MijnID - Security for an Android Application

Authors:
A. ten Napel  
4087798
A. Narwade  
4013085
T.P. van Helden  
4106725

Supervisors:
Dhr. S. Hadri
Dr. J.A. POUWELSE
Dr. M.A. Larson
Dr. F. Hermans

February 13, 2015
Abstract

This document reports the development of a proof of concept for the MijnID Android mobile application. Identity theft and identity fraud are serious problems, leaving thousands of Dutch citizens with debts and damages each year. MijnID aims to improve detection by providing users with insight on what is happening to their personal details as they are stored in government databases. This early detection is key to mitigate damages. In this report we discuss the problem, project aims, project outcome and further development. The core functional features for the app consist of verifying user identity with DigiD, logging in to the application with a PIN, viewing personal data and safely reporting changes in a database to the user. The project focuses on the security and privacy of data during transport and as it is shown to the user. We use scrum and test-driven development to guide the development process. Unfortunately the MijnID app will be a target for numerous types of attackers and attacks. We map these attacks, define a scope and base security requirements on the threats within that scope. Functional requirements are derived from the wishes of various parties involved. The design process is one of constant adjustment as changing requirements and limited experience play a large role. We design and implement the application as well as a dummy environment, which consists of a server and a database. This dummy environment mimics the behaviour of the real environment in which the app will function. Furthermore, we interact with the testing environment of DigiD. Automated and manual test outcomes are used to evaluate the requirements. We manage to meet 80% of all requirements and over 85% of the security requirements. Challenges the team faced are discussed as well. These challenges range from implementation issues to ethical considerations. Several recommendations are provided to deal with open security issues and to continue development in the future.
Preface

This is the final report which describes the development of the MijnID application as a proof of concept for a broader pilot program. The MijnID application is an app for Android devices that monitors changes made in the Municipal Personal Records Database. These changes occur when data about that person is changed as issued by themselves or by others. This second case occurs when identity fraud has occurred. The user is notified when a change is made. This makes sure the user sees these changes on time. Through this early detection, identity fraud can be spotted and dealt with earlier on. The main focus of the project was the security for this app.

This project was done for the course TI-3806 Bachelor Project 2014/2015 at the Delft University of Technology. This project assignment was issued by Milvum. Milvum is a start-up company which develops applications for various platforms. Their office is located in Den Haag in the Netherlands. All of the research and development was done at the office of Milvum.

This report will provide information on the project methodologies followed, design decisions made, testing practices, requirements, implementation, our final results and recommendations.

We would like to thank everyone at Milvum, specifically Salim Hadri, Viresh Jagesser, Arvind Jagesser and Randy Tjin Asjoe for all their support, guidance, time, space and lunch. Without them, we would not have had this opportunity. We want to thank Johan Pouwelse for all his feedback and expertise. Without that, we would have been left guessing whether our solution was proper. We would also like to thank Martha Larson and Felienne Hermans for allowing us to do this project outside the regular schedule. Next, we want to thank Eric Bouwers at the Software Improvement Group for his feedback regarding our code development. Finally, a special thanks to Bastiaan Reijm for putting us in touch with Milvum.
## Contents

1 Introduction .......................................................... 1

2 Project Description .................................................. 2

3 Project Strategy ....................................................... 4

4 Security Threats .......................................................... 7
   4.1 Attacker profile ....................................................... 7
   4.2 Typical attacks ..................................................... 7
   4.3 The security scope for MijnID ................................. 9

5 Requirements .............................................................. 10
   5.1 Functional Requirements ........................................ 10
   5.2 Quality Requirements ........................................... 11
   5.3 Security Requirements .......................................... 12

6 System Design & Implementation ...................................... 17
   6.1 Global System Design & Implementation ....................... 17
   6.2 Client Design & Implementation ................................ 19
   6.3 Dummy Server Design & Implementation ..................... 23

7 Testing and Evaluation ................................................ 25
   7.1 Automated Testing ................................................ 25
   7.2 Integration testing ................................................ 25
   7.3 Software Improvement Group Feedback ....................... 29
   7.4 Requirements evaluation ........................................ 30

8 Challenges and Recommendations .................................... 31
   8.1 Challenges ........................................................... 31
   8.2 Recommendations ................................................ 35

Conclusion ................................................................. 38

Glossary ................................................................. 40

Acronyms ................................................................. 41

Bibliography .............................................................. 42

Appendices ................................................................. 47
   A Project Proposal (Dutch) ........................................... 48
   B SIG feedback (Dutch) ................................................. 51
List of Figures

5.1 User interactions with the application ............................................. 10
6.1 The major components in our system ............................................. 18
6.2 An overview of all main classes and flow in the client .......................... 20
6.3 Registration process ....................................................................... 20
6.4 Activity navigation ......................................................................... 21
6.5 Overview of all classes and flow in the dummy environment ................. 23
7.1 Example branch, illustrating registration sequence followed by viewing personal details ......................................................... 26
7.2 Activities exported ......................................................................... 27
7.3 HTTPS connections intercepted ......................................................... 28
7.4 Mallodroid xml-output .................................................................... 28
8.1 Scrum burndown chart .................................................................... 32
8.2 First client design ........................................................................... 33

List of Tables

4.1 Attacks classified by target .............................................................. 8
4.2 Attacks classified by flaw ................................................................. 8
5.1 Functional Requirements ............................................................... 11
5.2 Quality Requirements .................................................................... 12
5.3 Client Security Requirements ......................................................... 13
5.4 Requirements to counter types of attacks ......................................... 14
7.1 Tests and corresponding classes ....................................................... 25
7.2 Requirements evaluation ............................................................... 30
1. Introduction

Identity theft and identity fraud are serious problems. They are hard to detect and the victim often does not know what hit them until they are thousands of euros in debt. Early detection is key to tackling this problem. To quickly detect identity fraud, one could look at how changes are made to the personal details of citizens in databases. Examples of these personal details are registered address, name, names of children, phone number, bank accounts and credit cards. If unauthorized changes are detected, it might mean that an identity has been stolen and that identity fraud is being committed. The perpetrator might open credit card accounts, register phone number, rent cars or buy houses.

This project aims to build a proof of concept for an application that can help Dutch citizens detect unauthorized changes in their personal details. The project focuses on the security aspects of the Android application. This project will not result in a production-ready product. This document talks about the general design and implementation of the MijnID application. Apart from this, there are specific parts dedicated to security. In chapter 2 we give a broader explanation of the problem that this project addresses and the project assignment. In chapter 3 we talk about the project strategy. This includes our strategy for teamwork and development. We also give an overview of our planning and the methodologies and tools we used. In chapter 4 we profile our possible attackers, classify their attacks and define a scope for our security design. In chapter 5 we discuss the requirements for the app. We define three types of requirements: functional, quality and security. Note that we will not go into great detail on solutions for all security threats. In chapter 6 we talk about the design and implementation of the entire system. We first talk about the global design and some of the core features that require the interaction of multiple components. We then turn to a more detailed explanation of how the client or app works. We wrap up chapter 6 with a description of the dummy environment that is used to simulate the future live environment. Chapter 7 will explain how we tested different features and which tools we used to ease this process. We distinguish between automated testing and integration testing. Since all our components are very dependent of each other, integration testing was a large part of our project. The feedback from the Software Improvement Group (SIG) will also be discussed. Chapter 7 finishes with an evaluation of our requirements, to see which ones we managed to implement and which ones we did not. Chapter 8 discusses the difficulties and challenges we faced during this project. We will explain how our lack of experience in some areas made the project more challenging. Planning issues and methodology drawbacks also played a role in this project. There were some issues during the implementation of the app, which will be explained as well. We also briefly touch on the main ethical question we faced during this project. Chapter 8 will end with recommendations we have for Milvum and other future developers, these include solutions to open issues in the application. Finally, we conclude the report with a conclusion. We review all chapters and with a focus on what we learned and what we recommend Milvum to do in the future. A glossary and a bibliography are included after the conclusion. The appendices contain the original project proposal and the original feedback from SIG.
2. Project Description

"Identity theft is when your personal details are stolen and identity fraud is when those details are used to commit fraud [43]". Identity theft is almost impossible to detect. The most effective counter to identity theft is to inform people of who they can trust with their personal details, but even then it happens a lot. Identity theft often turns into identity fraud, which is detectable, but still very hard to counter. This is because people do not see their personal information on a daily basis. For example, if someone would get a new credit card on your name, it could take a long time before you would notice, especially if they changed your address as well. This happens more often than people think and the result is often several thousands of euros of debts and damages. A study by PricewaterhouseCoopers [53] showed that between 2007 and 2010 as much as 5.6% of the total Dutch population was the victim of identity fraud. Financial damage to civilians was estimated to be over 1 billion euros. They also estimated that in 2011, these numbers would increase as another 127.000 victims and 0.42 billion euros in damages were expected. In the current situation, there is often no fast way to check if changes have been made to personal information.

This is a problem that concerns the Dutch people as a whole. This is why the Ministry of the Interior and Kingdom Relations (BZK)[34] is looking for a solution. As personal information on each Dutch citizen is at stake, it also affects the private sector. If you own a company and someone can impersonate you, this could cause reputation and financial damage to your company. So there is no question that this is a big problem, but what is the solution?

**Project assignment.** One way to defend against identity fraud is to develop an application that give users insight in their personal information and sends notifications when changes are made. This way, identity fraud can be countered faster, reducing the potential damage. If someone would get a passport or credit card on your name, you would get a notification, stating that a new passport or credit card is requested. With identity fraud, as with many other problems, detection is extremely important to prevent damages.

While working on their KopieID app [32] for BZK, Milvum [11], a mobile application development company, came up with this solution. BZK agreed to a pilot program to develop this application. Our project team was then assigned to build the initial proof of concept. Our main focus was the main concern of BZK: Can we guarantee privacy and security in mobile applications? The design of the application would be similar to that of the ING [30] and ABN-AMRO [1] online banking apps.

To build this proof of concept, we had to research cyber security. This was not only to have a better understanding ourselves, but also to pave the way for future research by Milvum when they would continue the pilot. The scope of the research was bigger than just the application, as any supporting components, like servers and databases, also needed to be researched. During the implementation though, all components other than the mobile application could be considered dummy versions.

The major components to be implemented were:

1. User can verify his identity by logging on to DigiD[26].
2. Once the user has verified his identity once, the user can log in using a PIN.
3. The application can show personal data to the user.
4. The application can notify the user when a change in their personal details occurs.

All the above were required to be implemented with an as high as possible level of security. This criterion was not very specific, which led us to research the boundaries of the scope ourselves.
Parties involved. There are several organizations that were involved in creating this solution. Milvum is one of these organizations. Milvum is a Full Service Mobile App Agency that helps companies and institutions map their mobile opportunities, choose the right strategy and realize success for the mobile platform. With this, quality is of the highest importance. Milvum helps prepare organizations for the future.

Delft University of Technology (TU Delft) [3] is the university where the project group studies. This project is a Bachelor End Project for the Computer Science course. Several people from the TU Delft were involved in the setup and supervision of this project.

BZK is the party providing many of the requirements. They launched the pilot program of which this project is a small part. To display personal data of the users we also needed access to the Municipal Personal Records Database (BRP). The BRP is a collection of data of citizens in The Netherlands, for example name and address. The BRP also contains the Citizen Service Numbers (BSN) of all citizens, an unique number identifying a specific Dutch citizen. Specifically, we needed access to the Municipal Personal Records Database Provider (GBA-V)[28], which is the actual database containing the data from the BRP.

Logius [9] is the institution of BZK that is behind DigiD. Naturally, they were an essential part in the development of this project. We spent several hours mailing and calling Logius and they were most helpful.

An external designer bureau was also involved. Their task was to supply the design for the GUI.

Digital Ocean [4] is a hosting company where we ended up hosting our server. They offer great support and offer low price, quick solutions.

Azure [10] is the massive hosting platform by Microsoft. They offer a big variety of packages with a lot of possibilities. We initially hosted our server there, but their budget options were limited.
3. Project Strategy

A good project setup can prevent problems down the line. Setting standards and rules is important to avoid conflict. A good organizational structure heavily influences the effectiveness and efficiency of a team. At the start of the project we put quite some time in developing a sound system to guide our team efforts. In this chapter we will discuss our project strategy, methodologies and the tools we used to guide our project.

