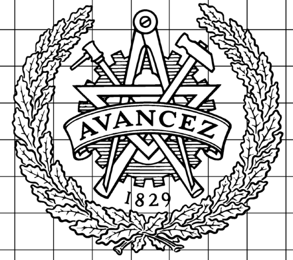
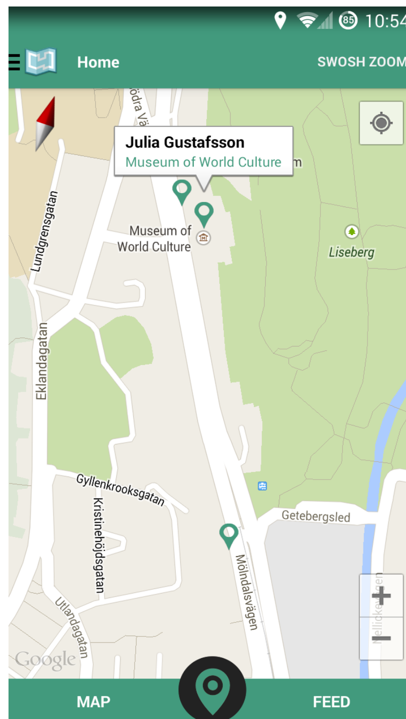
CHALMERS





Safe and Social

Location-based Social Network Focusing on Security

Bachelor of Science Thesis in Computer Science and Engineering

JULIA GUSTAFSSON
ANTONIOS KIOKSOGLOU
ANTON KLOEK
VICTOR LINDHÉ
BARNABAS SAPAN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Safe and Social
Location-based Social Network Focusing on Security

Julia Gustafsson
Antonios Kioksoglou
Anton Kloek
Victor Lindhé
Barnabas Sapan

© Julia Gustafsson, June 2014.
© Antonios Kioksoglou, June 2014.
© Anton Kloek, June 2014.
© Victor Lindhé, June 2014.
© Barnabas Sapan, June 2014.

Examiner: Jan Skansholm

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: Map view from the application of this project, as described in section 1.5]

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Foreword

This report is a bachelor thesis from a group of students at Chalmers University of Technology, Göteborg, Sweden. The project was carried out in the spring of 2014. Different security measures suitable for Android and web development were analysed, as a smartphone application was developed.

The authors wish to thank Andreas Abel for supervision of the project, and Tomas Olovsson for providing valuable input during an interview.

Abstract

Security breaches can be potentially harmful for individuals, companies, and authorities. As a result of this, it could be beneficial for developers to know how to best secure user data in the increasing area of smartphone applications.

This thesis discusses several security measures that can be taken by developers to secure sensitive data from attackers. During the investigation, an Android application was built together with a server backend.

To protect data, this project uses encryption, secure connections, database protection, and protection of the source code. Best practices for persistent login and server environments were explored.

The security measures investigated during this project are applicable on both Android applications and web servers. The result could be used both by companies and authorities to secure user data.

Sammandrag

Säkerhetsintrång kan innebära stora skador för individer, företag samt myndigheter. På grund av detta kan det vara fördelaktigt för utvecklare att veta hur man lämpligast säkrar data för mobila applikationer.

Denna uppsats undersöker olika typer av säkerhetsåtgärder som kan användas av utvecklare för att säkra känslig data, utifrån ett flertal typer av angripare. Under projektets gång utvecklades en applikation för Android, och en tillhörande back-end på en server för att hantera användardata.

Undersökningar om hur man lämpligast upprätthåller en anslutning mot server och servermiljön har genomförts. För att skydda data har ett flertal metoder används, såsom kryptering, säker dataöverföring, skydd mot databasattacker, server- och databasbehörigheter, samt skydd av källkod.

Säkerhetsåtgärderna som undersökts i det här projektet går att tillämpa på både applikationer för Android, och webbservrar. Resultaten kan vara till stor nytta för både myndigheter och företag som vill säkra deras användardata.

Glossary

.apk	Android application package file, containing the executable and other resources of an Android application
.dex	File format for files executable by the Dalvik Virtual Machine
Activity	Component for a single screen in Android, can be programmed using the Activity class
ADB	Android Debug Bridge, command line tool for debugging Android applications
AES	Advanced Encryption Standard, symmetric encryption algorithm
Application Manifest	File where the general behaviour of an Android application is declared, such as permissions needed
Apache	Software for running web servers
API	Application Programming Interface, a contract on how a set of programmed functions should be used
Broadcast receiver	Component for listening to broadcast intents in Android
Ciphertext	Encrypted text which is not readable
CBC	Chained Block Ciphers, mode for encrypting blocks by XORing previous block of ciphertext before encryption
Cookie	File stored with a client for persistent information, which a server can read and write
DVM	Dalvik Virtual Machine, a virtual machine built to run on hardware with limited resources
ECB	Electronic Code Book, mode for encrypting blocks separately

GCM	Google Cloud Messaging, service to send notifications from a web server to an Android application
Hashing	A method for turning plaintext into irreversible ciphertext
HTTP	Hyper Text Transfer Protocol, standard protocol for web communications in browsers
HTTPS	HTTP Secure, an extension of the HTTP protocol for using SSL/TLS
Initialization vector	Random string used when encrypting the first block with CBC
Intent	A message type for communication between Android components
Internal Storage	Private, persistent storage in Android to store files
JSON	JavaScript Object Notation, easy-readable data format based on key-value pairs
PHP	Scripting language, used in this project for generating dynamic responses on the server
Plaintext	Normal, readable text as opposed to encrypted ciphertext
Relational database	Database where data is stored in tables
Reverse Engineering	The process of reversing software back to source code
RSA	Public-key cryptographic algorithm
Salt	Random number to used to make identical passwords different after hashing
Session	Time frame during which a server can remember the actions of a client

SharedPreferences	Key-value based persistent, private storage in Android for simple data types
SHA-1	Secure Hashing Algorithm 1, popular algorithm for hashing
SSL	Secure Socket Layers, protocol for encrypting data sent between a client and a server. Predecessor of TLS.
SQL	Structured Query Language, used to manipulate data in relational databases
SQL injection	An attack on a database where SQL code is run remotely
Tapjacking	An attack on smartphone applications where touch events are forwarded to invisible layers, in order to deceive the user
TLS	Transport Layer Security, successor to SSL
Toast	A box used for showing short messages in Android, for a short time
VPS	Virtual private server, an instance of a virtual machine where the customer has full root access
XML	Extensible Markup Language, used for building views in Android and declaring application behaviour in the manifest

Table of Contents

1. Introduction 1
1.1. Background 1
1.2. Purpose 1
1.3. Risk analysis 2
1.4. Security goals 2
1.5. The application prototype 3
1.6. Report structure 4
1.7. Delimitations 4
2. Technologies of the Product 6
2.1. User permissions in Linux 6
2.2. Storing data in databases 7
2.3. The structure of an Android application 7
2.4. Communication between application and server 9
2.5. Notifying the application through Google Cloud Messaging 10
3. Security Background 12
3.1. Regarding data as sensitive 12
3.2. Protecting data with cryptography 12
3.3. Vulnerabilities of an Android application 14
3.4. Attacking a database through SQL injection 15
3.5. Data in transit is vulnerable 16
4. Method 18
4.1. Literature and research 18
4.2. Configuring and developing the server application 18
4.3. Developing the Android application 18
4.4. Testing the security measures 19
4.5. Other tools used in this project 19
5. Architecture and implementation 20
5.1. Sensitive data of this product 20
5.2. Overview of the server application 20
5.3. Functionality and security of the Android application 24
5.4. Product functionality 27
6. Testing and evaluation of security measures 30
6.1. Using reverse engineering with an obfuscated application 30
6.2. Attempting to start a tampered application 30
6.3. Attempting to start a the application on a rooted device 31
6.4. Reading from the encrypted application database 32
6.5. Testing the tapjacking protection 33
6.6. Listening to the data in transit 33
6.7. Reading from the server database 34
6.8. Attempting SQL injection on the server database 34
6.9. Attempting to sign in using stolen token 34
7. Result and Discussion 35
7.1. Evaluation of the test results 35
7.2. Discussing the security measures from a general perspective 36
7.3. Trusting third parties 37
7.4. Future work 37
7.5. Conclusion 38
References 39
Appendix A: Interview with Tomas Olovsson I
Appendix B: Libraries, tools and services II

1. Introduction

Due to the rising popularity of smartphone applications, it could be beneficial for developers to know how to protect the data of their users. This project aims to develop a simple prototype application for Android with a server backend, to investigate what kinds of security measures that can be taken by developers.

1.1. Background

The popularity of smartphone applications has exploded in recent years. In 2013, the use of mobile applications has increased by 115% (Khalaf 2014). Forecasts say that the total of application downloads will break 200 billion per year in 2016 (Stamford 2013). Social applications are increasing the most, with an increase of 203% in 2013 (Khalaf 2014).

With the rise of smartphone applications, it could be beneficial for developers to know how to best secure user data. Security breaches can be devastating for both users and the developer. Since it is common for applications to use some kind of server backend, developers should know how to secure data there as well.

Security breaches can be potentially harmful for individuals, companies, and authorities. In 2012, the Swedish IT-company Logica was attacked, which led to personal data of thousands of Swedish citizens being compromised, many of the victims having protected identities. Logica managed the transfer of personal data from the Swedish tax authority to several government agencies, such as the police (Åhlin 2012). Companies dealing with sensitive user data have a responsibility towards customers to make sure data is protected.

A good knowledge among developers regarding security could lead to more secure information systems around the world. Some argue that increased security online could save the global economy trillions of dollar every year (World Economic Forum 2014). Spending time and money on increasing the security of software products might therefore benefit the global economy in the long run.

The results of this project could be applied in many different scenarios such as applications for banks, hospitals, and law enforcement. Those kinds of organisations deal with data that is usually classified and should never be available to an unauthorised third party.

1.2. Purpose

This project aims to develop both a simple prototype application for Android and a server backend. The produced Android application will be an interface for a social network based on sharing locations. Along with the development of this prototype, solutions for common security issues will be investigated and presented in this thesis. The security focus will be on the basic measures that can be taken by developers to protect the sensitive data of users against attackers. This thesis will cover security measures suitable when developing and launching an Android application, together with a web server backend.

1.3. Risk analysis

A smartphone application that works together with a server backend is potentially vulnerable from multiple angles. Even if the application should be secured, data could be stolen from other parts of the system, which the application interacts with, as visualised in **Figure 1**. This project aims to secure data from attacks on the device, on the server backend, and in transit.

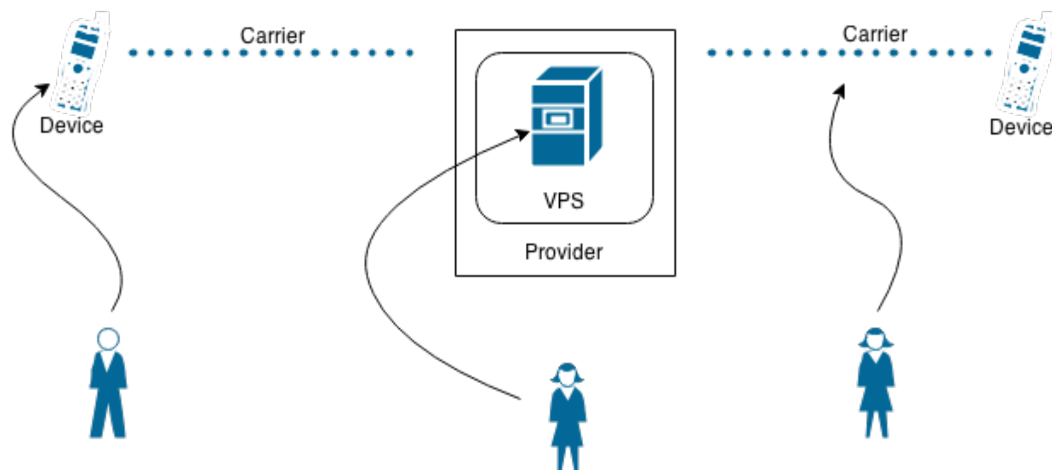


Figure 1: Abstract overview of a mobile application such as the one of this project, communicating with a server. Three different attackers are attacking different parts of the system in order to steal user data.

There are several risks for a system like the one developed in this project:

- Data storage, on both the server and the application, could be accessed through an attack by either a human or software.
- Databases, on both the server and the application, could be attacked by remotely running SQL code.
- Data in transit could be hijacked and its messages could be revealed.
- The Android application could be reverse engineered and its source code and server communication could be exposed.
- The Android application could be manipulated and released to users, who may use it believing it is the original one.
- Touch events on the Android device could be manipulated, triggering actions that the user remains unaware of.
- Login data used for persistent login could be stolen and used to sign in on another user's account.

1.4. Security goals

Given the security risks of a system like the one developed in this project, there are several measures that could be taken to minimise the risk of user data being stolen. The measures taken in this project will aim to fulfil the following goals:

- If the server database would be compromised, the data regarded as sensitive should still be encrypted and unreadable to the attacker.
- If the data in transit would be listened to, the data packets should not be readable.

