Mobile Application Security

an assessment of bunq's financial app

Bachelor End Project

Authors
K.Q. Lampe
J.C.M. Kraaijeveld
T.D. den Braber

Date
June 2015

TU Delft Coach
Dr. ir. S.E. Verwer

TU Delft Bachelor Project Coordinators
Dr. ir. F.F.J. Hermans
Dr. ir. M.A. Larson

Company Coach
W. Van

Preface

This report was written as part of the Bachelor Computer Science at the Delft University of Technology. The report contains all information, struggles and achievements of our project that was carried out over a period of ten weeks. The project was created by bunq, a technology company situated in Amsterdam, with the goal of testing their security. When we started with this project we had no experience with software security or mobile application development. Learning about software security and mobile application development was an important goal for ourselves.

The project consisted of research into the security of mobile apps in general and the assessment of the bunq apps in particular. The aim of this report is to inform the reader about the accomplishments made during the project, but it also contains recommendations on mobile application security. The recommendations are included, because security is not something that applies to a single development phase; it is also important during further development.

We would like to thank everyone at bunq for their support and kindness during the project. A special thanks goes out to Wessel, Sicco, Esan, Djoeri and Ali for learning us numerous things about coding, writing, unix and entrepreneurship. Finally, we would like to thank the table tennis table at bunq that pulled us through the hard times during the project.

June 26, 2015
Kees Lampe
Michel Kraaijeveld
Tom den Braber

Summary

Disclaimer: All found bugs are fixed. The app is secure on all the aspects that we examined.

bunq is an IT company that is developing a financial appliction for both iOS and Android. With all IT projects security is an issue, and especially with banking applications security is a very important aspect. The goal of this project was to assess the security of bunq's mobile applications. In order to achieve this goal, two applications were made that can perform security tests on mobile apps.

A taxonomy of possible attacks on the applications was used to describe the scope of the assessment. The scope of the security assessment focuses on the applications itself and not the network connection or the backend with which the apps communicate.

Both the Android and iOS operating systems have some basic measures to secure their apps. The measures of iOS are a bit stronger than those of Android. A first measure both operating systems use, is sandboxing to protect their apps from the intrusion of other apps installed on the same device. Signing the apps in both Android and iOS is another measure that prevents anyone to alter the app without knowledge of the private key that was used to sign it. Data that the app stores on the device needs to be as little as possible and well protected. To achieve this, developers can make use of the Android Keystore and iOS Keychain, which are encrypted data storages.

Additionally, certain security issues aimed at Android and iOS are discussed. Tapjacking is such an issue and can be preformed on Android by tricking the user into making UI interaction with an app without their knowledge. There are methods to protect your app from tapjacking and bunq also implements these methods. Other issues are taskmanage snooping and misuse of the push notifications functionality of the app.

Furthermore, on both Android and iOS, a user does not have access to system files by default, but this access can be gained by modifying the software of the device. Because these modifications brings rise to extra security risks, bunq tries to detect such modifications and disable their app if they are present. [redacted].

Crashes and bugs inside an application can possibly be exploited by attackers. An attacker can use a bug to perform a Denial of Service attack on the application and during the project such an attack was succesfully carried out. Bugs and crashes can sometimes also be used to execute arbitrary code or can lead to the leakage of data. The two applications that were created during this project are aimed at discovering these kind of bugs.

One of the first approaches one can take to examine an application, is by decompiling it. Decompiling is the act of restoring compiled code to code at a higher level. This can give insight in the code, and can reveal security issues in this way. A counter measure that developers can take is obfuscating their code, which makes it harder to read the after decompiling the code. However, it will always be possible for an attacker to reconstruct the original source code, as illustrated in this report. Therefore, developers should consider the source code of their application as known to outsiders.

Next to decompiling the app, as explained earlier, two applications were developed. The first application developed in this project is the FSM-learner. Finite State Machines (FSM) show the working of a system by displaying all the possible states the system can be in and the transitions it can make to go from one state to another. This application generates finite state machines of the

mobile apps on both iOS and Android. These FSMs can then be used to compare the implementation on iOS and Android of the same app. By examining them, one could also discover illegal state transitions that can form a security risk. During this project, the generated FSMs were evaluated and showed differences in implementation between the iOS and Android app. This could lead to potential security issues, although they were not encountered, as it indicates a possible lack of communication between the development teams.

The second application that was developed, is a fuzzing tool. Fuzzing an app is a way of sending random inputs to it to see how these are handled. The fuzzing tool was used on the bunq app and resulted in bugs being found that were not covered in traditional testing methods. An example of a found bug was the inability of the application to cope with a very high number, while all other numbers appeared to work fine. As this is not an edge case, traditional testing would not uncover it while the random inputs of the fuzzing tool did.

To ensure that the built applications were of high quality, some quality standards were set and adhered to. These included elements such as the amount of parameters a function should have at maximum and the maximum number of lines of code in a function. Furthermore, the applications were extensively tested and also reviewed by both externally, by a company called the Software Improvement Group, and internally by our supervisor. The results of these reviews were very positive, as it turned out that the code was maintainable above avarage.

In addition to the issues found in the app of bunq, it also exposed additional problems that were outside of the previously defined scope. [redacted].

To conclude, everything was carried out successfully and it showed that a mobile application can never be fully secure. However the bunq app has a high security standard. Still, the server should always carry out enough validation checks to ensure that everything goes as expected and no illegal actions are performed.

Table of Contents

1 Introduction **1**

2 Problem Analysis **3**

2.1 Problem Definition . . . 3

2.2 Success Criteria . . . 3

2.3 Taxonomy of Attacks . . . 4

2.4 Security Scope . . . 5

2.4.1 Outside the Scope . . . 5

2.4.2 Inside the Scope . . . 5

2.5 Current Situation . . . 5

3 General Mobile Application Security **7**

3.1 Android Applications . . . 7

3.1.1 Layers of Defense . . . 7

3.1.2 Signing . . . 7

3.1.3 Manifest File . . . 8

3.1.4 Data . . . 8

3.1.5 Tapjacking . . . 8

3.2 iOS Applications . . . 9

3.2.1 Permission System . . . 9

3.2.2 Sandbox . . . 10

3.2.3 Memory Randomization . . . 10

3.2.4 Signing . . . 10

3.2.5 App Store . . . 10

3.2.6 Data Protection . . . 11

3.3 General . . . 11

3.3.1 Root Access . . . 11

3.3.2 Push Notifications . . . 11

3.3.3 Task Manager Snooping . . . 11

3.4 Results . . . 12

3.4.1 Signing Android Applications . . . 12

3.4.2 Manifest File . . . 12

3.4.3 Android Data . . . 12

3.4.4 Tapjacking . . . 13

3.4.5 iOS Data . . . 13

3.4.6 Root Access . . . 14

3.4.7 Push Notifications . . . 14

3.4.8 Task Manager Snooping . . . 14

4 Crash Exploitation **16**

4.1 Denial of Service . . . 16

4.2 Arbitrary Code Execution . . . 16

4.3 Memory . . . 16

4.3.1 Using Stored Information . . . 16

4.3.2 Buffer Overflow . . . 16

5 Decompiling **18**
5.1 Research . . . 18
5.1.1 Exploiting Decompiled Code . . . 18
5.1.2 Tools . . . 18
5.1.3 Research Conclusion . . . 19
5.2 Goals . . . 20
5.3 Approach . . . 20
5.3.1 Android . . . 20
5.3.2 iOS . . . 21
5.4 Results . . . 22
5.4.1 Android . . . 22
5.4.2 iOS . . . 22
5.5 Conclusion . . . 22

6 Generating Finite State Machines **24**
6.1 Research . . . 24
6.1.1 FSM Diagram Analysis . . . 24
6.1.2 Tools . . . 24
6.1.3 Learning Algorithms . . . 25
6.1.4 Research Conclusion . . . 26
6.2 Goals . . . 26
6.3 Approach . . . 26
6.3.1 Global Set-up . . . 26
6.3.2 Alphabet . . . 27
6.3.3 Teacher . . . 28
6.3.4 Learner . . . 28
6.3.5 Design . . . 29
6.4 Results . . . 33
6.4.1 Adjusting the Goals . . . 33
6.4.2 Payment FSMs . . . 33
6.5 Conclusion . . . 35

7 Fuzzing **36**
7.1 Research . . . 36
7.1.1 Fuzzing Methods . . . 36
7.1.2 Tools . . . 36
7.1.3 Research Conclusion . . . 37
7.2 Goals . . . 37
7.3 Approach . . . 38
7.3.1 Drawbacks of Existing Tools . . . 38
7.3.2 Design Goals . . . 39
7.4 Design . . . 39
7.4.1 Global Set-up . . . 39
7.4.2 Configuration Files . . . 40
7.4.3 bunqFuzzer-pc-client . . . 41
7.4.4 bunqFuzzer-android-server . . . 42
7.5 Results . . . 42
7.5.1 Implementation Results . . . 42
7.5.2 Bunq Specific Configuration Files . . . 43
7.5.3 Bug Results . . . 43
7.6 Conclusion . . . 44

8 Code Quality **45**
8.1 Testing Framework 45
8.2 Code Coverage 45
8.3 Quality Assurance 45

9 Additional Findings **47**
9.1 Unlimited Pin Tries 47
9.2 Denial of Service 47
9.3 Bug Reporting 47

10 Discussion & Recommendations **48**
10.1 General Mobile Application Security 48
10.2 Decompiling 48
10.3 Generating Finite State Machines 48
10.4 Fuzzing 48
10.5 Code Quality 49
10.6 Overall 49

11 Conclusion **50**

Appendices **57**

A Original Project Description **57**

B Testing Techniques **58**

C What is a Finite State Machine? **59**

D Example XML File **60**

E Overview Code Quality **61**

F First Feedback from SIG **62**

G Second Feedback from SIG **63**

H Project Infosheet **64**

1 Introduction

The days that people put their money under their mattresses are long gone. Nowadays, money is mostly digital. bunq [1] is a company which is developing a mobile financial application. It is obvious that the customers of bunq do not want to lose their money or privacy and that the company itself does not want to lose its integrity as a business. However, keeping a mobile application secure is not that easy: a large amount of the most popular apps (mobile applications) have been hacked [2]. During this project, bunq's mobile applications were tested for security vulnerabilities; this report describes the goals and the used methods for doing so.

bunq has both an iOS [3] and an Android [4] version of the application. When the goals of this project were set, it became clear that the Android app potentially had more security problems than the iOS app. Therefore our research focused on the Android application. However, we also tried to compare both versions to see if they behave more or less the same.

The reports starts with a thorough analysis of the problem. This analysis leads us to the following research question: how can an assessment of the security of bunq's mobile application be made using black box testing techniques? Black box testing techniques are characterized by the fact that the testing makes no use of the source code. This analysis also contains a taxonomy of possible security issues. By using this taxonomy, the scope of this project has been defined; the duration of the project is limited, so this project focused on the security of the app itself, and not on the network or runtime environment. A description of the current situation concerning the testing that has already been done, is also given. The next secion explains how mobile applications are generally secured and what common security mistakes are made by mobile app developers. The app of bunq have been tested on these commonly made mistakes.

The report continues with an overview of methods in which app crashes can pose security threats. This is important because a large part of our project focused on finding bugs and crashes that can be possibly exploited. Thereafter, an introduction into decompilation - the retrieving of the source code of an app using its binary file - will be given, as well as an explanation of what it means for the security of bunq's apps.

Another way in which insight into the application can be gathered without having its source code is by generating a finite state machine (FSM) of it. An FSM is a representation of the application which shows its inner working by displaying the possible states the system can be in and how it can move from one state to another. During the project, an application was developed in order to generate these FSMs from bunq's app. The design choices made while developing this application are presented, as well as the results it yielded. The comparison of the generated FSMs revealed that the working of the Android and iOS apps differed on points where they should not.

The FSM learning process works by feeding the bunq app with semi-random input. The process of fuzzing, which is described next, also works by sending semi-random input to the app. In the case of fuzzing, this is done to let the application crash and find bugs. Because existing fuzzing tools did not suffice, a new fuzzing tool has been developed. The inner workings of our fuzzing tool are explained. Using this tool, numerous bugs were found, as well as some severe security issues.

While developing these two applications, we had to make sure that they were maintainable and developed well. Therefore, the report proceeds with the methods that were used to ensure high quality code. These methods include writing tests and doing code reviews.

Some bugs were found that did not fall into the scope we defined at the start of the project. Because some of these bugs concern the security of the bunq app, they are presented at the end of the report. When all the technical details have been covered, the report continues with a reflection on the process. Furthermore, we also reflected on the approach we have taken in order to answer our research question. The report ends with a conclusion, which can be summarized as follows: bunq should make sure that the backend is safe, as the app can almost always be misused.

Note: The tests were executed on an unfinished product that will be improved and throughly tested before released.

2 Problem Analysis

This section describes the problem that bunq originally proposed to us and our approach to solve this problem. In section 2.1 translates the problem into a research question. Then Section 2.2 describes the success criteria for our project. These criteria follow from the problem definition and they are reflected on at the end of the report. Section 2.3 gives a categorization of different approaches one can take to attack a system or application to narrow down the scope of this project. This scope itself is described in Section 2.4.

2.1 Problem Definition

The original project description, as bunq posed it to the TU Delft, can be found in Appendix A. The question that bunq essentially wants answered can be stated as: are the apps and backend of bunq secure? Because this question is to broad for us to fully answer within ten weeks, we decided, after meeting with bunq and our supervisor of the TU Delft, that we had to narrow it down. It was decided to focus on the security of bunq's apps, and mainly the Android app. The goal of the project is to investigate the security of mobile apps in general and the assessment of the bunq apps in particular.

This resulted in the following research question: how can an assessment of the security of bunqs mobile application be made using black box testing techniques? The choice for black box testing techniques was made, because an attacker typically does not have access to the source code. A brief introduction into several testing techniques is given in Appendix B. The first two weeks of the project consisted of doing research and writing a report to answer this question. The content of this research report has been included in this final report. The techniques that were found during the research were applied to bunq's application. The research question was used to further define the scope of our project, which resulted in the Sections 2.3 and 2.4.

From the research followed four approaches that we used to determine if the apps of bunq are secure. To start, Section 3 provides an overview of how mobile applications are generally secured and what security leaks are likely to be found in such applications. The bunq app has been tested on these leaks.

The second approach is decompiling, which will be discussed in Section 5. It will explain how decompiling is accomplished and what the possibility of decompiling the apps means for the security of bunq's platform.

The third approach concerns the generation of finite state machines of the apps. This is done by using an application that has been developed in the course of the project. More on this can be found in Section 6. The use of the finite state machines can help spot design flaws or inconsistencies between programs, which is especially relevant to our client as they have both an iOS and Android version of the app.

The last approach is fuzzing, which can be found in Section 7 and was conducted by using an application that was developed during this project for this purpose.

All these methods do not make use of the actual source code of the application. This means that they can be considered as black box techniques.

2.2 Success Criteria

The success of this project depends on two factors. The first goal concerns the testing process and its results. The second goal consists of the result we leave behind when the project is done.

It is difficult to determine criteria for a successful security assessment. Security testing methods can have either of two results: one or more security leaks are found, or no security leaks are discovered. In the first case, the testing has a result, which means that one or more techniques were applied in a proper manner; this implies that the project was in any case a success to a certain extend. However, even when the testing yields one or more security leaks, this does not mean that the entire project was executed properly; the supposed absence of more leaks does not mean there are none. In the second case, the same holds: the absence of any leaks does not prove that the application is secure. The success is thus hard to determine. In both the first and second case, the project will be deemed a success when the described techniques are used as best as possible, taking into account the knowledge at the beginning of the project and the limited duration thereof.

A one-time assessment of the security of the application is nice to have, but a repeatable procedure is of much more worth. The second goal of this project is leaving a repeatable procedure for assessing the security of the application. This goal is of more importance to us than the first goal. The bare minimum of this repeatable procedure consists of a step-by-step plan that bunq can perform in order to reassess the security of its application. It is even better to leave behind one or more applications that can be used to test the application.

2.3 Taxonomy of Attacks

In order to get an overview of possible weaknesses in an application, it is useful to categorize attacks. The authors of AVOIDIT [5] made an excellent taxonomy already, which will be referred to in this section. The taxonomy consists of different resources that can be targeted in order to obtain information of, gain access to or abuse an application or system. Therefore, not only does the focus lie on the application aspect, but also on other elements that are associated with it, such as network and users. Below the description of the different targets can be found.

Operating System
: To begin with, the operating system on which an application runs can expose certain threats. Each operating system has its own vulnerabilities which can be targeted to gain unauthorized access to a device and possibly the applications on it.

Local
: With local attacks, an attack is specifically aimed at a device. This includes attacks on the application or on the application's data from malware or a virus, as they are present on the local device itself and not bound to a certain application.

Network
: More and more devices are connected to the internet nowadays and this also brings an extra threat with it. For example, it can be possible to target weaknesses in network protocols to gain access to certain privileges or information. Another approach is to use the network connectivity as a resource to read the information that is being sent or received in order to find information that can be used.

