University of Amsterdam

System & Network Engineering

Security in mobile banking

December 23, 2012

Authors:
Thijs Houtenbos
Jurgen Kloosterman
Bas Vlaszaty
Javy de Koning

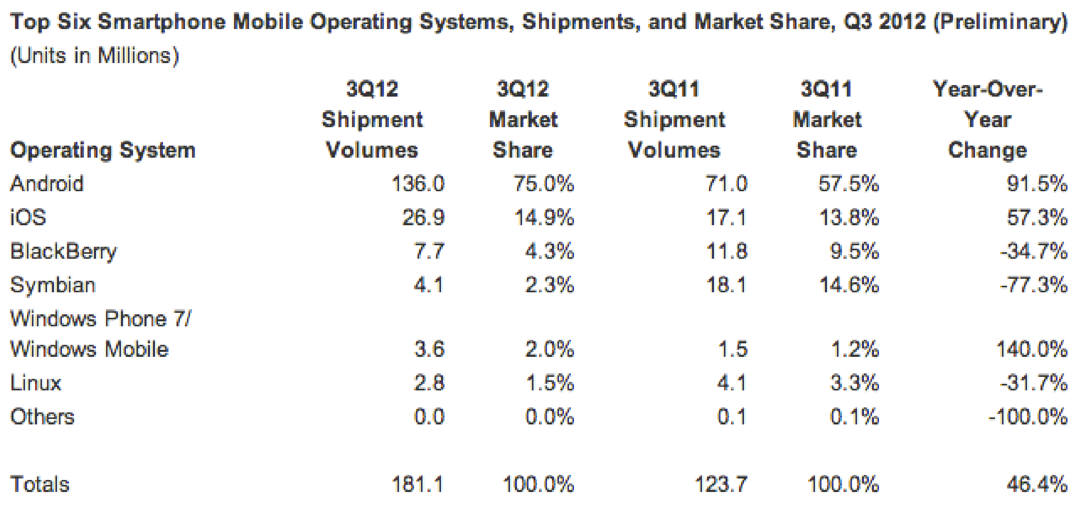
Abstract

The goal of the research described in this paper is to find out how the security used in Android based mobile banking applications is implemented. While monitoring the network traffic and an initial review of the source code from the most popular Android banking applications, the decision was made to focus entirely on the application released by ABN AMRO [1]. In order to find out what data was exchanged, the source code of the application was modified and a new APK file was created which now directs data to a valid SSL domain different from www.abnamro.nl before the data is sent to www.abnamro.nl. Fortunately, it became clear that the application accepts the new domain, therefore constituting a classic man in the middle attack. With the configuration in place, the next step was to inspect and understand the messages which are being sent when the user starts the application and enters a pin number. By assuming that multiple challenges all consist of the same structure, each containing a RSA public key, an idea was launched to encrypt the data with our own RSA public key. The response could then be decrypted using the private key, which gave the actual data. The main conclusions are that both SSL, RSA and exchange of challenges all depend on careful implementation, and that our recommendations having been implemented by ABN AMRO to further improve end user security.

Contents

1. Introduction

Nowadays almost everyone owns a smart-phone. We use smart-phones to call, text, listen to music, update social media, and so forth. Current developments such as Near Field Communication will even enable us to make payments with our phone. As part of a service to customers, several banks in the Netherlands developed mobile applications to enable customers to manage their bank accounts. All major banks currently have a smart phone app available in the Apple Appstore[3] or in the Google Play store[4]. These apps can be used to view account balance and history, but more important is the fact that an end user can initiate a transaction no matter where he or she is. Some of the applications allow users to set a private pincode to use as a means of authentication. And because the usage of these applications is exceeding the use of the banking website [2], security becomes even more important. This is also the reason why we would like to find out what the security features are that application developers in modern mobile banking applications have implemented.



Top Six Smartphone Mobile Operating Systems, Shipments, and Market Share, Q3 2012 (Preliminary)
(Units in Millions)

Operating System	3Q12 Shipment Volumes	3Q12 Market Share	3Q11 Shipment Volumes	3Q11 Market Share	Year-Over-Year Change
Android	136.0	75.0%	71.0	57.5%	91.5%
iOS	26.9	14.9%	17.1	13.8%	57.3%
BlackBerry	7.7	4.3%	11.8	9.5%	-34.7%
Symbian	4.1	2.3%	18.1	14.6%	-77.3%
Windows Phone 7/ Windows Mobile	3.6	2.0%	1.5	1.2%	140.0%
Linux	2.8	1.5%	4.1	3.3%	-31.7%
Others	0.0	0.0%	0.1	0.1%	-100.0%
Totals	181.1	100.0%	123.7	100.0%	46.4%

Figure 1: Android market share 3Q2012. Published by IDC on November 2nd, 2012

1.1. Focus on Android

The decision was made to focus entirely on applications for Android as this continuously developed, open platform has the greatest market share (75%)[1] when compared to other existing mobile operating systems. The number of activated Android based mobile phones have risen considerable in recent years and this popularity can also be noticed from the enormous amount of applications available at the Google Play Store

[1] `http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012/q3-2012-smartphones/`

and very notable the development of third party custom Android operating systems such as Cyanogenmod[2]. As end users are given the chance of developing their own Android application, it has become moderately easy to install new applications and to locate and decompile existing ones from a so-called ”rooted” device. And as anyone can submit an application to the Google Play store, apps with malware are therefore no exception.

1.2. Focus on ABN AMRO

Given the fact that there was only a limited amount of time available for this research project, it was clear that we would have have to focus our research. There was simply not enough time to thorougly investigate the applications of all three major banks in the Netherlands. To make an informed decision on which application to research further we first spent some time decompiling all of them. How this was done is described in chapter 4. We also tried to make some simple adjustments to them, recompile them, and run the new applications in an emulator. The experience of those tests were used as the basis for the decision which application to focus on. The applications and their observations are listed below.