Methodologies. Scrum [82] is an agile software development framework. In scrum you divide the project into sprints or cycles. At the beginning of each sprint a sprint planning session is held in which the tasks for that sprint are determined. A sprint backlog is used to write down each task and track the progress of the sprint. Each day, a daily scrum meeting is held to briefly discuss the most pressing matters. After that, everyone works on their tasks as described in the scrum backlog. Throughout the project we used scrum as a format to guide the design and implementation process. We chose this methodology for several reasons. First of all, it was the standard for Milvum. All their projects are done in this way because it is highly adaptable and scalable. This works perfect in a fast changing market like the mobile application market where customers want a demo every two weeks. Second, the project had a lot of insecurities. Scrum allows for quick pivoting because every week you choose a product or set of features to work on. We know now that certain requirements were still unclear in week seven. The agility that scrum offered us was a determining factor for the success of this project. Initially, the sprints were two weeks but we later changed this to one week to better fit our planning.

On the subject of implementation, we decided to use Test Driven Development (TDD) [47] as a methodology. TDD is a software development process that works as follows. The programmer creates a set of tests that represent the desired output given certain input. These tests will initially fail since there is no code yet. Once all tests have been written, the programmer writes code that makes the tests succeed. This forces the programmer to think about which input results in which output. This in turn will lead to cleaner, more focused and easily tested code. This project is just a part of a pilot, so extensibility is important. For this reason, clean code is very important. Secondly, nothing shouts attack me like untested code, so enforcing automated tests also seemed like a good idea.

Project roles. Throughout the project, everyone developed several roles. Leadership was exercised by different people in different situations. Thomas handled the initial contact with the company, which caused him to take a leading position at first. Once everyone settled into the project, Albert and Anish also took control in certain situations. Another interesting development was the specialization of each team member. Everyone started off relatively inexperienced in all areas, but soon everyone became a master of a specific aspect of the project. Anish focused on the Android code, designing most parts of the interface and using Google libraries. Albert managed the security aspect on the Android application and the communication between the server and the client. Thomas built most parts of the environments like the server and the database. This specialization helped us work very efficiently. For meetings it was always a natural decision who would be the leader. Often the person with the most pressing problem or most important feature for that sprint would take the lead. This diverted a bit from the standard scrum methodology but worked out well for us, especially when size of features increased.

Meetings. At the start of each sprint we held a sprint planning session. In these meetings we planned what we would do, how much time it would take and who would do it. For us this was a democratic process, as opposed to the normal scrum setup where mainly the product owner does this. Our approach kept everyone involved. This went well because we were a small group. In a larger group however, we would stick to the regular scrum approach with Scrum Master and Product Owner, as democracy tends to be more time consuming. Estimating how long tasks took to complete was difficult but we became better at it as the weeks passed. Salim Hadri would
often review the tasks planned for the week. We held daily scrum meetings with just the team members. In these meetings we discussed design issues but also interpersonal issues. These meetings kept the group together and helped keeping us goal oriented. When a big problem arose, the daily scrum meetings often acted like a reset button for our stress levels. Every Friday we held a meeting with Salim about how we progressed that week. We would talk about specific issues that we encountered and whether or not we had to prioritize certain tasks next week.

Apart from the meeting every Friday, we also had non-stop support from Milvum. Salim often met with us to give us updates on what happened with other parties involved. With so many stakeholders that was really helpful. The other programmers at Milvum also helped identify Android specific problems.

Johan Pouwelse was key in advising us on security. To explore such a vast area without any expert to verify you are on the right track is close to impossible. Meetings with Johan often meant we had to review a large part of our project as he was rather critical, but this led to more clarity for us and for Milvum.

Martha Larson skyped with us halfway through the project. This was initially just a checkup but we gained some valuable tips from her. She forced us to ask ourselves a couple of hard questions in terms of planning and reinforced our professional attitude towards Milvum.

**Planning** Planning was particularly interesting with this project. Because there were so many parties involved with different schedules and interests, it was hard to plan for everything.

The project started as five, two week sprints. The first sprint would be research, followed by three sprints coding and then one sprint wrapping up. That very quickly changed as the full specification were still unclear. We did not know yet which database we should or could connect to. It was also not clear whether we would get full access or only testing access to the DigiD platform. We decided to focus on what we knew for sure. The sprint length turned into one week to allow for more flexibility. Each sprint should still have a measurable result at the end of the week, so we had to restructure components to fit into one week of production. This new structure split the project into two phases.

The first phase would last until the end of 2014, leaving us four sprints to implement the components that did not require external databases or DigiD.

The second phase would last until the 13th February. Six weeks were left to implement a session system, connection to DigiD and connecting with all databases. Connecting to the testing environment of GBA-V was more of a bureaucratic nightmare than Milvum wanted to get into for a proof of concept, so we had to create a dummy database with test data provided by the BRP.

With these two phases defined, we planned each week according to features that were ready to be implemented or research topics that were ready to be researched. Each phase started with one sprint research. After that, we spent about a week implementing or updating the automated tests for the features we were implementing that phase. We would then spend a week to implement the features, followed by a week of integration testing and bug fixing. The last two sprints were used to put the report together as well as allow for possible delays.

In this first phase we implemented and tested the following:

1. Basic application with different activities
2. The basic dummy server
3. Logging in with a PIN
4. TLS 1.2 with self signed certificate

In the second phase we dedicated most of our time to:

1. DigiD
2. Sessions
3. Dummy BRP database
4. Debugging

5. Writing our final report and guides

**Tools.** During this project we used several tools to help us develop and test all the components. We will go over the reasons why we chose these tools.

We used Git[6] to share code. Initially we had the option of choosing Apache Subversion[2] but the majority of the group had better experiences with Git. SmartGit[13] is the Git client which we used. There are many alternatives, but we had great experiences with this one. Google Docs[7] was used to share documents because it is feature-full and easy to setup. We used Eclipse[5] as IDE because it has a good development environment for Android and came recommended by several programmers. It also works well for developing other types of projects, like setting up the dummy environment. We used Tomcat[16] to run our dummy server because of its ease of use. Certain code could be shared because both the server and client are implemented in Java. Using a Java server also meant not having to learn a new programming language. Security was not considered when choosing this tool. We used PostgreSQL[12] as our dummy database. Because we would only use it as a dummy, the security settings were not of the highest importance. We used Java Keytool[75] to generate server keystores and self signed certificates. Drozer[27] is a tool that checks the security settings of an Android application. We used this during integration testing to check our Android security settings. Cain & Abel[22] is a password retrieval tool that can sniff networks for data that is not sufficiently encrypted. It can also fake certificates and perform MITM attacks. We used it to test whether our client server connection was protected. Wireshark[17] was used in combination with Cain & Abel to check if our TLS was actually working. We used it previously during our studies and it is commonly used to inspect connections. Mallodroid[33] is a tool that helps detect broken SSL in Android applications. We came across this tool when researching security issues for Android. We planned to use this to check if our SSL/TLS settings were correct for our application. We used Advanced Rest Client Application[18] to manually test the dummy server.
4. Security Threats

This chapter is a product of our research sprints. We aim to establish how attackers work and what attacks they use. From this we define what attacks we will try to prevent and which ones we won’t.

4.1 Attacker profile

To correctly profile an attacker you need to examine who they are and what they want. By examining the targets, much can be learned about the perpetrator [70]. Is this attacker a group or an individual? What do they stand to gain by attacking this target? What kind of techniques are used? The application displays personal details of people. In the future, the application would show financial data, personal data and more. Attackers can do a lot with this kind of information.

The attacker could attack the system to disrupt it. This can be done to send a message, damage reputation or distract from other targets. The gain from this is usually political. The term hacktivism [79] is often used in this case. Our application is definitely a target for hacktivists because it will be a government app used by many people.

Another reason would be to extract information about a specific person. This can be useful if the personal details of this person can be used for something else. With this personal data an attacker could initiate other cyber-attacks, open new credit card accounts, create false identities and more.

A third reason to attack this system is to extract data on a large scale. Again, with the data extracted many things can be done. This time however, it can be done on a large scale. If done correctly, it can be harder to spot and the payoff is bigger. Also, with more targets, there are more entry points. Most attackers will probably attempt attacks on this scale. Breaking into the server or deceiving a large number of innocent civilians can very quickly turn into massive financial gain.

There are other possible scenarios where an attacker wants to attack our app. Some attackers do it just for fun. Some attackers might have a personal vendetta against certain users. The potential rewards for breaking in are high. The goals of the attack can be financial, political or even without clear purpose. The attackers can be individuals or groups looking to attack one or more users or institutions. There can be and will be different types of attackers and different types of attacks.

Most attackers do follow a similar pattern [59]. First reconnaissance is performed on the target to find as much information as possible. Then the target is scanned to find entry points and exploitable software or hardware. After they try to break in during the exploitation phase. After exploitation, hackers will often try to create a back door which allows them permanent access. Finally, good hackers will hide their tracks. Recognizing these patterns will help defend against attacks.

4.2 Typical attacks

There are many ways to attack or exploit a target. It is useful to categorize attacks, so that weak points can be discovered and can be focused on. To talk about attacks in an abstract manner, we will use two classification schemes provided by the AVOIDIT taxonomy [83]. Table 4.1 shows attacks classified per target. We give a description of the attack type and some concrete examples of these attacks.

These examples are by no means an exhaustive list. Often combinations of these attacks are used to gain access to a system. Furthermore, access to one component can also lead to access to other components. For example, access to the OS or root user easily turns into access to an application installed on that OS.
Table 4.1: Attacks classified by target

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Attacking the user to gain information or access</td>
<td>Coercion, regular theft, social engineering, fake applications [91]</td>
</tr>
<tr>
<td>Local</td>
<td>Breaking into a specific device</td>
<td>Malware [94], FROST [73], Cold boot attacks [63]</td>
</tr>
<tr>
<td>Application</td>
<td>Breaking into an application</td>
<td>Malware, brute force [66], injection [46], dictionary attacks [74], rainbow tables [71]</td>
</tr>
<tr>
<td>Operating System</td>
<td>Attacking a specific OS, user account or driver</td>
<td>Exynos exploit [40]</td>
</tr>
<tr>
<td>Network</td>
<td>Attacking a network</td>
<td>Man in the Middle (MITM) [51], Denial of Service (Dos) [69], ARP spoofing [92], certificate faking [72], DNS spoofing [86], session hijacking [67], Heartbleed [57]</td>
</tr>
</tbody>
</table>

With so many attacks and more it becomes hard to imagine how to defend against them all. To gain a better understanding of how to protect ourselves, let us look at attacks from another perspective. Table 4.2 shows how, using the AVOIDIT taxonomy, attacks can be classified according to attack vectors. Simply put: "What flaw can be used to break in".

Table 4.2: Attacks classified by flaw

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misconfiguration</td>
<td>Abuse misconfigured applications</td>
<td>Misconfigured SSL libraries [61], Abusing default passwords [87]</td>
</tr>
<tr>
<td>Kernel flaws</td>
<td>Abuse flaws in the OS</td>
<td>Exynos exploit [40]</td>
</tr>
<tr>
<td>Buffer overflow [54]</td>
<td>Exceed capacity of buffer to disrupt</td>
<td>Heartbleed [57]</td>
</tr>
<tr>
<td>Insufficient authentication validation</td>
<td>Exploit weak authentication measures</td>
<td>CSRF [49], unvalidated redirects/forwards [62]</td>
</tr>
<tr>
<td>Insufficient input validation</td>
<td>Abuse lack of input checking</td>
<td>SQL injection [64], XSS attack [46]</td>
</tr>
<tr>
<td>Symbolic links [89]</td>
<td>Create link to file that application has permission to write to</td>
<td>Wget Symlink attack [44]</td>
</tr>
<tr>
<td>File descriptor</td>
<td>Abuse file system index to access files</td>
<td>OpenBSD File Descriptor Vulnerability [37]</td>
</tr>
<tr>
<td>Race conditions</td>
<td>Abuse a time window where files are in between privilege modes</td>
<td>Abusing a race condition flaw in UNIX [48]</td>
</tr>
<tr>
<td>Incorrect file/directory permissions</td>
<td>Directly exploit unprotected directories or files</td>
<td>Cisco VPN local privilege escalation [24]</td>
</tr>
<tr>
<td>Social Engineering</td>
<td>Attacking a network</td>
<td>419 scam (&quot;Nigerian Prince&quot;) scam [84]</td>
</tr>
</tbody>
</table>
4.3 The security scope for MijnID

With these classification schemes we know what to look for during the design and implementation of security features. In theory, any of these attacks could be used to attack a mobile application that is connected to a network. To be able to establish requirements and to evaluate those requirements, let us first define a clear scope. Looking at table 4.1 there are scenarios where the device is accessed directly by hackers. There are also non-cyber attacks that deceive the user or coerce the user into giving access. Both these attack types are considered to be outside the scope of this project. Also, since we cannot alter the operating system itself, the options for protecting the OS are also limited. So we need to protect an Android application that is part of a network. This application is installed on a device that might be infected with malware or might be accessed remotely.