Application
: Instead of an operating system, an attack can also be targeting a specific piece of software. The impact such targeting can have, depends on the kind of software attacked. Some applications are stand-alone and are not connected to a server or to other devices; if such an app is attacked, only the app might get compromised. However, some applications are connected to other systems; when such an application gets compromised, not only the app could be affected but also the systems that are connected to it.

Users
: Whenever a user is targeted, the attack is most often aimed at retrieving personal information

or credentials. These can then be used to gain access to applications, without having to crack the application itself. This can be hard to detect, as authorized access has been granted. Such security breaches will only be detected when abnormal behaviour occurs.

2.4 Security Scope

Above, many targets are described at which attacks can be aimed. Now the type of attacks which this project focuses on are defined. The scope is divided in two categories. First, what is outside our scope is defined; these are subjects which we will not look at during the project. Then all the subjects that are inside the scope of our project are determined.

2.4.1 Outside the Scope

It is not possible to make changes to the different operating systems on which the app will run. An application runs on top of it, so any vulnerabilities in the operating system will also be a risk for the other applications present on the device. Since improving the operating system is out of reach, it is also considered to be outside the scope of this project. In addition, local attacks are also considered to be outside the scope as attacks aimed at a user's device cannot be stopped by the bunq app. For example, viruses and malware can infect a device without the use of the app. The aim is to keep the application itself safe, so nothing on the device will be able to breach it. Another aspect that could be considered, is whether all network connectivity is secure. When the application is secure, but the connection is not, it might still result in big security issues. Due to time constraints, this matter will not be discussed. However, during the project some issues with the network layer where found, these additional findings are reported in Section 9.

2.4.2 Inside the Scope

The application is the main target that is included in the scope. The application has to be as secure as possible, so therefore the focus is on attacks that target the application. However, these attacks can overlap with the other categories of the taxonomy above. For example, the operating system provides app developers with some functionalities in order to make their apps secure. These kind of functionalities can be seen as matter that lies in the scope of the OS but also in the scope of the application. Likewise, some local attacks can be directed at the application level. These kind of attacks, that are directed at the app and can possibly be prevented by the app itself are inside the scope of this project.

2.5 Current Situation

Currently bunq has an Android and an iOS application. At this point, bunq has done a lot of research in the security aspect of the apps. However, it hasn't been extensively tested to make sure it is secure and properly implemented yet. There is a pilot going on in which both apps are being tested and the bugs Because the full system of bunq is still under development it is changing rapidly.

When a user is setting up a bunq account for the first time, a token is used to verify that the user and device are connected. This connection is only needed once, meaning that a user is not bound to a device. It is possible to login on a friend's phone, or anyone else's, who also uses the bunq app.

The application is accessible with a 6 digit pincode, or by using the fingerprint scanner (Touch ID) incorporated in the latest iOS devices. After a pincode has been entered, a secure connection is set up to the server to verify it. The connection itself makes use of the Secure Sockets Layer protocol (SSL) [6] and it relies on the Secure Remote Password protocol (SRP) [7] to make sure your that even when the sent information is intercepted, it cannot be used to access an account.

[redacted]

[redacted]

3 General Mobile Application Security

This section explains how mobile applications are generally secured and identifies some common security mistakes. Section 3.1 shows how the android operating system secures the installed apps, and what basic setup one must use to keep an app secure. Section 3.2 does the same for iOS. In section thereafter some risks that lie in the local scope are examined. The risks here can be secured by the app itself. Therefore we examined them and checked if the bunq app was protected against them. Section 3.4 shows the results of these checks and how bunq uses the security provided by the operation system.

3.1 Android Applications

Android is an open source operating system. This makes it possible for developers to alter the system. However, it can also give attackers a lot of inside information for potential attacks. Android implements some layers of defense to counter these possible attacks. This section is dedicated to explaining how the Android operating system provides security for its applications.

3.1.1 Layers of Defense

The Android layers of defense as shown in Figure 2 are used to secure the operating system and other apps against potentially harmful apps. The first layer is the Google Play Store. To get your app available to the large public you can submit it to the Play Store. Google then verifies whether or not the app violates the platform's developer policies [10]. This check is less strict than the check Apple performs for their app store, and it does not guarantee that all harmful apps are rejected. On Android it is also possible to install apps from other sources then the Google Play Store, which makes it pretty easy to bypass the first layer. The verify apps consent layer checks the executable of an Android app against a database of malware before it is installed. The app is sandboxed which means that each app runs inside its own virtual environment to secure it from intrusion of other apps. What the app can do outside of this sandbox is restricted to the permissions granted to it. This granting of permissions has to be done by the user. Note that a lot of people do not really check what kind of permissions an app requests, thus this permissions layer could lead to a false sense of security. [11–13]

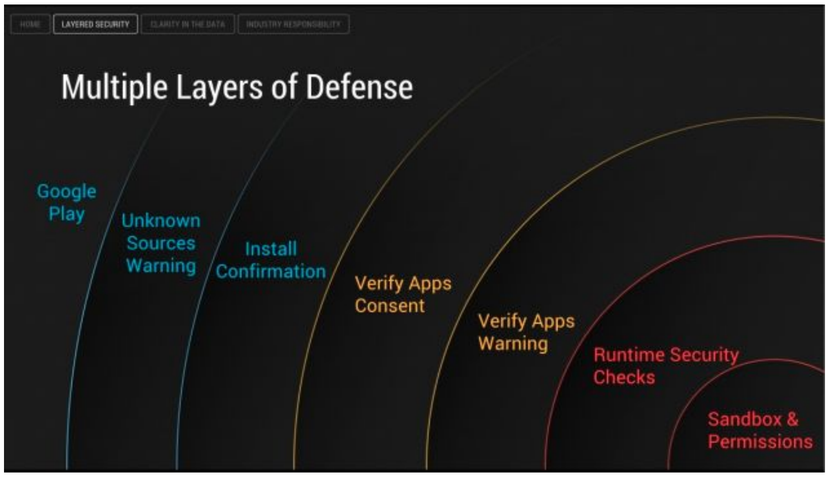


Figure 2: Android Layers of Defense [9]

3.1.2 Signing

All apps need to be digitally signed with a certificate before they can be installed on a device. By signing an app, the Android operating system can verify the author of the app and make sure it has not been tempered with by anyone else. The signature with which the app is signed is stored in a keystore which basically is an encrypted file that stores signatures. It is important to keep this keystore save.

If someone would get access to the private signature of someone else, one could potentially override the app of the other person or make apps signed with the same signature. This is harmful, because apps that are signed with the same signature can access each others data [14].

3.1.3 Manifest File

On Android there is an `AndroidManifest.xml` file that contains info about an app such as the permissions that are required to run it. Besides permissions, it is also used to set certain configurations. If this file is not configured correctly, the app will be less secure, which is not desirable. Settings that need to be disabled for the bunq app are for instance `allowbackup` and `allowdebug`. If these stay enabled, a user could connect to the bunq app and would be able to collect extra data which could be used to find weaknesses in the application itself.

Furthermore there is an export functionality that can be disabled for some, or all, components in the app. When the export of the components are disabled, it means that other apps on the same device will not be able to directly make a call to that component and potentially use it.

There are some limitations to the influence of the permissions set in the manifest file. An activity is a component of the app that represents a single thing that the user can do. It is possible to call every activity when a device is rooted, regardless of what was permitted in the manifest file. This is due to the fact that a rooted device has more system permissions and can therefore discard the permissions found in the manifest. However, bunq has taken measures as it will always require you to first login before any other activity can be loaded. It is therefore not possible to immediately start a payment by calling its activity and skipping the login.

3.1.4 Data

The bunq application should store as little information on disk as possible. Because all data that is stored on the device can potentially be collected by an attacker and due to the sensitivity of financial data, the risk of a data leak should be minimized by saving as little as possible on disk.

Android Keystore

In Android 4.3 and later a keystore is implemented. This keystore stores private keys in a container to make it more difficult to extract it from the device. The keystore is encrypted with the password of the device and is unlocked when the device is unlocked. When a key is inside the keystore it can be used for cartographic operations with the private key remaining non-exportable. This means that you can use the key to encrypt data, but it is not possible to see or export the key itself.

3.1.5 Tapjacking

Toasts are normally used to show small messages to respond to an user's action, an example can be seen in Figure 3. However, toasts are fully customizeable and can even fill the entire screen. When you tap on a toast the input is directed at the activity beneath it. This is what tapjacking makes use of. By showing a screen-filling toast that instructs you to tap certain points on the screen, the user is directed to perform instructions to another app underneath. Figure 4 shows an example of a tapjacking attack which directs the user to install an app.

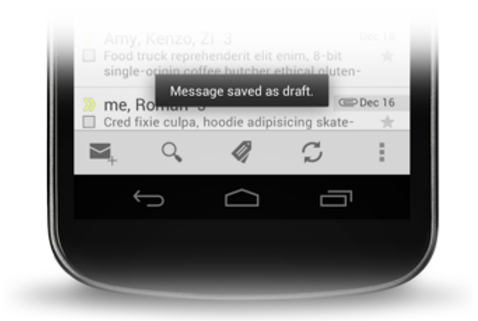


Figure 3: Normal Toast Example [15]

Since Android 2.3 it is possible to set the following attribute to your view: `Android:filterTouchesWhenObscured="true"`. This attribute makes sure that the view cannot be clicked when something is laying on top of it. Some blogs on the internet suggest that Google fully removed the click-through functionality of a toast in Android 4.0.3 [16, 17] However, no official statement of Google can be found confirming this suggestion. When we tried the tapjacking procedure on multiple devices running Android 4.3 and 4.4, it turned out that it was still possible.

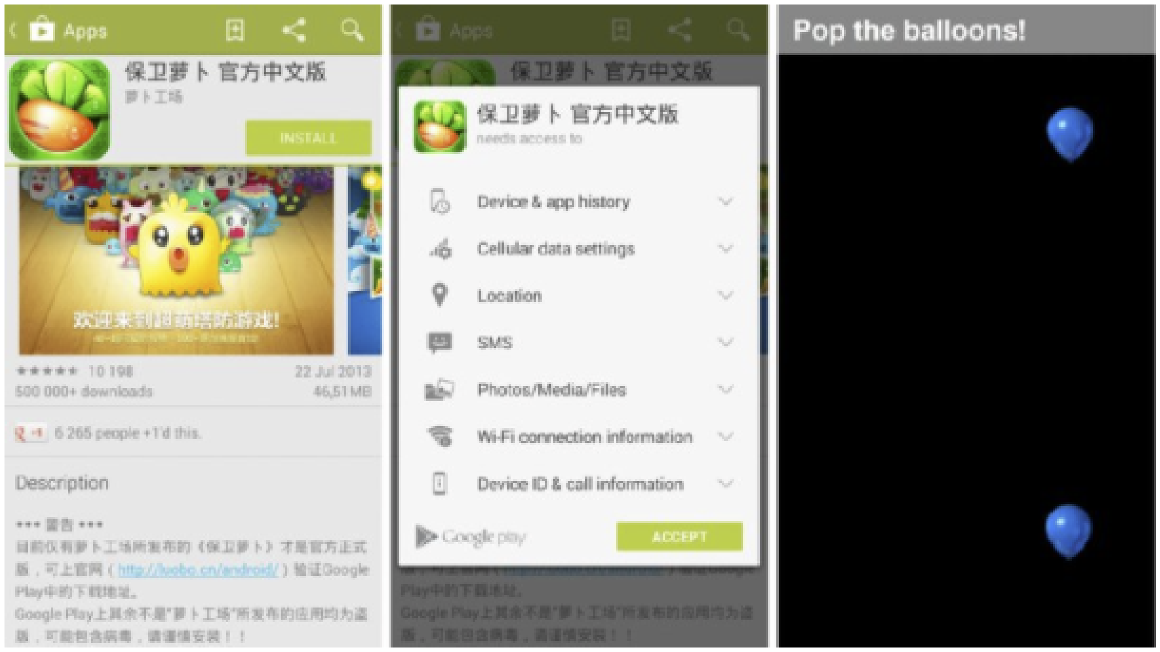


Figure 4: Tapjacking Example [18]

3.2 iOS Applications

Not much is known about the internal use of iOS, as it is closed source. Luckily, Apple released [19] a paper which explains some of the security measure that are present in iOS. This sections covers these measures and how the bunq application makes use of them.

3.2.1 Permission System

In iOS there are multiple security measures [19] to make sure applications are secure and limited to certain actions. First of all there is a permission system to withhold applications from accessing data or functionality they do not really need. These permissions are not shown when an app is downloaded from the AppStore, but rather when the app tries to use it. As an example you can have the Google Maps app which, of course, would like to use your GPS to indicate where you are located. When downloading the app, there won't be any permissions asked, but as soon as you run it it will indicate it needs to access your location and ask the user whether this is allowed.

3.2.2 Sandbox

Furthermore, as illustrated in Figure 5 iOS has a sandbox in which an app is ran. Because of this, an app is limited to the data it can reach and makes it impossible to exchange data with other apps on the device. As a result it is not possible for a malicious app to access (sensitive) data of other apps. There are however ways in which communication between apps can be achieved [20], but this needs additions to an app and cannot be done without the developer's consent. Whenever an app needs user information or other features such as iCloud or push messages, this can only be done by digitally signed entitlements [19,21]. These entitlements can carry out specific privileged operations that are not allowed from within the sandbox of the app. Due to the fact that the entitlements are signed and designed for only specific operations, they can not be misused for other operations than they are intended.

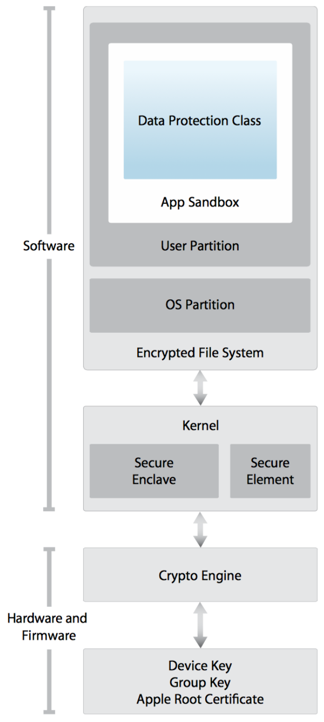


Figure 5: iOS Security Architecture [19]

3.2.3 Memory Randomization

In addition to the sandbox, iOS also makes use of address space layout randomization (ASLR) which randomizes all memory regions when an app is launched. This makes it a lot harder for potential attackers to misuse memory addresses as locations are no longer predefined. Furthermore, ARM's Execute Never (XN) feature marks certain memory pages non-executable, while others are both writable and executable and can only be used by apps under tightly controlled conditions. [19]

3.2.4 Signing

Just like with Android, iOS apps are signed before they can be installed on devices. The signature ensures that the apps are distributed from a known source that was approved by Apple. In other words, unsigned apps can not be run on iOS devices and the signature also ensures that an app was not altered since the moment it was signed.

There are two types of iOS distribution certificates by which an app can be signed for redistribution. The first one is a developer certificate that gives access to signing and redistributing an app through the official App Store. However, for big corporations that create apps for internal use, there is another kind of certificate available: an enterprise certificate [22]. With such a certificate it is possible to sign apps and redistribute them outside of the App Store as they should not be accessible to the public.

3.2.5 App Store

Normal iOS devices, meaning that no root-access has been granted by using exploits (Jailbreak), can only install apps that come from the official App Store [23]. Before an app can be found in the App Store, it has to go through numerous checks by Apple to make sure it complies to their standards [24]. This also includes a manual review process by Apple to check for anything that was missed by the automated checks. After the review process gives the green light, the app can enter the AppStore and it is considered to be safe to use. Because of this, it becomes very hard for malicious applications to infect an iOS device in general.

There is a small exception though, which includes apps signed by an enterprise certificate. With such a certificate it is possible to sign an app and install it on any non-jailbroken iOS device. In practice this does not occur often, as it still requires a location, such as a website, where the application can be downloaded from and this is not something users do easily. Besides, it is not easy at all to get into the Developer Enterprise Program [22], as one needs to have a legitimate business which can pass the credit-checks to prove its trustworthiness.

The signing protocols, together with the App Store, also make it hard for viruses to install themselves on iOS devices. Of course it is possible that a malicious application passes Apple's App Store checks, but in practice this rarely happens and malicious apps are immediately removed when discovered. The only other way in which viruses could potentially install themselves, is if the iOS device has been jailbroken as it is then possible to install apps that are not available in the App Store. However, jailbreaking is not done by the majority of users and those who do jailbreak their device, know about a decrease in security.

3.2.6 Data Protection

Although iOS has multiple security measures installed to ensure high encryption of user data [19], saved data can still be read when a device is unlocked. This leads to the possibility of an attacker getting access to data saved by the bunq app on the device. There is also a more secure storage available on iOS, called Keychain [25]. Here, more private data such as passwords, encryption keys and certificates can be stored.

3.3 General

There are also a couple of general security issues that apply to both iOS and Android. These will be discussed in the following section.