- ABN AMRO
  - Application decompiled well. Quite readable code, structure of the application could be easily identified from the source. Code could be adapted and run without problems.
- ING
  - Application decompiled well. Code not very readable, obfuscated method names, structure quite difficult to make out.
- Rabobank
  - Decompiled poorly, completly obfuscated code which gave us no insight in the application's internal workings.

Given these initial results, the decision was made to first investigate the ABN AMRO application, because it looked the most promising.

[2]`http://www.cyanogenmod.org/`

1.3. Research questions

In our research we want to answer the following research question.

- *What is the current state of security implementations in Android banking applications?*

Other sub-questions for this research are given below and these cooperate to answer the primary research question.

- *In what way do applications set up and maintain communications and how is traffic protected from eavesdropping?*
- *What personal information is stored by mobile banking apps, and how is it protected?*
- *What information is available in memory when the app is running, and can we extract that information?*

2. Introduction to the ABN AMRO application

To use the ABN AMRO mobile banking application, users need to activate their bank account within the application when they execute the application for the first time. The activation must be verified with an e.dentifier[5] challenge-response number and after that a 5-digit Personal Identification Number (PIN) is chosen by the user to get access to the banking account. Both the account and card number are stored in the application cache so to use the application the user only has to enter a PIN number. Multiple bank accounts can also be accessed from the application menu. The basic features of the application are comparable to that of the ABN AMRO web site, but are limited as the following short list shows. This comes as no surprise given the purpose of the application.

1. View account balance and transaction history
2. Set a daily limit between EUR 0,- and EUR 750,-
3. Transfer any amount below the daily limit to "known" accounts[3] with the PIN-code
4. Transfer a maximum of EUR 3000,- per transaction with e.dentifier verification to any account
5. Transfer a maximum of EUR 250.000,- between own accounts with the PIN-code

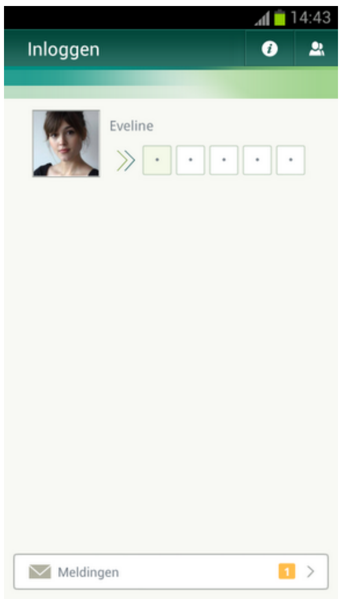


Figure 2: ABN AMRO Bankieren application on Android

[3]Known accounts are bankaccounts where money has been transferred to in the past 18 months from this account

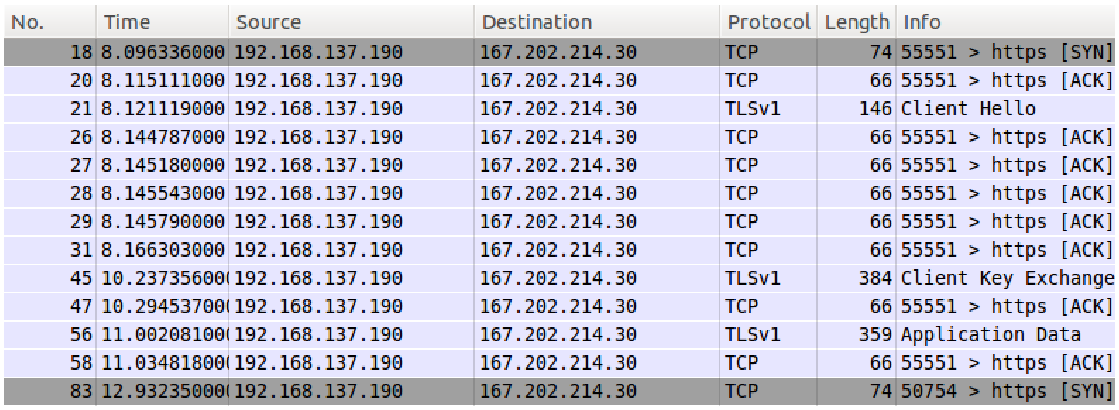
3. Traffic inspection

Although we suspect that communication with the bank website will be transferred over a secured SSL-connection we will inspect the traffic to verify if this is true. To inspect the traffic we've set up a laptop with two network interface cards.

1. Intel PRO Wireless 3945ABG [4]

2. TP-Link TL-WN422G [5]

The Intel network card is used to connect to an access point connected with the internet and the TP-Link Network Interface Card (NIC) is used as an access point for the Android device. Traffic is then routed through the laptop which we will use to inspect the traffic. We used Droidwall [6] on the Android phone to prevent other applications from accessing internet resources. Droidwall is an Android frontend for the popular iptables Linux firewall. By only whitelisting the network traffic from the mobile banking application we were able to filter out the traffic produced by other applications. From a technical



No.	Time	Source	Destination	Protocol	Length	Info
18	8.096336000	192.168.137.190	167.202.214.30	TCP	74	55551 > https [SYN]
20	8.115111000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
21	8.121119000	192.168.137.190	167.202.214.30	TLSv1	146	Client Hello
26	8.144787000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
27	8.145180000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
28	8.145543000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
29	8.145790000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
31	8.166303000	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
45	10.23735600	192.168.137.190	167.202.214.30	TLSv1	384	Client Key Exchange
47	10.29453700	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
56	11.00208100	192.168.137.190	167.202.214.30	TLSv1	359	Application Data
58	11.03481800	192.168.137.190	167.202.214.30	TCP	66	55551 > https [ACK]
83	12.93235000	192.168.137.190	167.202.214.30	TCP	74	50754 > https [SYN]

Figure 3: Wireshark traffic capture.

point of view, we will now describe the steps that were executed when the app is started. The app initially does an A-record DNS query lookup for www.abnamro.nl, which is answered with the DNS response which contains the IP-address 167.202.214.30. From then on multiple TLS handshakes are sent between different source and destination ports, after which encrypted data is exchanged. Our lab configuration is further explained in appendix D.