Looking at table 4.2, we can define file descriptor attacks and kernel flaw exploits as out of scope since we do not have privileges to access the OS in such a way. Social engineering can only be used on the user. So during the design and implementation we need to be mindful of misconfigurations, buffer overflows, strange input, symbolic links, weak authentication, file and directory permissions and race conditions. This last one is especially true when we are running multiple threads.
5. Requirements

We now provide an overview of the requirements. The requirements started off differently than what they ended up being. The original requirements can be found in appendix A. This change is discussed in chapter 8. In the sections below we will explain all requirements using tables. All three tables have a similar format. Table 5.1 shows a full overview all functional requirements. Table 5.2 shows the quality requirements. Table 5.3 shows all security requirements for the client. These tables also show ranks according to the MoSCoW [50] model, which subsystems are involved in the implementation and how the implementation is evaluated. In the MoSCoW model, the M stands for Must haves, S stands for Should haves, C stands for Could haves and W means Won’t haves. The subsystem components are abbreviated as with S, C, D, DB, ? which stand for Server, Client, DigiD, Database and unknown respectively. Unknown in this case means that there should be some other entity that is involved, possibly some governmental institution, but it is unclear at this point what institution that would be. Evaluation methods are abbreviated with the letters A, T, M, C, D, R which stand for Automated tests, test with a Tool as part of integration testing, Manual test, Code inspection, live Demonstration and Reading the documentation. These evaluations will be further discussed in chapter 7.

5.1 Functional Requirements

The functional requirements remained pretty much the same throughout the project. The use case diagram 5.1 shows most of the actions available to the user.

![Use case diagram 5.1: User interactions with the application](image)

Note that Login to DigiD is not executed in the client, as instead, the user is redirected to the DigiD login page. This is an external requirement by DigiD itself. Yellow represents actions that are part of the registration process. Registration without Choose PIN is called the verification process. Green actions can be done once the user has
registered. Registration and logging in is called authenticating. Blue actions are only available once the user has logged in or is authenticated. The requirements are ordered by feature type. Table 5.1 shows a full overview all functional requirements.

Table 5.1: Functional Requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Client application with minimal GUI with all activities</td>
<td>M</td>
<td>C</td>
<td>A,M,D</td>
</tr>
<tr>
<td>F2</td>
<td>Dummy server to handle data trafficking</td>
<td>S</td>
<td>S</td>
<td>A,M,D</td>
</tr>
<tr>
<td>F3</td>
<td>Dummy database to store account details: PIN, authentication tokens, app ID and BSN</td>
<td>S</td>
<td>DB</td>
<td>M</td>
</tr>
<tr>
<td>F4</td>
<td>Dummy database to act as GBA-V database</td>
<td>S</td>
<td>DB</td>
<td>M</td>
</tr>
<tr>
<td>F5</td>
<td>Server can connect with databases like GBA-V or BelastingDienst</td>
<td>C</td>
<td>S,C,?</td>
<td>M</td>
</tr>
</tbody>
</table>

Core components

<table>
<thead>
<tr>
<th>Core communications</th>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6</td>
<td>Server can connect with the database to store and retrieve data</td>
<td>M</td>
<td>S,?</td>
<td>A,M,D</td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td>Client can connect with the server to request data</td>
<td>M</td>
<td>S,C</td>
<td>A,M,D</td>
<td></td>
</tr>
</tbody>
</table>

Verification

<table>
<thead>
<tr>
<th>Verification</th>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>Client can connect with DigiD to verify the user</td>
<td>M</td>
<td>S,C,D,?</td>
<td>A,M,D</td>
<td></td>
</tr>
<tr>
<td>F9</td>
<td>Server can connect with DigiD to verify the user</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F10</td>
<td>User can authenticate multiple devices</td>
<td>C</td>
<td>S,C,D,?</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>F11</td>
<td>Block users</td>
<td>C</td>
<td>S,D</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

PINs

<table>
<thead>
<tr>
<th>PINs</th>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F12</td>
<td>User can choose a PIN</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F13</td>
<td>User can login with PIN</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F14</td>
<td>User can change PIN when logged in</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F15</td>
<td>User can restore PIN when forgotten</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F16</td>
<td>User can log out</td>
<td>M</td>
<td>S,C,D,?</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

Notifications

<table>
<thead>
<tr>
<th>Notifications</th>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F17</td>
<td>Server can detect changes</td>
<td>S</td>
<td>S,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F18</td>
<td>Server can send anonymous change notifications</td>
<td>S</td>
<td>S,C,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F19</td>
<td>User can view changes</td>
<td>S</td>
<td>C</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F20</td>
<td>User can view changes by clicking notifications</td>
<td>S</td>
<td>C</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F21</td>
<td>User can ignore changes temporarily in changes screen</td>
<td>C</td>
<td>C</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F22</td>
<td>User can accept changes and never see those changes again</td>
<td>C</td>
<td>S,C,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F23</td>
<td>User can restore previously ignored changes</td>
<td>C</td>
<td>C</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F24</td>
<td>Server can send reminder anonymous change notifications</td>
<td>C</td>
<td>S,C,?</td>
<td>M,D</td>
<td></td>
</tr>
<tr>
<td>F25</td>
<td>User can report strange activity</td>
<td>C</td>
<td>S,C,?</td>
<td>M,D</td>
<td></td>
</tr>
</tbody>
</table>

Personal Details

<table>
<thead>
<tr>
<th>Personal Details</th>
<th>#</th>
<th>Functional Requirement</th>
<th>MoSCoW</th>
<th>Subsystems</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F26</td>
<td>View personal details</td>
<td>S</td>
<td>C</td>
<td>M,D</td>
<td></td>
</tr>
</tbody>
</table>

5.2 Quality Requirements

Qualitative requirements in this case mean non functional requirements that are not part of the main focus: security. These were not the most important part of the project. Table 5.2 shows the quality requirements.
### 5.3 Security Requirements

Table 5.3 shows all security requirements. Notice that there is a difference with what we researched and what we intended to implement. These requirements are focused on the security of the application or client. Further security measures are provided in the security recommendations in section 8.2. We will also go into detail about these security requirements and how they protect the user from possible attacks.

In the previous chapter we looked at classes of attacks. Table 4.1 and table 4.2 showed different ways of classifying these attacks. Table 5.4 shows which security requirements from table 5.3 help protect the application from which type of attack.

**TLS.** Transport Layer Security [15] (TLS) is the way to secure communication between a client and a server. It effectively prevents almost all MITM attacks [55]. TLS comes in different flavors.

We chose Diffie-Hellman Elliptic Curve [38] for key exchange because it is an international standard, it is safe and very cost-effective [90, 35]. It also provides Perfect Forward Secrecy [68].

Once the shared key is negotiated, we will use AES [85] with 128-bit keys and GC-Mode [58]. Intuitively 256-bit keys feel safer, but the practical security difference is minimal. There is a slight computation cost increase for using 256-bit as well. GC-Mode defines how the single-block AES cipher is applied repeatedly to encrypt multiple blocks of data. There are several alternative modes, but GC-Mode is a good choice when integrity and authenticity are important [88]. GC-Mode also prevents a few attacks that were particularly successful against TLS [80].

SHA256 is a hashing function that is used to hash the data that is sent, the hash is added to the actual data. This hashed part of the package can be checked to see if the data has not been altered. We use SHA256 because it is a standard post-2010, but in the future the SHA-3 family might be a better choice [25].

As for verifying the servers identity, we should check if the certificate used by the server is still valid and if the signature is correct. If any of these two criteria fail, the client should terminate the connection immediately.
# Table 5.3: Client Security Requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Client Security Requirements</th>
<th>MoSCoW</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>TLS Connection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>Client forces TLS connection</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S2</td>
<td>Client TLS forces Diffie-Hellman Elliptic Curve</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S3</td>
<td>Client TLS forces AES 128 GC-Mode</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S4</td>
<td>Client checks certificate validity</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S5</td>
<td>Client checks certificate signature</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S6</td>
<td>Client terminates connection when unsafe</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>Client checks root for invalid authorities</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>2</td>
<td><strong>PINs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>User is locked out after three failed login attempts</td>
<td>M</td>
<td>M,D</td>
</tr>
<tr>
<td>S9</td>
<td>PINs are five digit</td>
<td>M</td>
<td>C,M,D</td>
</tr>
<tr>
<td>S10</td>
<td>Weak PINs are not permitted</td>
<td>M</td>
<td>C,M</td>
</tr>
<tr>
<td>3</td>
<td><strong>Sessions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>A combination of PIN and authentication token is necessary to start a session</td>
<td>M</td>
<td>C,M,D</td>
</tr>
<tr>
<td>S12</td>
<td>Without a session, no data can be requested</td>
<td>M</td>
<td>M,D</td>
</tr>
<tr>
<td>S13</td>
<td>Logging out makes session invalid</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td><strong>Data storage</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S14</td>
<td>Local data is stored with Shared Preferences Context.MODE_PRIVATE</td>
<td>M</td>
<td>T,M</td>
</tr>
<tr>
<td>S15</td>
<td>Data stored is kept to a minimum</td>
<td>M</td>
<td>C,M</td>
</tr>
<tr>
<td>5</td>
<td><strong>Android Specific</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S16</td>
<td>Allow only necessary permissions</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S17</td>
<td>Export activities only when necessary</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S18</td>
<td>Export services only when necessary</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S19</td>
<td>Intents have specific destinations</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S20</td>
<td>Variable intents are checked</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S21</td>
<td>AllowBackup set to false</td>
<td>M</td>
<td>C,T</td>
</tr>
<tr>
<td>S22</td>
<td>Application is signed</td>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>6</td>
<td><strong>Additional</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S23</td>
<td>No back door access to application</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S24</td>
<td>All testing accounts are removed</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S25</td>
<td>Used libraries are secure</td>
<td>M</td>
<td>C,R</td>
</tr>
<tr>
<td>S26</td>
<td>Incoming data is checked for correct format</td>
<td>M</td>
<td>A,M</td>
</tr>
<tr>
<td>S27</td>
<td>Package size is checked</td>
<td>M</td>
<td>R</td>
</tr>
<tr>
<td>S28</td>
<td>Avoid hardcoded credentials</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S29</td>
<td>Remove too verbose error messages</td>
<td>M</td>
<td>C,M</td>
</tr>
<tr>
<td>S30</td>
<td>Remove debugging/test code</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S31</td>
<td>Screen capture disabled in app</td>
<td>M</td>
<td>C,M</td>
</tr>
<tr>
<td>S32</td>
<td>Secure creation of files</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S33</td>
<td>Be careful writing to shared files or variables</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td>S34</td>
<td>Code obfuscated using ProGuard</td>
<td>S</td>
<td>C</td>
</tr>
<tr>
<td>S35</td>
<td>Show security tips in app</td>
<td>C</td>
<td>M</td>
</tr>
<tr>
<td>S36</td>
<td>Set Permission protection levels</td>
<td>S</td>
<td>M,R</td>
</tr>
</tbody>
</table>
Table 5.4: Requirements to counter types of attacks

<table>
<thead>
<tr>
<th>Attack</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>User targeted attacks</td>
<td>S16, S21, S22, S28, S29, S30, S34-S36</td>
</tr>
<tr>
<td>Device targeted attacks</td>
<td>S15</td>
</tr>
<tr>
<td>Application targeted attacks</td>
<td>S8-S14, S16-S34</td>
</tr>
<tr>
<td>OS targeted attacks</td>
<td>S15, S25, S32, S33</td>
</tr>
<tr>
<td>Network targeted attacks</td>
<td>S1-S7, S25, S29, S30</td>
</tr>
<tr>
<td>Misconfiguration exploits</td>
<td>S4, S5, S16-S18, S21, S32, S33</td>
</tr>
<tr>
<td>Kernel flaws exploits</td>
<td>S7</td>
</tr>
<tr>
<td>Buffer overflow exploits</td>
<td>S27</td>
</tr>
<tr>
<td>Insufficient authentication validation</td>
<td>S1-S13, S16-S20, S22-S24</td>
</tr>
<tr>
<td>Insufficient input validation exploits</td>
<td>S3, S20, S26</td>
</tr>
<tr>
<td>Symbolic links exploits</td>
<td>S32</td>
</tr>
<tr>
<td>File descriptor exploits</td>
<td>None</td>
</tr>
<tr>
<td>Race conditions exploits</td>
<td>S33</td>
</tr>
<tr>
<td>Incorrect file/directory permissions exploits</td>
<td>S15, S16, S23, S24, S32</td>
</tr>
<tr>
<td>Social Engineering</td>
<td>S22</td>
</tr>
</tbody>
</table>

because there is no guarantee that the server is who he claims to be. Trusting expired or unverified certificates is
the easiest way to nullify the security of TLS.