- If the data storage on the device would be compromised, the data regarded as sensitive should still be encrypted and unreadable to the attacker.
- If the application would be reverse engineered, the original source code should be obfuscated, making it more difficult to manipulate it.
- If a third party would manipulate the application, the application should not be able to run.
- If the application would run in an unsecure environment, the user should be warned.
- If the user would be deceived into unknowingly triggering actions in the application, the application should not respond to touch events.
- If the persistent login data would be stolen, there should be a way to detect this.
- If an attacker would attempt to run SQL code remotely, the code should be sanitised and treated like a string.

1.5. The application prototype

The application prototype developed during this project will be used to create a context for the security measures. The functionality of the application aims to work as a real social network, in the sense that users can interact with each other from the application.

Basic functionality such as registering, signing in and remaining signed in while not active, *persistent login*, will be implemented. Friends of a user will be able to see the current and previous locations. Users will be able to search for each other and to send friend requests, which must be accepted before user interaction can begin. The list of friends together with a notification of pending friend requests is shown to the right in **Figure 3**.

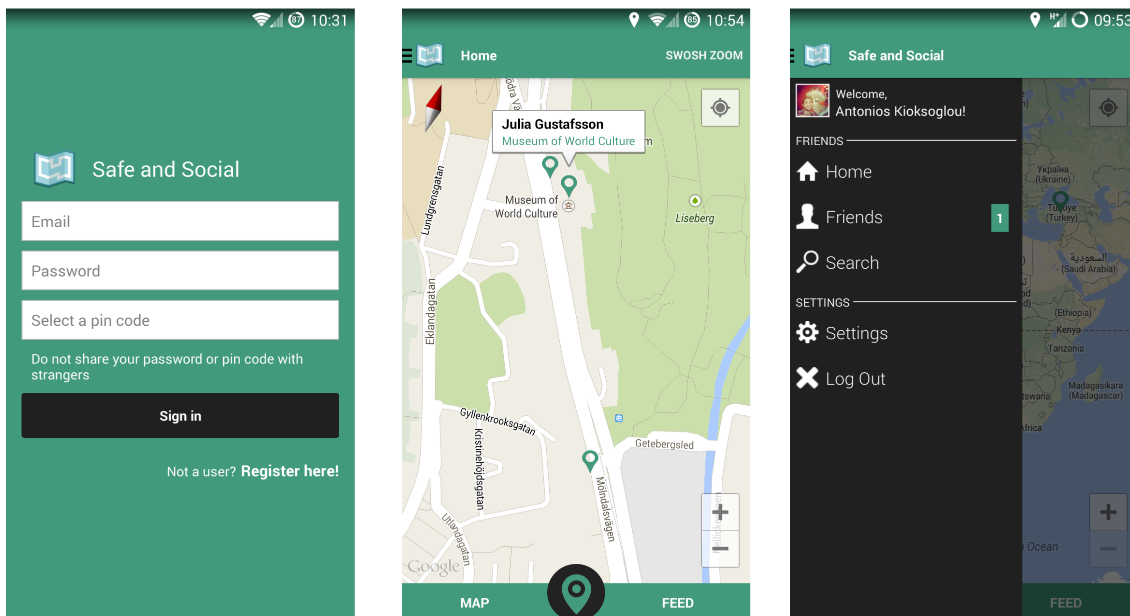


Figure 2: The login screen is to the left. In the middle, a map view is presented with locations of users together with more specific information if the user presses a location. To the right, the map is visible along with the navigation drawer.

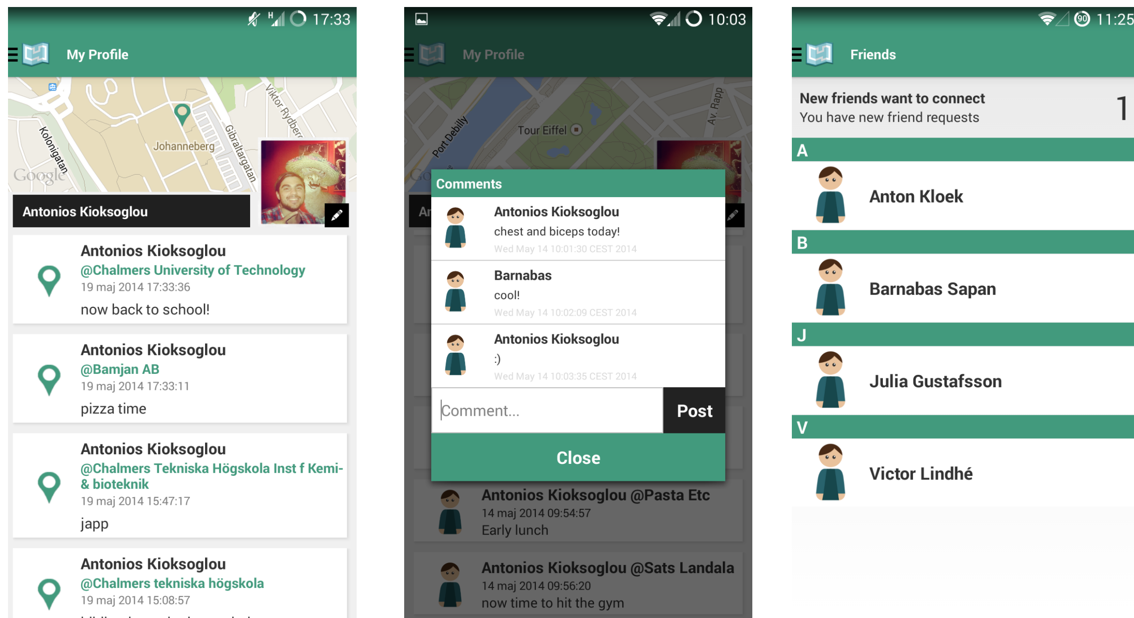


Figure 3: The user profile is presented to the left, along with visited locations. In the middle, the user is commenting a shared location. The list of friends is shown to the right.

Locations will be saved as geographical coordinates, together with information about the place where the user is, as seen in the middle of **Figure 2** and to the left in **Figure 3**. When sharing a location, the friends of the user will be notified through the application. Interaction is available through commenting on shared locations, as seen in the middle of **Figure 3**.

1.6. Report structure

An introduction to the technologies behind the product is given in Chapter 2. Section 2.3 focuses on the relevant parts of developing Android applications, with respect to the scope of this project. Section 2.4 focuses on the technologies used in the server backend.

The technical background is followed by an introduction to different security concepts in Chapter 3. These concepts will later be referred back to when describing implementation in Chapter 5. Chapter 6 focuses on testing the security measures taken during development. In Chapter 7, the result will be discussed and connected with the purpose of this project.

1.7. Delimitations

This project aims to investigate how a developer or a team of developers can secure sensitive user data, within the scope of a developed product. Therefore, necessary third parties are trusted. Third parties include the provider of the *virtual private server - VPS*, the Internet infrastructure in Sweden, phone carriers, mobile manufacturers, the Ubuntu operating system and the Android operating system. System administrators at the VPS provider are trusted as well. For more information on the VPS, see section 2.4.

Both the Android application and the server are built with several external libraries, which are trusted. See section 4.2 and 4.3 for more information about these libraries. In addition to the libraries used in the project, external *application programming interfaces*

- *APIs* - are also trusted since they provide a lot of functionality needed to make a usable product. During development, the tool *Git* and the repository service *Github* are trusted.

The investigation of this project will consider only the technologies used in this project. This means that, though probably applicable elsewhere, the solutions will only be tested and evaluated with respect to the technologies of this project.

Although important when building end-user applications, this thesis does not analyse user experience and design. Some resources during development will be used to design the prototype, but no analysis of the application from these perspectives will be made. Implementation of different functions of the product will only be covered when relevant from a security perspective.

2. Technologies of the Product

This project largely relies on Linux, SQL databases and different web communication technologies. The security concepts described in Chapter 3 are applicable for the technologies described in this chapter.

2.1. User permissions in Linux

In this project, two operating systems are used - Android, that runs on the mobile device, and Ubuntu, which runs on the web server. Both these operating systems are built on top of the Linux kernel (Android Developers 2014a; Ubuntu.com 2014). To secure these systems, it is relevant to have a basic understanding for how user permissions are handled in Linux.

In Linux, which is based on Unix, each and every file divides users into three categories. The first category is the owner of the file, and there can be exactly one owner. Secondly, there is a group of users, which the file has an association to. Any user belonging to that group are treated the same way by the file. Finally, there is the category of *other* users, which basically consists of everyone else (Snyder, Myer & Southwell 2010).

Each one of these categories has a set of permissions for a file. There are three kinds of permissions: *read, write* and *execute.* A category either has or has not anyone of these permissions. As an example, the owner could have permission to read and write a file, while the group is only having permission to execute a file. The execute permission is necessary for a user to be able to run a specific program (Snyder, Myer & Southwell 2010).

Directories also have permissions, which can override the permissions of the files inside. If a directory does not let a user to read from it, the user will neither be able to read from the files inside it. On the other hand, if a directory is accessible for anyone and a file is not, the file does not become more accessible, than the fact that it can be seen when listing the files in the directory (Snyder, Myer & Southwell 2010).

Every software program is considered as a user. From the perspective of the operating system, a program is called a *process.* A process needs to have the permission to access files it works with (Haas 2014). For example, a process running a web server needs to have access to the files of a website in order to show its content.

The root user, also called the *superuser*, is the only user who has access to all the files and directories in an instance of an operating system. For a program to have the rights of a root user, it has to be executed by the root user. The root user is the only user who can change the ownership of a file (Snyder, Myer & Southwell 2010).

2.2. Storing data in databases

In this project, a *MySQL* database is used on the server and two *SQLite* databases are used for the Android application. Both these database systems use tables for storing data, making them *relational database systems.* To manipulate data in relational databases, *Structured Query Language - SQL* - is used (Garcia-Molina, Ullman & Widom 2014).

In this project, only the most common SQL queries are used to work with data. These are:

- *SELECT:* This query fetches table rows from a database.
- *INSERT INTO:* This query creates a new table row.
- *UPDATE:* Updates one or several existing table rows.
- *DELETE:* Removes all table rows that match a certain condition.

Since databases are created for storing large amounts of data, they are among the most valuable targets for an attacker. In this project, databases are encrypted on both the Android device and the web server. In addition to encryption, the web server database manages permissions to restrict what can be done from the server application. This is further described in section 5.2.

2.3. The structure of an Android application

Android applications are mainly written in Java and *Extensible Markup Language - XML*. When compiled, an application is compressed into an *application package* file with the extension *apk* (Android Developers 2014b), where the executable *dex*-file is stored (Gunasekera 2012).

The Application manifest and application permissions

The core of every Android application is the *application manifest*, located in the file *AndroidManifest.xml*. The manifest is written in XML and is located in the project root directory. The permissions of an application are declared in the manifest, and must be granted by the user explicitly. This is usually done during the installation of the application (Gunasekera 2012). For example, an application must declare that it needs permission to access the GPS functionality of the device (Android Developers 2014c).

Using components for different purposes

An Android application is built up by parts referred to as *application components* or just *components*, with each type of component having a distinct purpose. All types of components are declared in the application manifest (Android Developers 2014b).

Components interact with each other using *intents* (Android Developers 2014d). Intents can be seen as messages between components (Six 2012). Intents can be divided into two types, with the first one being *explicit intent*, which is sent to a specific component. The second type is called *implicit intent* or *broadcast intent*, and is sent to the entire system (Android Developers 2014d). To prevent other applications from interacting with the components of this Android application, the *exported* flag for each component is set to *false*.

The component *activity* is a single screen, which the user sees and interacts with. Activities are implemented by inheriting the *Activity* class. An activity handles the logic of user interactions, such as different events (Android Developers 2014e). Applications usually consist of several activities, where one activity serves as the launcher activity, being the first one to appear when an application is started (Android Developers 2014e). This project, for example, has an animation screen - or *splash screen* - as launcher activity.

Another component used in this project is the *broadcast receiver.* A broadcast receiver works as a listener for broadcast intents in the system. With broadcasts, sensitive information should not be sent, as the information can be read by other applications with permissions to listen to the same intents (Android Developers 2014f). In this project, broadcast receivers are used to receive notifications from the server through Google Cloud Messaging, which is further described in section 2.5.

Displaying user interface with views

Some attacks on an Android application are targeting the user interface, which consists of *view* objects. Views are all the blocks displayed on the screen and are stored in XML files, where the layout is marked up. The behaviour of a view is programmed in Java classes. It is possible to extend the *View* class to create new types of custom views (Android Developers 2014g).

There are many simple types of views in Android, such as the *toast.* A toast is a simple popup that displays a text message for a period of time (Android Developers 2014h). A toast cannot be displayed more than a maximum time span defined by Android (Wei 2010). Toasts can be used as an attack on the application through tapjacking, which is described in section 3.3. In this project, toasts are used in the application to display short messages to the user.

Saving data persistently in Android

There are many ways to store data persistently in Android, and which one to use depends on the situation. In this project, three different types of storages are used for different purposes. The data stored in each of these three types is private and not accessible outside the application, except for the root user. See section 2.1 for more on permissions in Linux systems.

Primitive data types and strings can be saved using the Java class *SharedPreferences*. The data is stored using key-value pairs. If the application should be killed or the application should be rebooted, the data would still be persistent (Android Developers 2014i). Because of the simple structure of SharedPreferences, the application in this project uses it to save simple user data when a user is logged onto the application.