[4] `http://www.intel.com/products/wireless/prowireless_mobile.htm`
[5] `http://www.tp-link.com/en/products/details/?model=TL-WN422G`
[6] `http://code.google.com/p/droidwall/`

4. Decompiling the Application

There are several tools available which can be used to convert Android Package Files (APK) to a somewhat human readable format. The most popular ones give an output in Java and a special format called Smali. We used the following tools:

- dex2jar - http://code.google.com/p/dex2jar/
- jd-gui - http://java.decompiler.free.fr/?q=jdgui
- apktool - http://code.google.com/p/android-apktool/

dex2jar is a tool to convert Android dex files to Java class files, both binary formats.

jd-gui is a graphical tool that can displays Java source code from Java class files. Because of the decompilation the output is not perfect and can be hard to read but it can still give a good impression about what operations are executed in a section of code.

apktool is a tool to reverse-engineer and modify binary Android applications. It can decompile the application to smali format. The big advantage here is that the application can be reconstructed from these smali files, which allows to modify and recompile it.

The traffic inspection showed us that the application was using SSL/TLS to communicate with the bank servers. To be able to look inside the traffic we modified the base URL used in the application which is used to construct the request URL for the various actions.

```
.field public static final DEFAULT_SERVER_URL:String; = "
   https://www.abnamro.nl/"
```

We changed this to the address of a server in our control and recompiled the APK with this change included. Because we own the domain name of the new destination we were able to request a valid SSL-certificate for this address and present it to the application on connecting. We forwarded the requests we received towards the real ABN-AMRO servers so we could look for any additional security features inside the SSL-connection.

5. Man in the middle attack

A man in the middle (MiTM) attack is a form of active eavesdropping in which an attacker creates connections to the victim's phone and the bank's website and relays the messages between them. Our first attempt to perform a MiTM attack succeeded using our modified application. As part of our research we also investigated how the application handles invalid certificates. We wanted to include subject to be complete in our research, but did not expect to do any findings here since other banks recently had related problems. [7].

We used our own DNS-server to let the original application from the Play Store connect to our own server when resolving the www.abnamro.nl domainname. Our first test concluded of a self-signed certificate with a matching hostname (CN). The application behaved like expected in this test. It showed the user an error that mobile banking is not available at the moment and stops the connection process. In our second test we presented an SSL-certificate which was correctly signed by a trusted Certificate Authority (CA) but where the hostname (CN) in the certificate does not match the name of the requested site. The application did not handle this test as expected. No user warning was displayed and it continued the connection as if the real bank server had been contacted. This means we could now perform a MiTM attack on the mobile banking application by only redirecting the user traffic to our own server. There are several attacks available for this on local or remote networks such as DNS-spoofing[8] or ARP poisoning[9].

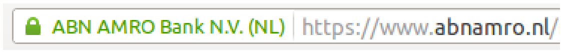


Figure 4: SSL-certificate validation as displayed in a webbrowser

When using a webbrowser for Internet banking from a standard PC the user is displayed a green bar next to the URL in all modern browsers to indicate a valid and Extended Validated (EV) certificate is used (Figure 4). The Dutch Society of Banks (NVB) even had a large campaign to educate users to check among others the certificate of the bank when doing online banking[10]. Because the mobile application handles all connectivity without user interaction, the user has no way of checking the connection is indeed with the real bank server as is possible in browsers.

[7]`http://www.eenvandaag.nl/binnenland/40032/mobiel_bankieren_ing_maandenlang_onveilig`
[8]`http://en.wikipedia.org/wiki/DNS_spoofing`
[9]`http://en.wikipedia.org/wiki/ARP_spoofing`
[10]`http://www.3xkloppen.nl/over-3xkloppen/`

6. HTTP Traffic inspection

After the application is started it first does a request for any news messages. When the user does an attempt to login the 5-digit PIN-code must be entered. The application then requests a token at the server and includes the account- and card number in the request. If the account- and card number are valid the bank sends a challenge back to the client (Figure 5).

6.1. Challenge

```
02000084b7a5f77cc0ffef50300045 0b359a3040100b7fefd0e198d9
```

Figure 5: Challenge in TLV (Type, Length, Value) format.

With the help of the decompiled application we were able to identify the challenge as a hex encoded Type, Length, Value (TLV) format which includes the following items.

- Random data (8 bytes, marked in blue)
- Unix timestamp (4 bytes, marked in orange)
- RSA Public key (256 bytes, marked in red)
- RSA Exponent (3 bytes, marked in pink)

Next to the challenge itself a challengeHandle is included which is probably used to identify which RSA public key was sent to the client. The challengeHandle changes every time the RSA key is changed.

The plain challengeHandle is included with the subsequent login request together with the account number and card number. There is also a 512 hex digit response in the login request. If the login attempt is successful then the server includes a session identifier cookie in the reply to the client which is from then on used in every request from the client. All further traffic between the client and server except the authorization of a payment with the PIN-code is sent through the SSL-tunnel without further encryption.