When all these criteria are met, the communication between the app and the server is secure.

**PINs.** User is locked out after three failed login attempts. This is a pretty standard counter measure to prevent
brute force attacks. When the user is locked out, he cannot try to login for a few minutes.

PINs being five digits was a design choice copied from the ING and ABN-AMRO bank application. Though
five digits is not a lot, there is also a limit to what users can remember and find easy to use. This is a trade-off and
we decided to stick to the design of the ABN-AMRO app and the ING banking app.

Checking for weak PINs is a logical step as well. Any hacker can guess the PIN 12345 so, for the users
protection, these types of PINs are not allowed. Again, this is a trade-off between usability and safety, but we
chose all sequential five digit combinations as well as all single number codes like 11111. This is a counter measure
to password guessing.

**Sessions.** Sessions are used to create a dialogue between the server and the client. During this session, personal
data can be transferred. Outside the session, no personal data can be transferred. To start a session, the client
needs the users PIN and their authentication token. Missing any of these two and the server will not start a
session, meaning no data can be requested. This prevents attacks where a third party application might have
extracted the authentication token for example. Sessions protect files and enforces permissions. Session cookies
are stored in memory.

When the user logs out manually, the session is killed automatically. The prevents damage if the session cookie
is obtained in some way and limits the time attacks have to exploit a valid session.

**Local storage.** The best way to protect data on a device, is to not have any data on the device. Unfortunately, this
cannot be done. When data has to be stored on Android devices, Shared Preferences [19] is the way to go. Shared
preferences stores a small collection of key-value pairs. It is important to set the Context.MODE_PRIVATE so
the pairs can only be accessed by the application. Context.Mode_PRIVATE is a good example of enabling the
right configurations. This makes reconstructing the application a little harder, protecting the user from potential
reconstructed fake applications.
**Android specific.** One of the major issues with Android is the coarse granular permissions [60]. Permissions are very broad and the user must accept all permissions to use an application. With more permissions come more exploits. It is because of this that we need to use only the necessary permissions. This protects against attacks targeting misconfigured applications. Also, if a fake version of this app is created, it will be easiest to spot by the user if it asks for more permissions. If the permissions are kept the same in a fake app, it can do a little less harm than with all permissions enabled.

Exported Activities are Activities that can be called by other applications. The easiest attacks exploiting this can bypass security checks like PINs easily. Setting the exported property to false for most screens protects the application from some basic attacks. The same goes for services.

Intents are calls to other applications, Services or Activities. When these are made variable, they become a target for injection attacks. Intents with specific destinations cannot be injected, thus protecting the user from being dragged to malicious apps or websites. This is a configuration matter. However, sometimes you need to use variable intents. An example of this would be the DigiD verification URL. With these intents, it is crucial to check whether these URLs are indeed correct before sending the user there. This reduces the risk of insufficient input validation exploits.

AllowBackup is a setting in Android applications that determines whether applications can be backup-ed. Allowing for the creating of these backups would make it significantly easier to reconstruct this applications. By default, this is set to TRUE, so it is important to check this.

Signing an application is vital for security, because it is the easiest way to determine whether an application is indeed your application or a fake/reconstructed version. Though the effects of this are not within the scope of this assignment, it is definitely a must for future developers.

**Additional security.** A back door can be an interesting point of entry for many attackers. To prevent unnecessary weaknesses, the application should not contain any back doors. The developers should not have access to the application while it is in use anyway. A similar thing is true for testing accounts. Testing or default accounts left in by the developers are often easy ways to get access to an application. Removing all test code and data before launching is a good practice in any case.

The usage of libraries is highly recommended when implementing security for applications, because these libraries have been tested thoroughly. It is up to the developers to check if they are indeed using safe libraries. Unsafe libraries can cause multiple exploits depending on the library and flaw.

Insufficient input validation is a core category in the attack vector classification 4.2. When data is read by the app, it is important to check if the format equals to what is expected.

Checking for the size of incoming packages can prevent buffer overflow which could otherwise disrupt the application. This is a problem especially for low level platforms.

Hardcoded credentials have many drawbacks. Once the code is reverse engineered, all these credentials are known to the attacker. This leads to more exploitation opportunities or, in the worst case, plain access to an application.

Gathering information is an important part of the attackers routine. Often they will send fake packages to see what the application sends back. If the application sends error messages that contain too much information, this could lead to easy exploits. The same goes for test responses.

There might be attacks where a third party app tries to make screen shots when the application is running. Disabling the screen capture will stop these attacks. On top of that, the user is prevented from storing sensitive information on their phone.

Symbolic link attacks strike when programs create files in an insecure manner. If we were to create files with the application, security measures must be in place to prevent these types of attacks.

Careful management of shared variables and files is key to prevent race conditions. This is also important for the proper functioning of the program, but we emphasize the possible security risks that come with it. Either not sharing variables or using synchronized [31] methods helps prevents race conditions from occurring.

Code obfuscation is the practice of replacing code with harder to read code, such as randomized variable names. This makes reconstruction a little bit more difficult. A persistent hacker would find ways, but it is a way to
prevent just anyone from looking inside your application.

The user is often the biggest threat to their own privacy. People are unaware of what applications they install exactly or go to unsafe websites without realizing that they are unsafe. When presented with permissions, the user almost always agrees. This is the reason most people get hacked. Users are unaware of what potential threats might be and how they can protect themselves against these threats. To ensure the safety of the users data, it is important to educate the user in the basics of cyber security.

The protection level for permissions indicate how the user is made aware of the permissions the application requires. Informing the user on this might help them spot fake applications later on.
6. System Design & Implementation

We will now explain how we designed and implemented the MijnID application. The requirements chapter gave a detailed list of what we Must, Should and Could implement. Throughout this chapter we will refer to these requirements. We first explain the bigger picture. This illustrates how components work together and allows us to define some terms that are shared across multiple subsystems. We then discuss the design and implementation of the client and the dummy environment.

6.1 Global System Design & Implementation

The design of the MijnID authentication system is based on the mobile applications by the ABN-AMRO and the ING banks. These applications use a special card reader to verify the user. The user enters their PIN and a code provided by the application. Afterwards the card reader gives a response code. After feeding the response code back to the application, the verification is complete. The user then stores a personal 5 digit PIN to authenticate themselves in the future. The difference between the MijnID application and the banking application lies in the verification process. This is done via DigiD and not a card reader. After logging in via DigiD, the user stores a PIN containing 5 digits for future use (S9). In our case DigiD is the one who verifies, returning whether the user logging in is really who they say they are. If a user has successfully verified with DigiD, then the users BSN will be returned. Once the user has authenticated and logged in with a PIN, the user should be able to view their data and changes in the GBA-V. So the main features of the application are:

1. one time verification mechanism with DigiD.
2. complete PIN system.
3. change detection and notification system.
4. data retrieval mechanism.

The global system requires a few core components that communicate with each other. Before we go into detail on each of the components specifically, let us look at how they communicate with each other. Figure 6.1 shows the main components and how they talk to each other. Purple indicates dummy components which correspond to requirements F2. Blue represents the MijnID app or client which fulfills F1. Our user is red, green is Google Cloud Messaging (GCM) and yellow is the DigiD testing environment. Our only security concern in this picture is the lines that are directly connected to the MijnID app.

**Messages & Requests.** The client and server have a unique way of communicating with each other. Whenever the client needs to communicate with the server, it uses a Request. The client wraps this Request in a Message and sends that to the server. The server in turn replies with a Message.

Messages are strongly typed data structures. Messages are converted hash maps with key-value pairs. All possible keys that the client expects to find in a Message are defined in the ClientMessageChecker. The server has a ServerMessageChecker. These checkers parse Messages, look for invalid input and convert the Messages into usable hash maps (S26).

Request are also strongly typed data structures that correspond to a type of information that the client wants. Each interaction with the app that requires data from the server in some way has a specific Request (F7). The Request types are Verify, Post-Verify, StorePin, Login, Data, Changes, ChangePin, ForgotPin and Logout. The server
also accepts Block and Post-Block-Verify, but these requests are not made from the client. Each of these Requests is initiated with the information the server needs to process the request made by the client. A Message from the client to the server containing a Request is simply called a request. Requests between other components can also be called requests, but Messages are only used between the client and the server.

When requests fail, a Message is sent by the server with a specific MiddlewareCode. This code represents what went wrong, without revealing any server implementation details to the client (S29). The client can directly translate these MiddlewareCode into correct error messages.

The data that is expected by the client and stored by the server is also predetermined. Data is also stored in a Message and then sent to the client. Because all possible data formats are defined, checking the data for injection is easier. All of these predefined structures are shared between the client and the server.

**User authentication: DigiD and PIN.** The verification process uses the connection between the client, the server and DigiD. DigiD only supports web applications and not mobile applications. After consulting Logius it became clear that logging in with DigiD is only allowed in a web browser. This means the application should start up the browser during the registration, and return to the application after logging in. The full process step by step (F8, F9):

C = MijnID Client, S = dummy Server, D = DigiD server

1. C sends a Verify request to S
2. S sends a verify request to D
3. D sends a reply with a unique URL for C to S
4. S sends the URL back to C
5. C opens browser with the URL and logs into DigiD
6. C gets redirected to our landing page on S and gets credentials
7. C sends the credentials to S using a Post-Verify request
8. S sends the credentials to D
9. D sends a BSN to S
10. S sends a verification successful reply to C
11. C gets redirected to the DigiD landing screen to register a new PIN
12. C sends a StorePin request to S. This includes a unique app ID.
13. S adds a new entry in the user database, generates a new authentication token and sends it to C
14. C stores the authentication token and gets redirected to the Login screen.

This process can be repeated for any number of devices as long as the app ID is unique (F10). Large parts of this verification process are also used for account recovery or account blocking. Step 1 to 10 from 6.1 are the same. For blocking accounts, the Block and Post-Block-verify requests are sent instead of Verify and Post-Verify.

A classic PIN system allows for the storing of PINs (F12), restoring PINs (F15), logging in (F13), changing a PIN (F14) and logging out (F16). The first two are heavily dependent on the verification process. Logging in, changing a PIN and logging out is done by sending a Login, ChangePin or Logout request to the server.

With this PIN also comes a session system. This session system helps with the verification process, but its main purpose is to force the client to provide correct login credentials without having the user logging in every ten seconds (S11). No data or changes can be requested without having a valid session token (S12). This token is only provided to the client if a valid login and authentication token is provided. The session automatically becomes invalid after 15 minutes of no requests. Logging out invalidates the session (S13) and deletes the session token on the device and on the server.

Change Detection, Notifications and Data Requests. The dummy server and dummy database are connected and exchange data in the default way (F6). The database contains a table for user accounts and a table to represent GBA-V [28] (F3, F4). The GBA-V is a governmental database that contains all the personal details of Dutch citizens. Normally the GBA-V has a system to detect changes this called the Wijzigingen Dienst, but we did not have the right security clearance to access it. So we had to create a system that detects changes. The database uses a trigger function to log changes. The server, in turn, parses the change logs, orders them, stores the parsed changes in a new table and sends notifications to the right people (F17).

This is where Google Cloud Messaging (GCM) comes into the picture. GCM allows the server to talk to a client with a specific app ID. As mentioned before, this app ID is stored during the initial registration process. All the server has to do is tell GCM to send a certain message to a device that runs an application with the specified app ID. This message does not contain any real information (F18). Since this is the only use for GCM for this application, once the client receives a message from GCM, it knows that a change has been detected.

Once the user has been notified, the client can request to see what actually changed. Requesting data is done with a Data request and the Changes request returns a list of all recent changes. RemoveChanges allows the user to acknowledge that certain changes are correct and can be ignored in the future.

6.2 Client Design & Implementation

An overview of all the classes and is provided in figure 6.2. The application shows all the Activities, the Wrong-PINService and the MijnIDApplication controller class. The MijnIDApplication contains a ConnectionManager that handles all communication with the server. Communication with DigiD is handled via the web browser. After verification DigiD redirects to a web page which runs on the dummy server. Communication with GCM can be used by importing the Google Play library and is handled in the MijnIDApplication. The various Activities in the app send Requests to the ConnectionManager. The ConnectionManager creates a TLSTask which sends converted Requests to the server and waits for a reply.