For larger amounts of data, Android provides an interface for SQLite databases (Android Developers 2014i). In this project, the databases are used to store friends and locations in the application after that the data has been fetched from the web server, together with necessary data from GCM.

The *internal storage* is for storing any type of data that can be stored as files. By default, files saved here are private, which neither other applications or the user can access directly (Android Developers 2014i). This project saves the profile pictures of users as files in the internal storage.

Executing the Android application

The executable of an Android application is stored as a *dex*-file in the application package. This file is created through a conversion from the standard Java classes that are usually stored in *jar*-files. A file in dex-format is smaller in size than a corresponding jar-file (Gunasekera 2012). Both dex- and jar-files are vulnerable to reverse engineering, which is described more in section 3.3.

When running an Android application, the *Dalvik Virtual Machine - DVM* - executes the dex-file. The purpose of using the DVM in Android is to make applications runnable on different platforms and hardware with limited resources, such as smartphones (Gunasekera 2012).

The DVM applies *sandboxing* when executing applications, by running each application in its own instance of the DVM. Sandboxing is the concept of restricting the permissions of an application and the possible damage it could cause to the system. Applications are running separately and need permissions to interact (Six 2012).

Secure distribution of an Android application

While applications for Android can be distributed in many ways, the official way is through Android's own application store, *Google Play* (Android Developers 2014j). To have an application distributed through Google Play, the developer needs to sign the application with a certificate. The certificate is also necessary to install an application, but during development applications are signed with a test certificate (Android Developers 2014k).

The certificate is used to verify the identity of the developer and to prevent an attacker from replacing an application under the developer's name. For an application to be updated through Google Play, it needs to be signed with same certificate as previous versions (Android Developers 2014k).

2.4. Communication between application and server

This project uses a web server for the Android application to communicate with. The server functions as a hub for the social network, and all interactions between users of the application go through the web server.

The application of this project communicates with the server over *HTTPS*, which is an extension of *Hypertext Transfer Protocol - HTTP* (Snyder, Myer & Southwell 2010). When sending requests to the server from the application, extra data is brought in key-value pairs inside the request message. This method is called POST (Network Working Group 1999).

Handling requests on a web server

The server of this project runs on a *virtual private server - VPS -* provided by *CityCloud*. A VPS runs on a virtual machine and the customer has root access to the operating system running on it.

It is likely for a web server to be a highly valuable target for attackers, since it usually stores both data and business logic of a software product. Linux web servers are mainly secured by managing permissions, to make sure the right users have the right access (Snyder, Myer & Southwell 2010). For more on how the web server is secured in this project, see section 5.2.

This project uses *Apache* as web server software and *PHP* as scripting language to create responses dynamically (The PHP Group 2014). Files containing PHP code ends with *php*, which tells Apache to process the content before returning the file to the client. In this thesis, the term *server application* refers to the PHP code and its connection to the server database.

Responses from the server are, except for images, sent back to the mobile application as *JSON*. JSON stands for *JavaScript Object Notation,* and is a data format based on key-value pairs, created to be easily readable by both humans and computers (JSON.org 2014).

Keeping connections alive with sessions and cookies

When a client initialises a connection to a web server, a *session* is started. A session is a time frame during which the server will remember the connection and be able to save and retrieve information in *session variables*, which persist throughout the session. Sessions exist and run independently from each other, and one session does not have access to the session variables of another session (Adobe 2014).

Cookies are used to remember data, after a session has expired. A cookie is a file stored client-side and contains information that is being sent to the server along with requests (Kurose & Ross 2014). An example of using cookies is when a user visits a website after several days, but does not have to login. In this case, the cookie may have saved the credentials necessary to login again.

2.5. Notifying the application through Google Cloud Messaging

To notify the application from the server without having the application constantly fetching data, *Google Cloud Messaging - GCM -* is used. Provided by Google, GCM allows the server to send messages directly to specific devices (Android Developers 2014l). This project uses GCM for *send-to-sync* messages, meaning that when the application is notified, it connects to the server to fetch new data (Android Developers 2014m). This way, no sensitive data is sent through GCM but only different codes, which only this application understands, along with an internal id for a user. For more on the implementation of GCM, see section 5.3.

When a message is sent through GCM, it is received on the device as a broadcast intent. As described in section 2.3, broadcast intents can be read by several applications. It is possible to prevent this by adding a special permission in the application manifest. The name of the special permission is a concatenation of the application package name and *.permission.C2D_MESSAGE* (Android Developers 2014n).

3. Security Background

The product developed in this project is vulnerable on different levels, as described in section 1.3. Like many other applications, it is not acting only on the device, but heavily relies on a server backend. Data should therefore be protected on the phone, on the web server and when travelling between these two parts as described in **Figure 1**.

3.1. Regarding data as sensitive

Data can be separated into two different categories, which are *personal data* and *sensitive data*. Personal data is data that would not harm the user emotionally or physically, if it would be leaked. Sensitive data, on the other hand, is data that should never be available for an untrusted third party (Gunasekera 2012). For more on selecting sensitive data in this project, see section 5.1.

3.2. Protecting data with cryptography

Cryptography is the process of securing data and making it unreadable for a third party (Rivest 1990). Unencrypted data is referred to as *plaintext*. The plaintext is encrypted into *ciphertext*, which must be decrypted for the message to be recovered.

When encrypting or decrypting data, a *key* is used. A key is a string, which only the communicating parties should have access to (Rivest 1990). Some algorithms use the same key for encryption and decryption, called *symmetric algorithms.* Algorithms using one key for encryption and another key for decryption are called *public-key algorithms* (Snyder, Myer & Southwell 2010).

Advanced Encryption Standard for symmetric encryption

The *Advanced Encryption Algorithm - AES -* is a symmetric algorithm. It was created through a competition arranged by the National Institute of Standards and Technology and officially adopted as a standard in 2001. Originally created for 128-bit keys, it was later changed to support up to 256-bit keys, which is about 40% slower (Schneier & Whiting 2000). The National Security Agency uses it for encryption, which together with high performance has led to widespread popularity of AES (Snyder, Myer & Southwell 2010). AES is sometimes referred to as *Rijndael*. In this project, AES is used for encryption, both on the server and on the Android application. It is also used when encrypting and decrypting data in transit using SSL/TLS.

Public key encryption with RSA

While symmetric algorithms use the same key for both encryption and decryption, an *asymmetric algorithm* or *public-key algorithm* uses different keys. A public key is used to encrypt the message, and a private key to decrypt. The public key is not sensitive, so everyone can encrypt a message but only with the private key can a message be decrypted (Jonasson & Lemurell 2004).

The public-key algorithm RSA was developed in 1977 (Snyder, Myer & Southwell 2010) and is today one of the most popular encryption algorithms for commercial use (Jonasson & Lemurell 2004). RSA is used when establishing a secure connection using SSL/TLS.

Different modes for encrypting blocks of data

When encrypting data with algorithms such as AES, the plaintext is encrypted in blocks of data. This can be done in many different ways, known as *modes*. The easiest way is to encrypt each block with the encryption key separately, using the *Electronic Code Book* - *ECB* - mode, which is visualised in the upper part of **Figure 4**. There is, however, a safety drawback using this method, since two identical blocks of plaintext would result in two identical blocks of ciphertext, making it easier for an attacker to decrypt certain parts of ciphertext (Gunasekera 2012).

Another mode is the *Cipher Block Chaining* mode, or *CBC.* When using CBC, every block of plaintext is XORd with the last block of ciphertext, with the first block being XORd with an *initialisation vector*. The initialisation vector is a random string with the same length as a block of ciphertext, which works as an extra encryption key. This way, identical blocks will not look the same since every block depends on the last one (Gunasekera 2012), as visualised in the lower part of **Figure 4**. This project uses CBC when encrypting and decrypting the data stored in the project databases, on the server and on the device.

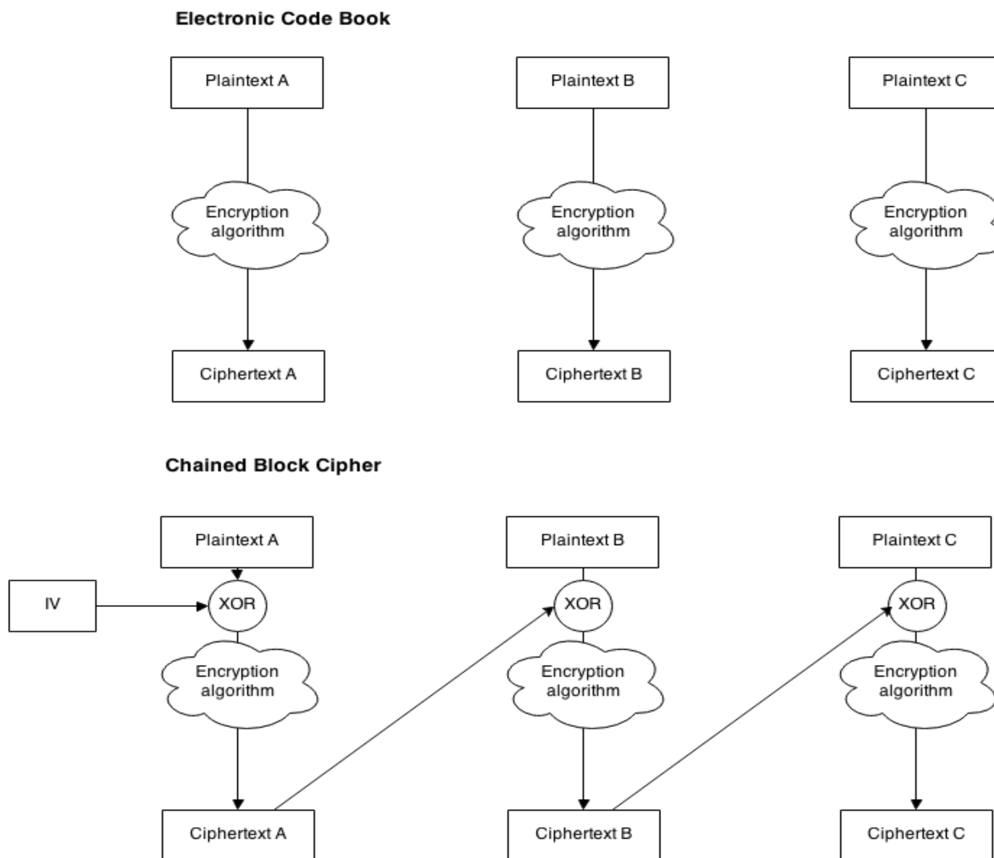


Figure 4: When using ECB, blocks are encrypted independently. Two identical blocks of plaintext would become two identical blocks of ciphertext. When using CBC, each block of ciphertext will depend on the previous block. This figure is based on two figures from the book Android Apps Security by Sheran Gunasekera (Gunasekera 2012).

Using hashing and salt to protect passwords

A technique similar to encryption is hashing. The first difference is that encryption uses a key, while hashing uses the plaintext string itself to convert it into ciphertext. The second difference is that encrypted messages are reversible while a hashed message cannot be recovered (Snyder, Myer & Southwell 2010).

This project uses the *Secure Hashing Algorithm 1 - SHA-1* - to hash passwords. Even though this algorithm has been considered weaker than newer versions of SHA, SHA-1 is still regarded as secure. In addition to that, public confidence in the implementations of the newer algorithms is still not as high as the implementations of SHA-1, even though the old algorithm itself is considered weaker (Snyder, Myer & Southwell 2010).

A drawback from using just hashing is that two identical passwords would look the same in the database, even when hashed. This is solved by concatenating a salt with the password before hashing. A salt can be any string, often a timestamp, which should be different between users. Two identical passwords with two different salts do not look the same at all when hashed (Snyder, Myer & Southwell 2010). This way it becomes more difficult to use common passwords to figure out the passwords of the users.

3.3. Vulnerabilities of an Android application

There are several ways to steal data from an Android application. Besides stealing data from the data storage of an application, an attacker can manipulate an application to share data or deceive users into sharing data without knowing it. This section gives a brief introduction to the various threats, against which this project's application is designed to protect.

Reverse engineering and tampering

The concept of *reverse engineering* in this context refers to the process in which the internal structure of an Android application is revealed, such as the source code (Manjunath 2011). This is a threat in several ways. The first one being intellectual property theft, since source code can contain important algorithms, which the creators do not want released publicly.

Another threat is that an attacker may release a *tampered* version of an application. Users could be deceived to download the modified application, believing it is the original one. For more on how reverse engineering and tampering are prevented in this project, see section 5.3.

An application has a lot of information about communications with a server backend. In a third scenario, a reverse engineered application could help an attacker to get knowledge about the server. Reverse engineering is therefore not only about protecting the application, but the whole software system as well.

Stealing touch events through tapjacking

Tapjacking is a method where an application deceives the user into triggering an action without the user knowing it, by forwarding touch events (Niemietz & Schwenk 2012). In 2011, Ken Johnson demonstrated an Android implementation of this idea (Johnson 2011). Tapjacking can be done by tampering an application or through another

application.

An attack based on tapjacking is carried out by drawing a layer on top of an existing user interface (Niemietz & Schwenk 2012). In the case of Ken Johnson implementation an expanded full screen toast was drawn on the top stack (Johnson 2011). As described in section 2.3, there is maximum time for displaying toast messages. This could however be altered and worked around by, for example, displaying the toast multiple times as described by Wei (Wei 2010).