We suspected the login response was RSA encrypted data since we already found a 2048-bit RSA public key was being transmitted to the application and the response had a length of exactly that size. Since we do not have access to the private key part of the public key we were unable to decrypt the payload. We do consider RSA to be secure at this time so did not look into breaking the algorithm, instead we wanted to focus on the implementation. One thing we thought would be worth trying was to replace the public key part with our own generated RSA public key, for which we also have the private key in our possession.

We replaced the original RSA public key from the bank server with our own key in the challenge. We then captured the encrypted response and tried to decrypt it with our private key. This indeed worked as expected and allowed us to look at the content of the response which we identifier as the following data:

- Header (1 byte)
- Random data (8 bytes)
- Unix timestamp (4 bytes)
- User Id (13 bytes)
- Secret (PIN-code) (5 bytes)

The PIN-code here is the actual PIN-code setup and entered by the user to authenticate against the account and can now be retrieved in plaintext with this attack.

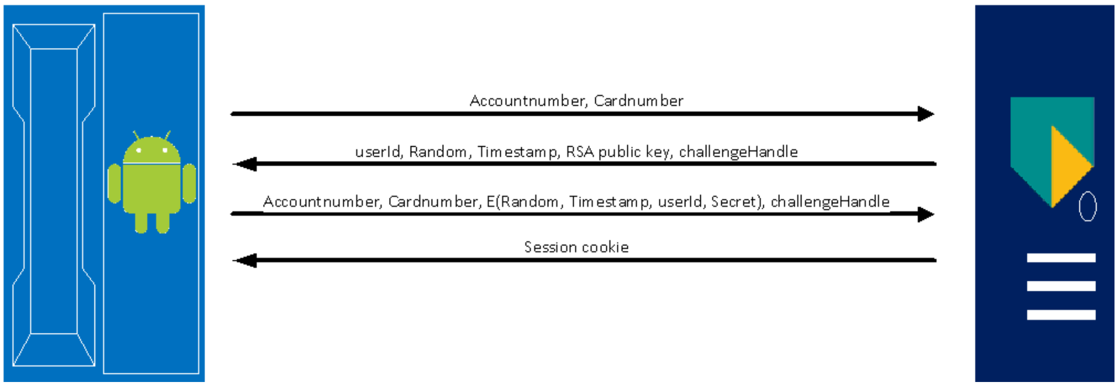

Accountnumber, Cardnumber
userId, Random, Timestamp, RSA public key, challengeHandle
Accountnumber, Cardnumber, E(Random, Timestamp, userId, Secret), challengeHandle
Session cookie

Figure 6: Communication between Bankieren app and ABN AMRO

7. File system caching

Android phones give the user access to two locations to save files. These locations are:

1. "/emmc": This is the internal memorycard.
2. "/sdcard" or "/sd-ext": This is the swappable memorycard.

We are interested in the data that the application writes to the storage. This data is usually placed in either one of the following partitions:

1. "/data" Also called user data, the data partition contains the contacts, messages, settings and apps that are installed. Erasing the data partition performs a factory reset of the device. The data we are looking for is probably located in this partition.
2. "/cache" This is the partition where Android stores frequently accessed data and app components. Wiping the cache doesn't effect the personal data but simply gets rid of the existing data there, which gets automatically rebuilt as you continue using the device.

7.1. Profile Database

After setting up the ABN AMRO application we noticed the directory "/data/data/-com.abnamro.nl.mobile.payments/databases/" got populated with 2 database files. The first one is the "profiles.db" database file which stores unencrypted account data. From this file we were able to read the bank account number and the card number which was setup in the application. We have replaced the account number with "123456789" and the cardnumber with "123" in the example below.

Listing 1: Profile database

```
sqlite> .databases
seq  name             file
0    main             /data/data/com.abnamro.nl.mobile.
   payments/databases/profiles.db
sqlite> .tables
abn_profile       android_metadata

sqlite> PRAGMA table_info(abn_profile);
0|counter|INTEGER|0||0
1|alias|TEXT|0||0
2|account_nr|INTEGER|0||0
3|card_nr|INTEGER|0||0
```

```
4|is_private_banker|INTEGER|0||0
5|image_hash|TEXT|0||0
6|image_data|BLOB|0||0

sqlite> select * from abn_profile;
58||123456789|123|0||
```

7.2. Caching Database

The second file, ”cache.db”, is a little more complicated. This file is created when the application retrieves the transaction history in a logged in session. When the user closes the application or when there is a 15 minute timeout the file gets deleted automatically. We examined this database and the data is encrypted with AES-128 bit encryption. The key is randomly generated every time the database is created.

Listing 2: 128bit AES encrypted transactions database

```
sqlite> .tables
MUTATIONS_4914862_1091039688  android_metadata

sqlite> PRAGMA table_info(MUTATIONS_4914862_1091039688);
0|_id|INTEGER|0||1
1|position|INTEGER|0||0
2|data|BLOB|0||0

sqlite> select * from MUTATIONS_4914862_1091039688;
1|-20|(encrypted data blob)
...
```

8. Memory analysis

One other area of attack we explored is the possibility to gain access to sensitive information through inspection of the application's memory heap. The application saves all data in the RAM, so we looked at the possibility of directly extracting the data from there.

To investigate if this was possible we used tools from the Android SDK [11]. In particular we used the android emulator to run the application. Then we used the Dalvik Debug Monitor Server (DDMS) to make a dump of the active application memory. After that Eclipse's Memory Analyser Tool (MAT) was used to analyse the memory dump and investigate if there was any sensitive data for us to extract.

[11] `http://developer.android.com/sdk/index.html`

8.1. Android SDK

Setting up the emulator to run the Android application is very straightforward. By installing the Android SDK and running the emulator it was possible to just install the application and run it.