3.3.1 Root Access

On both Android and iOS, a user does not have access to system files by default. This results in people wanting to have access to everything, which means that so called 'root access' needs to be gained. On Android, this can be done by 'rooting' a device and the equivalence on iOS is called 'jailbreaking'. However, since root access also removes any limitations there were, it brings security risks with it. Therefore, bunq has chosen to not support rooted or jailbroken devices by checking for this in their apps. But, since root removes any limitations, it is possible to circumvent this detection and still run the app. There are multiple ways in which this can be done, starting from decompiling the application (see Section 5) and fully removing the detection functionality, to simply installing an app such as RootCloak [26] (on Android) that does everything for you.

3.3.2 Push Notifications

Push notifications are used on both Android and iOS to deliver messages directly to a device. These messages pop up on screen and are shown in the notification balk. They are intended for a specific app on the device. But by looking at them in the notification balk their content can be seen without logging in on the specific app. The push messages are handled by external services from Apple (Apple Push Notification service [27]) and Google (Google Cloud Messaging [28]).

3.3.3 Task Manager Snooping

After you exit an application, by pressing the back or home button on the device, you can return to that app by using the task manager/recent apps list. This list shows all the recent applications that are actually still running in the background. This list consist of the application names and a thumbnail of a screenshot taken right before the app was left. Task manager snooping is the act of looking at these thumbnails for sensitive data.

Of course bank accounts can contain highly privacy sensitive data, such as your account balance and payments. These could then be seen in the thumbnail of the taskmanager. Even after the app logs you out due to a timeout, the thumbnail still shows your last opened screen in the app. Thus, when someone would get their hands on your phone and start the task manager, he or she could see this privacy sensitive data.

On Android this issue can be avoided by adding the following line of code:
`.getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE)`
This turns off the screenshot feature for that activity and thus prevents the task manager from showing a recent screenshot, instead a black or white screen is shown.

On iOS there are guidelines on 'being a responsible background app' [29]. These guidelines also include a section that explains on how to remove sensitive data before the app moves to the background, as it is not allowed to disable the screenshot functionality on iOS. The idea behind the approach is that it can be detected when an app is transitioned to the background and to show a different screen just before this transitioning happens. Then a screenshot will be made of this new screen, on which you can decide for yourself what to show: for instance a blank page or a logo.

3.4 Results

Some of the security measures discussed in the previous sections could also be applied to the bunq apps. The results of this are shown in the following section. The results are divided into the same sections as they were in the general explanation, meaning that the missing sections indicate that it was not applicable to the bunq app.

3.4.1 Signing Android Applications

The private key that bunq will use for signing the production version of the Android app is not yet created. The versions they are currently testing are signed with temporary test keys. This is good practice because there is no possibility that the signing private key can get compromised now. The keystore of the production key will most likely be stored on an encrypted USB drive and put into a safe to make sure it is secure.

3.4.2 Manifest File

Settings that need to be disabled for the bunq app are for instance `allowbackup` and `allowdebug`. These settings are indeed disabled. In the bunq app, for most components the export functionality was disabled as expected. However, for some components the export is necessary. [redacted]. However, bunq has taken measures as it will always require you to first login before any other activity can be loaded. It is therefore not possible to immediately start a payment by calling its activity and skipping the login.

3.4.3 Android Data

[redacted]

[redacted]

Android Keystore
Since Android 4.3 the file can also be stored in the Android Keystore, which is more secure place to store private keys. The first problem bunq encounters if they want to use the keystore, is that it is not available for Android 4.0 to 4.3. For bunq it is not possible to implement its own keystore for the following reason: this keystore would need a password/pincode remembered by the user to be safe. However, requiring the user to remember two pincodes for one app (one for their account, one for the keystore) is not very user friendly. Using the same pincodes for the keystore which is device specific and the account is not possible because one device can be used by multiple user account, whom do not have the same pincode. Even if it was possible to use the same pincode, it would expose the keystore to extra risk, due to the fact that when there is an implementation flaw in the self-developed keystore - or if the pincode of the keystore would get compromised - the pincode of the account is also compromised.

One more problem arises: the implementation of the keystore on Android 4.3 came with a bug. Due to a buffer overflow it that can possibly be exploited [30]. However, exploiting this bug is much harder than getting root access on a device. We therefore advised bunq to implement the keystore on 4.3 devices.
[redacted]

3.4.4 Tapjacking

Using an example of tapjacking [31] and customizing it we were able to perform a tapjacking attack on the bunq app. After our finding bunq has set the `filterTouchesWhenObscured` attribute to all its views which now fully protects the app against this vulnerability.

3.4.5 iOS Data

To ensure that a possible leak of data will not pose a risk for the users, the data stored by the iOS app was collected and investigated. The first thing that is stored by the iOS application, is information about the device such as the device's authorization token. [redacted]. When one only has access to an authorization token and not the corresponding private and public keys, the token can be considered useless as it alone cannot be used to register another device correctly. It can therefore not be used to trick the server in thinking The private and public keys are stored in the iOS keychain, which means that they cannot be (easily) retrieved.
[redacted]

[redacted]

[redacted]

3.4.6 Root Access

[redacted]

Unfortunately there was no iOS device with root access available, so the actual circumvention could not be tested. It was however possible to decompile the iOS app, meaning that theoretically it would be possible.

3.4.7 Push Notifications

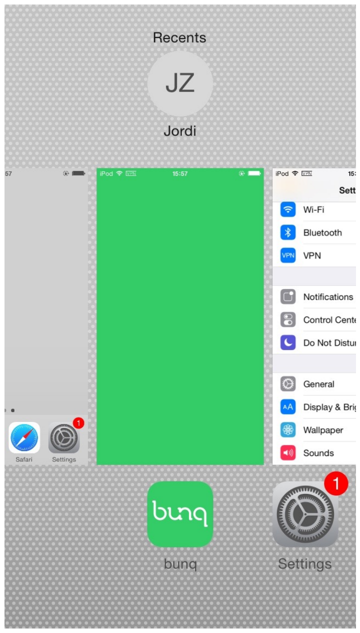
bunq also uses push messages to inform users on payments that are made to them. These messages include sensitive information such as the amount of a payment this gives rise to some privacy related issues in the case of bunq. This sensitive information can be seen by anyone who has access to the device, even if they are not logged in to the bunq app. bunq is aware of this problem and at the time of writing is working on a solution.

However, one of the features of the bunq app is being able to login on multiple devices with the same account. If this account is later on not manually removed from the device, that device will also start to receive push notifications intended for that account. This means that another person would be able to see everything that happens on your account, by just looking at the received push notifications. These notifications contain privacy sensitive data, and users are most likely not aware of the need to actively remove their account from oneothers device. Therefore it would be better to disable the push notifications function by default, and make an option in the settings menu to enable them for the current device.

3.4.8 Task Manager Snooping

At this point, bunq has not yet implemented the secure flag method as it can be useful during the testing phase to take screenshots of the app. In the production version of the app, the flag will be enabled, so it will no longer be possible to snoop around.

In iOS it is possible to make screenshots while the task manager thumbnail shows only a green screen. An example of this can be found in figure 6b. Figure 6a shows how the task manager would look if it was not covered with a green screen.



(b) Secure task manager, shows a green screen

Figure 6: Task manager iOS examples

4 Crash Exploitation

The previously mentioned methods - decompiling, generating finite state machines and fuzzing - can all be used to find conditions in which the application crashes or produces an error. Of course this is not a desired feature of the application, because users can lose data and will have to restart their application or system. But when an application crashes, security issues also arise. Not every crash is exploitable, however when an application crashes it means that something unexpected happened or something is not properly understood. If a situation is not expected nor understood, you have no way of knowing if that situation is safe and therefore you must assume that it is not. This section shows why crashes are a security risk by explaining how a crash can be exploited.

4.1 Denial of Service

After a crash an application needs to be restarted, which (shortly) denies a user of the service. If an attacker replicates the conditions leading to the crash, your service will suffer from prolonged outage, which thus denies the user from using bunq's services. In the case of bunq, a potential Denial of Service (DoS) attack was discovered by us on which more can be found in Section 9.2.

4.2 Arbitrary Code Execution

In case an application crashes because some input was not processed correctly, an attacker could find out which exact input leads to the faulty procedure. By analyzing which inputs lead to crashes, the attacker could infer details about the system and its internal workings. The insight could then result, together with the faulty procedure, in an attempt to execute arbitrary code. An example of arbitrary code execution is a so-called SQL injection [32]. In most situations, this is not possible. However, if the inputs of the application are not sanitized, the application is likely to crash or present an error message when special characters like quotes (') are inserted. This gives the attacker the notion that an SQL injection - which is a form of arbitrary code execution - is possible. Using the unsafe input, the attacker can use SQL to retrieve, edit and delete data.

Another form of arbitrary code execution is XSS, which stands for Cross Site Scripting. With an XSS attack, the attacker manages to inject a malicious piece of code in a client-side scripting language into a webapplication. The victim then unknowningly executes this script, which is often used to steal credentials. [33] [redacted]. [redacted].

4.3 Memory

Since memory is used by an application to store information, it is a common target for exploitation. A couple of ways in which this can actually be done will be discussed next.

4.3.1 Using Stored Information

The information that is stored in RAM can be viewed when a memory dump has been made. It is possible that information that is stored in this memory can be used to bypass security measures. An example of misuse of in-memory data is session hijacking, if a session key is stored in RAM.

4.3.2 Buffer Overflow

Executing malicious code can also be done by a so-called Buffer Overflow attack. Such an attack consists of two parts. The first is concerned with putting malicious code in the address space - or buffer - of the application under attack. A possible way to do this is by using this code as an input to the program. The second part consists of making the application under attack jump to the malicious code. This can be achieved by overflowing a buffer that has weak or non-existent boundary checks on its inputs. The overflow then can be used to corrupt a pointer in memory to point to the malicious code injected in the program's address space in part one of the attack. The only thing left to complete the attack is to activate a part of the application that uses the pointer that was corrupted. [34]

5 Decompiling

One of the first approaches one can take to examine an application, is by decompiling it. What exactly is meant by decompiling and how can this be put to practice? Questions like these is what will be covered in the following section. Furthermore an overview will be given of the possible decompilation programs that are available and which ones will actually be used during the project. It also contains the goals that were aimed to be achieved with decompiling, how it was actually carried out and the found results.

5.1 Research

Decompiling is restoring compiled code to code at a higher level, for example Java; the result of decompiling can be quite close to the source code of the original software [35]. Decompilation does not work by 'fuzzing' the application, but by analyzing the machine code; because the original source code is not available when decompiling, it could be considered as a black box technique. The reason this technique is mentioned and used in this project, is because decompiling is a way to restore the source code of an application. Since possible attackers do not have direct access to the source code, they depend on a techniques like this to recreate the source code themselves. By looking at the code from an attacker's point of view, it is easier to see which vulnerabilities can be used and what kind of information can be retrieved, without needing the actual source code.

5.1.1 Exploiting Decompiled Code

Many exploits do not need a clever approach for discovering leaks, but result from static analysis of the code [36]. This means that by retrieving the source code, by decompiling, one can find security vulnerabilities by using a static analysis tool. People who want to obtain insight in the vulnerabilities of an app, can also decompile it and try reading the source code. This is a tedious procedure and it is hard to find bugs in this way.

However, a counter-move is possible: one can obfuscate the code of an application, which makes decompiling the code a lot harder. Obfuscating means transforming a program P into a program P', where P and P' have the same observable behaviour [37]. The goal is to make P' more obscure then P: less readible and understandable, both for computers and humans. When someone decompiles P', it is a lot harder to retrieve the actual meaning and execution flow of the program then when that person had access to the source of P.

5.1.2 Tools

When reviewing the possible approaches and programs, only the programs that fit a certain set of criteria are included in the report. These criteria consist of programs that are recently updated, they should not be dependent on the operating system and they should be affordable (so they could actually be put to use in the company). These criteria where set as there are numerous applications available, which would result in an almost inexhaustible list of possibilities.

In order to check whether code obfuscation is applied, the application first has to be decompiled. After decompiling, a quick look at the code will show if it's obfuscated and also to what extent. Therefore this section will not go into depth about code obfuscation, but it will list programs which can be used to decompile the application.

Not all of the programs that were researched fit the earlier set criteria. These are therefore not discussed in this section, but will be quickly mentioned: IDA Pro [38], JEB [39], JAD [40], VTS [41] and APK Studio [42].

dex2jar

The first option that was considered, is the application dex2jar [43]. This application makes it possible to convert DEX files (a filetype in which Android applications are stored) to JAR files (a filetype which consists of archived Java files). The JAR file can then be converted into readable Java files again, for which a program called JD_GUI [44] is available. The only downside of this approach is that dex2jar does not decrypt the Android binary XML file format. Luckily there are simple programs such as AXMLPrinter2 [45] that are able to do this. Another downside is that the created Java code is not compile-ready, as assumptions are made to recreate the higher-level programming language.

APKtool

A somewhat similar tool to dex2jar is APKtool [46]. With this command-line program it is possible to convert an APK file, which contains the Android application files, directly to Smali files, which contain code in an assembly-like language. It also automatically decrypt the Android binary XML file format. The downside however is the smali language to which the files are converted. This language is hard to read and there is just a single program [47] available which is able to convert Smali to Java files. Advantages are that it is able to recompile the application. This means that it is possible to alter the Smali files, built a new APK from it and run it on a device.

Decompile Android

There is also a website available that is capable of decompiling APK files: Decompile Android [48]. This website makes use of some of the programs discussed earlier, such as Dex2Jar and APKtool. When decompiling the APK has been finished, it shows the Android manifest file [49] and gives the option to download an archive containing the decompiled files. Unfortunately, the website is not very quick at decompiling and the programs that it uses can be downloaded for own use. Furthermore, it might not be very secure to upload an APK file of an application that is still under development.

Dare

Dare [50] is a replacement of the older version called DED [51]. The program aims at Android application analysis and is capable of decompiling APK files to Java Class files. A big disadvantage of the program is the time it takes to decompile. Most of the other programs discussed in this section take around a minute to decompile a test application, while Dare needs over an hour. This is due to the fact that it also decompiles the Android library files. The only way to circumvent this is by just decompiling the files and afterwards run Soot [52] to optimize only the needed files. Without this optimization, there is very much redundant code that makes it harder to understand the decompiled program.

Hopper

For decompiling iOS applications, there is a program called Hopper [53]. Since Objective-C, which is the language used for iOS applications, is directly translated into machine-code, it is very hard to decompile it to a higher-level language. Therefore, Hopper is only able to decompile it to a 'pseudo-C' language and not objective C. Hopper is not a free solution, but it is very reasonably priced in comparison to other programs.

Since iOS apps get encrypted when they are put in the App store, the application first has to be decrypted before Hopper can be put to use. Luckily, there are tools available which can decrypt iOS applications, such as oTools (which is included in the developer tools of Xcode [54]) and Clutch [55].

5.1.3 Research Conclusion

After a careful evaluation, the programs that were chosen to be used are dex2jar and APKtool, for Android, and Hopper, for iOS applications. Dex2jar had the best decompiled source compared to the other possible programs, and it was in readable Java instead of Smali. This makes it easier to understand the source code and find any vulnerabilities in there.

APKTool, because it makes use of Smali, has the ability to build a new APK from the sources even after altering them. This gives way to more interesting vulnerabilities, as it is possible to adjust the application to one's liking. Furthermore, both programs are easy to use, well documented and don not require many other programs to function.

Since there are not any other viable solutions for iOS applications, Hopper was an easy choice. The only real competitor is IDA Pro [38], but its licensing fees start at $1000 and is therefore not considered to be a good fit for this project.

5.2 Goals

Since a possible attacker does not have direct access to the source code of the application, it is necessary to recreate it. This is where decompiling steps in, as it is a procedure which results in (somewhat) understandable code. Depending on whether obfuscation techniques or programs are used, e.g. Proguard [56] on Android, this code will be easier or harder to understand. One of the goals is therefore to find out to what extent the code is obfuscated, and whether it can be improved, in order to make it harder for a potential attacker to understand how the application works. This also results in a more difficult understanding of possible flaws or weaknesses in the application, making it harder to find and exploit them in a timely manner.

A risk associated with decompiling a program is the possibility to change aspects of a program and issue this modified application to the public. If a skilled attacker modifies the application in such a way that it will expose sensitive information of the users, or include malicious code, it will result in big security risks. During this project, it will be assessed to what extent it is made harder for a potential attacker to be able to change the original application to its own liking. This also includes checks to see whether a signature of the application, by a different certificate and key combination, results in the application being unusable, or if this makes no differences.

A minor goal that will also be achieved by using decompiling, is to check whether settings are set correctly. This includes some simple checks of the Android Manifest file, but also what will happen when these settings are changed and if the application is still secure in that case.

Most of the above goals are directed at Android, as decompilation on that platform leads to much better results (for instance Java or smali code). On iOS, decompiling would result in assembly code, which essentially does the same as obfuscation: the code is not easily readible by everyone. Of course it will still be useable for skilled attackers, but the same applies to code obfuscation.