The result is that a user interface can be drawn on top of another, as shown in **Figure 5**. This could deceive the user into triggering touch events that the user remains unaware of, as described by Richardson (Richardson 2010). One of the dangers with tapjacking is that the malicious application does not need any special permission granted by the user (Qiu 2012).

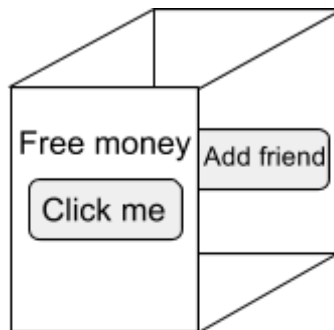


Figure 5: This figure illustrates the concept behind tapjacking on Android. An activity is displayed in front of another activity and lets touch events pass through to the underlying application while the user remains unnoticed. The figure is based on another figure originally created by Niemietz & Schwenk (Niemietz & Schwenk 2012).

Risks when using a rooted device

Another security risk in Android is running the device as the root user. If this is the case, the device is referred to as *rooted* since this is not the default state in Android. As described in section 2.1, the root user of a system has access to everything in Unix-based systems. Therefore, Android does not prevent a root user from getting access and modifying the operating system, the underlying kernel or any other application (Android Developers 2014o). For more about how this application handles rooted devices, see section 5.3.

3.4. Attacking a database through SQL injection

SQL injection is when an attacker manages to run SQL code in the database by manipulating the input to the database (Snyder, Myer & Southwell 2010). Since SQL queries are separated by semi-colon, and strings are surrounded by quotes, an attacker can use this syntax to end a query and insert a new one after, like in the following example:

1. Assume that the server is supposed to run the following SQL query after the user fills in the login form in an application: `SELECT * FROM users WHERE email = 'email@fromuser.com';`
2. The user enters the following input instead of an email address: `1'; DELETE FROM users; SELECT * FROM users WHERE email = '`
3. That would result in the following query executed: `SELECT * FROM users WHERE email = '1'; DELETE FROM users; SELECT * FROM users WHERE email = '';`
4. Which would result in deleting the entire user table.

There are many ways of preventing SQL injection. The method used on the server in this project is to escape strings by adding a backslash before quotes, so that quotes remain but are interpreted by SQL as a part of a string (Snyder, Myer & Southwell 2010). To protect against SQL injection on the Android application, standard helper methods such as *query()* are used to build queries.

3.5. Data in transit is vulnerable

Data sent between the application and the server is vulnerable to someone stealing the data through a Man-in-the-Middle attack. This is solved by letting a third party verify the server, and encrypting all the data sent between the communicating parties.

Stealing data with a Man-in-the-Middle attack

A *Man-in-the-Middle* attack is where an attacker is able to intercept information from both a client and a server (Gunasekera 2012). This can be done because a server can pretend to represent a URL of another server. For example, it is possible for a server to pretend that it is the Google server. During a Man-in-the-Middle attack, the attacker is able to listen to and manipulate the data in transit.

Securing connections with SSL/TLS

To prevent Man-in-the-Middle attacks, *Transport Layer Security - TLS* - and its predecessor, *Secure Sockets Layer - SSL* - are two protocols used to keep a connection secured between a client and a server (Snyder, Myer & Southwell 2010). The purpose is to make sure that the client communicates with the right server, and that a third party cannot read the data transmitted. When SSL/TLS is used, the data is transmitted using the HTTPS protocol as mentioned in section 2.4.

A server using SSL/TLS must be verified by a third party (Snyder, Myer & Southwell 2010). This third party is a *certificate authority*, which must be trusted by the client. The server confirms that it is trusted by having a server certificate signed by a certificate authority. For example, the certificate authority RapidSSL is trusted by all major browsers (RapidSSL 2014), and is the certificate authority used for issuing a certificate in this project.

The most widely used implementation of SSL/TLS is *OpenSSL*. Netcraft reports that 66% uses the web server, in a recent report from April 2014 (Netcraft 2014). OpenSSL is used in this project to enable a secure connection with Apache. For more on how SSL/TLS is configured on the server, see section 5.2.

During the development of this project, a vulnerability called *the Heartbleed bug* was found in the OpenSSL library. The bug allows an attacker to steal data from a server using the TLS Heartbeat extension (Heartbleed.com 2014). The purpose of Heartbeat is to keep a connection alive when no data is transmitted (IETF 2012). The bug was solved immediately by the OpenSSL team (The OpenSSL Project 2014).

4. Method

This project was carried out by building the Android application and the server backend simultaneously. In the initial phase of the project, basic functionality was developed without any security measures. The reason for this was to make sure the product worked before making development more complicated by adding security features throughout the system.

Security measures were later added with respect to the different security scenarios described in section 1.3. As for functionality, these measures were implemented simultaneously on the Android application and the server.

By the end of the project, the security measures were tested and evaluated with respect to the security goals of this project. The results of these tests are described in Chapter 6 and discussed in Chapter 7. The tests were done by using several tools, described in section 4.4. The tools, libraries and services described in this chapter are listed in Appendix B.

4.1. Literature and research

The research was mainly carried out by studying literature, papers and various blogs on the Internet. Especially when researching some of the security threats to Android, blogs have been useful. The greatest part of information regarding Android development was gathered from the official documentation.

4.2. Configuring and developing the server application

The server application was built on the local computers of the developers, and then uploaded to the server. For writing PHP code, the text editor *Sublime Text* was used. *MySQL Workbench* was used for working with the server database.

Two libraries are used on the server. Firstly, the library *mcrypt* is used for encrypting and decrypting the server database. Secondly, when sending data through GCM, the server uses the *Client URL* library, or *cURL,* to make HTTP requests to the GCM servers.

4.3. Developing the Android application

The development of the Android application was done through the, by Google recommended, combination of the Eclipse *integrated development environment - IDE* - and the *Android Developer Tools - ADT* - plugin (Android Developers 2014p). For testing purposes and debugging, both real devices and emulators were used.

The data returned from the server is, as described in 2.4, formatted as JSON. While it is possible to parse this format with functionality provided by the Android API, the library *Jackson* was chosen to provide a larger flexibility in parsing.

To support the use of Google Maps, the application uses the official library *Google Play Services*. This allows the application to use a Google Maps view of user locations.

The library *SQLCipher* is used for encrypting and decrypting data stored in the device database. The classes of the library are inheriting the standard SQLite classes of Android. SQLCipher's classes can be used exactly like the SQLite classes, except that an encryption key is needed. SQLCipher uses the AES algorithm and CBC mode, which are described more in section 3.2.

4.4. Testing the security measures

By the end of the project, the security measures were tested in order to evaluate the success of the project. Some tests required the use of external tools.

The first tool used, when testing reverse engineering, was *dex2jar*, which converts the application's executable dex-file to a jar-file. As described in section 2.3, the dex-file is created by converting the jar-file when the project is built. The reason for converting it back during reverse engineering is because it is possible to convert jar-files back to Java code.

Another tool used, when testing to reverse engineer the application, was *JD-GUI*, which is a Java decompiler. JD-GUI also has a navigator, making it possible for the user to navigate through the source code of a decompiled jar-file. This way, the result of obfuscation can be tested since the source code will be visible.

When attempting to tamper the application, *apktool* was also used. This tool can be used for both reverse engineering the application, and building a modified application project back to an installable apk-file.

To test the secure connection between the application and the server, two different tools were used. Firstly, the Android application *tPacketCapture* was used to save down captured data traffic to a file. This was later analysed on a computer with the tool *Wireshark*, to see if captured data was encrypted or not.

When evaluating the encryption of the application database and SharedPreferences, the Android SDK command line tool *Android Debug Bridge - adb* - was used to transfer the data to the computer. *Visual Studio* was used to look at the database content and *Atom* was used for the content of SharedPreferences.

4.5. Other tools used in this project

For this project, the tool *Git* have been chosen as version control system. Git was initially developed by Linus Torvalds to suit the need of managing the growing Linux kernel (Somasundaram 2013). Github was used to host the git repositories of this project. Since the source code of the project should not be visible to third parties, choice of host is a security matter. As mentioned in section 1.7, Github is a trusted party.

5. Architecture and implementation

The product is divided into two parts, the Android application and the server backend. The Android application works as a frontend for user interaction, and is the only thing that users will experience as the product. All communication between instances of the application goes through the server backend.

5.1. Sensitive data of this product

As described in section 3.1, data is sensitive only if it should never be available for an untrusted third party. This project uses a different approach in selecting sensitive data. All data is regarded as sensitive, unless it fulfils two conditions:

1. The data can be accessed by anyone through using the product.
2. It would be impossible to scale the system while regarding that data as sensitive. This can be the case in scenarios where a large amount of data has to be decrypted, which is performance heavy.

As a result of this, id of the user, username, email, and profile pictures is the only data that is regarded as non-sensitive. Firstly, username and email are searchable through the application search functionality. Secondly, having these attributes encrypted would require a decryption of all users before search could begin. That would not be feasible as the number of users grows. While the path to profile pictures is encrypted, the files are not.

The id of a user is never visible when using the application, but works as an internal id in the logics of the application and the server. When notifying applications through GCM, the user id is sent as data. The reason for this is for the application to fetch the correct data. The user id cannot be used for anything harmful unless the database is compromised and decrypted. In that scenario, all data would already be revealed.

5.2. Overview of the server application

The main purpose of the server application is to organise the data of the social network. This is done by storing the data in a MySQL database, and by providing several functions for the Android application to use by communicating with the server over HTTPS. The functions are divided as different pages, all having a distinct purpose. The structure of the server application is visualised in **Figure 6**.

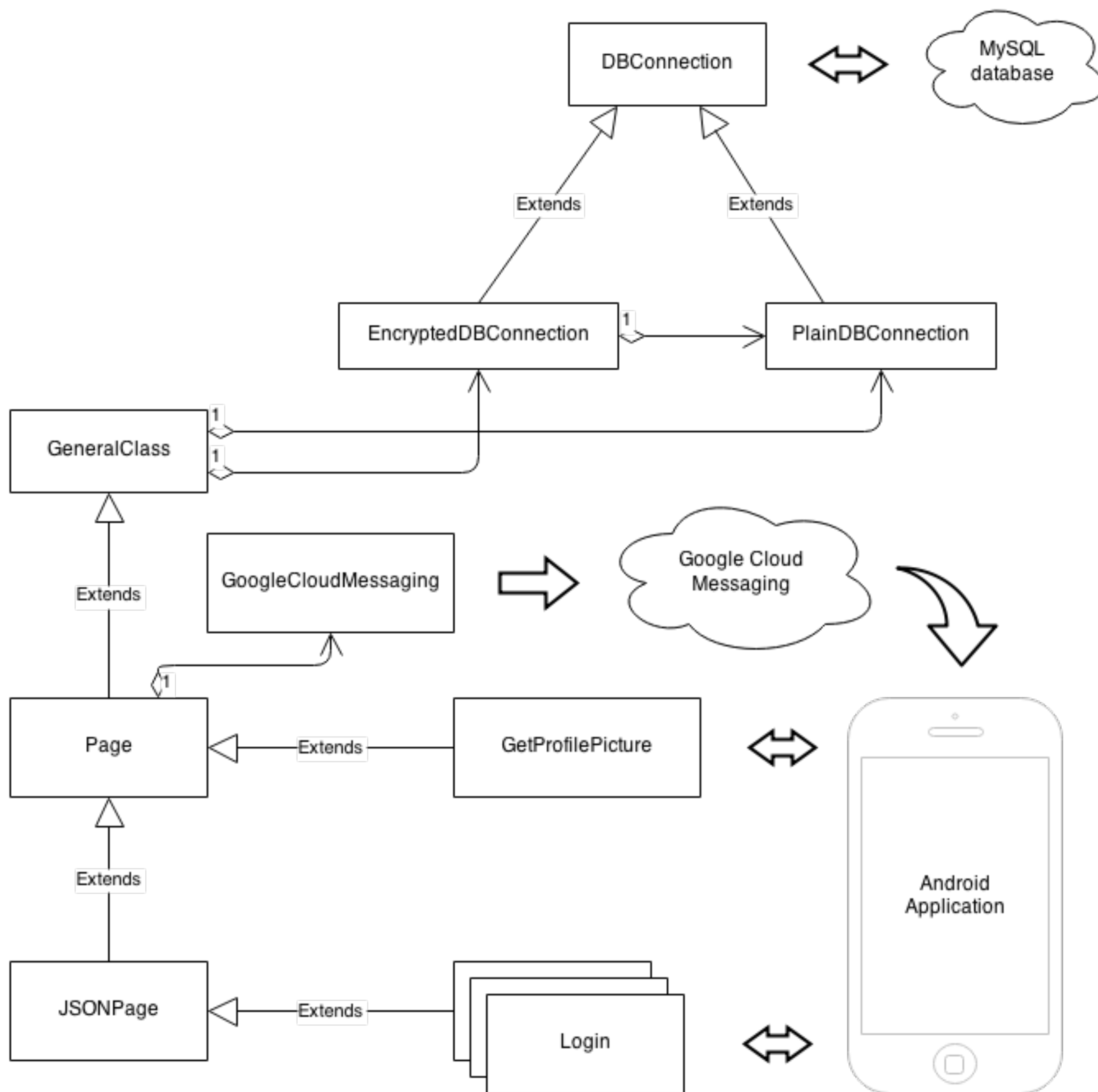


Figure 6: The structure of the server application. GeneralClass connects the database classes with the Page class and its subclasses, being the core class of the server application.