8.2. Dumping the memory

With the application running in the emulator the next step is to dump the memory to a file. To do this, we use the DDMS from the Android SDK.

To be able to do a thorough analysis of the memory, two different memory dumps where made. One before the user is logged into the application, and one just after the user is logged in. The idea is that possibly by looking at the differences of the two dumps, it would be possible to more easily find the newly stored variables, like passwords or other sensitive data.

After making the dump, it is necessary to convert the memory dump into a format that can be read by the memory analyzer tool. To do this, hprof-conv is used. This tool converts a .hprof dump into the necessary file format and is part of the Android SDK.

8.3. Analyzing the dump

Now we have a complete dump of the memory, MAT was used to analyze the memory. In MAT it is possible to browse through all the Java objects, and their contents. This way it is possible to for instance read the values of all String objects used by the application. It is also possible to use OQL-queries on the dump to select and filter through instances of all objects. [12] [13]

8.4. Results

Analysis of the dumps yielded moderate results. Things we found:

- The 5-digit PIN-code
- Accountbalance
- Accountnumber
- Session identifier

Things we looked for but did not manage to find:

- Key for encrypted cache database
- RSA key

[12] `http://en.wikipedia.org/wiki/Object_Query_Language`

[13] `http://help.eclipse.org/indigo/index.jsp?topic=2\%Forg.eclipse.mat.ui.help\%2Freference\%2Foqlsyntax.html`

Given the fact that we did not have any previous experience with memory analysis, we consider this a reasonable result. Perhaps with more experience and skill in this area it would be possible to also extract the encryption keys from memory, but given the time and our capabilities we did not manage to do that.

The things we have found are still quite concerning. The 5 digit PIN-code is used to transfer money to known associates, for amounts within the daily limit. It is also used for the transfer of money between for instance savings accounts. This means that with knowledge of this pincode an attacker could potentially transfer small amounts of money, typically up to 500,-, to his account, provided the victim has transferred money to this account before. This means it is not very realistic to exploit this on a large scale, because the victim would probably need to know the attacker personally.

For the account balance and the account number the biggest problem is perhaps a privacy concern. It is sloppy for an application to not handle information like this more carefully, but it is not a real threat to security.

The fact that the session key can be captured makes it possible for an attacker to request information about the account, like balance and past transfers. For any real attacks on the account, like for instance transferring money, the attacker would need the 5 digit PIN-code. Another reason why this is not a major problem is that a session key is only valid for 15 minutes. This means that would this be exploited, the attacker would have access to account information for a maximum duration of 15 minutes. Using the 5 digit PIN-code this would also be possible, therefore the added value for an attacker of having the session key is limited. As with the account balance and account number, the major concern here is privacy. Information about past transfers should be strictly confidential.

8.5. Threat of an attack

To perform an attack like this in practice, an attacker would first need to get a memory dump from the victim's phone. In "Acquisition and analysis of volatile memory from Android devices" [14] the writers propose a method of dumping the memory of a rooted Android phone. This shows that it is possible, though still difficult, to acquire such a memory dump. When an attacker has a memory dump from the phone, the 5 digit PIN-code is the most valuable piece of data the attacker will try to recover. When he has access to this, there are a number of things he can do:

- View account data
- Transfer an amount up to the daily maximum amount to a known account number

For the first attack the only risk is privacy. The confidentiality of your account can not be guaranteed. The second attack is potentially a much bigger security risk, because of the possibility to transfer money. However there are two restrictions in place:

- Daily limit is set to some value between EUR 0,- and EUR 750,-, so larger transactions are not possible without the e.dentifier verification.

[14]`http://www.sciencedirect.com/science/article/pii/S1742287611000879`

- It is only possible to transfer to an account number that is 'known', meaning that a previous transfer to that account number has to have been made. In other words, before an attack can take place, the victim has to have transferred money to the attacker in a previous transaction.

Considering these limitations, the potential of using this attack on a large scale is very limited.

9. Proof of Concept

To test our findings we executed a man in the middle attack on the application when connected to our own wireless network. We setup a DNS server on this network which resolved www.abnamro.nl to an IP-address of our MiTM server. The server presents an SSL-certificate which is correctly signed by a trusted CA for another domain name (CN) (Section 5).

On the MiTM server we run a custom build script which accepts the SSL-connections and forwards the requests to the real www.abnamro.nl server. The script rewrites the RSA public key in the challenge response with our own public key and forwards that back to the application (Figure 7).

When the application sends a login request with the encrypted PIN code the script can decrypt this with our own private key, retrieve the plain PIN-code and re-encrypt the data with the original public key from the bank. The re-encrypted data is again forwarded to the real bank server which can then validate the details and start an authenticated session if they are correct. At this point we have all the details needed to perform a login on the user's account (accountnumber, cardnumber, pin) and are still in the middle of the current user session.

The rest of the session does not have any further encryption except for the SSL-session itself. This means we can read the account balance, transaction history and any payment requests.

Encryption is again used in the same way as in the login process when a transaction is authenticated with the PIN-code. But we can do the same trick with the RSA key here since it does a new request for the public key first before sending the encrypted message.

Because we are also able to see any payment requests we tried to see if we can alter the destination of the transfer to a different bank accountnumber. To hide this from the user we can setup our attack in such a way that it only changes the transaction when the user already expects the e.dentifier needs to be used to verify the transaction. We can do this by forwarding the original transfer request to the bank, and only change the transaction if the bank indicates the e.dentifier verification is needed. Since there is no relation between the e.dentifier code and the amount or destination of the transaction, and because the application cannot verify if the current e.dentifier code is actually linked to the requested transaction, the user has no way of knowing that he is in fact authorizing our altered payment.