Activity flow. Figure 6.3 shows the flow of the client Activities during registration. It is important that the user verifies with DigiD and registers a PIN during the first use. We made the FirstUseScreen which only launches if the user has not registered yet. During registration, the user opens a browser to verify with DigiD. After being redirected by DigiD, the user is brought to the DigiD_LandingScreen. This is where the PIN is chosen. The
Figure 6.2: An overview of all main classes and flow in the client

Figure 6.3: Registration process
From left to right: FirstUseScreen, DigiD login page, DigidLandingScreen and LoginScreen
user is then brought to the LoginScreen to log in. After logging in, the application opens the MainScreen. This is our navigation screen through which the rest of the application is reachable. The options from this point on are displayed in figure 6.4. The user can choose to view the contents of several predefined data sets, view recent changes or change their PIN. The predefined sets currently consist of “person data” and “address data”. Clicking on one of these sets sends a request to the server to get the data. A DataScreen opens with the right data loaded into it (F26). Choosing to view recent changes opens the ChangeScreen in which changes will be shown. Within this change screen, changes are displayed in a list made of Checkboxes. The user can choose to temporarily hide certain changes from view (F21), restore the full view (F23) or to remove changes completely (F22). This last option should be used when the user knows the changes made were legitimate. The last button leads to ChangePincodeScreen, an Activity that lets the user change their PIN. This is done by entering the old PIN once, and the new PIN twice. Figure 6.4 shows these Activities. These screens are only accessible if the client has a valid session token. If the server tells the client the session token is invalid, the user is forced back to the LoginScreen.

Notifications are received through GCM. Once the notification arrives, it is displayed in the notification area of the device. This notification, as we mentioned earlier, does not contain specifics about the user. “Er zijn wijzigingen gedetecteerd” will be shown to the user. Next to this message the GCM API allows us to let the device vibrate and make a notification sound, after setting the corresponding permissions.

If the application is uninstalled, all of the data is removed and the user does not receive notifications any more. After registering again, a new app ID will be generated and sent to the server during registration.

Secure connection with the server. The ConnectionManager manages the connection between the client and the server. The ConnectionManager can accept a Request. Upon receiving a Request, a new TLSTask will be created and a request will be sent. The TLSTask encrypts the requests. TLSTask is implemented with the javax.net.ssl.HttpsURLConnection. This HTTPSURLConnection checks the validity and signature of the server certificate. If this does not match the certificate in our keystore, it will reject the connection (S4, S5, S6). This keystore is generated using the java.security library. Apart from the certificate, the hostname is also checked. To enforce TLS version 1.2 with DHEC, AES 128 GC-Mode and SHA256, we implemented a custom SecureSSLSocketFactory by extending the SSLSocketFactory. SecureSSLSocketFactory creates SSL sockets that only allow specific protocols and cipher suite (S1, S2, S3).

Our cipher suite is only supported by API version 20+. We wanted to support devices with API 18+ so we
used the SpongyCastle library. SpongyCastle is an Android port of the BouncyCastle library. BouncyCastle provides secure implementations of many cryptographic algorithms, including our chosen algorithms.

**Secure connection with DigiD.** The risks that come with DigiD lie mostly in the parties that use it. If the safety requirements DigiD provides are not met, the system will not be secure. A good implementation of the service is very important because part of its security relies on this.

We received documentation regarding the implementation of DigiD. DigiD does not provide a Android API to work with. We had to make a secure connection between the application and the Chrome browser ourselves. This is done using the appropriate *intent filters*. These are filters defined in the *Android Manifest* to check whether the browser was a valid source and contains the right parameters when redirecting back to the application. Opening the browser is done by requesting the URL from our server which is encrypted. When you log in using your DigiD, a secure connection is formed between the client and DigiD. This means all of the data sent is also encrypted.

Apart from this, we also had to catch all of the errors provided by DigiD in their documentation. After logging in, DigiD returns certain parameters, indicating whether the login succeeded or not. These parameters will help us indicate what kind of error occurred and take the appropriate measures as described in the documentation.

**GCM security.** Our implementation regarding the GCM only allows us to receive messages from the GCM server. These downstream messages are SSL-encrypted and bound to the specific installed application. We did not implement upstream messaging. As we mentioned before, the messages sent by the server do not contain sensitive information. If the user clicks on a notification, the application opens and starts the *LoginScreen*, after logging in the user will then be send to the *ChangeScreen*, where his changes will be shown (F19, F20).

**Other security measures** We minimized data stored on the device itself. This reduces the damages when the application is attacked. All personal data viewed by the user is requested from the server every time. No personal data is stored locally. The only data that are stored on the device are:

1. an authentication token
2. a session token
3. an app ID

To ensure the safety of the authentication token and app ID, they are stored in Shared Preferences. This is a private space on the device only accessible by the application. The session token resides in memory and becomes useless after the session is terminated by the server. Memory is protected by the Android OS.

We implemented a strength constraint for PINs. This means that we do not accept PINs consisting of only one digit like “1111” or sequential numbers like “45678” and “98765”. Any PIN with patterns like these will be denied. The MijnID app runs one Service called *PincodeWrongService*. This is a security measurement preventing brute force attacks from the application. If a faulty PIN is entered three times, the Service is activated and blocks every *LoginRequest*. The user is not allowed to log in for 15 minutes. The time is stored in Shared Preferences. If the app or Service is destroyed, the timer will remain valid. Re-installing the app does destroy the timer, but this would also force verification with DigiD.

The settings in the Android Manifest file are very important for security. We ensured that only the Activities and Services that had to be available to other applications were exported. The *FirstUseScreen* has to be available for launch. The *LoginScreen* is required when a notification is pushed and users need to log in to view data. The *DigiDLandingScreen* must be callable from Chrome during the registration process. All other Activities and Services cannot be called directly. We made sure that Intents have specific destinations when starting Activities or Services. The only variable Intent is the one that leads the user to the DigiD website. This is a URL that is sent by the server. This Intent is checked in three ways. Before the Intent is executed, the URL is checked. During the verification process, Chrome will check the certificate of DigiD. On return to the app, the client will check the return values. To make deconstructing the app as hard as possible we made sure that the AllowBackup property is set to false.
After creating the app, all test code, test accounts and back door access was removed (S23, S24, S30). The client is implemented using Android, Java and Google Play libraries. Apart from this we used the Spongy Castle library. Security for these libraries was last evaluated on 12th of February 2015. No known issues were found (25). All Messages and Data constructs have a predefined format and are checked by a ClientMessageChecker before being used by the client (S26). Because our application runs on Android, which runs on Java, the JVM will automatically check for buffer overflows [77] (S27). The only credentials that are used in the MijnID app are either stored on a remote database or safely as discussed earlier. Naturally, there are no hardcoded credentials (S28). We also disabled screen capture in each screen (S31). Symbolic link attacks can only occur when an application creates files, which MijnID does not do (S32). Variables are shared as little as possible and synchronized methods are defined to write to these variables (S33), when dealing with concurrent methods. Synchronized methods prevent concurrency issues and race conditions.

6.3 Dummy Server Design & Implementation

The application will be functional in a complex environment. To properly build this application, a good representation of the future environment has to be build. The real environment is a massive network of government databases and servers, so we only went so far to mimic this environment. This dummy environment includes a server and database shown in purple in figure 6.1. The server has to support all features implemented in the client. The database should store user account data and personal details similar to GBA-V. An overview of the environment design and flow is shown in figure 6.5.

![Figure 6.5: Overview of all classes and flow in the dummy environment](image)

**Supporting features** There was no version of the server yet, so there was nothing we could copy. Security was not taken into consideration, so we chose easy solutions to support the client. Tomcat was an obvious choice because it runs Java. We use PostgreSQL as a Database Management System (DBMS). The server and database had to support the client with the following features:

1. TLS
2. DigiD verification
3. PIN system
4. Sessions
5. GBA-V dummy
6. Change detection and notification

7. Data and Changes requests

**Communication.** To handle the TLS with the correct cipher suite, we had to do two things. First we had to enable TLS in Tomcat. After that we had to build a Servlet class that handled all incoming TLS connections. This was done with the `javax.Servlet` package. A self signed certificate and keystore are used which were generated with Java KeyTool. This self signed certificate is installed on the client (S4, S5). The connection with DigiD is handled by `DigiDRequester`. DigiD officially has two way authentication, meaning both parties need PKI-Overheid certificates. In the test environment, DigiD also accepts self signed certificates. The connection with GCM is made in the `ChangeNotificationThread`. This is done with the help of the `com.google.android.gcm` package. The connection between server and database is implemented using the PostgreSQL database driver for Tomcat. The `java.sql` package is used to manage the connection.

**Authentication.** The full registration process is explained in section 6.1. The server has to store data and provide the client with the right information. First the server must accept a `Verify` request, requests a login URL from DigiD and send it back. The `DigiDRequester` requests the URL which is sent back via the `Servlet` (F9). Next, the client will send a `Message` containing data about his verification attempt. The `DigiDRequester` checks this with DigiD. If the user provided correct credentials, DigiD confirms this and the server asks the client for a PIN and an app ID (F12). Once these are provided, the `DBManager` sends all data to the database which creates a user account (F3, F6). This user account contains an authentication token, app ID, hashed PIN and a BSN.

Whenever the server receives a `Login` request from the client, it checks if an account exists for that users authentication token. If so, the PIN is checked by comparing it with the hashed PIN stored in the database. This hashing is an additional security feature that was not in the requirements. A further explanation about this is given in chapter 8.2. If the PIN is correct, a session is started for the user (F13). The session expires after not receiving a request from the user for 15 minutes. The session is aborted when a correct `Logout` request is received (F16). Sessions were implemented using `javax.servlet.http.HttpSession`. Changing PINs is simply updating the database after the old PIN is checked (F14). Restoring a PIN, as mentioned before, works almost identical to registering the device on first use (15). The difference is what is stored in the database. With restoring a PIN, a new PIN is saved instead of generating a new user account.

**GBA-V, change detection, notification and retrieval.** To represent the GBA-V we used a data set provided by BRP to create a dummy database. It contains 247 fictitious details on 342 fake people. We considered this to be a good test set as the most important columns of GBA-V are represented in this set (F4).

Change detection and notification is handled by the database and the `ChangeNotificationThread`. First the database uses a trigger function which is linked to the table containing BRP data (GBA-V dummy). This trigger function registers when the table changes and what changes. These changes are stored in a different table. When the user makes a `RemoveChanges` request, changes are deleted from this table (F22). The `ChangeNotificationThread` is an asynchronous thread that checks the database for changes frequently (F17). It runs on a timer which can be set to any length. If changes have occurred, the `ChangeNotificationThread` will send notifications to the users using their app ID and GCM. Again, these notifications do not contain information on what changed (F18). GCM will forward the notification to the MijnID app.

Data and Changes requests are very simple. If the user provides a valid session token and data exists, data is returned. If not, an appropriate error code is returned.
7. Testing and Evaluation

Software tests have to be repeated often during development cycles to ensure quality. Every time source code is modified, tests should be repeated. This is very time consuming. This application consists of many elements, varying from GUI elements to connections and other functionality. Defining automated tests for these elements saves time because once created, automated tests can be run over and over again at no additional cost. For the remaining parts, we can use manual tests and open source tools to test. In chapter 5 we discussed the requirements of the application. They are depicted in tables 5.1, 5.2 and 5.3. These tables contain all the features of the application and how they are tested.

7.1 Automated Testing

For the automated tests, we implemented a number of test classes executing methods and comparing them to the predicted outcomes. Since we are doing Test Driven Development it was necessary that these tests worked without any errors and as fast as possible. Next to JUnit, we used Robolectric [39] which also is a unit testing framework. The Java helper class ReflectionMethods was needed to help us access private variables and methods. For each of these requirements we implemented several test classes covering them. The automated tests were indicated with the "A" in the column. The table 7.1 lets us see the requirements marked with the "A" and the corresponding test classes. All client tests were implemented with Robolectric. There is one security requirement tested using automated tests. S26 is checked by the MessageTest class.

7.2 Integration testing

Integration testing is a very important part of the project. To ensure that all components work well together, we used tools as well as manual testing.

7.2.1 Manual Tests

Like we did in section 7.1, we shall first take a look at the requirements tables 5.1, 5.2 and 5.3. In these tables, the to be manually tested features are marked with a "M" in the "evaluation" column.

Manual client tests. The functional requirements are divided into six different categories. For most of these tests we started up the application after installing it, testing these features by hand. We can find defects by playing the

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Corresponding Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>DataScreenTest, DigiDLandingScreenTest, LoginScreenTest, MainScreenTest, ChangePincodeScreenTest, ServiceInterruptionTest</td>
</tr>
<tr>
<td>F2</td>
<td>DBManagerTest</td>
</tr>
<tr>
<td>F5</td>
<td>DBManagerTest</td>
</tr>
<tr>
<td>F6</td>
<td>ConnectionManagerTest, MessageTest, RequestTest</td>
</tr>
<tr>
<td>F7</td>
<td>DigidLandingScreenTest</td>
</tr>
</tbody>
</table>

Table 7.1: Tests and corresponding classes.
role of an end user. To correctly simulate the role of an end user, we must look at all of the possible use cases. The use cases as seen in 5.1 will come in handy here. We can take all of the possible paths and test these manually. As an example, one of the branches is given in figure 7.1

![Figure 7.1: Example branch, illustrating registration sequence followed by viewing personal details](image)

By testing all paths, most of the functional requirements are tested. However, there are requirements that are not tested by testing all paths. These requirements are: F10, F17, F18 and F31.