Structure of the server code

To prevent an attacker who succeeds to steal the source code from stealing the encryption keys, they are not located in the server code. Therefore, passwords and API keys are located in Apache environmental variables and are redefined as PHP constants in *globals.php.* The Apache environmental variables are stored in the *httpd.conf* file. This way, it would also be easier to setup a separate instance of the server application without having to think about the secrecy of the keys when copying the source code. For more about the permissions on the server, see section 5.2.

The database connection goes through *DBConnection,* which is an abstract class. In order to make it easy to build SQL queries, methods are there to help the developer. Protection against SQL injections is handled in this class, by escaping all values before queries are built. By building all queries using this class, there is minimal risk that SQL queries are used without being sanitised.

DBConnection has two subclasses with the first one being *PlainDBConnection*. All data sent to and from the database is in plaintext when using this class. This is used for the plaintext version of the user table, when searching for users using the MySQL *LIKE* operator, which is used for pattern matching in strings. No data in this table is regarded as sensitive, as described in section 5.1.

The second subclass is *EncryptedDBConnection*. This class handles manipulation of encrypted tables. Data is automatically encrypted with 128-bit AES when inserted into the database and automatically decrypted when fetched, so outside the instance of this class, nothing extra has to be done. Encryption and decryption is handled in the *SOANAPEncryption* class.

These two classes work exactly the same on the outside so there is no difference in using them. EncryptedDBConnection has a function to sync an encrypted table with a plaintext table. This is used to sync the ciphertext table *users* with the plaintext table *users_plain*, when the new user is added.

GeneralClass is used to provide its subclasses with instances of the database classes, along with other functionality used throughout the server application. For example, GeneralClass provides functionality to check the friendship between two users to make user only friends can access location data.

The class *Page* inherits GeneralClass and is located in the *pages* directory. The Page class handles things like checking if a user is logged in, saving request data and loading users based on either email or id. The Page class also has an association to the class *GoogleCloudMessaging*, which is used to notify the application. The reason for this is, as mentioned in section 2.5, to avoid the Android application from having to fetch data from the server all the time.

JSONPage inherits Page and deals with requests when the response will be a JSON object. Every JSON object has a status and message variable, and then a number of specific variables to the response. The JSONPage class also defines a number of constants used in JSON responses and GCM codes, which are used by the subclasses.

Securing the server environment with permissions

To protect the server code, permissions of the source directory are restricted. As described in section 2.1, each process is its own user in a Unix-based system. The Apache server's user belongs to the group *www-data* and has the rights to read write and execute the files in the directory *social-android-server*. The directory is inaccessible for other users than these and the owner of the directory, which is the root user. This is to prevent unauthorised users to access the source code.

Passwords and encryption keys are located in environmental variables of the Apache server, allowing only the root user to access and Apache to read. If an attacker would get access to the source code of the server application, it does not have to mean that the database is compromised since the keys are not in the source code. All that the attacker would find is a lot of ciphertext, apart from the *users_plain* table where the data is retrievable by search anyway.

As an extra security measure, it is not possible to browse the server directories via a web browser. By default, Apache allows directories and files to be listed, but this has been changed in the Apache configuration file *httpd.conf,* for all directories and subdirectories belonging to this project's server application. This is to prevent a potential attacker from using the browser to get a clear overview of the server application.

Securing user data with MySQL permissions

The accounts available for accessing the database are *root* and *soanap_user.* To protect the database, anonymous accounts are removed. Anonymous accounts could be used to connect to the database, and in some systems even allow the user to manage databases (Snyder, Myer & Southwell 2010). The root user is only used for the administration of the database and has a different password than the root user of the server. This is to prevent anyone breaching into the server from automatically having full access to the database management system.

To work with the database, Apache connects to the database using soanap_user as programmed in the server code. To prevent a hijacked account from causing too much damage, this user only has the permissions to perform SELECT, INSERT, UPDATE and DELETE-queries. For more information on these SQL queries, see section 2.2.

The reasons for not using the MySQL root user when connecting to the database, and instead a user with restricted permissions, are several. Firstly, if soanap_user would be hijacked, there would be nothing in the way of protecting the database or other databases on the same server. Despite the permissions being enough to delete all entries in the database, a backup database for example would still be protected. Secondly, the root user can create other users and manage privileges. This way, an attacker would be able to create an account and use it without an administrator knowing about it.

Configuring SSL/TLS to secure data in transit

This project uses a server certificate from RapidSSL[1] to enable a secure connection between the Android application and the server. This way, the application will trust only the correct server and the data in transit will be encrypted. The certificate uses RSA with 2048-bit keys for key exchange and AES with a 128-bit key for encrypting and decrypting the data in transit.

OpenSSL is used on the server to support the secure connection. As described in section 3.5, the Heartbleed bug became publicly known during the development of this project. Therefore, the server's instance of OpenSSL was upgraded to version 1.0.1g, which contains a fix for that bug.

[1] RapidSSL is a certificate authority that issues server certificates at a relatively low price. RapidSSL is a trademark of GeoTrust, one of the world's leaders in digital certificates (GeoTrust 2014).

To make Apache work with SSL/TLS, the Apache module *mod_ssl* has to be enabled. Configurations are also made in the *default-ssl* file in the */etc/apache2/sites-available/* directory. These are done so that Apache will find the certificate and the private key. When Apache is started, the passphrase for the certificate has to be entered since the passphrase is not saved anywhere on the server.

5.3. Functionality and security of the Android application

The Android application is built to fulfil the functionality- and security goals of this project. The application uses an encrypted database and encrypted SharedPreferences to protect user data, along with several security measures against various threats. External APIs such as Google Maps and Google Places are used for location sharing functionality.

Build target for the application is set at the latest current release, version 19. Minimum API level to run the applications is 15. This was chosen to maintain compatibility with a large user base, without losing the ability to use modern security features in the later versions of the API (Android Developers 2014q).

Asynchronous communication with the server

All classes for the communication with the server are located in the package *com.safeandsocial.networkutil*. For sending data to the server through POST requests, the class *PostToServer* is used. The class *ServerResponse* is used by PostToServer upon receiving the response from the server, and allows the response to be handled without temporarily freezing the user interface.

The JSON responses from the server are parsed by subclasses of the *AbstractParser* class, which uses the Jackson library. Each subclass represents one type of response. For example, there is one subclass parsing the response from *login.php*, while another for parsing the response from *getFriendRequests.php*.

Receiving notifications through Google Cloud Messaging

The push notifications from GCM are received by a wakeful broadcast receiver. If the application is running, the data is fetched immediately. If the application is in standby, the user has to enter a pin code when entering the application, before starting to fetch the data. The usage of the pin code is described in the next section. All data fetched is inserted in the application database and all views are updated.

Protecting data and encryption keys using a pin code

The Android application uses two SQLite databases, one for sensitive data and one for data necessary for the application to receive GCM notifications. The database for sensitive data and parts of SharedPreferences are encrypted using AES with a 256-bit key, using CBC mode. The encryption and decryption of one of the databases is handled by SQLCipher, a package of extensions of the standard Android classes for SQLite databases.

The application uses the series identifier, provided by the server when logging in as further described in section 5.4, as an encryption key for the database. This means that the key is different every time a user is logged in with email and password, and that the database is removed if a user logs out. This minimises the potential damage if the key should be stolen.

During the time a user is logged in, the database encryption key is stored in SharedPreferences. The encryption key for SharedPreferences is derived from a pin code, which the user enters when starting the application. The pin code has to be at least 4 digits.

Preventing tapjacking with view settings
The application developed during this project protects against tapjacking, which is described in section 3.3. This is done by setting the Boolean variable *setFilterTouchesWhenObscured* to true on all views in the application (Android Developers 2014g). The result is that touch events are discarded if the view is obscured by any view on top of it, and protects the user from triggering actions in this application without knowing it.

Implementing tamper detection
As described in section 3.3, it is possible to reverse engineer an application and release a tampered version. This could lead to users having a malicious version of the application without knowing about it. This section focuses on how prevent tampering by certain checks, when starting the application. For information on how reverse engineering is prevented, see section 5.3. In case of detecting tampering, the user is notified and the application terminates.

Three different safeguards are implemented to protect the user from running a tampered version of the application. The first one is a check to see if the application is installed from Google Play. This is done by using the standard Android class *PackageManager* to compare the installing application to Google Play.

The second safeguard is to validate the signature of the binary using checksums. A string in the application represents a valid signature, and is compared to a signature calculated using the binary. If the binary is changed, the signatures will not match and the application will not start.

The third way is to prevent an attacker from debugging the application, thus making it more difficult to tamper it. This is done by checking the debug flag during launch. The debug flag is an attribute set in the AndroidManifest file. When this attribute is set to true, it is possible to debug the application with external software, thus making it easier to tamper the application.

Warning dialog when running on rooted device

If rooted device is detected during launch, a message will appear to warn the users of the potential security risk of being rooted. The danger of rooted devices is that other applications may get access to the data storage of other applications. A user with a rooted device can still use the application, after closing the warning dialog. To determine if the device is rooted, the application runs different tests to determine the root status of the device.

The first check is to check the Android system build keys. Two different keys exist, either *release keys* or *test keys*. Official Android releases are released with signed release keys, so if these are found, the device is most likely not rooted. The opposite holds true if test keys are found - the build is a custom build and is most likely rooted.

Another action to check if the device is rooted is to look for applications and binaries associated with rooting the device, which can be used to manage root access for other applications. An example is the binary *su*, which is a Unix command for getting root user privileges.

Obfuscating and optimising the application with ProGuard

Reverse engineering is the first step of tampering and could lead to theft of intellectual property. To protect against reverse engineering, this project uses the tool *ProGuard.* ProGuard is a generic optimiser and obfuscator for Java bytecode (Saikoa 2014). ProGuard is distributed along with the Android SDK and is also integrated into the process of building Android applications (Saikoa 2014; Android Developers 2014r). The operations that can be done to the application are shrinking, optimisation, obfuscation, and preverification (ProGuard 2014a; Lafortune 2013). All these steps are optional and mutually exclusive. Preverification has been turned off, since preverification is irrelevant for the dex compiler and the DVM (ProGuard 2014b).

In this first step, shrinking*,* the ProGuard analyses the code recursively. Unused code as in classes and members will be flagged and discarded after this step. A successful shrinking could also result in an apk-file that is 5-70% smaller in size (Lafortune 2013).

During optimisation of the application, there are several methods that can be used (Lafortune 2013). One optimisation method, *peephole optimisation,* rewrites instructions in more optimised ways (Tanenbaum, van Steveren & Stevenson 1982). A typical peephole optimisation is replacing times two with the signed left shift operator. Another method, *method inlining,* replaces method calls by the body of the method. This removes the overhead of calling a method (IBM 2006).

During installation of an application on an Android device, the DVM also runs some optimisations on the dex-file (Bornstein 2008). Some of these optimisations might already been done by the ProGuard, and the installation might be completed sooner. As Dalvik performs some of its own optimisations, some improvements made by ProGuard might be incompatible with Dalvik (Norbye 2012). To avoid some of these known incompatibilities, some small range of ProGuard optimisations have been explicitly turned off (ProGuard 2014b).

Obfuscation has the purpose of making reverse engineered source code practically unreadable for humans (ProGuard 2014c). This is done without destroying the functionality of the source code, meaning that it is still executable when compiled. However, it would require more time for a human to understand the code.

During obfuscation, most classes and methods will be renamed with semantically obscure names. These names will be combinations of letters without any connection to the implementation of the class or method (ProGuard 2014c). For example, classes may end up with names such as *aa, ab, ac* etc. Renaming is not applied to classes and methods, for which the name is necessary for the execution of the application, such as the *main* method. Other actions taken during obfuscation are to remove all logs and merge all packages into one. This makes it more difficult to make sense of a reverse engineered application, since the structure is lost.

5.4. Product functionality

The functionality of the product depends on both ends of the system. As the application is an interface to the server backend, all interactions between users go through the server. It is the responsibility of the server to provide the right data to the application, and how that is done is described in the following sections.

Registering as a new user

Users register themselves by entering user information, password, pin code, and an optional profile picture into the application. The application performs a check so that the user enters a password of at least 3 digits, at least one special character, 6-40 letters with at least one uppercase letter and at least one lowercase letter. The pin code is validated as well.

When saving the new user, a salt and hashed password are generated, and saved together in the table *users* along with username, email and password in the server database. If the user provides a profile picture, the path to the uploaded image file is stored in the table *profile_pictures*. The server signs the user in, and returns the user id, a token and a series for persistent login. For more about persistent login, see next section.

Profile pictures are stored as files in the *profile_picture_files* directory at the server. They are not accessible through the browser by entering the path. This was configured in the Apache configuration file. In order to see a profile picture, the user must be signed in and use the *getProfilePicture.php* file.

Signing in using different methods

Login can be done either by signing in by standard login or persistent login. The first type is for signing in when not having used the application for a while. The second type is when the server session is finished, but a valid token is still stored for the user. In this way the user does not have to enter username and password every time the session expires, but instead is signed in again automatically.