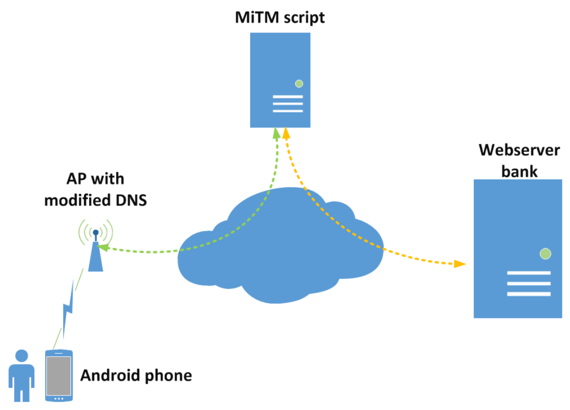


Figure 7: Proof of concept setup

But we believe most users will also enter an e.dentifier response when requested to do so when they do not expect this. Our reasoning is that the user trusts the banking application to be secure and will enter the code when requested to do so. If we take this into account in the attack we can alter every payment the user initiates.

We could have even gone one step further and keep a history of real and altered transactions, which we can use to change the transaction history between the bank and the application, so that the altered transactions are hidden from the user and the original transaction is shown. This will keep the user from knowing about the attack until the account is checked without our interference.

In our proof of concept we demonstrated that our theory works. We can read all the details needed to login on the user's account and gather even more information like account balance and history. We can alter payment requests and get the user to authenticate them while hiding this from the user until a safe network is used. All this is done by only having the mobile device running the application connected to a network where we can alter the DNS-record or take over the default gateway.

10. Security remarks

In this paragraph we will point out our concerns regarding the security of the application. We will also explain how we feel that security could be improved.

10.1. Verification of SSL certificate

The first, and probably most important issue we found is that the hostname of the SSL certificate is not validated. When the application connects to `https://www.abnamro.nl` it downloads the encryption key part of the security key with the certificate and the application uses that to encrypt all messages being passed back to the server. The content is unable to be converted back into readable text without the decryption key which remains in the security certificate stored on the server so that the server hosting the site is the only place where that content can be decrypted.

To secure the communication, you have to be sure that you are communicating with a ”trusted” source whose identity you can be sure of. The application does validate the SSL certificate but it does not verify that the SSL certificate matches the hostname for `www.abnamro.nl`. This makes it possible to do a man-in-the-middle attack, you are encrypting traffic but you are not sure who you are encrypting it for.

10.2. Implementation of RSA

The application uses a 2048 bit RSA key to encrypt the communication between the phone and the server. This key is rolled over multiple times a day and the public key is sent from the server to the client in a challenge. RSA recommends a key size of 2048 bits to keep data confidential until at least 2030. They predict that a 10 million dollar machine will need approximately 5 months to factor out a 2048 bit RSA key in the year 2030. [15]

Rolling this key over that much does not improve the security of the application. We feel it would be safer to include one, or multiple public keys within the application. That way the traffic cannot be intercepted when a man in the middle attack is performed.

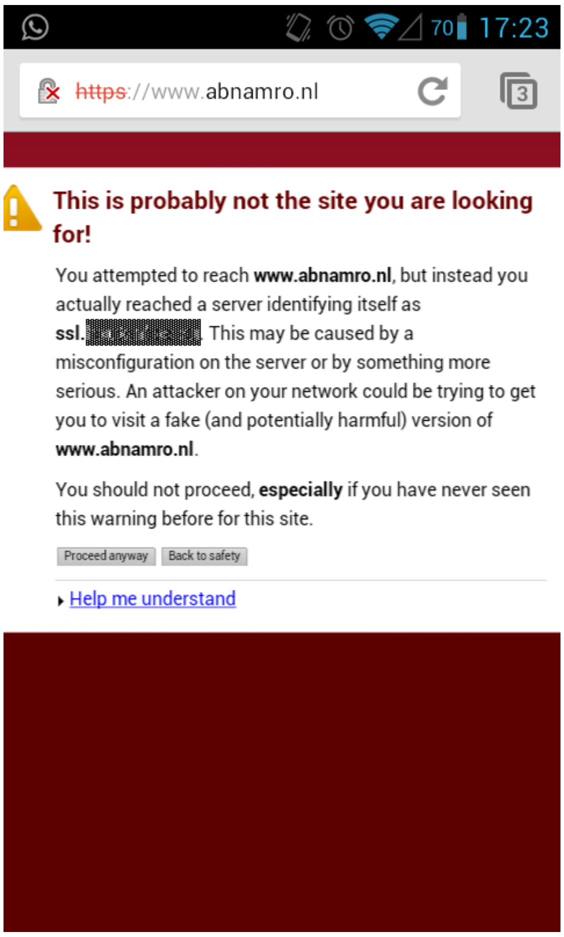
10.3. E.dentifier codes

The e.dentifier codes are used to verify the transaction. However there is no known visible relationship with the between the transaction and the codes used. A user is not able to see if the data has been tampered with and will verify the modified transaction. To improve security another validation number could be used. For example a validation number based on:

1. The total amount of money you would like to transfer.
2. The account number where you are transferring money to.

Building in these checks would make easy for the user to spot a malformed transaction.

[15] `http://www.rsa.com/rsalabs/node.asp?id=2004`



~~https~~://www.abnamro.nl

This is probably not the site you are looking for!

You attempted to reach **www.abnamro.nl**, but instead you actually reached a server identifying itself as **ssl.**[redacted]. This may be caused by a misconfiguration on the server or by something more serious. An attacker on your network could be trying to get you to visit a fake (and potentially harmful) version of **www.abnamro.nl**.