5.3 Approach

Android and iOS are different operating systems and therefore require different tools to decompile an application. Therefore this section is split into an Android and iOS part, showing what was needed to successfully decompile the application on both systems.

5.3.1 Android

As discussed earlier, there are numerous tools available which can be used to decompile an application. The dex2jar tool will be used first to retrieve, hopefully readable, Javacode from the application. It can then be checked to what extent this code is obfuscated and how this obfuscation can possibly be improved.

The use of the dex2jar tool is fairly easy. First an APK file of the application is needed, which can either be downloaded from the Google Play store [57] or, in this case, be retrieved from the developers directly. Then the APK file is renamed to ZIP, in order to unpack it with any unarchive program. After the ZIP file has been unpacked, it will consist of a folder in which a file called 'classes.dex' can be found. This file contains all the code of the application, but in Dalvik format which can be read by the Android system. At this point dex2jar is used to convert the DEX file into a JAR file, which consists of the Java sourcecode files of the application. This can be done by using the following command: `sh ./d2j-dex2jar.sh classes.dex`. The JAR file can then be loaded by the JD_GUI program, showing the actual Java classes which are (human) readable. The readability of these files is depending on whether code obfuscation has been used or not. A shortcoming of dex2jar is that the Android manifest file is still encoded. In order to make the manifest file also readable, AXMLPrinter2 is used with the following command: `java -jar AXMLPrinter2.jar OriginalAndroidManifest.xml > ReadableAndroidManifest.xml`.

Furthermore another program is used to not only decompile the application, but have more possibilities such as building a new one from the decompiled sources. The application that is capable of doing this, is APKtool. For this program to be used, the only thing needed is the APK file of the application. Whenever the APK file has been obtained, the following command can be run to decompile it: `apktool d application.apk`. This results in a folder in which a couple of files can be found, including the decompiled DEX file (to smali format) and some other XML files such as strings used in the application. At this point, the files that are interesting are the one with the smali extension as these can be altered. In order to find which files should be altered, dex2jar was used as this gives better readable code (Java) than the assembly-like smali. Whenever the right file is found to be edited, the actual editing will be done in the corresponding smali file. After modifying one or more files, the application can be built by using the following command: `apktool b application-folder`. This will create an unsigned APK file with the edited files included. In order to run this modified APK on an Android device, it needs to be signed again. For this, the program SignAPK will be used which comes with its own certificate and key to sign the APK. With the following command, the new APK can be signed: `java -jar signapk.jar certificate.pem key.pk8 app-unsigned.apk app-signed.apk`. This will generate a new, signed, APK called `app-signed.apk`. This APK can then be installed and ran on an Android device.

5.3.2 iOS

For iOS there isn't much choice, so Hopper will be used. For this, the IPA file has to be retrieved from the iOS device. Unfortunately, this is not possible with the standard programs from Apple such as iTunes, therefore an alternative called iTools [58] will be used.

After the IPA file has been retrieved, it can be renamed to ZIP and extracted. This will result in a folder which contains a binary file that can be read by Hopper. When the file is processed by Hopper, it will show the contents in assembly. The assembly code can then be modified and recompiled into a useable IPA file, which can then be installed and ran on an iOS device.

Please note that in this case the IPA was never distributed through the AppStore and therefore not encrypted by Apple. Normally it would mean that the IPA is encrypted and cannot be read by Hopper as-is. This will require some additional steps involving oTools to remove the encryption. Since it was not necessary in our case, these steps are also not explained in detail in the approach and results section.

5.4 Results

As there are different approaches, the results also differ between Android and iOS. Therefore this section is also divided into two parts, each covering their own results.

5.4.1 Android

The first, and easiest, part that was checked, was code obfuscation. After decompiling the application with dex2jar and APKtool, both the outputs where checked and it showed obfuscated code where possible. Most classnames, functions and variables were replaced by short variants such as 'a' or 'b'. Some parts, however, were not fully obfuscated due to the fact that these were declared in XML files. For instance activities in Android can only be used when their original name is known, otherwise it simply does not work so code obfuscation cannot be applied in those cases. In short, most code has been obfuscated and this makes it very hard to understand everything that is happening in the application.

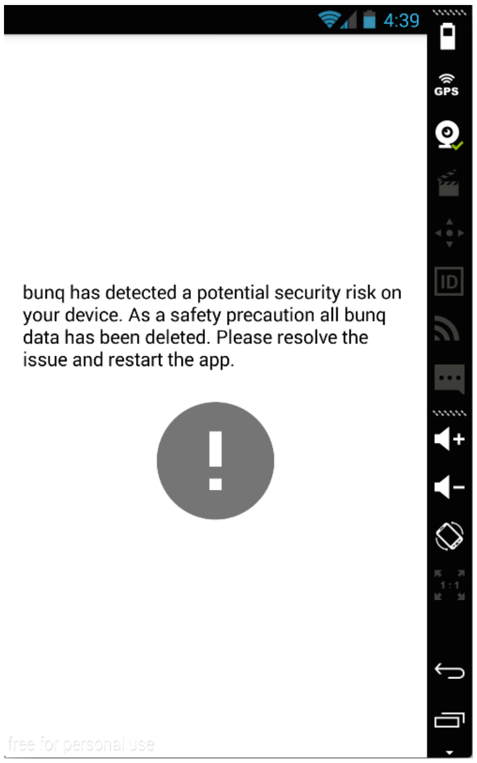
Next, the application was signed with a different certificate and installed on an Android device. On this device, the original application was present, which resulted in an error since the signatures did not match. Therefore the original application first had to be removed, after which the new one could be installed and configured. [redacted]

5.4.2 iOS

As expected, the IPA file was not encrypted and could be easily read by Hopper. This resulted in lots of decompiled assembly code, which turned out very hard to read. An advantage of the assembly code in comparison with the Java (or smali) code that is generated by decompiling an Android app, is that code obfuscation is not necessarily needed: the assembly code already takes care of this. The drawback of this was that we were not able, also due to the limited time of the project, to actually change parts of the iOS application and recompile it. In theory however, this would be perfectly managable for people that are skilled in assembly. Hopper also supports recompiling the modified sources, meaning that a new IPA could have been made and run on an actual device.

5.5 Conclusion

Code obfuscation is made use of, resulting in code that is hard to understand. This slows down the process of understanding the application by an attacker, so it is therefore encouraged to be done. [redacted]



bunq has detected a potential security risk on your device. As a safety precaution all bunq data has been deleted. Please resolve the issue and restart the app.

(a) Root detection enabled

[redacted]

Figure 7: Root detection examples

6 Generating Finite State Machines

One of the ways to assess the security of the application is by creating and examining the Finite State Machine diagram of the application. First, a research concerning the generation and the use of FSMs will be presented. Thereafter, several goals will be presented; these goals will show what we hope to achieve with the generation of FSMs of the bunq app. After the goals have been set, an approach is given in which an attempt will be made to achieve these goals. Subsequently, the results of the used approach are presented. This section closes with the conclusions that can be inferred from the acquired results.

6.1 Research

A finite state machine (FSM) is an abstraction that can reveal the inner working of an application. A short introduction into finite state machines is given in Appendix C. One way of reverse engineering is to fuzz an application; one could use the output the application generated to build a finite state machine diagram of that application.

For example, an iOS application can be reverse engineered using dynamic analysis of the runtime behaviour of the app. Using heuristics, new interface states in the User Interface (UI) can be discovered when they are activated by events in the UI. From the data that was found, states and state transitions can be constructed, which can be further analyzed. [59]

6.1.1 FSM Diagram Analysis

When an FSM has been inferred from an application, it can be analyzed. By analyzing an application's FSM, one can see wether certain state transitions or even entire states are present while they should not. For example, during a recent research, an application on a banking smartcard was found to be vulnerable to a certain sequence of actions. This was done by examining the generated FSM of that application. [60]

Another interesting aspect is that when there are multiple implementations available for a specific protocol or specification, the FSMs of the different implementations could be compared [60]. Because subject application of this research has two implementations, one for Android and one for iOS, the FSMs of those implementations can be compared to check if their behaviour is the same and discover potential security vulnerabilities.

6.1.2 Tools

In order to create finite state machines, a couple of programs can be considered. There are not a lot options available, probably due to the advanced algorithms contained in such applications.

iCRAWLER

An interesting option for retrieving the state machine of an iOS applications appeared to be iCRAWLER [59]. However, after contacting the researchers who wrote this application, it turned out that the application was not released for public use yet.

LearnLib

One often used library for generating state machines is LearnLib [61]. This Java library contains several learning strategies and can be used to create a program that is possible to retrieve the state machine of an application. Recent projects have shown that LearnLib can be used on various real-world systems like banking cards [62] and the bounded retransmission protocol [63]. LearnLib communicates with a SUL (System Under Learning) using TCP/IP. To use LearnLib, an application needs to be written that functions as a connection between the SUL and LearnLib.

LibAlf

LibAlf [64] is a second option for learning state machines of applications. However, there are not many research papers which use LibAlf; also, the website has not been updated for the past few years. This was a great indication that the program is no longer actively developed and supported, which made it unreliable for creating an FSM.

The most promising FSM-learning application appeared to be LearnLib, so in the next section, this application will be used to evaluate the different learning algorithms.

6.1.3 Learning Algorithms

The earlier mentioned applications both implement several algorithms that are used to actually learn the state machine of the SUL. Generally speaking, a learning algorithm for FSM's does two things. First, it makes all sorts of combinations, called queries, with the input alphabet and queries the SUL with those combinations. Using the output it gets from the SUL, the algorithm tries to form a hypothesis. Thereafter, the learning algorithm constructs queries with which it tries to validate the hypothesis; in other words, it tries to find a counterexample for its hypothesis. If the algorithm can't find such a counterexample, the learning algorithm apparently has constructed the correct FSM and it stops the learning. [65]

LearnLib implements the following learning algorithms. The most important aspects of each one is given below. The description of these algorithms is quite technical; for a more detailed explanation of the algorithms, please see the referenced papers.

L* [65]
The L* algorithm uses observation tables, so that it can gradually build a hypothesis, instead of constructing a hypothesis in each round. This leads to a relatively large memory footprint, but also a good performance.

Maler/Pnueli [66]
This algorithm is a modification of the L* algorithm. The learning process is characterized by executing a search for a counterexample after every run, instead of only in the end. A run consists of all combinations of length x of the words w in subset Y of alphabet A.

Direct Hypothesis Construction [67]
The Direct Hypothesis Construction (DHC) algorithm continuously maintains a hypothesis. It uses a breadth-first manner to explore the system and discover new states. The algorithm has a smaller memory footprint than the earlier discussed L*, because it does not keep an observation table like L*. It can be used to analyze very large systems.

Kearns/Vazirani [68]
The Kearns/Vazirani algorithm uses discrimination trees to keep its information about the states it learnt. The leafs of this tree represent the states of the current hypothesis.

TTT [69]
The TTT algorithm is characterized by the redunancy-free storage of its observations. It thoroughly analyzes the counter-examples and stores only those parts of the information that are needed for the refinement of its hypothesis.

Several other algorithms are available, for example Rivest/Schapire [70] and NL* [71], but these are not applicable to our context for reasons that we won't go in to here.

We compared the algorithms in terms of correctness and speed. All algorithms were run on the same set-up, which had a small subset of the capabilities of the bunq app. The results of this comparison can be found in Table 1.

Algorithm	**Running time (mm:ss)**	**Correct result**
TTT	01:12	no
Kearns/Vazirani	01:30	no
L^*	02:47	yes
DHC	03:30	yes
Maler/Pnueli	04:03	yes

Table 1: A comparison of the different algorithms

6.1.4 Research Conclusion

We can conclude from the research done that for evaluating the security of an application, an FSM can potentially provide crucial information. To generate such an FSM, several applications were examined. Because there is more research available that uses LearnLib as a learning tool, we will use LearnLib to learn the finite state machines of the apps. The different algorithms that LearnLib offers were evaluated too. The ones that failed to return the correct result will obviously not be used. Because speed can be an issue in a learning process, we will use L^* to examine the SUL as it appeared to be the fastest in our tests.

6.2 Goals

The first goal is purely focused on the security: are there unwanted state transitions or even entire states? These potential security leaks can be discovered by tracing the state transitions in the FSM.

Because there are two implementations of the bunq app, two FSMs can be created. The second goal is concerned with the similarity of the two applications: do they have the same security measures? And are the actions that can be performed in each state in both applications equal to each other? These questions too can be answered by analyzing the state machines.

6.3 Approach

Now that the goals have been determined, a method is needed in which those goals can be achieved. This section describes how the goals concerning the learning of the FSMs were achieved. It starts with giving an overview of the system; then it will explain what an alphabet is and what the alphabet used for this project looks like. Subsequently, the different parts of the learning system are described. This section concludes with an in-depth look into the system design of the software that was created to learn the FSMs of the bunq apps.

6.3.1 Global Set-up

LearnLib is, as the name indicates, just a library; to use it, an application needs to be written that actually uses the classes and methods of the library. This application communicates through Appium [72], which is an application that provides functionality to execute commands on mobile devices, with the System under Learning (SUL). In our case, the SUL is the mobile application of bunq. Figure 8 contains the architecture that we will build in order to use LearnLib to learn the state machine of the bunq app.

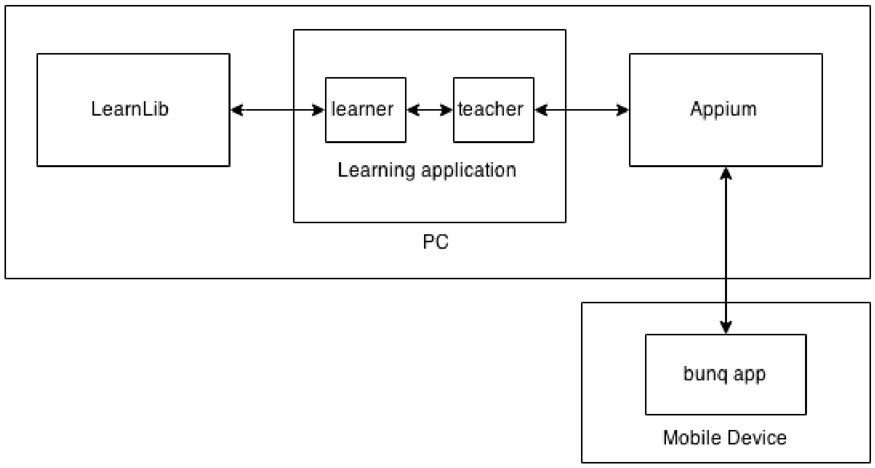


Figure 8: A global architecture overview of our FSM learning approach

Since bunq has an application for both iOS and Android, it would be useful if parts of the LearnLib application could be used independently of the operating system. The standard developer programs that come with Android (UIAutomator [73]) and iOS (UIAutomation [74]) to communicate to devices, unlike the names would suggest, differ greatly from each other. Because of this, it would result in two different programs that would perform the same tasks, just with different devices. In order to overcome this, different programs were assessed that could both communicate with iOS and Android devices. The decision was quickly made in the favour of Appium [72], as it is updated regularly and provides the communication for the different operating systems on which the apps run. This resulted in the ability to build a common learner application, which uses Appium to communicate with the different devices. Under the hood, Appium uses UIAutomator and UIAutomation; however, it provides a wrapper around those two systems so that both Android and iOS can be instrumented in a very similar manner.

At this point the only component that slightly differs for both operating systems, is the part that creates a readable alphabet file from the application. This is not a surprise, as the applications are slightly different and make use of different ways to identify certain fields or inputs.

6.3.2 Alphabet

To examine a SUL, LearnLib queries the system. These queries consist of words, that contain the actions to be carried out, from a pre-defined alphabet. The words that are sent by the learner should be understood by the SUL; after the SUL has parsed the words, it should execute the actions that they indicate and return the output the system generated.

For the SUL to understand the alphabet words, all words should have the same format. The format chosen for this project is as follows:

```
action%param1#param2#...#paramx
```

`action` represents the action that the SUL has to execute. The `%`-sign separates the action from the parameters. The `#`-sign separates the parameters from each other. Each parameter provides the SUL with the needed information to perform the given action. Each action will be performed on an element of the user interface; the parameters should give enough information to find the exact element on which the action should be performed.

Because Android and iOS do not provide us with an universal method of retrieving elements from the user interface, two different versions of the alphabet are needed.

For the Android version of the application, an element can be found by tracing it's xpath [75]. Each word of the Android version of the alphabet contains such an xpath as first parameter. The following word is an example of a word from the Android alphabet:

```
push%//android.widget.LinearLayout[1][@index='0' and @resource-id=''
    and contains(@text, '') and @content-desc='']/android.widget ...
    #540#115
```

Please note that the dots (...) are not part of the actual word; they indicate a repetition of elements, similar to the first UI element in this xpath. This was done since the actual word would span around half a page which was not considered useful.