In the case of requirement F10 "User can authenticate multiple devices", we used multiple devices to authenticate ourselves via DigiD and register. This worked fine and was registered correctly in the database. Requirements F17 and F18 both concerned detection and notification of changes by the server. We had to change some data in our dummy GBA-V and see whether the notifications were being sent or not. We tested these two different scenarios: when the user was logged in and when the user was logged out. The messages being sent were displayed. From the time a change was made to the notification arriving at the client took thirty seconds on average. We did not encounter problems here. F31 was tested by simply trying to make a screen capture on every screen. We received the expected error message on each screen.

All quality requirements tested manually are all tested using one of the methods mentioned above. Q12 and Q14 showed no consistent patterns, taking very long or short depending on the connection and device. Q13 was the most consistent. Requirement Q15 was met.

We also manually did a “stress” test by for example quickly opening and closing the application numerous times in a short duration.

Finally, the security requirements S8, S9, S10, S13 and S29 are also tested using the aforementioned methods. S26 was checked at the client by typos we made during server implementation. Apart from this, we also purposely tested client input validation (in roughly the same way).

**Manual server tests.** For testing the server side of the application, we use Advanced Rest Client Application. This allows us to send post requests to an IP address. With this we are able to send multiple HTTPS requests to our server and see what we get back. Though we used a tool, we still considered this manual testing, as the tool itself did not perform any checks. With this we were able to test F3, F6, F11-F16, S1, S11-S13. During the final tests, no bugs were found. Problems that arose during earlier test are discussed in 8.1.2.

**Manual connection tests.** We tested the security of the connection manually. The security requirements S1, S2, S3 and S4 were tested by configuring the accepting connections at the server. If the accepting connection type at server side differs from the connection type the client uses, the server will give an error. We did this for every type of connection to see if the server accepts it or not.

The security requirements S4, S5, S6 were tested using fake certificates on the server. We set our server time back in time to create expired certificates. This way we could test the validity and signature. If the client gives an error using these fake certificates, the system works correctly.
7.2.2 Tests with tools

All requirements marked with a "T" under evaluation were tested with tools. We used Drozer, Wireshark, Cain & Abel and Mallodroid.

**Drozer** Drozer checks the security settings of Android applications (S16-S21). During this project we used Drozer twice. The first run was halfway through the implementation and the second right before the end. In this section we will present our findings.

To identify the attack surface of our application, we can make use of some of the commands Drozer provides. The following command shows the attacking surfaces of the application. These surfaces are vulnerable through Android’s IPC (Inter-process Communication) resulting in leakage of sensitive data to other applications. The surfaces are the Activities, Broadcast Receivers, Content Providers and Services. One of the outcomes of our first Drozer run was showed that 4 Activities were set to be exported (meaning accessible by other apps). At this point in time, we only had 4 Activities in total. So all our Activities were accessible. We changed this in the Android Manifest. Drozer also allows the tester to open Activities. After we changed the permissions, Drozer could not perform this type of attack.

During the second test run indicated that the Broadcast Receiver we used was exported as seen in figure 7.2. This is the "receiver" for the notifications. If we leave it as is, other applications could interfere with the notifications received.

We can put the `android:exported="false"` in the Android Manifest to make sure this Broadcast Receiver is not exported. We use filters to check if the incoming notification is from a valid source. Only then, the notification is being received and displayed. The exported Activities are the ones used to access the app. All Activities that require authentication are not exported. For each of the Activities that are not exported, we can try and open them via Drozer like we did in the first run. Here we see that these Activities could not be opened. Furthermore we do not export any Services and we do not make use of Content Providers.

**Cain & Abel and Wireshark** To test the connection between the server and client (S1-S3) we used WireShark 1.12.2. It allows for packet sniffing. After noticing our connection was TLS encrypted, Wireshark became obsolete. Cain & Abel (Cain for short) can do similar things as Wireshark, but can also perform MITM attacks by ARP spoofing and faking certificates. We used these tools once to check whether we were able to intercept communications between the client and the server (S1-S6).

We sniffed the network for available hosts and our Android testing device is selected as the "victim" for an ARP spoofing. All HTTPS connections are intercepted as shown in figure 7.3. Cain also creates self-signed certificates to try and trick the victim in accepting these certificates. When we went to Google or Facebook on the Android testing device in the browser, the connection was not accepted and a warning was shown in the browser. A similar thing happened to our application. Our own applications traffic was not caught by Cain and thus could not be decrypted or analyzed. Cain could not crack our application. Also when attacking the client or server, the app no longer was able to connect to the server. When trying to attack the
server (server as victim) the same happened.

**Mallodroid** Mallodroid is a small tool built on Androguard to find broken SSL setting and certificates (S4, S5). It creates *xml* output of the suspicious code after reverse engineering the .apk-file. We exported a version of the application as an .apk-file to test with Mallodroid. This gave us the *xml* output seen in figure 7.4. The *xml* output has no parameters, indicating that there are no errors. Errors are be indicated by the `broken="True"` or `broken="Maybe"` parameter given within the *xml*. This means our SSLSockets are not broken and we verify our host names correctly.

```
<?xml version="1.0" ?>
<result package="com.wilvun.rijnuid">
  <trustmanagers>
    <trustmanagers>
      <hostnameverifiers>
        <hostnameverifiers>
        </hostnameverifiers>
    </hostnameverifiers>
    <allhostnames>
    </allhostnames>
  </trustmanagers>
</result>
```

*Figure 7.4: Mallodroid xml-output*
7.3 Software Improvement Group Feedback

The Software Improvement Group (SIG)\cite{14} gives feedback regarding the implementation of code, infrastructure, testing and so on. During this project, we had our code evaluated two times by SIG. For each of these, SIG provided us with recommendations. These recommendations can be found in appendix B.

**First feedback.** In the first week of January 2015 we sent the first version of the MijnID application to SIG. We received four stars. The recommendations for this code review showed that we had different subjects to work on namely: unit size, code complexity and code duplication. These are the parts that scored the lowest.

The percentage of methods that were too long was above average. Splitting these long methods into smaller methods made sure the code was easier to understand. This improved unit size and code complexity.

The third issue was code duplication. This was harder to counter. There was a lot of code that looked alike but was different in terms of functionality. These situations occurred many times. Where possible, we created new methods and classes to counter this duplicate code. The biggest problem with this code duplication is the implementation of Activities. These all have similar layouts, but very different methods and checks behind them. Some code duplication was cause by the default Android layout generated by Eclipse.

**Second Feedback.** One month later we sent another version to SIG for our second evaluation. Within a day we received the results that can be found in appendix B. The issues were about the Unit Size and Code Duplication like the first recommendation. SIG mentioned that the code duplication was raised by 3% since the first evaluation, and that we should look at this problem before we hand in the final code.

We tackled the unit size issue by checking every class and searching for methods that could be split up. Every Activity has a method that runs when the Activity starts. These methods were relatively long, implementing several procedures before starting up the Activity. We split these procedures up, making it a sequence of methods. Furthermore we looked at all the remaining methods used for defining several GUI elements and saw that these could also be split up into a sequence of methods.

Since SIG mentioned that we should be critical looking at code duplication, we looked at the methods that almost looked the same. For some methods, splitting up was handy because this counters duplication without compromising the functionality. After splitting up, we searched for the duplicate methods and merged these. However, there are many GUI components that have very specific checks built in. In these cases, splitting off small parts and defining more smaller functions is only a slight improvement in terms of duplication and it would hurt the code readability.

The feedback we received during the second evaluation was, in our opinion, minimal. We looked at the issues addressed by SIG, and contacted them afterwards about the code duplication in LoginScreen and ChangeScreen but could not get any details. We also sent in some JavaDoc, which was not mentioned in the feedback. We also did not receive a score for the code maintainability. With the limited response we got the second time, we felt that SIG did not give our second submission much attention.
7.4 Requirements evaluation

In table 7.2 we listed the requirements we were able to finish on time and the ones that are still unimplemented. Refer back to tables 5.1, 5.2 and 5.3 to see descriptions of the corresponding requirements. We were able to implement all Must requirements and almost all Should requirements. Functional requirements that we did not manage to implement were both Could Haves and were not implemented because of time constraints. The quality requirements were not all met. This is largely because we did not have enough time. The only security requirements that were not met are the ones that have to deal with user interaction and releasing the app. Both these aspects are handled by Milvum later on.

Table 7.2: Requirements evaluation

<table>
<thead>
<tr>
<th>Functional Requirements 5.1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Done</td>
<td>F1, F2, F3, F4, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, F16, F17, F18, F19, F20, F21, F22, F23, F26</td>
</tr>
<tr>
<td>Not done</td>
<td>F5, F24, F25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality Requirements 5.2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Done</td>
<td>Q1, Q2, Q3, Q5, Q7, Q10, Q11, Q13</td>
</tr>
<tr>
<td>Not done</td>
<td>Q4, Q6, Q8, Q9, Q12, Q14, Q15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client Security Requirements 5.3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Done</td>
<td>S1, S2, S3, S4, S5, S6, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33</td>
</tr>
<tr>
<td>Not done</td>
<td>S7, S22, S34, S35, S36</td>
</tr>
</tbody>
</table>
8. Challenges and Recommendations

In a project like this there are many challenges. Having gained experience by facing these challenges and researching solutions, we wish to give some recommendations to the future developers of the MijnID application.

8.1 Challenges

Throughout this project we were constantly reminded that security was not a topic in our bachelor curriculum. This caused two major problems. We had to research everything and we had trouble assessing whether we had sufficient knowledge to start designing. Not knowing anything about a topic means you have to research what to research. Then you have to assess if you have found all topics to research and do research on them. There is a vicious circle in this process that was hard to overcome without the help of an expert. Of course Johan was there to help, but there were moments when we were dependent on him to make any progress. During the implementation these problems haunted us further.

An example of this was during our initial research. We were confident we used a secure cipher suite, but during the integration testing in phase 1, we discovered that our chosen cipher suite didn’t have any Perfect Forward Secrecy [68]. This means that if a single private key was compromised, all keys could be compromised. We then quickly changed our cipher suite to one that did provide Perfect Forward Secrecy.

8.1.1 Planning Issues

Because of the many parties that were involved, we were naturally dependent on others. Many of our questions had to go through Milvum to other people, slowing down our process. BZK and Logius were often slow because of bureaucracy. The design team was supposed to deliver a design but failed to do so before our project was done. Our main contact with all these parties was through Milvum which slowed things down significantly. Below, we give a broad explanation of the major planning issues we encountered.

- A full list of requirements was not established until week seven. Requirements came partly from BZK, partly from Milvum and partly from ourselves. The requirements mentioned in appendix A had some unmentioned prerequisites that we did not foresee.

- We had to spend some time on figuring out which resources and platforms we had access to. At first, we had testing accounts at Microsoft Azure to host our server. We installed our server there and spent some time talking to the help desk why things did not work. Apparently, running a Tomcat server on a trial account was not allowed and we had to link our credit card. Milvum was surprised by this and we started looking for different solutions. We ended up at Digital Ocean, where most problems were fixed within a day.

- Another resource that was valuable was the PKI-Overheid certificate. With this certificate we would be able to get access to the Security Assertion Markup Language (SAML) [41] platform that DigiD will use in the future. It is also a requirement to connect to many government and municipal databases. We tried to get one, but this was expensive. We were in the process of getting a certificate via contacts of Milvum, but it would not arrive before the end of the project. This was a large part of the reason why we could not connect with testing environments for GBA-V or BelastingDienst [20]. Once we noticed this would take a long time, we decided to connect to the Common Gateway Interface (CGI) [23] platform instead, as this did not require the certificate.
• A design company would deliver designs. Next to making the application aesthetically pleasing, this would help us understand the flow of the program. Since this design had to wait, we made our own temporary GUI based on what would seem logical to display in such an application.

• One difficulty with Scrum at the start was the correct estimation of time it took to complete tasks. In the beginning we found that we easily underestimated tasks, which resulted in certain tasks having to be pushed to the next sprint. Luckily, every sprint our estimations became more accurate. During the final sprints we put in some extra work. A burnout chart can be seen in figure 8.1.

