listing D.2: Manipulated source code

```
.method public doCompleteUpdate()V
 .locals 6
 .line 89
 sget-object v0, Lse/raddning/lekra/Manager/MapManager;->TAG:Ljava/lang/String;
 const-string v1, "Updating..."
 invoke-static {v0, v1},
      Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
 .line 90
 move-object v0, p0
 .line 91
 .local v0, "mMapManager":Lse/raddning/lekra/Manager/MapManager;
 new-instance v1, Lorg/json/JSONObject;
 invoke-direct {v1}, Lorg/json/JSONObject;-><init>()V
 .line 93
 .local v1, "postData":Lorg/json/JSONObject;
 const-string v2, "battery_level"
 invoke-virtual {v1, v2},
      Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
 .line 95
 const-string v2, "lat"
 invoke-virtual {v1, v2},
      Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
 .line 96
 const-string v2, "lon"
 invoke-virtual {v1, v2},
      Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
 .line 97
 const-string v2, "role_id"
 invoke-virtual {v1, v2},
      Lorg/json/JSONObject;->opt(Ljava/lang/String;)Ljava/lang/Object;
 .line 99
 try_start_0
 new-instance v2, Lorg/json/JSONObject;
 invoke-direct {v2}, Lorg/json/JSONObject;-><init>()V
 .line 101
 .local v2, "pos":Lorg/json/JSONObject;
 const-string v3, "lat"
 iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocat
      Landroid/location/Location;
      ion:Landroid/location/Location;
 if-nez v4, :cond_0
 sget-object v4, Lorg/json/JSONObject;->NULL:Ljava/lang/Object;
```
goto :goto_0
:cond_0
# Test implementation: Change Longitude attribute: lat
const-wide v4, 0x404dbfb1af3a14cfL  # 59.49761
invoke-static {v4, v5}, Ljava/lang/Double;->valueOf(D)Ljava/lang/Double;
move-result-object v4
:goto_0
invoke-virtual {v2, v3, v4}, Lorg/json/JSONObject;->put(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;
.line 102
const-string v3, "lon"
iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->latestKnownUserLocation:Ljava/location/Location;
if-nez v4, :cond_1
sget-object v4, Lorg/json/JSONObject;->NULL:Ljava/lang/Object;
goto :goto_1
:cond_1
# Test implementation: Change Longitude attribute: lon
const-wide v4, 0x40321268a0d349bfL  # 18.071909
invoke-static {v4, v5}, Ljava/lang/Double;->valueOf(D)Ljava/lang/Double;
move-result-object v4
:goto_1
invoke-virtual {v2, v3, v4}, Lorg/json/JSONObject;->put(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;
.line 104
const-string v3, "battery_level"
iget-object v4, p0, Lse/raddning/lekra/Manager/MapManager;->app:Lse/raddning/lekra/LekraApp;
invoke-virtual {v4}, Lse/raddning/lekra/LekraApp;->getBatteryPercentage()F
move-result v4
const/high16 v5, 0x42c80000  # 100.0f
mul-float/2addr v4, v5
invoke-static {v4}, Ljava/lang/Math;->round(F)I
move-result v4
invoke-static {v4}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v4
invoke-virtual {v1, v3, v4}, Lorg/json/JSONObject;->putOpt(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;
.line 105
const-string v3, "pos"
invoke-virtual {v1, v3, v2}, Lorg/json/JSONObject;->putOpt(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;

# Test implementation: Change Permission level attribute: role_id

const-string v3, "role_id"
const/4 v4, 0x1
invoke-static (v4), Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v4
invoke-virtual {v1, v3, v4}, Lorg/json/JSONObject;->putOpt(Ljava/lang/String;Ljava/lang/Object;)Lorg/json/JSONObject;

try_end_0
.catch Lorg/json/JSONException; {:try_start_0 .. :try_end_0} :catch_0

.line 112
.end local v2 # "pos":Lorg/json/JSONObject;
goto :goto_2

.line 110
:catch_0
move-exception v2

.line 111
.local v2, "e":Lorg/json/JSONException;
invoke-virtual {v2}, Lorg/json/JSONException;->printStackTrace()V

.line 114
.end local v2 # "e":Lorg/json/JSONException;
goto_2

.iget-object v2, p0, Lse/raddning/lekra/Manager/MapManager;->requestManager:Lse/raddning/lekra/Manager/RequestManager;

const-string v3, "/api/update"
new-instance v4, Lse/raddning/lekra/Manager/MapManager$L2;
invoke-direct (v4, p0, v0), Lse/raddning/lekra/Manager/MapManager$L2;-><init>(Lse/raddning/lekra/Manager/Lse/raddning/lekra/Manager;Lse/raddning/lekra/Manager/MapManager;)V

invoke-virtual {v2, v1, v3, v4},
Lse/raddning/lekra/Manager/RequestManager;->doPostRequest(Lorg/json/JSONObject;Lse/raddning/lekra/Utilities/RequestCallback;)V

.return -void
.end method