The iOS version of the alphabet is a lot more readable; user interface elements in the iOS app can be found by just the name. Sometimes, an additional value is needed if there are more elements with the same name. The following word is an example from the iOS alphabet:

```
push%Back
```

As can be seen from the examples, the Android and iOS alphabets do not look like each other. Because one of the goals was the comparison of the Android and iOS implementation, a top-level alphabet was created by hand. This alphabet has the same format, but is more readable for humans. This top-level alphabet corresponds with the words in the two alphabets that were mentioned earlier. If a word in the iOS and the Android alphabet has the same meaning, this will result in one word in the top-level alphabet. Some actions could be platform specific; if this is the case, this action will be also present in the top-level alphabet but in a more readable fashion.

6.3.3 Teacher

As can be seen in the general overview given earlier, one part of the learning application is called the teacher. This is the counter-part of the learner. In this analogy, the learner asks questions to the teacher. The teacher then uses its tool, in our case this tool is Appium, to retrieve the answer from the SUL.

To be able to execute actions on a mobile device automatically, external software was needed. Initially, we chose to use UIAutomator [73] for Android and UIAutomation [74] for iOS. However, using these two systems proved to complicate the design of the application. For Android, for example, UIAutomator runs on the device, not on the system that is connected to the device. This means that a socket communication would be needed in order to send assignments to the device. On top of that, the runtime environment on a mobile device differs from the runtime environment on a normal computer; in our case, this meant that we could not use certain features of the Java language that we did need. Luckily, Appium proved to be a cross-platform alternative.

The teacher package also contains a procedure to parse and execute the actions it receives from the learner. By using reflection, an action-string will be parsed to a Method object and directed to either of the two instrumentator classes. These instrumentator classes then use Appium to execute the assignment.

6.3.4 Learner

The learner component of the application ensures that certain assignments are created and that these are sent to the teacher component to execute them. Furthermore the response of the teacher to the assignment is saved to 'learn' about certain states that are available in the application.

To achieve this behaviour, the learner first has to know about the possible actions that are available in the application. For the Android application so called 'window dumps' are created with UIAutomator of each of the screens in the application. For iOS the same is done, but by using UIAutomation. This results in XML files, or more precise a PLIST file for iOS, which contain all user interface elements that are present in the application.

Through two specially designed XML parsers in the learner component, the input files are read and an alphabet word is created for each interactive option available. The options that were considered 'interactive' contained actions such as pushing a button, checking a checkbox or entering text in an inputfield. These actions are represented in the alphabet as `push`, `check` and `enterText`. All these alphabet words combined will result in the actual alphabet to be used. Since some screens in the application have the same interactions, which would result in an alphabet with multiple instances of a certain action, only one instance of an action is saved. This means that pushing the 'back' button and pushing the 'next' button are both saved in the alphabet, but not a second instance of pushing the 'back' button.

When the alphabet has been generated, it is written to a file which can be interpreted by the learner. This file is then read and each of the alphabet words is sent to the teacher to carry them out. The response of the teacher is then saved and, by using the functions provided by the learnlib library, turned into a finite state machine diagram.

6.3.5 Design

To give a better understandig of code of the application, this section will cover the design aspects. An overview will be shown in which the package structure can be found, but also more in-depth explaination will be given on some classes and used architectures. It won't be possible - or exciting - to cover every corner of the code, therefore only the most striking design consideration will be selected.

Packages

In figure 9 an overview is given of the different packages in the Learnlib application. Each of the packages focuses on a certain aspect of the application, which will be discussed below:

com.bunq.learner
: Consists of the Learner class that interprets the responses from the teacher and create a finite state machine diagram from it by using the LearnLib library. Furthermore it contains the SulAdapter class that acts as a communication channel between the Learner and Teacher.

com.bunq.learner.alphabet
: Contains the class needed to parse the created XML files, generate an alphabet and write it to a file.

com.bunq.learner.reset
: When a set of instructions has been carried out by the teacher and returned to the learner, the application environment, and possibly the database, have to be reset; which the classes in this package take care of.

com.bunq.teacher
: This class contains the Teacher class and additional classes it relies on. This includes the instrumentators that handle the connections to the Android and iOS devices.

com.bunq.util
: Since many components of the program can change, for instance when a different menu is created in the application or dependencies are updated, it is useful to only have to chance a single file

in order to keep the program useable. Therefore the util package contains a property class that can read and parse the config files belonging to the application.

com.bunq.main

This packages only contains the Main class that controls the executing of the program.

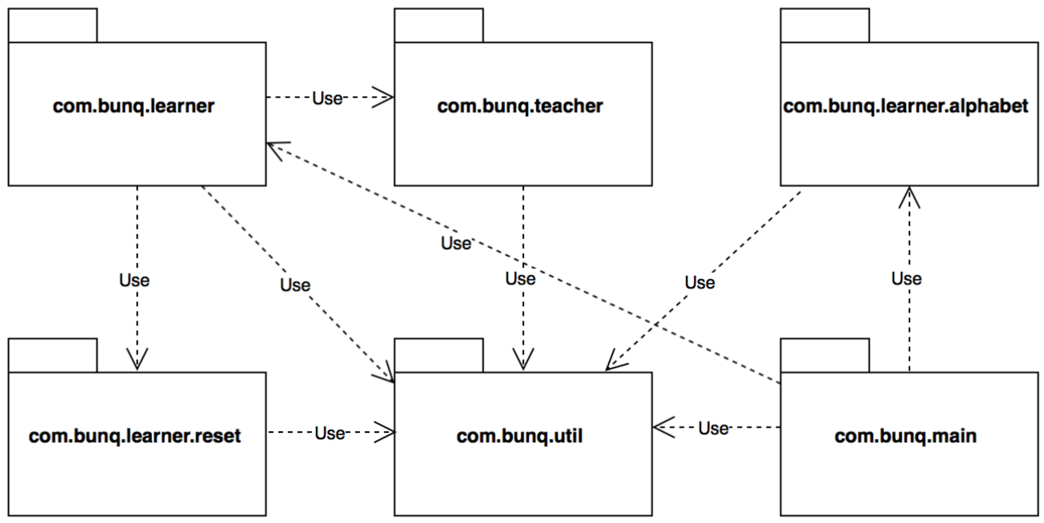


Figure 9: Package overview of the FSM application

Architectural Choices

As explained earlier, not every part of the code is useful to be covered in this report. Therefore only the most important aspects and choices will be illustrated in this section.

Singleton pattern

A singleton design pattern has been used in the Property class. This was done to ensure that the config files that are read, are not read multiple times when different classes need access to a certain property. The config files are not changed during runtime, so whenever the file is loaded, all the properties are already present. Therefore reading the file once is enough and this makes it a perfect example for turning it into a singleton class. Whenever a property is needed in the application, the class checks whether this is the first time or if an instance has already been created. If there's an instance present, this instance is returned instead of reading the whole file again.

Interchangeable

The SulAdapter class takes care of the communication between the learner and teacher. Since the learner only communicates with the SulAdapter and doesn't know of any 'teacher', it would be possible to interchange different teacher implementations without having to adapt the learner. This also makes the application better maintainable, as any future changes to the teacher class can be added without breaking the application itself.

Abstract classes

Whenever functionality was shared between multiple classes, an abstract class was created where possible. This resulted in a SulReset class that is extended by both the JailReset and

DatabaseReset classes. The functionality both classes had in common, was moved to the SulReset class to make it easier maintainable and keep the other classes from getting bloated with duplicated code.

Interfaces

Since there are both an Android and iOS application, and who knows which future operating systems will be supported, it sometimes comes in handy if an interface exists to instantiate certain classes. This is for instance the case with the UINode and Instrumentator classes. The Android and iOS applications might differ, but certain functionality is needed in both cases. That is also the reason why an interface can be perfect here: the functionality can be differently implemented on Android and iOS, but in the end the functionality is there.

Hashsets

Every word that is being generated, should only occur once in the eventual alphabet. Otherwise, certain words would be asked by the learner more frequently than other words, which would not lead to a proper examination of the SUL. To ensure this is done quickly, as there are many identical buttons and fields available on different screens, a hashset has been used to save the words. When adding the words to the hashset, it can immediatly check whether a word is already present and discard it if necessary.

Reflection

In the generated alphabet file that is used by the Learnlib library, function calls are also contained. This means that at runtime the file has to be read, a function has to be extracted from it and called with the specified parameters. In order to achieve this in Java, reflection is used to call the desired function.

By using this, it becomes possible to expand the alphabet files with new functionality by only adding the appropriate functions to the Instrumentator classes. The other parts of the application won't need any adaptations as at runtime the actual functions will be parsed and called. Furthermore the alphabet files differ between operating systems and this makes it possible to create a general function call, without having the need to check which operating system is currently ran. This also creates the possibility to easily extent the application to other operating systems if a bunq app would be created for them.

Performance

Because the user interface (UI) elements of the application are needed to interact, the performance drastically goes down. In order to combat this, a couple of smart systems were implemented to ensure speed could be won where possible. First of all, the application had to be reset to a consistent clean state. To ensure this is always the case, the database can be reset to undo any changes that were made. However, not every test makes changes to the database, so why would the database be reset after every run? Therefore a system was made with different levels of reset states: ranging from only using the 'back' button to get to the initial state, to resetting the whole database if changes were made to it.

Another way in which the application could gain in performance, was by “fast-forwarding” some queries when they became to obsolete. An obsolete query could occur whenever a query contained an action on an UI element that could not be found. Figure 10 illustrates this situation. In run 1, the application is initially in A_0. The query that has to be executed consists of 3 actions, $w1$, $w2$ and $w3$, represented by the arrows. The first action succeeds and takes the application to state A_1. The second action contains a UI element that could not be found; when it would have been found, it would have taken the application to state A_2. However, because the UI element is not present, the action is not properly executed and the application stays in state A_1. The next action can be executed and takes the application from state A_1 to state A_3. Because of the missed element, this query is obsolete: it is covered by queries in run 2 and 3 that are surely to be executed by the learning algorithm. Run 2 has a query containing 2 actions: $w1$ takes the application from state A_0 to A_1; then, it tries to execute $w2$, but the UI element that is needed for the execution of this word cannot be found. The learning algorithm now has found that the sequence $w1$, $w2$ terminates: a UI element cannot be found. Run 3 also contains 2 actions: $w1$ again takes the application from state A_0 to A_1; then, $w3$ is executed; this application takes the application to A_3. This means that the query of run 1 has been covered completely by the queries in run 2 and 3 and thus can be skipped. However, LearnLib needs a result for every query it asks. To work around this problem, each query that contains a word that cannot be executed due to missing UI elements will be “fastforwarded” to the end by immediately returning that the query contains a word that returns a negative result.

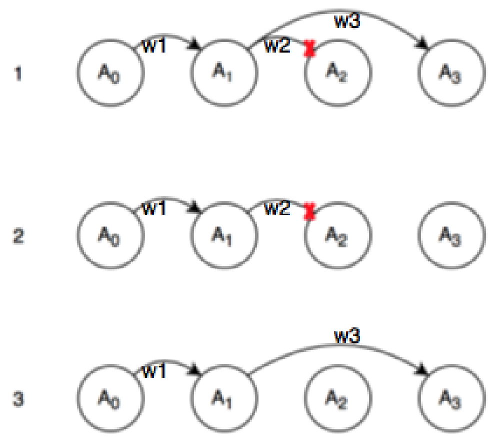
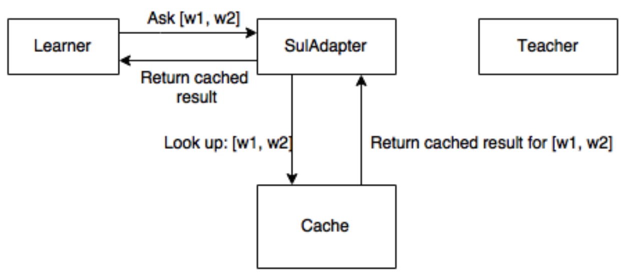
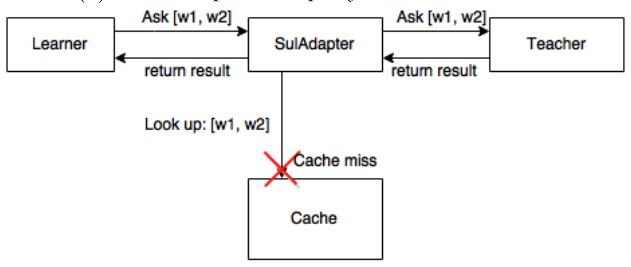


Figure 10: Performance improvement on runs

To improve the performance of the system even more, a cache was programmed. Each query that the learner asks will first be looked up in the cache. A query consists of a series of words; the SulAdapter class, through which the query is asked, first tries to lookup the query in the cache; if the query is in the cache, the result of this cached query is returned without consulting the teacher as can be seen in Figure 11a. Because of the slowness of the teacher, this gives a huge performance improvement. When the query can not be found in the cache, the query will be asked to the teacher, as in Figure 11b. Every query that is asked to the teacher will be saved, so the queries have only to be asked once.



(a) An example of a query that was cached



(b) An example of a query that was not cached

6.4 Results

This section contains the results of the generated finite state machines. As there were some unexpected difficulties, the goals were slightly adjusted and a section covers this adjustment in more detail. Furthermore, examples of generated FSMs are given and the results that can be concluded from them.

6.4.1 Adjusting the Goals

Our first observation concerning the generation of FSMs was that the instrumentation of mobile applications is very slow. All applications that were tried out - UIAutomator, UIAutomation and Appium - suffered from this problem. Although the performance of the learner application was improved a lot by the use of a "fast-forward" mechanism and a cache, the generation of an FSM that covered the whole bunq app would still cost way too long.[1]

A second observation concerns the stability of the application. Because the bunq app is still in its development phase, crashes could occur during the generation of the FSM. When the bunq app crashed, it obviously returned other output then when the app was running normally. LearnLib then tried to compose an FSM from these conflicting outputs, but failed to do so: when the same situation yields different outputs due to a crash, LearnLib does not understand what to do.

Because both the performance of mobile instrumentation as well as the the stability of the bunq app were a hurdle in generating an FSM of the application as a whole, the choice was made to cover only certain parts of the application. The most important parts of the bunq app are all those parts that have to do with payments. Because the likelihood of an app crash is lower when a lower amount of actions is executed and the time needed to generate an FSM of a small part of the app is significantly lower, only the parts of the app in which actual payments are happening are put in the FSM.

6.4.2 Confidential

This section has been removed as it contains sensitive information.

[redacted]

6.5 Conclusion

[redacted]. [redacted]. Although this difference between iOS and Android turned out to be no security leak, we were happy that the generated FSMs showed this difference: it shows that FSMs are useful for analysing applications concerning security, because this difference could have been a severe security issue.

From the FSMs, no security leaks could be deduced. There were no unwanted state transitions, that bypassed authentication or that allowed a user to spend more money than there was in his bank account. However, one cannot conclude that the app is safe based on the generated FSMs, for only certain parts of the bunq app were included.

[1]Generating an FSM with n words and m took about will take O(nm) queries of maximum length O($n+m$) [76]. A set-up with n=4 and m=3 on the bunq app took 5 minutes. This means that every word could be executed in roughly $nm * (n+m)/t = 4 \times 3 \times 4 + 3/(5 * 60) = 0.28$ s An alphabet that would cover the whole application would consist of 50 words; the number of states can only be guessed; 70 seems an appropriate guess to us. This would then take $t * nm * (n+m) = 0.28 * 50 * 70 * (50+70) = 117600$ s $\approx$ 32 hours. Although the cache does reduce this number, it would still be very high. The time to reset the SUL is also not taken into account.

Although the generated FSMs do not reveal any security leaks in case of the bunq app, they do reveal differences in implementation between the Android and iOS variant. [redacted]

7 Fuzzing

Fuzzing is a black box testing technique that can reveal possible security leaks. This section provides more information on fuzzing and which tools are currently available to fuzz test mobile applications. In addition, the limitations of current fuzzing tools will be covered and how this was overcome by creating our own fuzzing application. For creating a new tool, certain goals had to be set and design decisions had to be made. These are also described. This section concludes with what was learned about fuzzing mobile applications with the use of our own tool is summarized.

7.1 Research

Fuzz testing (fuzzing) is a technique of testing, often automated, which uses invalid, unexpected and random data as inputs to the application. The program is monitored on how it reacts to these invalid inputs [77]. Fuzzing is useful for security testing, because the types of bugs that can be found with fuzzing often form a security threat. These types include insufficient error-handling and memory-leaks.

Section 7.1.1 will explain in more detail what fuzzing is and which different fuzzing methods exist. Then Section 7.1.2 discusses the different existing tools that can be used to fuzz test applications. Finally, Section 7.1.3 summarizes what is learned in the research phase and which tools could possibly be used to fuzz test the bunq apps.