When the server handles standard login, the entered password is concatenated with the salt stored for the user in the server database. The concatenated string is then hashed using the SHA-1 algorithm. The stored hashed password is compared to the newly hashed password, and if they are equal, the user is signed in. For more on hashing and salting, see section 3.2.

Persistent login is used when the server session expires. The application sends the user id, series identifier, and token to the server, which are all retrieved during standard login. If all three variables are correct according to the entry in the *tokens* table, the user is signed in. If only the user id and series are correct, but not the token, the user notified about a security breach in the application and the token is removed. The user is then required to sign in again (Jaspan 2006).

The reason for having this kind of persistent login is to minimise the damage of stolen credentials. Even if stolen credentials can help an attacker to login, it would only be possible until the ordinary user tries to use the credentials. The series identifier helps the server to see the difference between a token theft and a random login attempt (Jaspan 2006). **Figure 7** explains a scenario where is technique increases security.

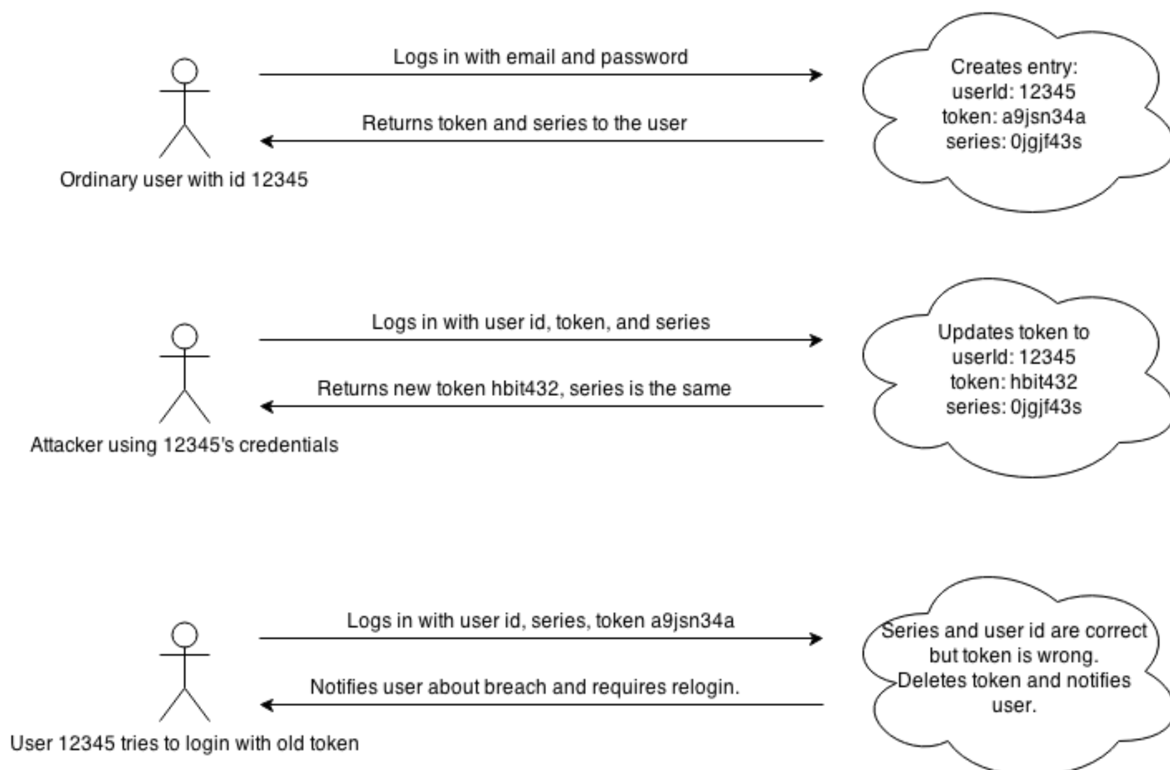


Figure 7: An example scenario on what may happen when credentials for persistent login are stolen. This concept of handling persistent login was written about by Barry Jaspan (Jaspan 2006).

When a user is signed in, a 30-minute session starts. A timestamp is stored in a session variable at the server and every time a request is made, the server checks the timestamp. If the last request was made less than 30 minutes ago, the timestamp is renewed. If it was made more than 30 minutes ago, the timestamp is removed and the application is required to login again.

Connecting with friends

In order to socialise, users must be friends with each other through the social network. The first step is to search for users by using email or username, so that a user can be added. This is done with the MySQL LIKE operator on the table *users_plain*, where data is stored about users in plaintext. As described in section 5.1, username and email are not considered to be sensitive.

When adding a friend, the server notifies the other user through GCM and saves the friend request in the server database. The only data sent through GCM is the code for fetching friend requests along with the id of the requesting user. If the added user accepts the request, the requester is notified through GCM as well and the friendship is stored in the database. In this case, the database id of the user is also sent but is not regarded as sensitive, as described in section 5.1. If the request is ignored, no notification is sent. GCM allows the application to just wait for notifications without having to fetch data all the time.

Socialising using locations

The core of the application is the *socialise* functionality, which allows users to update and share their location status with their friends. Information about the location is fetched from the *Google Places API.* Many locations have several places for the user to choose from, for example a user standing close to many restaurants. This makes the application more realistic, since people do not think of locations in terms of geographical coordinates.

When the place information has been fetched by the application and sent to the project server, it is saved in *user_positions* along with a server timestamp and the id of the user. The server selects all friends of the user who saved the location, and notifies them through GCM. The id of the sharing user is sent along with a code that tells the application to fetch this user's positions. As mentioned in section 5.1, user id is not regarded as sensitive. This way, the application fetches only the positions of right user to increase performance.

Commenting on shared locations

While friends and their latest shared locations can be fetched at the same time, this is not the case for comments on locations. This is because one location can have many comments, and since one person can have many friends, the number of comments to fetch would grow exponentially when a user starts using the social network. To solve this, comments are fetched only when being displayed.

Comments are only visible to the friends of the user who shared the location. Before the comments are returned, the server checks this friendship. Shared locations are not available for users who are not friends, but the check should still be performed to avoid someone circumventing the business rules of the application.

6. Testing and evaluation of security measures

In order to evaluate how the implemented security measures work, a number of tests have been done. The tests were done at the end of the project, as the application and server were finished. Each test is evaluated with respect to the security goals of this project, which focus around protecting sensitive user data.

6.1. Using reverse engineering with an obfuscated application

As described in section 5.3, ProGuard is used to obfuscate the Android application. The purpose of this is to make it harder to recover the source code. To examine how the source code looks after obfuscation, the application has been reverse engineered using a number of tools. The tools dex2jar and JD-GUI where used during this test, as mentioned in section 4.4. An example of how the source code of the reverse engineered application looks like in JD-GUI is shown to the right in **Figure 8**.

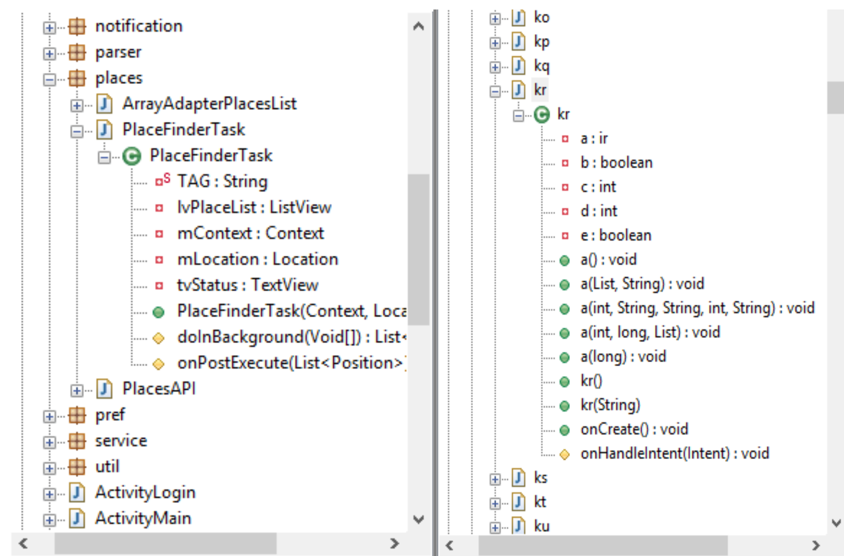


Figure 8: As displayed in the screenshot, to the right, of the output produced by the JD-GUI tool, classes have been divided and renamed along with methods and variables. The screenshot to the left presents how a non-obfuscated application looks like when reverse engineered.

As displayed in **Figure 8**, classes and methods are renamed with obscure names. Exceptions from this are classes and methods, which must remain the same for the Android application to work, such as *onCreate* and *onHandleIntent*.

6.2. Attempting to start a tampered application

The second step in preventing tampering, besides obfuscation, is a number of checks during the launch of the application, as described in section 5.3. As mentioned in section 4.4, apktool was used to reverse engineer the apk-file in this test.

When testing the tamper detection, this project's application was reverse engineered and new code was inserted. The code inserted allowed the application to read text messages sent to the device. Changes were also made to images and texts. Finally, the application's debug flag was set to true.

After modifications were made, the application package was rebuilt using apktool. As described in section 2.3, the application has to be signed with a certificate in order to be installable. Trying to install it without a valid certificate resulted in a message from the system, as shown to the left in **Figure 9**.

After signing the tampered application with a valid certificate, the application was installed and started. As described in section 5.3.5, the application will check if the certificate is the right one. Since it was not during this testing, the application refused to start.

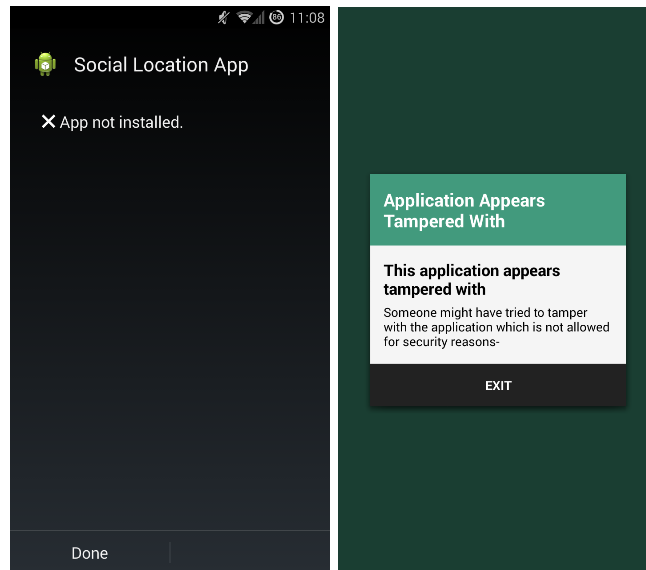


Figure 9: The left shows the result of trying to install a tampered version of the application without a valid certificate. The right shows the result of starting an application, which is signed with a different signature.

6.3. Attempting to start a the application on a rooted device

Different checks have been implemented to detect if the application is being run on a rooted device. As discussed in section 3.3, such devices expose user data for extra risk. Therefore, a warning dialog appears when the application start to warn the user, if the application runs on a rooted device. **Figure 10** shows the warning as the application was started on a rooted device.

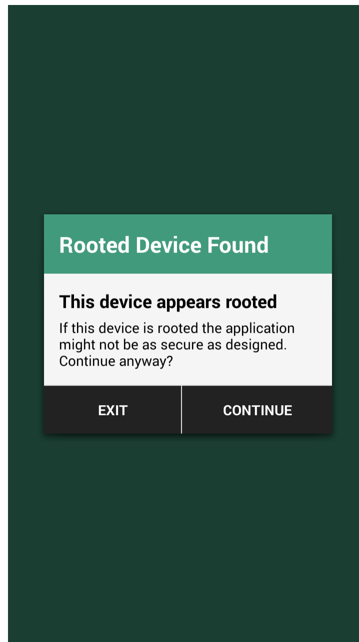


Figure 10: Showing the warning that the user receives if starting the application on a rooted device.

6.4. Reading from the encrypted application database

As described in section 5.3, one of the application databases and SharedPreferences are encrypted with AES. During this test, the data from both storages were imported using *adb*. See **Figure 11** for the result of reading an encrypted SQLite database and **Figure 12** for the data from SharedPreferences.


```
00000ca0  58 F7 55 E4 4F BF 5B BD  F5 B2 30 8C 3B BA 4D B3  X.U.O.[...0.;.M.
00000cb0  B4 8A E4 FB 82 45 FD 4E  A4 FA 04 B4 16 60 9E EC  .....E.N.....`..
00000cc0  75 FE F1 10 9D 2D 14 1C  26 A1 AF BB E1 89 0D 47  u....-..&......G
00000cd0  F2 7A 3D 38 F4 F9 9C AA  84 22 45 AE 4A 16 FA 5E  .z=8....."E.J..^
00000ce0  5E CC 6A 7E 41 A4 78 5A  1F E1 72 F7 BF 35 9F 2C  ^.j~A.xZ..r..5.,
00000cf0  E3 DF 94 BA 7F 14 F6 F8  B1 ED FB 1E 0E 86 01 D1  ................
00000d00  2F A0 E1 C9 BB CB A7 CB  F6 45 CE 31 3C 3E 52 DE  /........E.1<>R.
00000d10  85 23 EE 59 EE 4F 8C 8E  60 81 FF 26 92 00 63 25  .#.Y.O..`..&..c%
00000d20  8B D4 14 79 C0 66 10 C5  AA FA 6D E9 48 A5 FD 79  ...y.f....m.H..y
00000d30  35 34 06 DF 2D 41 1A 6C  5E 49 AF 6A 30 E7 97 54  54..-A.l^I.j0..T
```


Figure 11: Showing content from the application database after encryption, as shown using Visual Studio.


```
1  <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2  <map>
3      <string name="series">HHAPsm5nybRmnGh6UjQzcmChGbUM8b30f0PIAsAe80I=]i5eTi59lL
4      <string name="token">zPeJmfxVQU5f+dFt06ouOY6rirdzkLwttcu4FnyHM34=]GvSn07xDAw
5      <string name="userId">B9Yi4PzadHOO17py2oxMKokwiZBioBaQpqy0T3MIU0E=]bHGE8wmFh
6      <int name="app_version" value="1" />
7      <string name="reg_id">8mlRVjxRw7Ls/4IyYSvbqLWzM1ITJYNctuugVTnikRU=]1wVcCkWra
8      <string name="userName">3sbpsvCpOWrr30fjc9EF0f6UH433zKJUp6HiT+eUGR8=]cbVu1k0
9  </map>
```


Figure 12: Data from SharedPreferences. The keys are in plaintext, but the values in ciphertext, as shown Atom.