You should not proceed, **especially** if you have never seen this warning before for this site.

Figure 8: Incorrect certificate in webbrowser

11. Conclusion

In this paper, we have presented methods that obtain information about either the information that is being saved on a mobile device and the communication between that device and the infrastructure of the bank. Initially, we were very skeptical about the feasability of this project, but that changed after did our first discovery. This project shows that although the specification of security features in an application on itself should be sufficient, but the implementation of the actual application that is being used by around 500.000 end-users in the Netherlands is in fact not.
When we started with the project we were quickly able to find out how the traffic is communicated. Later we found out that the traffic is apparently not so secure due to an problem in the implementation of checking the validity of the SSL certificate. By setting up a rogue access point it therefore became possible to effectively eavesdrop the traffic and find out valuable data such as bank and card number and more importantly, the pin number. Based on this information, we were able to create a proof of concept in

which the actual payment request was modified in transit.
The information stored on the device should be protected in a way that if your phone is stolen it should be able to find any useful information. We were able to find some important data by obtaining a memory dump and we think that it is concerning regarding end user privacy, but not very realistic to be actually exploited in a large scale and the victim should have made previous payments to the attacker.
We therefore think that is reasonable to conclude that the current state of security implementations in Android mobile banking applications definitely needs improvement. Either if it is because of end user privacy or the proof of concept we discussed in this paper, security in Android applications should definitely be improved with regard to the expectation of an explosion of Android malware in 2013 and beyond.

12. Future work

A recent publication released by ESET [16] predicts that malware developed for Android will rise explosively in 2013 and it leaves no imagination that developers should adapt secure programming practices in the most sensible way. Given the fact that anyone is in the position to develop and submit an Android app to the Google Play Store, the process of adding a newly developed app to the Google Play Store is easy compared to the cycle one has to complete before an app is accepted in the iTunes App Store. [17] Coupled with the fact that the default Android antivirus application needs refinement [18], we think that future work should primarily consist of secure programming practices developers and a more strict evaluation on the side of the Google Play Store. With regards to the application discussed in this paper, we also think that there should be a standard practice for the use of certificates in Android applications, as we expect that our findings are certainly not limited to this particular case.

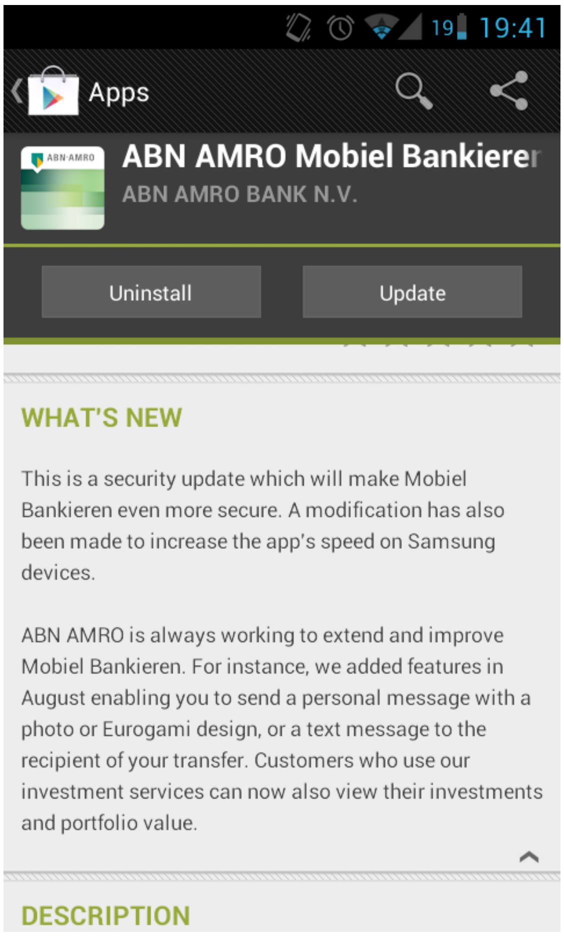
[16] `http://go.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf`

[17] `http://mobiledevices.about.com/od/additionalresources/a/Ios-App-Store-Vs-Google-Play-Store-For-App-Develo`
`htm`

[18] `http://www.av-test.org/fileadmin/pdf/avtest_2012-02_android_anti-malware_report_`
`english.pdf`

12.1. Patched Application

On Tuesday December 18th, 2012, only four days after being informed about the issue, ABN AMRO released a new version of the Android Application in the Google Play store, patching the issues described in this paper.



WHAT'S NEW

This is a security update which will make Mobiel Bankieren even more secure. A modification has also been made to increase the app's speed on Samsung devices.

ABN AMRO is always working to extend and improve Mobiel Bankieren. For instance, we added features in August enabling you to send a personal message with a photo or Eurogami design, or a text message to the recipient of your transfer. Customers who use our investment services can now also view their investments and portfolio value.