7.1.1 Fuzzing Methods

To fuzz test an application, a tool called a fuzzer needs to be written that generates semi-valid input data, submits it to the application and determines whether the application fails [78]. There are two forms of fuzzing: mutation- and generation-based fuzzing. Mutation-based fuzzing takes a valid input, changes it at a random point and sends this mutated input to the application. This is also called 'dumb fuzzing' because it doesn't require any knowledge of the program. Generation-based fuzzing generates inputs based on the documentation or specification of the application and adds an anomaly to them. This kind of fuzzing takes longer and requires knowledge of the program; it is therefore also called 'intelligent fuzzing' [79].

To find out which inputs can be sent to an application, one needs to determine the input surface. The input surface corresponds to all the ways in which an application can be initiated or accessed. An example on how such an input surface can be determined, is illustrated by Mahmood et al. [80]. They used a tool called MoDisco [81] to parse source code which they obtained by decompiling. In addition, they looked at the resources and configuration information that was present. By combining these findings, possible inputs could be retrieved that were then used to fuzz the application. The creators of MobiGUITAR [82], a tool for model-based testing of mobile apps, did something alike by using state-machines to create an input surface for their testing tool.

7.1.2 Tools

There are a lot of fuzzing tools available which are not device or OS specific (e.g. Peach [83], SNOOZE [84], SPIKE [85]). However, a lot of customization has to be done before they will work with Android applications. Because there are also tools available specific for Android, it was decided to focus the research on those instead.

Choudhary, Gorla and Orso [86] recently listed all major testing tools for Android. The testing tools which can be used for fuzzing include: Monkey [87], Dynodroid [88], DroidFuzzer [89] and Intent Fuzzer [90, 91]. Some other testing tools are: Appium [72], Robotium [92], MonkeyTalk [93], MonkeyRunner [94], Sikuli [95], MobiGUITAR [82], but these are not specific for fuzzing. Some of them need the source code while others mainly test the GUI. The benefit of using these tools would be that some of them can also be used for iOS. However, a lot of work will go into creating a real fuzzer out of these tools.

DroidFuzzer
: Only generates inputs for activities that accept MIME data types. DroidFuzzer is not publicly available.

Intent Fuzzer
: Intent Fuzzer mainly tests how an app can interact with other apps. There is only very limited documentation available. The fuzzer has to be installed on the device itself. From there an application which is present on the device can be selected and fuzzed. The aim of the application is to use $NULL$ values on inputs that expect a value, as this often means that they will fail, which will result in a crash.

Monkey
: Monkey is most frequently used and implements the most basic random strategy. It is developed by Google and is standard available in Android, which means it can be directly used on an emulator or USB connected device. The program is run on a computer on which an emulator is installed or a device is connected to, and is able to generate all kinds of inputs: from system inputs to touch and click events. The use of Monkey is well documented and indicates all possible options that can be used with it.

Dynodroid
: Dynodroid has some functions that make its exploration more efficient compared to Monkey, but it is less used. Dynodroid has some documentation, albeit outdated since it focuses on Android 2.3. After contacting the developer of the tool, a newer version was sent which is able to be used with Android 4.3. This version however needs certain changes in the standard UIAutomator that is integrated in Android, meaning that the Android sources have to be modified and a new image has to be built.

7.1.3 Research Conclusion

Fuzzing is a black box testing technique that comes in two forms: mutation-based and generation-based fuzzing. Either of those forms can be a useful method to find bugs that form a possible security threat.

DroidFuzzer is not publicly available, therefore it is not used. Intent Fuzzer has shown to be effective at revealing security issues. However, this application is not used often and it requires to manually select each activity that should be targeted. Therefore, it is not sufficient for testing the entire bunq app. The fuzzing tools Dynodroid and Monkey seem to have sufficient functionality to fuzz test the Android bunq app and both of these apps were considered to be used. However, after using the tools on the bunq app, it appeared they did not fulfill our needs. We will go more in depth on this matter in Section 7.3.

7.2 Goals

The goal of using fuzzing for our project is to find bugs. As explained in Section 4 all bugs should be seen as a security risk, because there is a possibility it could be misused by an attacker. With fuzzing we do not try to assess the found bugs in terms of their risk and how they could possibly be misused, but simply aim at finding any bugs in the first place. After some bugs have been found, they are looked at individually to see if they form a real security issue.

Fuzzing is used because it is a black box testing method which could also be used by potential attackers who do not have access to the source code of the application. The focus lies on the Android application of bunq. In theory the iOS application should also be able to be tested using our fuzzing approach, but due to time constraints we did not bring this in practice.

7.3 Approach

As indicated earlier, the found tools did not fulfill the our needs. In this section it is explained what the drawbacks of the tools were and which functionalities they lacked. Because the tools were not useable, the decision was made to create our own custom fuzzing tool. The design goals of this tool can be found in Section 7.3.2 and the design itself in Section 7.4.

7.3.1 Drawbacks of Existing Tools

All the researched tools posed drawbacks. Even the most promising ones, Monkey and Dynodroid, did not offer enough advantages to be used for the bunq app. The three main issues with the fuzzing tools are listed below:

Documentation and Version

Since most fuzzing tools are developed by researchers who did not put a lot of effort into writing documentation, configuring some of the tools took a lot of time. The lack of documentation also makes it hard to understand how specific functionality of the fuzzing tools work and what it would generate as output. Furthermore, some tools are not actively maintained, for instance Dynodroid, which makes them incompatible with the newer versions of Android on which the bunq app runs. Even the newest version that we got from the developer of Dynodroid himself, did not run on the current Android versions without modifying core files.

Randomness

Both Dynodroid and Monkey generate user interface (UI) input that is too random to be actually of use. For the bunq app, some structure is needed to perform basic actions, such as making a payment, as it depends on other actions. With Monkey this immediately resulted in problems, as it always pressed buttons that returned it to the previous screen, or filled in random text when an amount or person was expected. Configuring Monkey differently did not yield better results and also the configuration possibilities were very limited.

Another common issue with the random fuzzing tools is that they press the logout button and are not able to login again. As a result, they are stuck on the login screen for most of the time which will not yield viable results.

When it was clear that the fuzzing tools where too random, other fuzzing tools were considered such as MonkeyTalk, Monkeyrunner and others. However, for most of these tools a change in the source code is needed before it can be used. Besides, these tools had the reversed issue of not doing any random actions at all. These random actions are useful because they by doing some random actions you will be able to find unexpected bugs. As explained in the research phase in Section 7.1.

Recognizing Bugs

Another problem is that the existing fuzzing tools do not recognize all application crashes and do not recognize any functional bugs. Most of them do report it when an application fully crashes, but not the exact reason that lead to the crash. Furthermore, none of them recognizes functional bugs (bugs that do not make the application crash, but resulted in a system behaviour than it should).

Because of all the drawbacks the existing tools had, the decision was made to create our own fuzzing tool. This fuzzing tool is not too random, as it has the ability to follow certain paths with additional random inputs. It can therefore complete a payment, but also tries to feed the application with random inputs, such as invalid amounts. The fuzzing tool uses knowledge of the structure of the app to explore it and is therefore considered to be a generation-based, or 'intelligent', fuzzer for mobile applications.

7.3.2 Design Goals

With our fuzzing tool we want to make a proof of concept for our idea of fuzzing that is not completely random, but follows a given structure. To emphasize this point the fuzzing tool will not be specific for the bunq app or OS. However, the tool has been configured in such a way that the Android bunq app can be tested. However, a user of the tool should be able to change this configuration to test other apps on both Android and iOS.

Obviously, the problems that were encountered by using the existing tools should be avoided. This will be done by following the methods described in Section 8. The three drawbacks of the existing tools will be countered in the following ways:

Documentation and Version

The tool should be well documented and work with the latest version of Android. The tool should be easy to configure for different apps. This is accomplished by writing a readme that fully explains how the tool has to be set up and which options can be configured. For developers, it should be easy to modify the source code, so that they can get the tool to work under other operation systems. This is done by creating javadoc for all the methods and classes. In this way the developers are able to fully understand the source code of the tool and thus change it if they need to.

Randomness

The randomness of the tool should be able to be configured according to the tester's wishes. This can be done by the use of XML configuration files. These are the files which the testers need to write specifically for the app. In these files he should be able to tell the tool what testing paths it should take and how much random actions it should take in between.

Recognizing Bugs

The tool should be able to recognize system crashes and functional bugs. It should also report what steps it took to produce these bugs, so that they can be reproduced later on.

7.4 Design

Since not all of the written code is useful to discuss, only some highlights are portrayed in this section. This includes the parts for which design choices needed to be made, or where special code structures were used. The following section will layout the global set-up of the fuzzing tool. The sections thereafter will go further into the different parts of the tool; this includes the configuration files and package structure.

7.4.1 Global Set-up

The fuzzer exists out of three parts. The first part are the XML files. These files are created by the tester and hold paths through the app that the fuzzer should test with a certain chance. The other two parts are two programs. An UML diagram of these two programs is shown in figure 16.

One program, the bunqFuzzer-pc-client runs on a computer. It parses the XML files to actions and input objects and sends these objects as UI instructions to the server.

The other program, called bunqFuzzer-android-server, is a server that runs on the android device which runs the app under test. The bunqFuzzer-android-server application gets instructions from the bunqFuzzer-pc-client. These instructions are executed by the bunqFuzzer-android-server on the device.

Because of the division into two programs it is also possible to make a server on iOS or Windows Phone which translates the same instructions from the client to UI interactions. Although the earlier described FSM learning application uses Appium to execute its UI interactions, the fuzzer uses UIAutomator to execute actions on Android devices. Appium makes it easier to make the translation from Android to another operating system, but it also slows down the application. Because fuzzing uses random operations which do not always make sense, you want to do a lot of them. Therefore it is important that you are able to send a lot of instructions within a certain time period.

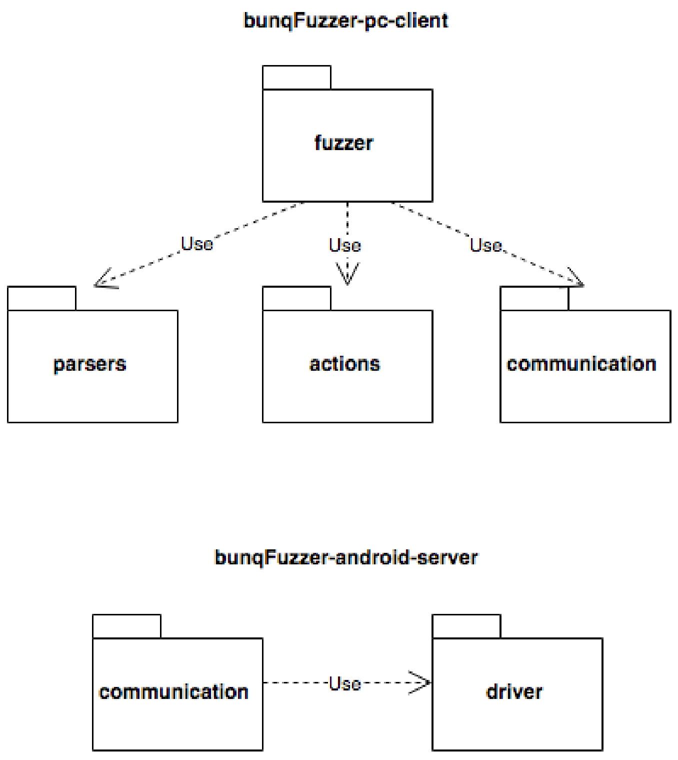


Figure 12: Fuzzer applications package overview

7.4.2 Configuration Files

The fuzzer is instructed by XML files. These files are created by the tester. They hold information about all the possible actions that can be executed on the app under test. Appendix D shows an example of such an XML file.

The root node of these XML files is an `actionset` node. An `actionset` holds other actionsets and actions. An `action` is a single UI instruction that can be executed on the device. Sometimes an `action` holds an input. For example, when something needs to be entered into a textfield the `action` and its input could look like this:

```
<action operation="insert" id="com.bunq.android:id/etAmount"> 100 </
    action>
```

An input for such a field could also exist out of an `inputlist`, then it would look like this:

```
<action operation="insert" id="com.bunq.android:id/etAmount"> <file
    name="amount.xml" /> </action>
```

An `inputlist` is an separate XML file which holds multiple inputs. When an `action` is sent to the device to be executed one input out of its `inputlist` is randomly chosen.

There are three attributes that can be set to an `actionset`:
chance indicates the probability that the actions that this `actionset` hold are sent to the device.
times is the amount of times the actions that this `actionset` holds are executed.
order can be set to one, random or inorder:
- **one** indicates that only one `action` in this `actionset` will be sent to the device.
- **random** indicates that the actions will be sent in a random order.
- **norder** indicates that the actions will be sent in the order they appear in the XML file.

An `action` itself can also have three attributes:
chance if it is set the action will only be sent with this probability to the device.
operation the kind of UI operation that the device should do.
id the id of the UI element that the operation should be executed on.

An input can also hold a chance parameter. This indicates the chance that this input is selected out of the inputlist to act as the actual input of an `action` when it is executed on the device.

7.4.3 bunqFuzzer-pc-client

The bunqFuzzer-pc-client has three functions. It has to read the XML files and parse them to action objects. The second function is holding the action objects and translating them to ActionInstructions while taking their chances into account. Finally, it has to send these ActionInstructions to the server. These functionalities are separated by the use of packages.

com.bunq.parsers

This package holds two parsers: the ParseActionXML and ParseInputXML classes. These classes both extend the abstract class ParseXML. ParseXML holds functions that both of the parser classes need.

ParseActionXML uses recursion to make a tree structure of all actionsets and actions of the XML file it is reading. The ParseActionXML class can return the root actionset. To keep a clean structure in the XML files it is possible to nest files in one another.

ParseInputXML is called by ParseActionXML when it encounters an inputlist file. It creates an inputlist object and the input objects that it holds. And returns this inputlist to ParseActionXML which on his turn adds this inputlist to the action object that it should hold.

com.bunq.actions

The actions package holds the ActionSet, Action, InputList and Input classes. These are all the classes needed to create ActionInstructions objects that can be sent to the server. An ActionSet object holds other ActionSets and Actions. An Action object can hold an Input or an InputList with Inputs.

ActionSet and Actions implement the interface IAction. This means that they both should have an `createActionInstructionList()` method that returns an ArrayList of ActionInstructions. When this method is called on the root ActionSet it recursively calls the `createActionInstructionList()` method of all the actions it holds. All the lists of ActionInstructions are recursively combined in one ArrayList of ActionInstructions, which can be sent to the server.

`createActionInstructionList()` calculates in which order the Actions of an ActionSet should be sent and how many times they should be sent. It also calculates the probability whether they should be sent. For Actions holding an Inputlist, it asks the InputList to calculate which input should be used.

com.bunq.communication

The communication package holds the ActionInstruction class that can be created by the Actions. An ActionInstruction is the object that is actually sent to the server. It is a different object than the Action objects themselves because it has to be much smaller and should only hold the data that the device needs to execute the UI operation.

ClientSocketWrapper is the other class in communication which sends ActionInstructions to the server over a socket and returns the response from the server.

7.4.4 bunqFuzzer-android-server

The bunqFuzzer-android-server has two functions. First it needs to start a server socket to communicate and receive ActionInstruction objects from the client. Secondly it has to execute these instructions; this is done by UIAutomator. These two functions are again separated by the use of packages.

The server is kept really small on purpose. By keeping it small and letting the client do all the calculations and parsing of XML files it will be easier to implement another server for iOS or another OS in the future.

com.bunq.communication

This package holds the ActionInstruction class, which is the same as on the client. An instance is created when an ActionInstruction is received by the ServerSocketWrapper. The ServerSocketWrapper connects with the ClientSocketWrapper on the client side..

com.bunq.driver

This package holds two classes that extend UiAutomatorTestCase and can be started on an Android device by an adb call. UIAutomator is integrated in Android. It makes it possible to trigger UI interactions without physical interaction.

7.5 Results

This section will present the results of our own fuzzing tool. First the implementation of the tool itself is discussed. Then the choices in the XML files specific for testing the bunq app are explained. Lastly, the bugs that are found with this tool are examined.

7.5.1 Implementation Results

Although the design of using our own server with UIAutomator makes the fuzzing tool quicker then by using Appium, it still is not very fast. Therefore, it is not able to execute as many actions as desired. Currently, the only way to improve this speed is by directly using ADB to send UI commands. This approach has a big downside; using ABD it is impossible to select UI elements at runtime. To be able to interact with UI elements, the location of all the elements that have to be clicked should be known by coordinates, which then should be put in the XML files. Constructing the XML files will then become a time consuming task, which is not desired.

The design goals set in Section 7.3.2 where met in the following ways:

Documentation and Version

The tool is well documented as can be seen above. Besides this report there is also a readme file available which gives basic instructions on how to set up the tool and construct the XML files. Because of the separation between client and server and the small size of the server, it is easy to adapt the server when someone would like to use the tool on another OS.

Randomness

The randomness can be completely configured by the tester when he composes the XML files. This was the most important aim of the custom tool. However, there is still room for improvement: more parameters could be added to the XML files, so that the tester could have even more control.