6.5. Testing the tapjacking protection

To test how the protection against tapjacking works in this application, a toast is drawn on top of the main view with a button that forwards the touch event to the socialise button, as seen in **Figure 13**.

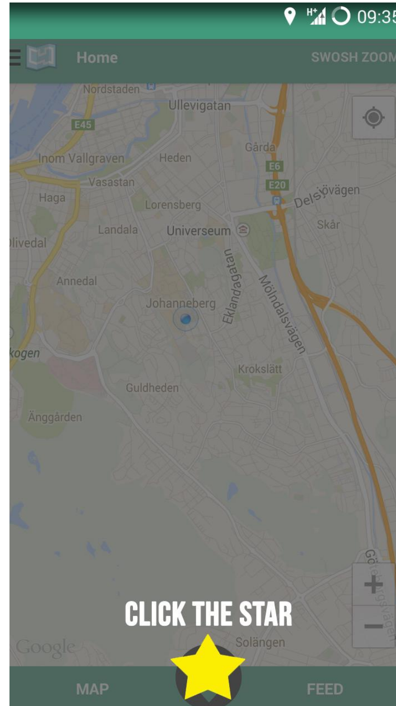


Figure 13: A toast is drawn on top of the other components and is deceiving the user into pressing the socialise button.

As described in section 5.3, the views in the application are programmed to discard touch events. As a result of this, the socialise button did not react during this test.

6.6. Listening to the data in transit

Testing the security of the data in transit was done using the Android application *tPacketCapture*, together with the data capture tool WireShark. The tPacketCapture application created a log file of the communication, which could later be analysed using WireShark. The results of capturing data with and without SSL/TLS are shown in **Figure 14** and **Figure 15**.

No.	Time	Source	Destination	Protocol	Length	Info
28	17.603016	10.8.0.1	141.255.189.134	TCP	54	40980 > http [ACK] Seq=1 Ack=1 Win=14600 Len=0
29	17.653317	10.8.0.1	141.255.189.134	HTTP	267	POST /social-android-server/pages/login.php HTTP/1.1 (application/x-www-form-urlencoded)

```
⊞ Frame 29: 267 bytes on wire (2136 bits), 267 bytes captured (2136 bits)
⊞ Ethernet II, Src: Google_00:00:01 (00:1a:11:00:00:01), Dst: Google_00:00:02 (00:1a:11:00:00:02)
⊞ Internet Protocol Version 4, Src: 10.8.0.1 (10.8.0.1), Dst: 141.255.189.134 (141.255.189.134)
⊞ Transmission Control Protocol, Src Port: 40980 (40980), Dst Port: http (80), Seq: 1, Ack: 1, Len: 213
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: application/x-www-form-urlencoded
    email=anton%40gmail.com&password=anton123
```

Figure 14: Showing a selection of the communication between the application and the server during user login. SSL/TLS was turned off and the password and email is sent in plaintext to the server, as seen in the last row of the figure.

```
No.  Time        Source           Destination      Protocol Length Info
  13 7.787944    141.255.189.134  10.8.0.1         TLSv1    101 Change Cipher Spec, Encrypted Handshake Message
  14 7.838458    10.8.0.1         141.255.189.134  TLSv1    292 Application Data
  15 7.839222    141.255.189.134  10.8.0.1         TCP      54 https > cognex-dataman [ACK] Seq=1117 Ack=609 Win=65
⊞ Frame 14: 292 bytes on wire (2336 bits), 292 bytes captured (2336 bits)
⊞ Ethernet II, Src: Google_00:00:01 (00:1a:11:00:00:01), Dst: Google_00:00:02 (00:1a:11:00:00:02)
⊞ Internet Protocol Version 4, Src: 10.8.0.1 (10.8.0.1), Dst: 141.255.189.134 (141.255.189.134)
⊞ Transmission Control Protocol, Src Port: cognex-dataman (44444), Dst Port: https (443), Seq: 371, Ack: 1117, Len: 238
⊟ Secure Sockets Layer
  ⊟ TLSv1 Record Layer: Application Data Protocol: http
      Content Type: Application Data (23)
      Version: TLS 1.0 (0x0301)
      Length: 233
      Encrypted Application Data: b43f6b08c3fdee7702f0688cb7049c641538178d4ae7b160...
```

Figure 15: An excerpt from the communication between the application and the server when using SSL/TLS. As seen in the last row, data is encrypted.

6.7. Reading from the server database

If an attacker would get access to the server database, would that not necessarily mean that the data is compromised since the data is encrypted as shown in **Figure 16**. The encryption key is stored in along with other server keys in the apache configuration files, which only the user of the Apache process and the root user may access.

id	username	email	password	salt
ð}L	Íj¥...	F1ÅÅÁÄàè¬R]...	õ™†N½ÆÈþÎµ,...	á/÷žÃ,àŸVÖà<ÿÝ/
-Ì82)š,Å:|Ÿ4"	°°rpQ‡};ú",x;K	Yv|þ°ÖgÏ(ý ...	Äîïþ.q?Èa,pÇª...	°©0}f]dMz$+ ÚÖ
0‰ÚØâ"o'Ùpn...	Øµo4 ¥¦Ìž^FÇj¨	PùZn:¯z©Â©"...	...z[™ÃñBzIÉø...	á/÷žÃ,àŸVÖà<ÿÝ/
;[ÛåçØÓÞ¨5ZvD'	ƒ$²»£ 9¦ögøjÆkë	ïœÉ¤Þb8Äo¡w"ùZ	{€I[7bÆ´Jò]æ¿...	á/÷žÃ,àŸVÖà<ÿÝ/
d¡€"+‹¬7ô—N...	Ö8'ÅÁÙö9½oš...	÷Ñîª›'ž‡+Ê–h...	x{cT«w:[C„KFr...	Ü~¦yî]M"oá'ì|È
FÝ¥ÜÉ½»+æ$...	°°rpQ‡};ú",x;K	*õÏÛƒ.´ˆø¿Y}BzÕ	³JÜ=Òò¹GK7R...	Ô>Ý¤š/³‰¾çÈêÉ
}T·¥í.ôˆ&xRŠbì	°°rpQ‡};ú",x;K	o%!Öù]Zv|ÐO"...	½ÕYôé7õµBþ...	ë8Ñ©%ñØ›ÒrõÑbú,,
ƒ×¨/óâ«}n¬Œ...	Ù»€x2;%Ó§‡'...	vÌ´§I`mÂDIÉP...	®þEÓ£å,T÷ï¯...	á/÷žÃ,àŸVÖà<ÿÝ/
‹€¼ûghW<ñZª...	Ö8'ÅÁÙö9½oš...	¹Nx€—JÝô: Š\...	l~b<4|{Ó›ŠÃœ...	ë8Ñ©%ñØ›ÒrõÑbú,,
@'ªËwÿyŸ<Fä...	Ö8'ÅÁÙö9½oš...	ŠH†v^%{jÔÂbÁÅ"	!¨+qÐcœ»8'À...	CÙi{¶"¥]®ç'
Ãò°e—8Ç'û¡@#	Y‹@vvO`UAH...		`(îÑÈðnžp¶lK...	á/÷žÃ,àŸVÖà<ÿÝ/
NULL	NULL	NULL	NULL	NULL

Figure 16: The encrypted table users *in the server database, as seen by using MySQL Workbench.*

6.8. Attempting SQL injection on the server database

During the development of the server, SQL injection was tested through test forms. Despite removing the protection against SQL injection, no query was successfully run. The reason for this is that the encryption functionality destroys any SQL code, by encrypting the strings into ciphertext.

6.9. Attempting to sign in using stolen token

The persistent login security was tested through the JavaScript tests that the server had for debugging reasons. The JavaScript code sent requests to the server, trying to do different things such as logging in and registering positions. When the JavaScript attempted to sign in using a different token, the server denied access and removed the token, with the following response:


```
{"response":105,"message":"Account frozen. Login with email and password."}
```

7. Result and Discussion

A working Android prototype has been developed, together with a server backend. The application allows users to share their locations with friends and interact around these through commenting. The security goals described in section 1.4 have also been fulfilled as seen in Chapter 6.

7.1. Evaluation of the test results

The server database was encrypted using 128-bit AES, using CBC with an initialisation vector. The reason for not using 256-bit AES is that the server has to handle much more data than one instance of the Android application. As mentioned in section 3.2, 256-bit keys make AES about 40% slower. This indicates that it is not possible to optimise both security and performance at the same time, and that trade-offs have to be made when developing software.

The database encryption on the server added an extra security layer against SQL injection, as described in section 6.8. This was nothing planned from the beginning of the project, but something that became clear when attempting to remove the security checks of the PHP code. Even if this unplanned effect was positive from a security perspective, it shows that unplanned effects may happen in the other direction and make a system less secure. As for the sanitation of the input, standard PHP functions were used. It would have been possible to write own functions for this, but since PHP is a widely popular scripting language, it was decided to use the standard functionality for this.

The database and SharedPreferences of the Android application was encrypted successfully. The key to decrypt the data is derived from the pin code that the user has. Although making it more difficult to decrypt data, it would still be possible if an attacker knew the pin code. Most users would probably use the same pin code from time to time, so that if an attacker would know the pin code and how the encryption key is derived, a breach could be possible. It is however less likely to recover the pin code than an encryption key stored in the application.

Encryption has not only been successfully used for the data storages. When travelling between the Android application and the server, the data is encrypted, as shown in section 6.6. This project uses OpenSSL and a certificate from RapidSSL. Since the purpose of the project is to explore basic security measures, using a free SSL library and a cheap certificate was considered as the best alternative.

Several security measures have been implemented on the application. Reverse engineering and tampering have been made more difficult, as seen in section 6.1 and 6.2. It is however not impossible to reverse engineer the application and to remove the checks in the code, but it is more difficult since the structure of the code is completely changed.

Another security measure to protect the application is root device detection, which is largely based on a check for applications associated with rooted devices. A weakness of this method is that the application would have to be constantly updated to check for new

applications of that type. Another weakness is that the application does not deny the user from using it, and it is the user's responsibility to avoid using the application. However, the warning may be an eye-opener for users who do not know that rooting the device is associated with security risks.

Protection against tapjacking works and the interface of the application does not respond when touch events are forwarded. A drawback from using this method is that it could sometimes be necessary for an application to have layers on top of other layers for graphical reasons, thus making this solution inappropriate.

The system has security measures to protect against theft of login data as well. As described in section 6.9, the server denied access when using the wrong token but the correct user id and series identifier. This solution allows an attacker to sign in as long as the ordinary user does not sign in. In theory, the ordinary user could wait a long time before attempting to sign in and this would allow the attacker to use the account freely during this period of time. Despite this, the solution of this project still increases the security since the attacker would be denied access when the ordinary user signs in.

The result of this project could be used as the first steps when developing any Android application that deals with sensitive user data. Securing user data could be relevant in many different kinds of applications, from simple games to complex systems used by for example law enforcement agencies or the government.

7.2. Discussing the security measures from a general perspective

Encrypting the database is one of the more obvious security measures, especially if the server stores sensitive data. If a database should be stolen, encrypted data would make it useless. Any encryption can be cracked given enough time, but if the complexity of the encryption is good enough, the time needed to crack the encryption could make the stolen information useless.

Another problem with encryption is how to handle the key[2]. In this project, only the root user and users in the same group as Apache have access to keys on the server. If an attacker would get the root user password, all security would fail. Another vulnerability is the Apache process. If an attacker would be able to control Apache, the keys would be compromised. On Android, the user is responsible for the encryption key, which is derived from the pin code.

Encryption is not only beneficial for persistent data, but for data in transit as well. As seen in section 6.6, data in transit can be hijacked when not using SSL/TLS. Since many people tend to use the same password on many websites and applications, a hijacked password could be disastrous for a user.

Secure connections, however, are not completely safe. The Heartbleed bug, as briefly described in section 3.5, showed that even if sophisticated and popular software is used, there is no guarantee that the software is completely secure. It is important that developers stay updated and upgrades software as bugs are revealed, both in third-party

---

[2] See Appendix A: Interview with Tomas Olovsson.

software and own software.