• The last major planning issue was the session system. We first created a session system of our own using encryption libraries. We encountered many errors using this system mostly decryption errors. During communication, the session token were changed in some way. We fixed this bug largely, but there were still some problems. After some thinking and feedback from Johan we decided it was best to use the session system from the standard Java library.

![](burndown.png)

**Figure 8.1:** Scrum burndown chart

### 8.1.2 Implementation Issues

**Global.** Throughout the project we did a lot of redesigning. As more requirements became clear and our understanding of security increased, we became aware that our application required new security features. New security features often meant restructuring the app. For example, when we discovered that GCM allowed us to send messages to Android devices using application IDs, the whole registration process had to include storing this
data. Refactoring usually leads to some errors. Tracking down what went wrong exactly can be hard, especially if Eclipse does not find the bug immediately. Pair debugging was often used.

**Client.** During this project, we needed a way to detect bugs quickly and fix them. In the *LogCat* [8] of Eclipse, we could filter the messages and track the bugs easier with this logger. The GCM we used had its own logger, we also used this logger to filter messages and display them in *LogCat*. However, this was not always sufficient.

One of the biggest problems we encountered was our unsupported cipher suite. Initially we used the Apache SSL library to handle our TLS, but for some reason it would not actually initiate a TLS connection. The error messages were not clear at all. There seemed no clear answer, so we ended up rebuilding the entire features with the *HttpsURLConnection* library to get better error messages. We then discovered that the Android version we were developing for did not support our specific cipher suite. We ended up using the Spongy/Bouncy Castle Library together with Java cryptography libraries to get a working TLS connection.

In the client there was an error that certain Activities were initiated when they already existed. When clicking on a notification, the client would start a new Activity, even though the app might still be running an activity. This Activity stacking was solved by setting additional parameters on intents. This way, Android would check what the application was doing before launching a new Activity, allowing us to kill or continue a previous Activity.

An interesting error occurred when the user would press the back button on the phone. This did not always link to the previous screen. This was fixed by setting the parent of each activity in the Manifest file.

![Diagram](image.png)

**Figure 8.2: First client design**

Figure 8.2 shows our first design. A lot of things changed since the initial design, largely because of new insights about security and requirements.

As seen in the diagram, our original plan was to have two services *AppManager* and *DataChangeMonitor*. *AppManager* would manage app crashes. *DataChangeMonitor* would periodically ask the server if there were any changes to the users personal details. *AppManager*’s function was later handled by the *MijnIDApplication*. The function of *DataChangeMonitor* was implemented on the server.
We also added a lot of Activities. This was done to allow for more complex control flows. Each Activity has their own permissions as to what data can be displayed and who has clearance to access that page. Some pages require a working session, some pages require the user to never have logged in at all. All of this was not fully clear in the first design. We spent a lot of time redesigning the flow of the app.

**Dummy Environment.** In the beginning we anticipated that we would have to connect to the GBA-V testing environment and our own database. Because of this, we spent some time figuring out how to dynamically connect Tomcat to databases. This would have also allowed a dynamic setup for TLS for Tomcat. This turned out to be more troublesome than expected. The feature was dropped when we heard that we were not going to get access to the GBA-V testing environment.

We moved change notification entirely to the server to reduce traffic. But we had no idea how the server could know which change belonged to which device. Or how to determine if the message arrived. This was solved once we found GCM. GCM only needs an application ID to locate the device and can ensure that the device receives the message. The downside is that our application has to send notifications via Google. It is partially for this reason that notifications contain no data.

The GBA-V test set that was provided was very useful, but that was not our first test set. Initially we had a different set, which was smaller and contained less strange values. The second set we found was a lot larger and seemed like a better representation of the actual database. It did however contain a lot of null values and strange non-ASCII values. We had to spend time manipulating it before our database accepted it.

We tried to optimize change detection. We had two options. The first option would be to have the database detect them, let the server loop over all changes and store them in a good format and notify users. The second option would be to have the database detect changes and store them in a good format and then have the server send notifications. It was a trade-off between memory usage, speed and ease of implementation.

When testing strange values (non-ASCII) the server showed incorrect errors. Specifically, when first provided with a correct login and directly afterwards with a non-ASCII login. This seemed to be a bug with the Advanced Rest Client and not our application. Nevertheless, we added an ASCII check to see if input received at the server is indeed ASCII.

When providing the server with strange data or SQL Injections, the server replied with stack traces. On further inspection, our RequestParser itself required some tuning. This used to be a class that parsed incoming requests. On inserting strange or no values to the URL, exceptions were thrown but not dealt with. We modified the RequestParser so it would only accept specific values. We also changed the parser on the client side to prevent similar issues. Later on we replaced the RequestParser with the Message-class and MessageChecker-classes, which both check the messages for incorrect values and can also handle parsing UTF8 values and not only ASCII.

### 8.1.3 Ethical considerations

With this project came a great ethical question. Can we provide appropriate safety for users and their data? If this project, based on our security insights were to make it to the market and it contained security flaws, we would be partly responsible for the damages. It was hard for us to assess whether we could provide decent enough security. Johan did provide us with his experience which helped out a lot. The problem with cyber security is that you never know what the next attack might be like. We did our research on what attackers are like, what attacks they perform, and how to counter these attacks. But if we failed in our research, it could cause serious problems for our client and for the possibly millions of users. The implementation also has to be flawless, otherwise attackers will exploit our mistakes. We built the application with the help of standardized security features, but if these are compromised than our application is at risk. It is for all these reasons that this report is so important. Through this report, a thorough documentation is given on the design choices we made. Because of this, Milvum and others working on this application will be able to tackle possible weaknesses in this system and successfully continue our work.

At the end of this project we are confident enough to claim that what was implemented was done correctly. As mentioned before, this only includes the application. We discussed the possible threats and made Milvum aware
of what to look out for during further stages of their pilot.

8.2 Recommendations

In this section we will give some broad recommendations on how to build a safe system, including server and database. We will also share ideas on functional extensions and maintainability.

We recommend the team responsible to implement the rest of this system to have a security expert on board. A security expert can give advice, check solutions or even fully develop certain security features. This project was a great exploratory process, but building large secure systems requires more expertise.

The general rule of thumb is: Do not invent your own security. When using the security implemented by others, check the source thoroughly. Keep used libraries and supporting software up to date.

Client. At the start of this project we made a list of all security requirements that were in the scope of our assignment. What follows is a list of the main possible security risks, how to counter them and some general recommendations.

- **Security**
  - At the moment it is only allowed to access DigiD with an internet browser. This means the application is dependent on the security of the browser and the security would thus be out of our control. It would be safer if it was allowed to start the session from the application itself, making it possible to enforce higher security standards.
  - With less than 100,000 pin numbers available, logging in is a feature that might be the least secure. Passwords offer more security but the reduce ease of use. Locking and unlocking a device using a fingerprint is getting more common. Using fingerprint scans to identify the user might increase the security and increase the ease of use. The security of fingerprint scans requires research before implementation.
  - Reading the memory gives access to data and session tokens. Using certain attacks the memory of the device can be extracted and analysed. This would be disastrous since the attacker could just read out the personal data that is stored in memory or get hold of a session token.
  - Session token can be stolen and used by attackers to request data. For this to happen, the session token has to be extracted from the TLS encrypted channel, the memory of the device or from the server just after the user logged in to the application. All three of these are tough to crack. Once this happens, the attacker has to use the token before the user logs out again. Most of these attacks are hard to do in real time so this is considered safe. Should the future bring faster attacks for memory access, then the session system should be modified. He can also keep the session alive by sending requests within the session lifetime, unless the original user logs out.
  - Attacks where ROOT can be access are a nightmare for any system. Though it is hard to gain ROOT access, especially through our application, but not impossible. The problem largely lies with the unaware user who can unknowingly install malware. This will make it easier for attackers to gain ROOT access. ROOT access gives attackers many options to exploit any application installed on the device. More research should be done to look for potential attacks that can give access to memory. Ideally the app should clear the memory when it notices an attack. We had insufficient time to do research on appropriate solutions.
  - An application can be installed on a device that already has fake certificates installed. When installing the app, it would be wise to check this as well. How this is done, or if it is even possible, requires research.
  - It was already mentioned in the requirements, but signing the application greatly improves the protection of an application because it makes the app distinguishable from other applications. The server will needs to be able to tell the difference between the real app and fake apps. Otherwise, fake applications could request data or try to get information from the server.
Using ProGuard to obfuscate code also reduces the risk that attackers will deconstruct the app. ProGuard settings should be checked before uploading the app to the Google Play.

The biggest improvement to the application would be to add a function that allowed users to notify authorities when unknown changes has been detected. This would point to identity fraud. Using this function, the authorities could be informed instantly with the right details on when changes occurred and where.

- **Maintainability**
  - Logging errors is good for maintainability but is currently not implemented. This is done for security reasons, as they should not be store locally. A solution where error logs are generated but not stored on the device needs to be implemented. We propose sending error reports to a remote server when errors occur.

- **Usability**
  - To make this application more user friendly, the GUI needs a lot of work. We specifically left this part out of the project to focus on other matters. We only had so much time and GUI development is more of a speciality of Milvum. Also, there was no final design ready for the GUI, so building a good one might have been a waste of time.
  - Currently the notifications are sent to user when they are detected. Once the notifications have been sent, no reminding notifications are sent later on. This might be desirable and can easily be implemented. After the notification has been sent, the changes are removed from the change log by the server. This causes them to never be detected again, so no new notification will be send. With an additional flag, this can be changed so it will repeat sending notifications until the client sends a message that it has seen the notifications or the changes.
  - At the moment any errors will cause the application to stop doing what it is doing and show an error message instead (in most cases). We recommend that the designers spend time improving the user friendliness of this process.

**Server.** The server already has a few basic features, including TLS, a session system and some input verification. But of all these features are solely implemented to create the right development environment for the client. For the real implementation, the design should be re-evaluated and improved.

- **Security**
  - A complete list of risks needs to be setup for the server. A decent list of threats can be constructed using the OWASP top 10 [93] and the CWE/SANS top 25 security errors [52]. These risks can be classified according to the AVOIDIT taxonomy same as in section 4.2. From there a security requirements list can be produces which will be a good starting point for the construction of the actual server. We will name a few common threats and solutions.
  - A solid secure system starts with a solid OS and solid supporting software. The more features an OS or servlet container has, the more entry points an attacker has. We recommend the server runs on OpenBSD [36] OS because it is a lightweight OS with a security focus [56]. Johan pointed this out to us.
  - Attacks will often scan a server for open ports and try attacks on all of them. The easiest counter to this is to minimize the amount of open ports on the server. Of course, a dedicated server should be used.
  - The current dummy version uses the TLS built into Tomcat. All traffic in an out of the server must be protected by TLS to prevent MITM attacks. As Tomcat will be replaced by a more lightweight server program/container, the TLS has to be implemented again as well. The cipher suite used in the current system can remain the same. The main change would be that the server checks the clients credentials as well. To do this, the application must be signed.
To safely store pin numbers they need to be hashed. This way, even if the database is breached the attacker still won’t have the PIN. The same PIN will always hash to the same hash code, but going from an hash code back to the PIN is impossible. The attacker would have to hash all possible PINs and check each hash code with the stolen hash code to see which one match. We have implemented hashing on the server already with the Bcrypt [78] algorithm. The Bcrypt hashing algorithm is based on the Blowfish [81] cipher and is specifically designed to hash passwords. Bcrypt is the default hashing algorithm in OpenBSD. Bcrypt protects against precomputed dictionary attacks and hardware improvements.

Similar to the client, servers are also threatened by injection attacks. To prevent this, all input must be validated. An example structure is provided by the Message and ServerMessageChecker classes. In order to prevent SQL injection attacks, PreparedStatements should be used. PreparedStatements will escape input so that injected SQL code will not affect the query. An example implementation of PreparedStatements can be found on the server.

It is important to keep the PKI-Overheids certificate for the server and keep it up to date. Even though the server might be relocated to a government owned location, it is still important to keep checking if the certificate is up to date. A system operating on expired certificates is useless.

Our final advice is to carefully manage access to the server. This is a no-brainer, but if all security can be bypassed because the admin password is admin, there is no point. Remove all default and test accounts and check for any back door access.

**Database.** The dummy database currently in use is a default implementation of PostgreSQL. This database is by no means secure. In the future, this database must be moved to a secure location with all the necessary security features.

Similar rules apply as with server design. A comprehensive list of security threats must be constructed from which security requirements can be deduced. The team responsible for designing this database should include a security expert. The most likely scenario would be that the government implements this part themselves.