Recognizing bugs

The tool does recognize bugs and fully reports all steps it takes to produce them. However, the tool is not able to discover functionality bugs itself. In order to discover functional bugs, the tester should monitor the application under test while it is being fuzzed. The tool missed some errors of the app due to its inability to discover functional bugs. This will be further addressed in Section 7.5.3.

7.5.2 Bunq Specific Configuration Files

The XML files to test the bunq app with our tool are divided in actionset and inputlist files. For each window an actionset file was created. One main file refers to all the other actionset files. Almost all possible paths though the app can be taken by following the actions in these files. Due to the limited time the settings menu is not fully included in the actionsets.

The paths that lead to a payment have gotten the most thought, because bugs in these paths could lead to the biggest issues for bunq. Also all possible input fields that are sent to the server have gotten extra attention, because they might be vulnerable to injections.

To reveal possible injections, all sorts of injections are included in the inputlist files. These injections can be inserted in all input fields. To make sure that a payment can be sent, the probability that the input fields for creating a payment get valid inputs was set to be relatively high.

7.5.3 Bug Results

In the process of making this fuzzer and testing it on the bunq app, several bugs where found. The first one was a bug that typically would not have been found with general testing techniques. It turned out that the Android application of bunq was unable to handle really large numbers and therefore would crash when you tried to make a really large payment. With classical testing techniques, one would probably test some edge cases such as zero, a negative and a positive number. Because the fuzzing tool did pick a random input, it discovered that the application crashed when very large numbers were entered. This bug has been fixed and did not form a real security risk.

The next bug was one that classic testing techniques would also have found. In the bunq app it is possible to make a payment reoccur at a certain date. When you would set this date and send the payment to the server the app would crash due to a problem with the formatting of this date. This bug has been fixed and did not form a real security risk.

[redacted]

[redacted]

[redacted]

7.6 Conclusion

Fuzzing has shown to be a testing technique that can reveal security issues of an application. There are some existing tools to fuzz test mobile applications. However, these have the issue of taking a testing approach that was too random for our purposes: in our case, they were not able to test the entire application.

Our own tool overcomes this issue. Due to the design of our tool it is possible to adjust it to test any app on any operating system. The tool has two drawbacks, the first being that the it is currently not capable of recognizing functional bugs. The second drawback is concerned with performance: the execution of actions on the UI is quite slow. Both of these issues are hard to improve. Further research should be done to find solutions for them.

For bunq the tool discovered some bugs that helped improve the quality of their app. [redacted]

8 Code Quality

Since the aim of both the Learnlib and fuzzer application is to be used in testing the bunq application for possible flaws, they themselves should work as expected. The diagrams generated by the Learnlib application should not result in false interpretations and the fuzzer should not encounter problems due to bugs in its own implementation. Therefore both the applications were thoroughly tested to minimize the risks of errors and inconsistencies. Furthermore, they are both software applications and should always be tested to ensure that everything behaves as expected.

8.1 Testing Framework

Both the Learnlib application and the fuzzer are written in Java, therefore JUnit [96] was chosen to be used for creating and conducting tests. For each of the classes that were written, a testclass was created to ensure everything works as intended. Because some classes depend on other classes, and since it is not feasible to create valid instances of every class it depends on, a mocking framework called Mockito [97] was used to mock these dependencies. This made it possible to test functionality that would otherwise not be possible with the standard testing methods that JUnit offers. There were cases in which even Mockito did not provide enough functionality, therefore PowerMockito [98] was used to extent the functionality of Mockito. This made it possible to also mock static classes and constructors.

8.2 Code Coverage

To make sure the tests did not miss any - important - functionality of the code, code coverage tools where extensively used. The code was written in Java by using the Eclipse IDE [99], EclEmma [100] is a well known tool that can be used in Eclipse to calculate code coverage. However, it turned out that Eclemma did not collaborate well with the PowerMockito framework and because of that the coverage reports were not reliable anymore. This resulted in a switch to Clover [101], which was able to work together with PowerMockito and also added the ability to collect metrics on the written code. Everytime something changed in the code, new tests were added and the Clover program was run to create a report. This made sure that everything that could be covered by testing was actually covered by the created tests.

8.3 Quality Assurance

Next to code coverage and traditional testing, the code was reviewed in multiple ways to ensure that it was also of a high quality. First of all, an overview was made to indicate what we believe to be code of good quality. The full overview can be found in Appendix E. This overview was used while writing the code to make sure we adhered to it.

Next, the metrics that Clover provides were used to get a better understanding of the code. The metrics showed for instance the complexity of classes and methods, and made it possible for us to refactor them where necessary. By lowering the complexity of classes, they could be tested more easily and this result in less bugs in the applications.

Another way in which quality was assured, was by peer-reviewing each other's code. This resulted in finding small errors which were overlooked by the original creator and also refactoring of certain functionality as it could be implemented differently. Not only was the code reviewed by our own group, but also by someone from bunq itself.

And last, but certainly not least, the written code was sent to the Software Improvement Group (SIG) [102]. They assessed the code twice and responded with an indication of what could be improved and gave an overall grade each time. After the first assessment the code received a 4.5 out of 5, meaning that it was maintainable above average. The feedback contained only one remark that a few methods were too long and could be split up. The first feedback of the SIG can be found in Appendix F. After the second assessment, the SIG noticed that all the long methods were refactored out of the code; however, the system had grown a lot in the time between the first and the second evaluation, which lead to a slightly lower score on other factors. This lead to a score of again 4.5 out of 5. They wrote that it is normal that a growing systems slightly lowers the score, and they did not have any point for further improvement. Their conclusion said that we learned from their earlier remarks and that our high score indicates that the systems are well maintainable. The remarks concerning the second assessment can be found in Appendix G.

9 Additional Findings

As explained in Section 2.4, the scope of the project was focused on the local application. However, while testing the bunq applications, it sometimes resulted in security vulnerabilities or bugs that fell outside of the defined scope. Some of those things were investiged a bit more and will be shortly explained in this section.

9.1 Unlimited Pin Tries

It is possible to login on multiple accounts on a single device, a security vulnerability was discovered. As a user, you can try three times to fill in the right pincode, after which a temporary block is issued. However, it turned out to be possible to circumvent this blockage by filling in a wrong pincode twice and then login in a different account. After the successful login, the amount of pin tries get reset and it is possible to try two more pincodes on the other account. With the right setup, one would be able to keep on trying different pincodes for an account by simply resetting the amount of tries each time. Eventually, this will result in the right pincode and the account being compromised. This problem has been solved immediately by bunq.

9.2 Denial of Service

[redacted]

9.3 Bug Reporting

If ACRA (Application Crash Report for Android) is not able to send its crash report due to a network error, this report is stored on the device. This crash report contains a lot of information about the hardware and the state of the system at the time of the crash, but luckily no privacy sensitive data. Georgiev et al. reported in 2012 in their paper [103] that ACRA (prior 4.4) overrides the default trust manager and claimed that any app using ACRA is insecure against man-in-the-middle attacks. In a response ACRA stated that the content of their reports where indeed vulnerable to a man-in-the-middle attack, but the app using ACRA would not be fully insecure. ACRA updated to make a secure connection possible: *"ACRA v4.4.0 has been modified to use SSL certificate validation by default. If you send your reports to your own server via SSL with a self-signed certificate, you have to set the option* `disableSSLCertValidation` *to true (annotation or dynamic config)."* [104]

[redacted]

[redacted]

10 Discussion & Recommendations

In this section, we reflect on the contents of this project. The methods that are used are evaluated and we try to point out which parts of the research could be improved, and what we would do differently if we had more time.

10.1 General Mobile Application Security

At the start of the project, none of the team members possessed a lot of knowledge on security in software, let alone mobile application security. That's why all the research had to be done from scratch. This implies that it is possible that some important points concerning the security of Android or iOS have been overlooked. The section on general mobile security can be seen as an inexhaustive list: probably in a few months, new methods for bypassing the system's security will be found. However, given the knowledge at the start of the project, we are happy with the fact that we have researched the field of mobile app security to this extent; we were even able to use several known exploits, such as tapjacking and retrieving private keys on Android.

10.2 Decompiling

Decompiling the Android application went better than expected and produced a very useful result. [redacted].

An improvement for the decompilation procedure, would be to have more understanding of the assembly language. This would prove very helpful for decompiling the iOS application, as it produces assembly code which was not possible to edit due to our limited understanding of it.

10.3 Generating Finite State Machines

Considering the generation of FSMs, several things could be improved. The first point that could be done differently, is the moment of generating the FSMs. Because the bunq app is still in development, crashes could occur while running the FSM-learner program. Often, the result of such a crash was that the learning process had to be restarted. Having a fully stable version of the bunq app would improve the learning process a lot. This would also mean that an FSM of the entire app could be generated when time is not an issue.

Another issue concerns the implementation of the learner application. Because the learning process is very slow due to the used mobile instrumentation software (and due to the earlier mentioned instability), we were not able to generate an FSM of the bunq app as a whole. A possible gain in performance could be achieved by the use of multiple devices simultaneously. Each of these devices should then be instructed by the same learner application; however, they should be connected to a dedicated database, as the different mobile devices should not be able to affect each other's state. Because the time of this project was limited, this approach could not be implemented.

10.4 Fuzzing

Similar to the learner application, the fuzzer also has room for improvement. The first issue that we would like to mention also occurs with the learner application: the speed of the application is not very high. This could be improved by the use of ADB; however, as has been pointed out earlier, this approach comes with its own drawback in the form of very complicated configuration files. If the performance cannot be improved, the fuzzer should only be ran at night, so that it does not take costly developing time.

The second improvement is concerned with the working of the fuzzer. Our tool is able to spot application crashes. However, some bugs do not crash the application but alter the state of the application in an illegal way. These functional bugs cannot be spotted by the fuzzer. This is a drawback, because the fuzzer now can sometimes trigger illegal behaviour without noticing it. A possible solution could be to implement an invariant, that should always be true. When the invariant is not true anymore, it is likely that a functional bug has been found. However, the implementation of such an invariant is not easy; due to the limited time, this could not be done during this project.

Lastly, the fuzzer could give the tester even more control. The fuzzer reads XML configuration files and bases its actions on those files. By expanding the number of possible actions, the application could be tweaked to behave exactly as the tester wants it to.

10.5 Code Quality

Both the fuzzer and the learner application were tested thoroughly and well documented. The applications made use of packages and classes so that classes and methods with similar functionality could be grouped together. This can also be concluded from both SIG evaluations, as they indicated that the software was of above average quality.

10.6 Overall

The combination of the learner application and the fuzzer provides even more testing possibilities. A tester can first use the learner to discover the paths that are present in the application under test. The tester could then put these paths in the configuration files of the fuzzer in order to see how those paths react to all kinds of input.

Overall, we were happy with how the project evolved. Due to the limited time, the software has still some room for improvement and the methods we used to assess the security of bunq's applications could be researched in more depth. However, given the duration of the project and the knowledge at the start of the project, we are proud that the methods we applied to the bunq app to assess its security yielded the results as presented earlier in the report.

We also made some recommendations to bunq, which include an encouragement to keep on using the developed application as they can be used for additional testing and proved to be successful. In order to do this, it is strongly advised to optimalise the speeds of the applications where possible. Furthermore, bunq should keep updated on the latest trends in Android and iOS development, to make sure the apps are secure against the latest vulnerabilities.

11 Conclusion

At the start of the project, bunq wanted to see a way in which their mobile application (app) could be tested on possible security risks. Therefore, research was done into different techniques and how these techniques could be used in assessing security. This lead to multiple black box approaches that were successfully carried out during the project.

These approaches included generally known security vulnerabilities on both iOS and Android. Examples of these are data storage leaks, tapjacking, task manager snooping and root access which all showed areas on which could be improved. So, by doing research in the general security of an operating system, potential security vulnerabilities were already discovered.

Decompiling the app was a success, which showed that an app can never be fully trusted as someone might have changed parts of it. [redacted].

Furthermore, two black box testing applications were developed to generate finite state machines and to fuzz the mobile application. Due to time constraints and the current instability of the bunq application, it was not possible to generate an FSM of the full application. For the parts that were examined, no security issues were found, but it did show implementation differences. These differences indicate a small lack of communication between the two development teams, which could potentially lead to security issues.

The fuzzing application did find security vulnerabilities and can certainly find more defects in the future, because it makes use of random inputs. Traditional testing methods can never fully cover all possible inputs to an application, meaning that the randomness of the fuzzer has the advantage of being able to find flaws that are normally missed.

By just using the application, we also encountered security issues that fell outside our scope; these bugs often involved the behaviour of the back-end or the communication with it.

To conclude, the applied black box testing methods showed to be useful, as defects have been found that were missed by traditional testing methods. During the project, the techniques can therefore be considered to be successfully applied. Because the two applications were created, it is possible for bunq to keep on testing their future apps after the project has been concluded. The used methods showed also that an app in itself is not safe; we would like to advise to bunq to consider each app as under control by a potential attacker and keep the backend as safe as possible, as the backend is much harder to compromise.

References

[1] bunq. [Online] Available: `http://www.bunq.com`. Accessed on: 29-April-2015.

[2] Arxan 2014 state of mobile app security. [Online] Available: `https://www.arxan.com/wp-content/uploads/assets1/pdf/State_of_Mobile_App_Security_2014_final.pdf`. Accessed on: 29-April-2015.

[3] Apple - iOS 8. [Online] Available: `https://www.apple.com/ios`. Accessed on: 21-April-2015.

[4] Android. [Online] Available: `https://www.android.com`. Accessed on: 21-April-2015.

[5] C. Simmons, C. Ellis, S. Shiva, D. Dasgupta, and Q. Wu. Avoidit: A cyber attack taxonomy, 2009.

[6] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[7] T. Wu. The SRP Authentication and Key Exchange System. RFC 2945 (Proposed Standard), September 2000.

[8] Deloitte. [Online] Available: `http://www.deloitte.com/nl/nl.html`. Accessed on: 11-June-2015.

[9] Google android security presentation. [Online] Available: `https://docs.google.com/presentation/d/1YDYUrD22Xq12nKkhBfwoJBfw2Q-0ReMr0BrDfHyfyPw/pub?start=false&loop=false&delayms=3000&slide=id.g1202bd8e5_05`. Accessed on: 4-june-2015.

[10] Google blog on play store verification. [Online] Available: `http://android-developers.blogspot.co.uk/2015/03/creating-better-user-experiences-on.html`. Accessed on: 4-june-2015.

[11] Source android security. [Online] Available: `https://source.android.com/devices/tech/security/`. Accessed on: 4-june-2015.

[12] Lifehacker blog article on android security. [Online] Available: `http://lifehacker.com/how-secure-is-android-really-1446328680`. Accessed on: 4-june-2015.

[13] Developer android security tips. [Online] Available: `http://developer.android.com/training/articles/security-tips.html`. Accessed on: 4-june-2015.

[14] Developer android permission element. [Online] Available: `http://developer.android.com/guide/topics/manifest/permission-element.html#plevel`. Accessed on: 4-june-2015.

[15] Toasts. [Online] Available: `http://developer.android.com/guide/topics/ui/notifiers/toasts.html`. Accessed on: 19-june-2015.

[16] Android forums about tapjacking. [Online] Available: `http://androidforums.com/threads/commonsguy-tapjacking-example.842850/`. Accessed on: 17-june-2015.

[17] Stackoverflow about tapjacking. [Online] Available: `http://stackoverflow.com/questions/22179773/android-tap-jacking-how-to-prevent-it`. Accessed on: 17-june-2015.

[18] Shaun Colley Ollie Whitehouse Dominic Chell, Tyrone Erasmus. *The Mobile Applicaiton Hacker's Handbook*. John Wiley & Sons, 2015.

[19] Security of ios. [Online] Available: `http://www.apple.com/business/docs/iOS_Security_Guide.pdf`, 2015. Accessed on: 10-June-2015.

[20] Inter app communication. [Online] Available: https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html#//apple_ref/doc/uid/TP40007072-CH6-SW2. Accessed on: 10-June-2015.

[21] Entitlements in ios. [Online] Available: https://developer.apple.com/library/ios/documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/AboutEntitlements.html. Accessed on: 11-June-2015.

[22] Apple enterprise program enrollment. [Online] Available: https://developer.apple.com/programs/enterprise/enroll/. Accessed on: 11-June-2015.

[23] App store. [Online] Available: https://itunes.apple.com/nl/genre/ios/id36?mt=8. Accessed on: 11-June-2015.

[24] App store review guidelines. [Online] Available: https://developer.apple.com/app-store/review/guidelines/. Accessed on: 11-June-2015.

[25] Keychain ios. [Online] Available: https://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile/why-ios-is-safer-than-Android.aspx. Accessed on: 19-june-2015.

[26] Rootcloak. [Online] Available: http://repo.xposed.info/module/com.devadvance.rootcloak. Accessed on: 11-June-2015.

[27] Apple push service. [Online] Available: https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html. Accessed on: 17-june-2015.

[28] Google cloud messaging. [Online] Available: https://developers.google.com/cloud-messaging/. Accessed on: 17-june-2015.

[29] Apple ios background execution. [Online] Available: https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html. Accessed on: 18-June-2015.