Protection has not only been implemented through encryption. The obfuscation led to a source code that is almost completely unreadable for a human, making it more difficult for an attacker to tamper the application. A patient attacker could reverse engineer the application and remove these safeguards. Therefore, obfuscation is not a guaranteed protection against reverse engineering and tampering, but should nevertheless be considered a good practice for application developers.

Smartphones and touch devices have other vulnerabilities than desktop computers, like tapjacking. Tapjacking is something neither developers nor users are aware about. Although data is not directly stolen through tapjacking, a user could be deceived into sharing data without knowing about it. The solution implemented in this project prevents this from happening with the scope of this social network.

7.3. Trusting third parties

Trusting third parties is necessary to build advanced software, since there is seldom time to reinvent technology and products that have already been invented. As this project focuses on the basic security measures that can be taken by application- and web developers, trusting external APIs is relevant for discussion.

This project uses both Google Maps API and Google Places API in order to work. One could argue that it is unsafe to use APIs from Google, due to Google’s business model of data tracking and advertising. On the other hand, since Google owns Android, the user is already exposed in a way. In addition to that, the application does not send any information about the user to Google when working with the APIs. While Google may track the position of the device, they do not necessarily know who owns it, from the data used by this application.

Selecting provider for the server is also a choice to consider carefully. Some employees at the provider company may have access to the servers they provide. The risk that they abuse their permissions and steal data must be weighed against the benefits. Such benefits are easy setup, backups, low cost, and power supply. It would also be devastating for a server company if it should be publicly known that they steal customer data, so it is expected that employees do not steal data.

7.4. Future work

During the initial phase of the project, the plan was to implement two-way authentication between the server and the mobile device, using a unique certificate in the Android application as well. The purpose with this was to make sure the right application is communicating with the server. This plan was eventually discarded since it became clear that it would take too much time. Trade-offs have to be made at the cost of security to get a project done in time[2].

Application functionality that would be implemented in future work:

- Removing friendship
- Editing and removing shared locations
- Editing and removing comments on shared locations

If the product would be used in a production environment, these items of functionality would have been implemented. The reason this functionality not implemented during this project was lack of time.

7.5. Conclusion

Several methods that can be used by developers to protect user data in Android applications, web servers, and transit have been explored. Although implementing these security measures, there are always more measures that can be taken. Computer security is a vast subject, and time is always a limiting factor for software developers.

There is no such thing as completely secure systems and there are still several vulnerabilities to the system. Despite this, it would probably require much more effort from an attacker to steal sensitive data from this product than a similar without the security measures taken.

References

Adobe. (2014). *Configuring and using session variables.* http://help.adobe.com/en_US/ColdFusion/9.0/Developing/WSc3ff6d0ea77859461172e0811cbec22c24-7c48.html [2014-05-10]

Android Developers. (2014a). *Android, the world's most popular mobile platform*. http://developer.android.com/about/index.html [2014-04-22]

Android Developers. (2014b). *Application fundamentals.* http://developer.android.com/guide/components/fundamentals.html [2014-05-10]

Android Developers. (2014c). *Location Strategies.* http://developer.android.com/guide/topics/location/strategies.html [2014-05-14]

Android Developers. (2014d). *Intents and Intent Filters.* http://developer.android.com/guide/components/intents-filters.html [2014-05-02]

Android Developers. (2014e). *Activities*. http://developer.android.com/guide/components/activities.html [2014-05-10]

Android Developers. (2014f). *BroadcastReceiver.* http://developer.android.com/reference/android/content/BroadcastReceiver.html [2014-05-10]

Android Developers. (2014g). *View*. http://developer.android.com/reference/android/view/View.html [2014-05-02]

Android Developers. (2014h). *Toasts.* http://developer.android.com/guide/topics/ui/notifiers/toasts.html [2014-05-02]

Android Developers. (2014i). *Storage Options*. http://developer.android.com/guide/topics/data/data-storage.html [2014-05-05]

Android Developers. (2014j). *Distribute Apps.* http://developer.android.com/distribute/index.html [2014-05-10]

Android Developers. (2014k). *Signing Your Applications.* http://developer.android.com/tools/publishing/app-signing.html [2014-04-15]

Android Developers. (2014l). *Google Cloud Messaging for Android.* http://developer.android.com/google/gcm/index.html [2014-05-07]

Android Developers. (2014m). *GCM Advanced Topics.* http://developer.android.com/google/gcm/adv.html [2014-05-07]

Android Developers. (2014n). *Implementing GCM Client.* http://developer.android.com/google/gcm/client.html [2014-05-09]

Android Developers. (2014o). *Android Security Overview.* http://source.android.com/devices/tech/security/ [2014-04-04]

Android Developers. (2014p). *Get the Android SDK.* http://developer.android.com/sdk/index.html?hl=sk [2014-04-27]

Android Developers. (2014q). *Platform Versions.* http://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net [2014-05-13]

Android Developers. (2014r). *ProGuard.* http://developer.android.com/tools/help/proguard.html [2014-05-13]

Bornstein, D. (2008). *Dalvik VM Internals.* http://fiona.dmcs.pl/podyplomowe_smtm/smob3/Presentation-Of-Dalvik-VM-Internals.pdf [2014-05-16]

Garcia-Molina, H & Ullman, J & Widom, J. (2009). *Database Systems: The Complete Book. 2nd Edition.* Boston: Pearson.

GeoTrust. (2014). *About Us.* http://www.geotrust.com/about/ [2014-05-13]

Gunasekera, S. (2012). *Android Apps Security.* New York: Apress.

Haas, J. (2014). *Process.* http://linux.about.com/od/linuxdocumentation/a/Process.htm [2014-04-24]

Heartbleed.com. (2014). *The Heartbleed Bug.* http://heartbleed.com/ [2014-05-17]

IBM. (2006). *Java method inlining performance considerations.* http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzaha%2Fjmiperf.htm [2014-05-16]

IETF. (2012). *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension.* http://tools.ietf.org/html/rfc6520 [2014-05-17]

Jaspan, B. (2006). *Improved Persistent Login Cookie Best Practice.* http://jaspan.com/improved_persistent_login_cookie_best_practice [2014-05-19]

Johnson, K. (2011). *Revisiting Android TapJacking.* http://blog.nvisium.com/2011/05/revisiting-android-tapjacking.html [2014-04-05]

Jonasson, J & Lemurell, S. (2004). *Algebra och diskret matematik.* Lund: Studentlitteratur AB.

JSON.org. (2014). *Introducing JSON*. http://www.json.org/ [2014-04-22]

Khalaf, S. (2014). *Mobile Use Grows 115% in 2013, Propelled by Messaging Apps.* http://blog.flurry.com/bid/103601/Mobile-Use-Grows-115-in-2013-Propelled-by-Messaging-Apps [2014-04-21]

Kurose, J & Ross, K. (2010). *Computer Networking: A top-down approach, Fifth Edition.* Boston: Pearson.

Lafortune, E. (2013). *ProGuard and DexGuard.* http://www.saikoa.com/downloads/ProGuard_DroidconLondon2013.pdf [2014-04-04]

Manjunath, V. (2011). *Reverse Engineering Of Malware On Android.* Swansea: SANS Institute.

Niemietz, M. & Schwenk, J. (2012). *UI Redressing Attacks on Android Devices.* Bochum: Ruhr-Universität Bochum

Netcraft. (2014). *April 2014 Web Server Survey.* http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html [2014-04-27]

Network Working Group. (1999). *Hypertext Transfer Protocol -- HTTP/1.1.* http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5 [2014-05-19]

Norbye, T. (2012). *ProGuard Improvements.* http://tools.android.com/recent/proguardimprovements [2014-04-04]

ProGuard. (2014a). *Introduction* http://proguard.sourceforge.net/#manual/introduction.html [2014-04-04]

ProGuard. (2014b). *A simple Android activity* http://proguard.sourceforge.net/#manual/examples.html [2014-05-13]

ProGuard. (2014c). *What is obfuscation?* http://proguard.sourceforge.net/index.html#FAQ.html [2014-05-08]

Qiu, Y. (2012). *Tapjacking: An Untapped Threat in Android.* http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/ [2014-05-17]

RapidSSL. (2014). *99+% Browser Recognition.* http://www.rapidssl.com/learn-ssl/ssl-browser-recognition/index.html [2014-05-15]

Richardson, D. (2010). *LOOK-10-07 TapJacking.* https://blog.lookout.com/look-10-007-tapjacking/ [2014-04-05]

Rivest, R. (1990). *Cryptology.* Cambridge: MIT.

Saikoa. (2014). *ProGuard.* http://www.saikoa.com/proguard [2014-04-04]

Six, J. (2012). *Application Security For The Android Platform*. Sebastopol: O'Reilly.

Schneier, B & Whiting, D. (2000). *A Performance Comparison of the Five AES Finalists.* https://www.schneier.com/paper-aes-comparison.pdf [2014-06-05]

Snyder, C & Myer, T & Southwell, M. (2010) *Pro PHP Security: From Application Security Principles to the Implementation of XSS Defenses, Second Edition.* New York: Apress.

Somasundaram, R. (2013). *Git: Version Control for Everyone*. Birmingham: Packt Publishing

Stamford, C. (2013). *Gartner Says Mobile App Stores Will See Annual Downloads Reach 201 Billion in 2013.* http://www.gartner.com/newsroom/id/2592315 [2014-04-21]

Tanenbaum, A & van Staveren, H & Stevenson, J. (1982). *Using Peephole Optimization on Intermediate Code.* Amsterdam: Vrije Universiteit.

The OpenSSL Project. (2014). *TLS Heartbeat read overrun.* http://www.openssl.org/news/secadv_20140407.txt [2014-05-15]

The PHP Group. (2014). *What is PHP?* http://www.php.net/manual/en/intro-whatis.php [2014-03-28]

Ubuntu.com. (2014). *About Ubuntu.* http://www.ubuntu.com/about/about-ubuntu [2014-04-22]

Wei, J. (2010). *Indefinite Toast Hack.* http://thinkandroid.wordpress.com/2010/02/19/indefinite-toast-hack/ [2014-04-05]

World Economic Forum. (2014). *Increased Cyber Security Can Save Global Economy Trillions.* http://www.weforum.org/news/increased-cyber-security-can-save-global-economy-trillions [2014-05-19]

Åhlin, D. (2012). Skatteverket hackat. *TechWorld,* 29th of March. http://techworld.idg.se/2.2524/1.440750/skatteverket-hackat [2014-05-13]

Appendix A: Interview with Tomas Olovsson

During the last month of the project the group conducted an interview with Tomas Olovsson, associate professor in the field of Networks and Systems. The original interview was conducted in Swedish. A summary of the interview is listed below.

Should we bring up direct and indirect attacks in the report?
- It depends, if you are protecting against it then you could, otherwise it is not necessary. It is important to consider the time of the project. A trade-off always has to be made between what you are trying to protect against, time and effort in the terms of money etc.

Regarding cryptographic keys, where should they be stored?
- The keys are always the problem. Whichever cryptographic algorithm you choose, a key is a key, which can open up the data. Trust has to be made, should the server store the key or is it the users responsibility? It could depend on what the project is set to protect against. I would recommend doing a risk analysis on what the project is set to protect against and start from that.

Should we encrypt the data at the phone also?
- It's a good idea; the problem is still how to handle the encryption key. Some newer phones are starting to get special hardware for storing private keys where the private keys cannot be exported. The most secure way is then to handle the key in the hardware part, or if the key is not stored at all and must be memorised by the user. In the case of letting the user remember the key, it is expected that the user will not write it down.

Any general advice regarding this project?
- As stated previously, come up with scenarios that you want to protect against and then see what can be done to mitigate the issue. Do some kind of risk analysis and see what is likely to happen and what is worth protecting. Do not forget about the time constraints for the project.

Appendix B: Libraries, tools and services

Tools

- **Android Debug Bridge:** http://developer.android.com/tools/help/adb.html
- **apktool:** https://code.google.com/p/android-apktool/
- **Atom:** https://atom.io/
- **dex2jar:** https://code.google.com/p/dex2jar/
- **Eclipse with ADT:** http://developer.android.com/tools/sdk/eclipse-adt.html
- **Git:** http://git-scm.com/
- **JD-GUI:** http://jd.benow.ca/
- **MySQL Workbench:** http://www.mysql.com/products/workbench/
- **MySQL:** http://www.mysql.com
- **OpenSSL:** http://www.openssl.org/
- **Sublime Text:** http://www.sublimetext.com/2
- **SQLite:** http://www.sqlite.org
- **tPacketCapture:** https://play.google.com/store/apps/details?id=jp.co.taosoftware.android.packetcapture
- **Visual Studio:** http://www.visualstudio.com/
- **Wireshark:** http://www.wireshark.org/

Libraries

- **cURL:** http://www.php.net/manual/en/book.curl.php
- **Google Play Services:** http://developer.android.com/google/play-services/index.html
- **Jackson:** https://github.com/FasterXML/jackson
- **mcrypt:** http://se1.php.net/manual/en/book.mcrypt.php
- **SQLCipher:** http://sqlcipher.net/

Services

- **CityCloud:** http://www.citycloud.se
- **Github:** https://github.com/