It is key that the right Database Management System(DBMS) is used as one DBMS is significantly less secure than another. Among big names that can be trusted with this level of security concern are Oracle [76] and IBM DB2 [65].
Conclusion

The MijnID app helps users detect identity fraud, thereby preventing damages. Our goal was to create a proof of concept that demonstrated core functions and security features. We performed research and built the application, as well as a dummy environment to test the application. The initial design changed a lot. We frequently redesigned the app as requirements changed and our understanding of cyber security grew. With many refactoring steps came new bugs which slowed down our development process. Waiting for other organizations to give clearance or instructions also influenced our schedule. In the end we made an application that is resistant to most attacks in the current environment, but there is still work to be done. It was a great experience working on this project. We enjoyed the great environment of working in a young company with like-minded people. Working at Milvum was motivational and educational.

We used the skills that we learned during our bachelor program. We used scrum and test driven development to help guide our project. We used skills from different areas including database design, software design, project management, data structures, object-oriented programming, computer networks, software testing and scientific research. Using these skills allowed us to developed a deeper understanding of the things we already knew. We learned a lot about security, bureaucracy and the real difficulties of software projects. This took a lot of time, but in the end we were able to make clear statements about our abilities and responsibilities. We had meaningful discussions about security, design and ethics and how our project might affect others. This even changed some of our ideas about our personal future development.

There are many recommendations we would like to give to Milvum for future development of this system, but we will only name a few here. When maintaining the application it is important to refresh certificates and to keep libraries up to date. Future updates should include security features like giving the users the ability to report strange changes. Be aware that attackers will always find a way to get in eventually. Never invent your own security! Educating users is a large part of providing security. Examples of functional updates are reminder notifications and logging in with fingerprint scans.
Glossary

**Activity**  An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map.

**Android Manifest** A file where all of the Activities, Services and Application extensions come together to form the app.

**App** Application, usually used to mean an application on a mobile device.

**Application Programming Interface** Is a set of routines, protocols, and tools for building software applications.

**Authentication** The ways of a user to authenticate themselves, this could be via DigiD or via a PIN.

**Burger Service Nummer** The citizen service number (BSN) is a unique personal number allocated to everyone registered in the Municipal Personal Records Database (BRP). Also called the Citizen Service Number.

**Cipher Suite** A cipher suite is a named combination of authentication, encryption, message authentication code (MAC) and key exchange algorithms used to negotiate the security settings for a network connection using the Transport Layer Security (TLS) / Secure Sockets Layer (SSL) network protocol.

**Google Cloud Messaging** GCM for Android is a free service that helps developers send data from servers to their Android applications on Android devices, and upstream messages from the user’s device back to the cloud.

**Google Play** A digital distribution platform operated by Google.

**Integration Testing** Is the phase in software testing in which individual software modules are combined and tested as a group.

**LogCat** A log output in Eclipse.

**Logius** Governmental branch behind DigiD.

**Municipal Personal Records Database (Dutch Basisregistratie Personen)** Contains information about everyone who have lived, or is currently living in the Netherlands. Also called the Registration Municipal Personal Records database. BRP used to be called GBA.

**Municipal Personal Records Database Provider (Dutch Gemeentelijke Basis Administratie Verstrekkingsoorziening)** This is the central component database, containing the information from the BRP. BRP used to be called GBA.

**PKIOverheid Certificate** Digital certificate ensuring safe trafficking of information.

**Registration MijnID** The complete registration process for the MijnID application. This consists of a verification via DigiD and storing a new PIN for future use.
Scrum  Scrum is an agile way to manage a project, usually software development.

Service  A Service is an application component that can perform long-running operations in the background and does not provide a user interface.

Session  A session is a way to store information (in variables) to be used across multiple pages. Unlike a cookie, the information is not stored on the users computer.

Shared Preferences  Android API for saving key-value pairs.

Tomcat  This is an open-source web server and servlet container.

TU Delft  Delft University of Technology.

Verification via DigiD  The process of verifying who you are by logging into DigiD.
Acronyms

API  Application Programming Interface.
ASCII  American Standard Code for Information Interchange.
BRP  Municipal Personal Records Database.
BSN  "Burger Service Nummer".
BZK  Ministry of the Interior and Kingdom Relations.
CGI  Common Gateway Interface.
DigiD  Digital Identity.
GBA-V  Municipal Personal Records Database Provider.
GUI  Graphical User Interface.
HTTPS  Hypertext Transfer Protocol Secure.
JVM  Java Virtual Machine.
MITM  Man In The Middle.
OS  Operating System.
PIN  Personal Identification Number.
SAML  Security Assertion Markup Language.
SIG  Software Improvement Group.
SSL  Secure Sockets Layer.
TDD  Test Driven Development.
TLS  Transport Layer Security.
Bibliography


[70] Larisa April Long. Profiling hackers. SANS Institute Reading Room, 26 January 2012.


Appendices
Bedrijfsomschrijving. Milvum is een Full Service Mobile App Agency en is gepassioneerd over het creëren van succesvolle apps! Wij combineren strategie, concept, design, interactie, technology en content. Wij maken apps die er toe doen, apps die het beste presteren, apps die geld opleveren.

Project beschrijving. Het Ministerie van BZK is telkens op zoek naar manieren en methodes om alle vormen van fraude, waaronder identiteitsfraude tegen te gaan. Eén van de problemen hierbij is dat identiteitsfraude heel lang onopgemerkt kan blijven, doordat de burger niet dagelijks bezig is met zijn of haar gegevens.

Oplossing. De MijnID Android prototype App biedt de oplossing op het probleem dat identiteitsfraude lang ongedetecteerd blijft, doordat burgers geen inzicht hebben over hun informatie en over de verwerking hiervan. MijnID geeft de burger inzicht in de informatie die bekend is over hem of haar bij de Rijksoverheid. Tevens wordt hij of zij op de hoogte gehouden van wijzigingen in deze informatie, zodat een onverwachte wijziging direct duidelijk is. Omdat deze wijzigingen een indicatie kunnen zijn van identiteitsfraude, kan de burger hier snel actie op ondernemen. MijnID biedt een bijzonder snelle, reactieve oplossing voor problemen bij identiteitsfraude en vervult ook een informatieve rol, doordat de burger beter inzicht krijgt in de processen van de Rijksoverheid omtrent de burger’s identiteit.


Door de complexiteit willen we ons vooral focussen op de volgende twee onderdelen;
   A. Technisch vooronderzoek (erg belangrijk)
   B. Prototype MijnID App

Functionaliteit. De minimale functionaliteit van MijnID Android App is als volgt:

1. Je moet kunnen inloggen met DigiD om vervolgens blijvend gebruik te kunnen maken van de app

2. De gebruiker moet de App kunnen openen door te bewijzen dat hij of zij toegang tot de App heeft, bijvoorbeeld door een persoonlijke pincode.

3. (Optioneel) De App ontsluit de volgende informatie aan de gebruiker: Beschikbare informatie uit het Basis Register Persoonsgegevens (BRP)

A. Technisch vooronderzoek. Vanwege de complexiteit van de App en de systemen van de Rijksoverheid waaruit de informatie over de burger gehaald moet worden, zal er een vooronderzoek moeten plaatsvinden om de technische knelpunten en oplossingen hiervoor vast te leggen. Dit onderzoek zal minimaal de volgende punten
uitlichten:

1. **Inloggen via DigiD.** Het inloggen met DigiD is iets wat nog niet geformaliseerd is voor mobiele Apps. Daarom is het noodzakelijk om met de betrokken instanties te onderzoeken wat een veilige manier is om op een gebruikersvriendelijke manier de gebruiker te laten inloggen met DigiD. In het vooruitzicht is gesproken over het model van huidige Bankieren Apps, zoals die van de ABN AMRO en de ING Bank, als voorbeeld te gebruiken.

2. **Koppeling met BRP.** De App moet een koppeling maken met het BRP (Basisregistratie Personen), om zo deze informatie (read-only) aan de burger te kunnen tonen. Hiervoor moet de structuur van deze informatie bij het BRP geanalyseerd worden, om zo inzicht te krijgen hoe deze informatie efficiënt en veilig in de App getoond kan worden. Verder moet uitgezocht worden, gezien de wens om ook inzicht te krijgen in wijzigingen van deze informatie, of er mogelijkheden zijn om deze wijzigingen te detecteren.


4. **Veiligheidseisen app.** Omdat de App privacy-gevoelige informatie ontsluit, is het belangrijk dat deze informatie goed beschermd wordt. Hiervoor moet veiligheidseisen opgesteld worden die gebruikt kunnen worden als advies voor de betrokken instanties, zodat de App volledig beschermd is tegen misbruik.

**B. Prototype.** De gebruiker kan uiteindelijk zijn of haar informatie bekijken via de MijnID app. De App toont deze informatie op een veilige en heldere manier, zodat de gebruiker direct kan zien of er iets vreemds aan de hand is. Er wordt gezorgd dat de App zo veilig mogelijk is, waarbij de precieze eisen zullen blijken uit het vooronderzoek. In ieder geval zal de gebruiker moeten inloggen met DigiD en moet de gebruiker bij herhaald bewijzen dat hij of zij toegang heeft tot de app. De App zal dus de volgende functionaliteiten hebben;

1. Inloggen via DigiD
2. Gebruiker identificeren bij herhaald gebruik
3. (Optioneel) Het tonen van informatie over de burger
4. (Optioneel) Wijzigingen melden aan de gebruiker
5. (Optioneel) Geanonimiseerde push notifications veilig versturen.

**Platformen.** Om de app voor een zo groot mogelijk publiek beschikbaar te maken, maar toch vat te houden op de ontwikkelteun en -complexiteit, wordt de app ontwikkeld voor Android 4.1 en hoger.

**Extra informatie** In dit project zal gebruik worden gemaakt van Agile Scrum met sprints van twee weken. De eerste twee weken worden gebruikt voor planning, het schrijven van nodige plannings documenten en voorbereiding op de eerste sprint. Daarna volgen er drie sprints (6 weken) voor het programmeren. De laatste twee weken zullen besteedt worden aan afronding en de presentatie van het project. In deze laatste twee weken zal alle code en de documentatie gerevieuwd worden. Elke student wordt geacht 40 uur per week te werken.
aan het project. Als het prototype voor de MijnID app goed functioneerd zal de technologie gebruikt in het project ook gebruikt worden in de officiele app. In dit geval zal Milvum de ontwikkeling overnemen en voortzetten.

**Project team.**
Supervisie:
- Salim Hadri, salim@milvum.com, 06 81 47 37 36
- Johan Pouwelse, peer2peer@gmail.com, +31 15 2782539

Student team:
- Albert ten Napel, 4087798, aptennap@gmail.com, 06 57 17 19 83
- Anish Narwarde, 4013085, anish.narwade@gmail.com, 06 29 16 50 12
- Thomas van Helden, 4106725, t.p.vanhelden@gmail.com, 06 42 05 84 32
SIG feedback (Dutch)

First feedback

[Aanbevelingen]
De code van het systeem scoort net 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size, Unit Complexity en Duplication.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de ‘changePincode’-methode, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaar zoals bijvoorbeeld ‘// check if the current pincode is valid’ en ‘// Check if new pin code is equal to the repeated one’ zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methodes ook naar voren als de langste methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Voor Duplicatie wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren. In dit systeem is er met name duplicatie te vinden in de ‘ChangePincodeScreen’ en ‘MainScreen’. In het laatste geval zijn bijvoorbeeld de methodes ‘onResume’ en ‘onRestart’ vrijwel hetzelfde. Door het gedeelde stuk naar een aparte methode te halen zorg je ervoor dat de functionaliteit consistent blijft en de code makkelijker te begrijpen is. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Second feedback

[Hermeting]
In de tweede upload zien we dat de omvang van het systeem bijna is verdubbeld, maar dat daarbij de score voor onderhoudbaarheid is gedaald. Deze daling is met name toe te schrijven aan een lagere score voor Unit Size en Duplicatie.

Voor Unit Size zien we dat de toegevoegde methoden relatief lang zijn. Ook zijn de bestaande methodes zoals ‘changePincode’ niet gerefactored, ondanks dat er wel veranderingen in zijn aangebracht. Qua duplicatie zien
we een groei van 3%. De nieuw toegevoegde classes zoals 'ChangeScreen', 'LoginScreen' en 'DigiDRequester' bevatten interne duplicatie, bijvoorbeeld in de implementatie van de verschillende 'MessageCheckers'.

Als laatste zien we een kleine stijging van de hoeveelheid test-code.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie niet zijn meegenomen in het ontwikkeltraject. Het is sterk aan te raden om voor de laatste oplevering in ieder geval kritisch naar de duplicatie in het systeem te kijken.