[30] Security intelligence, android keystore buffer overflow. [Online] Available: http://securityintelligence.com/android-keystore-stack-buffer-overflow-to-keep-things. Accessed on: 17-june-2015.

[31] Tapjacking example. [Online] Available: https://github.com/mwrlabs/tapjacking-poc. Accessed on: 17-june-2015.

[32] WG Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[33] David Endler. The evolution of cross site scripting attacks. Technical report, Technical report, iDEFENSE Labs, 2002.

[34] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[35] N.A. Naeem and L. Hendren. Programmer-friendly decompiled java. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 327–336, 2006.

[36] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51, Jan 2002.

[37] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[38] Ida pro. [Online] Available: `https://www.hex-rays.com/products/ida`. Accessed on: 29-April-2015.

[39] Jeb. [Online] Available: `https://www.pnfsoftware.com`. Accessed on: 29-April-2015.

[40] Jad. [Online] Available: `http://varaneckas.com/jad`. Accessed on: 29-April-2015.

[41] Vts. [Online] Available: `http://virtuous-ten-studio.com`. Accessed on: 29-April-2015.

[42] Apkstudio. [Online] Available: `https://apkstudio.codeplex.com`. Accessed on: 29-April-2015.

[43] Dex2jar. [Online] Available: `https://github.com/pxb1988/dex2jar`. Accessed on: 29-April-2015.

[44] Jd_gui. [Online] Available: `http://jd.benow.ca`. Accessed on: 29-April-2015.

[45] Axmlprinter2. [Online] Available: `https://code.google.com/p/android4me/downloads/list`. Accessed on: 29-April-2015.

[46] Apktool. [Online] Available: `http://ibotpeaches.github.io/Apktool`. Accessed on: 29-April-2015.

[47] Smali2java. [Online] Available: `https://github.com/darkguy2008/smali2java`. Accessed on: 29-April-2015.

[48] Decompile android. [Online] Available: `http://www.decompileandroid.com`. Accessed on: 29-April-2015.

[49] Android manifest. [Online] Available: `http://developer.android.com/guide/topics/manifest/manifest-intro.html`. Accessed on: 12-May-2015.

[50] Dare. [Online] Available: `http://siis.cse.psu.edu/dare/index.html`. Accessed on: 29-April-2015.

[51] Ded. [Online] Available: `http://siis.cse.psu.edu/ded/index.html`. Accessed on: 29-April-2015.

[52] Soot. [Online] Available: `http://sable.github.io/soot`. Accessed on: 29-April-2015.

[53] Hopper. [Online] Available: `http://www.hopperapp.com`. Accessed on: 29-April-2015.

[54] Xcode. [Online] Available: `https://developer.apple.com/xcode`. Accessed on: 29-April-2015.

[55] Clutch. [Online] Available: `https://github.com/KJCracks/Clutch`. Accessed on: 29-April-2015.

[56] Proguard. [Online] Available: `http://proguard.sourceforge.net`. Accessed on: 12-May-2015.

[57] Google play store. [Online] Available: https://play.google.com/store. Accessed on: 11-June-2015.

[58] Itools. [Online] Available: http://itools.en.uptodown.com. Accessed on: 9-June-2015.

[59] M.E. Joorabchi and A. Mesbah. Reverse engineering ios mobile applications. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 177–186, Oct 2012.

[60] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. Protocol state machines and session languages: specification, implementation, and security flaws. 2015.

[61] Learnlib. [Online] Available: http://www.learnlib.de. Accessed on: 28-April-2015.

[62] Fides Aarts, Erik Poll, and Joeri de Ruiter. Formal models of banking cards for free! *Unpublished*, 2012.

[63] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *ICGI*, volume 21, pages 4–18. Citeseer, 2012.

[64] Libalf. [Online] Available: http://libalf.informatik.rwth-aachen.de. Accessed on: 28-April-2015.

[65] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.

[66] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. In *COLT*, volume 91, pages 128–138, 1991.

[67] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. *Automata Learning with On-the-Fly Direct Hypothesis Construction*. Springer, 2012.

[68] Borja Balle. Implementing kearns-vazirani algorithm for learning dfa only with membership queries. In *ZULU workshop organised during ICGI*, pages 12–19. Citeseer, 2010.

[69] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification*, pages 307–322. Springer, 2014.

[70] Ronald L Rivest and Robert E Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[71] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *IJCAI*, volume 9, pages 1004–1009, 2009.

[72] Appium: Mobile app automation made awesome. [Online] Available: http://www.appium.io. Accessed on: 30-April-2015.

[73] Uiautomator. [Online] Available: https://developer.android.com/tools/testing-support-library/index.html#UIAutomator. Accessed on: 9-June-2015.

[74] Uiautomation. [Online] Available: https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html. Accessed on: 9-June-2015.

[75] Android developers | javax.xml.xpath. [Online] Available: http://developer.android.com/reference/javax/xml/xpath/package-summary.html. Accessed on: 26-May-2015.

[76] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[77] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[78] Peter Oehlert. Violating assumptions with fuzzing. *Security & Privacy, IEEE*, 3(2):58–62, 2005.

[79] Charlie Miller. How smart is intelligent fuzzing-or-how stupid is dumb fuzzing. *Defcon 15*, 2007.

[80] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22–28. IEEE, 2012.

[81] Modiscor. [Online] Available: `http://www.eclipse.org/MoDisco`. Accessed on: 29-April-2015.

[82] Domenico Amalfitano, A Fasolino, Porfirio Tramontana, Bryan Ta, and Atif Memon. Mobiguitar–a tool for automated model-based testing of mobile apps. 2014.

[83] Peach fuzzer. [Online] Available: `http://www.peachfuzzer.com`. Accessed on: 29-April-2015.

[84] Snooze. [Online] Available: `https://seclab.cs.ucsb.edu/academic/projects/projects/snooze`. Accessed on: 29-April-2015.

[85] Spike. [Online] Available: `http://www.immunitysec.com`. Accessed on: 29-April-2015.

[86] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

[87] The monkey ui android testing tool. [Online] Available: `http://developer.android.com/tools/help/monkey.html`. Accessed on: 29-April-2015.

[88] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[89] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.

[90] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM, 2014.

[91] Intent fuzzer. [Online] Available: `http://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx`. Accessed on: 29-April-2015.

[92] robotium: The world's leading android test automation framework. [Online] Available: `https://code.google.com/p/robotium`. Accessed on: 30-April-2015.

[93] Monkeytalk | mobile app testing tool. [Online] Available: `https://www.cloudmonkeymobile.com/monkeytalk`. Accessed on: 30-April-2015.

[94] Monkeyrunner | android developers. [Online] Available: `http://developer.android.com/tools/help/monkeyrunner_concepts.html`. Accessed on: 30-April-2015.

[95] Sikuli script - home. [Online] Available: `http://www.sikuli.org`. Accessed on: 30-April-2015.

[96] Junit. [Online] Available: `http://junit.org`. Accessed on: 9-June-2015.

[97] Mockito. [Online] Available: `http://mockito.org`. Accessed on: 9-June-2015.

[98] Powermockito. [Online] Available: `https://code.google.com/p/powermock/`. Accessed on: 9-June-2015.

[99] Eclipse ide. [Online] Available: `https://www.eclipse.org`. Accessed on: 9-June-2015.

[100] Eclemma. [Online] Available: `http://www.eclemma.org`. Accessed on: 9-June-2015.

[101] Clover. [Online] Available: `https://www.atlassian.com/software/clover/overview`. Accessed on: 9-June-2015.

[102] Software improvement group. [Online] Available: `https://www.sig.eu/`. Accessed on: 9-June-2015.

[103] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[104] Acra. [Online] Available: `https://github.com/ACRA/acra`. Accessed on: 4-june-2015.

[105] Mohd Ehmer Khan et al. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–14, 2011.

[106] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[107] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[108] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

[109] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

Appendices

A Original Project Description

This is the original project description as bunq posted it on BepSys.

Keep them backends safe, mmkay? Each company that uses critical infrastructures needs to know when a glitch in the Matrix occurs. This means that no one enters or alters anything without permission. If they would do, we definitely need to know about it. That's where you come in. We need a state-of-the-art intrusion detection (and ideally, prevention) system that will automatically warn us when something is even slightly off. From SQL injections, illegal transactions, port scanning - we have to be able to notice it all, and preferably even prevent it before it happens (through e.g. honey pots). Shady connections should stay out of our system. Possible research questions: Which types of attacks do companies with critical infrastructures need to look out for? To what extent can we prevent or detect these types of attacks? Which methods are available that could prove effective? We'd like to see a proof-of-concept or prototype regarding intrusion detection and/or prevention.

B Testing Techniques

No application is perfect, so tests are needed to check whether it performs as expected. The idea of the tests is to uncover any hidden faults, which could also include potential security issues. There are multiple approaches how tests can be created, which will be discussed in this section.

White box

With white box testing, the underlying code and functionality of an application is known. This is considered as one of the most effective ways in validating an application, as it can target special cases that are not easily found with other testing techniques. [105] An example is for instance the following piece of code:

```
int function(int x){
  int y = x + 1;
  if (y == 403) doOtherFunction(); //error
  return y;
}
```

When the code is known, it can easily be seen that a test should be written which includes '`x = 403`', as this would trigger the function call.

However, white box testing also has it downsides. First of all, a choice should be made for what paths in the code will be covered; as it is simply impossible to check them all (except when it is a very small application with very limited input). [106] Therefore white box testing is not able to find all possible errors in an application, but can uncover some very specific ones as mentioned earlier.

Black box

With black box testing, the source code is either not known or simply not used while testing an application. [107]. Because there is no insight in the application, it is considered as a 'black box': some input is sent to it and this results in certain output. Examples of techniques that make use of this approach are for instance fuzzing, which will be discussed in Section 7, and the creation of finite state machines, which is found in Section 6.

A disadvantage of black box testing is that it depends on certain random input. Because of this, it might not cover states which can be easily found when the code is known. With the code example in the previous section, it is for instance a chance of 1 in 2^{32} to randomly input '403' with a black box test, while white box testing can find it instantly. Therefore, in order to get a better overall test coverage of an application, black box testing is often accompanied by white box testing. [106]

Gray box

In between white and black box testing, there is another possibility called gray box testing. Gray box testing is defined as black box testing, but with the addition of insights gained by reverse engineering an application. [77] Reverse engineering is the process of extracting information about the design and building abstractions that have a smaller dependence on a certain implementation. Reverse engineering is mostly used to find the different system's components and their interconnections, or to make representations of an application in a different form or at a higher abstraction level [108].

C What is a Finite State Machine?

In the formal definition, a finite state machine consists of a set of states, an input alphabet, a transition function, a start state and a set of accepting states. The set of states covers all the possible states the machine can be in. The alphabet consists of all the tokens the machine can read. An FSM can move from one state to the other by reading its input; the transition function describes for every combination of a state and an element of the input alphabet which state transition should occur. An FSM terminates with a 'success' result when, after reading all available characters from its input, the state it is in belongs to the set of accepting states. [109]

D Example XML File

This is an example of an XML file that can be used as input for the fuzzing tool.

```xml
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<actionset order="INORDER">

        <action chance="1" operation="ifExistsInsert"
                id="com.bunq.android:id/etPin">123456</action>

        <actionset order="RANDOM" times="1">

                <actionset order="INORDER" times="1">
                        <action chance="1" operation="clickParent"
                                id="Overview" />
                        <file name="res/xml/overview.xml" />
                </actionset>

                <actionset order="INORDER" times="1">
                        <action chance="1" operation="clickParent"
                                id="Nearby" />
                        <actionset order="INORDER" times="5">
                                <file name="res/xml/nearby.xml" />
                        </actionset>
                </actionset>

                <actionset order="INORDER" times="1">
                        <action chance="1" operation="clickParent"
                                id="Support" />
                        <actionset order="INORDER" times="5">
                                <file name="res/xml/support.xml" />
                        </actionset>
                </actionset>

        </actionset>

</actionset>
```

E Overview Code Quality

In order to keep high quality code, an overview was made to indicate when code was considered 'good', 'moderate' or 'bad' in our opinion. Below, the table can be seen that shows the different quality assessments:

	Considered		
Measurement	**Good**	**Moderate**	**Bad**
Method size (nr of lines)	<20	<30	>30
Javadoc	Every function has appropriate javadocs	Most functions have javadocs	Some/none functions have javadocs
Test coverage	>80%	>65%	<65%
Deprecated code left in comments	None	Some	Much
Amount of parameters in function	<4	<6	>6
Expressive function and variable names	Names are clear and indicate function correctly	Some names are clear	Names are very unclear (e.g. 'a' or 'b' variables)
Packages	Classes separated according to same functionality	Some packages are present, but classes could be separated more	No use of packages
Exceptions	When exceptions occur they are either resolved immediately or a clear error message is given	Exceptions are caught, but the error message is not useful	Functions silently fail

F First Feedback from SIG

This is the original feedback we got from Dennis Bijlsma of the SIG.

De code van het systeem scoort 4,5 ster op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een iets lagere score voor Unit Size.

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 IOSInstrumentator.enterText, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels als "Find the first element with the right value" zijn meestal een goed teken dat een blok code eigenlijk een aparte methode zou moeten zijn, in dit geval bijvoorbeeld findElementByValue (of iets dergelijks). Idealiter doet een methode één ding, en dat ene ding wordt aangegeven door de naam. Als jullie 5 sterren willen halen is het aan te raden om nog eens door de langere methodes heen te lopen en te kijken of er nog deelmethodes te maken zijn.

Daarnaast complimenten voor het schrijven van unit test-code. Jullie hebben momenteel ongeveer 650 regels testcode op ongeveer 1000 regels productiecode. De optimale verhouding is 1:1, en jullie zitten daar dus dicht tegenaan.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

G Second Feedback from SIG

This is the second feedback we got from Dennis Bijlsma of the SIG.

In de tweede upload zien we dat het codevolume is gegroeid, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven. Jullie zitten met 4,5 ster dus nog steeds erg hoog.

Tijdens de analyse van de eerste upload werd Unit Size als mogelijk verbeterpunt aangemerkt. Hier is duidelijke verbetering te zien: het systeem bevat nu geen lange methodes meer, hulde. Deze verbetering leidt echter niet tot een verbetering in de totaalscore, aangezien jullie op andere aspecten licht zijn gedaald. Dat is normaal, het is moeilijk om een systeem goed onderhoudbaar te houden op het moment dat het gaat groeien, zeker als je zo hoog zit.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de volgende evaluatie zijn meegenomen in het ontwikkeltraject.

H Project Infosheet

Title: Mobile Application Security: an assessment of bunq's financial app
Client: bunq
Date: June 26, 2015

Description

bunq is developing a mobile financial application for both iOS and Android. Security is an important issue in IT projects; this is especially true for financial applications. During the project, an assessment of the security of their apps was made. This was partially done by using two applications that were developed by us during the project.

Challenges: The curriculum of the Computer Science bachelor of the TU Delft does not contain much about security or mobile application development. Therefore, a lot of research had to be done in order to make a decent assessment of the security of bunq's app.
Research: For the first two weeks we only did research. Because of the challenges stated above, we also did a lot of research during the project. We learned about basic security measures, app development and mobile application specific security issues. We also learned about security testing techniques, which have been applied later in the project.
Process: We worked with three separate git repositories; each software application was stored in its own repository. The report has been written in LaTeX and was stored in a third repository. Each day we reported our status to each other and to our supervisor, Wessel Van. At the end of each week, a meeting was held with Wessel Van to discuss our progress.
Products: Besides performing a lot of security checks, we also developed two applications. The first application generates finite state machines. These finite state machines then can be used to compare the implementation of the Android an iOS app and check the application for unwanted transitions. The second application is called bunqFuzzer. This tool can be used for fuzzing mobile applications in a structured way. The tools were extensively tested using JUnit and Mockito; in order to improve the test coverage, Clover was used as a coverage tool.
Outlook: Both products were used to find numerous bugs in the bunq apps. Some of these bugs revealed security issues. bunq was informed about all these bugs and fixed most of them during the project. The developers at bunq have expressed their intention to keep on using our products to find bugs and security leaks in future versions of their app.

Members

Michel Kraaijeveld J.C.M.Kraaijeveld@student.tudelft.nl
Was responsible for the iOS component of the FSM learner application, decompiling the bunq apps and writing the corresponding parts in the report.
Kees Lampe kees.lampe@gmail.com
Was responsible for the bunqFuzzer and wrote most of the code and report for it.
Tom den Braber tomdenbraber@gmail.com
Was responsible for developing the Android and the learner component of the FSM learner application and writing the corresponding sections in the report.

All team members contributed to the report, the final presentation and finding additional bugs in the bunq applications.

Client Supervisor: Wessel Van - Lead Android Developer at bunq - wessel@bunq.com
TU Delft Coach: Sicco Verwer - EEMCS - Intelligent Systems - Cyber Security - s.e.verwer@tudelft.nl

The final report of this project can be found at: `http://repository.tudelft.nl`