DESCRIPTION

Figure 9: Application update in the Play Store

A. Acronyms

AES	Advanced Encryption Standard
APK	Android application PacKage file
CA	Certificate Authority
CN	Canonical Name
DDMS	Dalvik Debug Monitor Server
DEX	Dalvik Executable
EMMC	External Multi Media Card
EV	Extended Validation
HTTP	HyperText Transport Protocol
JAR	Java ARchive
MITM	Man In The Middle
RSA	Ron Rivest, Adi Shamir and Leonard Adleman
SDK	Software Development Kit
SQL	Server Query Language
SSL	Secure Sockets Layer
UvA	Universiteit van Amsterdam

B. Bibliography

References

[1] **ABN AMRO** ABN AMRO Bank N.V. is a Dutch state-owned bank with headquarters in Amsterdam. It was re-established, in its current form, in 2009 following the acquisition and break-up of the original ABN AMRO by a banking consortium consisting of Royal Bank of Scotland Group, Santander Group and Fortis. Following the collapse of Fortis, the acquirer of the Dutch business, it was nationalized by the Dutch government along with Fortis Bank Nederland.
`https://www.abnamro.nl`

[2] **Application usage statistics:** On November 9th, 2012 ING reported that their mobile banking app exceeded the use of their web-application. More then a million customers are using the mobile application.
`http://webwereld.nl/nieuws/112415/ingmobiel-bankieren-wint-van-internetbankieren.html`

[3] **Apple App Store apps:**
`https://itunes.apple.com/nl/app/rabo-bankieren`
`https://itunes.apple.com/nl/app/ing-bankieren`
`https://itunes.apple.com/nl/app/mobiel-bankieren`

[4] **Android Playstore apps:**
`https://play.google.com/store/apps/details?id=com.abnamro.nl.mobile.payments`
`https://play.google.com/store/apps/details?id=com.ing.mobile`
`https://play.google.com/store/apps/details?id=nl.rabomobiel`

[5] **E.dentifier:** Device used in combination with your bankcard to generate an access code. De e.dentifier is een paslezer waarmee u toegang krijgt tot bijvoorbeeld Internet Bankieren en Telefonisch Bankieren. De e.dentifier2 is de opvolger van de e.dentifier.

C. Contribution

Thijs Houtenbos	Scripting, SSL man in the middle, demo, challenge cracking, report
Javy de Koning	Lab setup, traffic inspection, presentation, challenge cracking, report
Jurgen Kloosterman	Lab setup, traffic inspection, report
Bas Vlaszaty	Memory analyzing, source code inspection, report

D. Traffic inspection set up using WireShark

Install and enable both network cards while using the default drivers. Then open a Command Prompt in Windows 7 with Administrator privileges and execute the following commands specifying the SSID of the secondary network card and a password for the Android phone to authenticate to.

Listing 3: 128bit AES encrypted transactions database

```
netsh wlan set hostednetwork mode=allow ssid=<ssid_name> key
   =<password> keyUsage=temporary
netsh wlan start hostednetwork
```

Next configure the settings of the primary adapter to share the Internet connection (Internet Connection Sharing, ICS) and also to restrict the traffic to HTTP/HTTPS. Then start Wireshark and select the secondary interface and start to capture network packets. In some occassions Wireshark generates an error when the WinPcap driver (called NPF) can't be initiated. This can be fixed with the following commands:

Listing 4: 128bit AES encrypted transactions database

```
sc qc npf
sc start npf
```

References: [19] [20] [21] [22]

[19] http://www.wireshark.org/
[20] http://ask.wireshark.org/questions/1281/npf-driver-problem-in-windows-7
[21] http://www.melbpc.org.au/pcupdate/3006/3006article8.htm
[22] http://wiki.wireshark.org/CaptureSetup/CapturePrivileges

E. HTTP Session

Listing 5: 128bit AES encrypted transactions database

```
==========================================

GET /session/loginchallenge?accessToolUsage=SOFTTOKEN&
   accountNumber=123456789&cardNumber=123 HTTP/1.1
Content-Type: application/json;charset=UTF-8
Accept-Language: en
User-Agent: [Bankieren]/[3.0.1] [Samsung]/[GT-I9100] [
   Android]/[4.1.2] [null] [] []
Host: www.abnamro.nl
Connection: Keep-Alive
Cookie: LBCSS=
00000000000000000000000000000000000000000000000000
0a000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000
Cookie2: $Version=1

HTTP/1.1 200 OK
Date: Thu, 22 Nov 2012 12:00:37 GMT
Server: IBM_HTTP_Server
Expires: 0
Last-Modified: Thu, 22 Nov 2012 12:00:37 GMT
Pragma: no-cache
Content-Length: 730
Set-Cookie: LBCSS=
00000000000000000000000000000000000000000000000000
0a0200000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000; Path=/
Cache-Control: no-store, no-cache=set-cookie
Keep-Alive: timeout=15, max=95
Connection: Keep-Alive
Content-Type: application/json
Content-Language: en-US

{"loginChallenge":{"userId":"0123456789_12","challenge
   ":"020008
   ebdac96801d257f103000450ae13e5040100d9c166061a8d097c006f}

==========================================

PUT /session/loginresponse HTTP/1.1
Content-Type: application/json;charset=UTF-8
Accept-Language: en
User-Agent: [Bankieren]/[3.0.1] [Samsung]/[GT-I9100] [
   Android]/[4.1.2] [null] [] []
Content-Length: 703
Host: www.abnamro.nl
Connection: Keep-Alive
Cookie: LBCSS=
0000000000000000000000000000000000000000000000000000
0a02000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
Cookie2: $Version=1

{"accountNumber":123456789,"cardNumber":123,"challengeHandle
   ":"1331699728","response":"872
   f1b0fe850a7c2341fc133840ad4c*"
HTTP/1.1 500 Internal Server Error
Date: Thu, 22 Nov 2012 12:00:38 GMT
Server: IBM_HTTP_Server
Expires: 0
Last-Modified: Thu, 22 Nov 2012 12:00:38 GMT
Pragma: no-cache
Cache-Control: no-store
Content-Length: 197
X-Cnection: close
Content-Type: application/json
Content-Language: en-US

{"messages":[{"messageKey":"MESSAGE_SEC02L_S_0030",
"params":null,"messageType":"ERROR","messageText":"
The identification code entered is incorrect. Please enter
the correct identification code."}]}
```