(In)Security of Mobile Banking...and of Other Mobile Apps*

Eric Filiol and Paul Irolla
ESIEA - Laboratoire de virologie et de cryptologie opérationnelles
France
{filiol,irolla}@esiea.fr

March 26, 2015

Abstract

Mobile banking is about to become the *de facto* standard for banking activities. Banking apps on smartphones and tablets are becoming more and more more widespread and this evolution aims at strongly limiting the classical access to banks (physical, through PC browser, through ATM). The aim is to first i cut the cost but also to make the personal data explode.

Then three critical issues arise because we entrust those mobile applications by feeding them with passwords, private information, and access to one of the most critical parts of our liking (money): Do those applications protect our private life and especially which kind of information is leaking to the bank? Are they containing vulnerabilities that could be exploited by attackers?

In this talk, we are going to present a deep analysis of many banking apps collected in the world as well as the Facebook app which is likely to be the most used app in the world. We have performed static and dynamic analysis based on the binaries AND the source code. We will show that almost all apps are endangering our private data (sometimes severely) but in a few cases the presence of vulnerabilities are extremely concerning.

Keywords: Android App Security - Mobile Banking - Static Analysis - Dynamic Analysis - Data privacy.

1 Introduction and Background

Mobile devices (tablets, smartphones and soon connected watches) have now invaded our life and there is no single aspect that is not impacted by mobility:

*This work has been presented at Black Hat Asia 2015.

health, everydays common tasks and uses, banking, business, social media.... Mobile devices are becoming more and more the unique entry point of our life. In this respect two main issues are critical:

- the security of our data and mobile environments. Malware can infect our devices. Vulnerabilities can be exploited to enable illegimitate access to them by an attacker;
- the protection of our privacy since many data may naturally leak to providers/companies that are selling/proposing not only the devices but also the different services on them. Most of the times, these companies and us, as customers, do not share the same interests.

In a mobile device, apps are in fact the most critical parts. We install them more or less voluntarily (more and more services are now available ONLY with mobile apps). They get more or less extensively access to the operating system and our data. They are suspected to do far more that the intended and official actions they are supposed to. In a word, can we trust them?

As far as mobile banking is concerned, those issues are of course even more critical. It relates not only to our money and assets but also to all the aspects of our life that are impacted by money: what we buy, what we do... While mobile banking becomes more and more invasive. Banks intend to reduce costs by closing more and more bank (physical) agencies to develop virtual agencies that can be accessed only through mobile banking. This the reason why – without loss of generalities – we focused on banking apps in this paper. However the reader must keep in mind that from the code perspective there is no fundamental difference with the other application domain. Insecurity and privacy are common issues which concern all apps.

This study has been conducted in the context of the OpenDAVFI project [9] which is the official, free and open fork of the former DAVFI project [5]. Funded by the French Government (6 millions euros with 0.35 % of funding), its goal was to design and develop a sovereign and trusted new generation antivirus software for Android, Linux and Windows. The main part of the project was conducted by our laboratory.

As far as the Android version is concerned, we have deeply modified the operating system to provide more security: file system encryption, preventing physical access-based attacks (enabling thus to access to data and to the system directly) and provided many additional security features like SMS encryption, VoIP encryption, fake geolocation system... But one of the key features is that all apps are available on a secure market only. Beforehand each app is fully analyzed (static and dynamic analysis including a reversing step). Whenever safe AND compliant to our security policy, the app is then certified and digitally signed before made available on the secure market. It means that contrary to most existing app markets (including that of Google), every app goes through a deep security analysis.

Very soon in our study we have been aware that simply asking to any app not being a malware was not sufficient. We discover that even legitimate apps can

be malevolent when it comes to targeted marketing and user tracking capabilities. A few apps contains severe vulnerabilities. Thus, the classical "malware" definition needs to be extended.

An app is trustworthy according to our Trust Policy if and only if the following mandatory conditions are met:

- It does not contain hidden functionalities.
- User information collection must be motivated by explicit functionalities.
- Web communications involving personal user information must be encrypted.
- The app does not contain known vulnerabilities.

In this paper, we thus describe the tools and the methodology we have developped to analyze and certify mobile apps. This paper is organized as follows. In Section 2, we present the different tools we have delopped for static analysis, dynamic analysis and apk database enlargement. In Section 3, we first present the results regarding the app which is likely to be the most used app: *Facebook* app to illustrate our methodology. It represents the worst case situation in terms of privacy violation and infringement. In Section refbankapps, we present the detailed results regarding the security of banking apps we have analyzed throughout the world and we present a world tour of insecurity by esposing results for a few bank apps. In particular, we compare the comparison between the Western world and the Asian world since we observed a significant difference in terms of development security and privacy protection. Section 5 will conclude and present future work.

It is worth mentioning that most banks have been contacted to provide (for free) all technical details. Up to now, only a very few have answered but did not take our recommendations into account. As a general observation is is very difficult to identify the suitable contact point in a bank. That is why we expect that people in charge of app security in their respective bank will contact us.

2 Our Analysis Tools

We have developped three main tools for analyzing and certifying android apps:

- *Egide* which performs static analysis and malware detection.
- *Panoptes* providing advanced dynamic analysis (network communications analysis at runtime).
- *Tarentula*, a complex web crawling framework to collect malicious and legit apps throughout the world.

2.1 The *Egide* Tool

Egide is a prototype for static analysis based on advanced and innovative data-mining techniques. It detects malware by exploring similarities (as defined in data-mining techniques) with known malware. Moreover, it produces a static analysis report which is a direct support to the manual analysis, for research and deep analysis purposes.

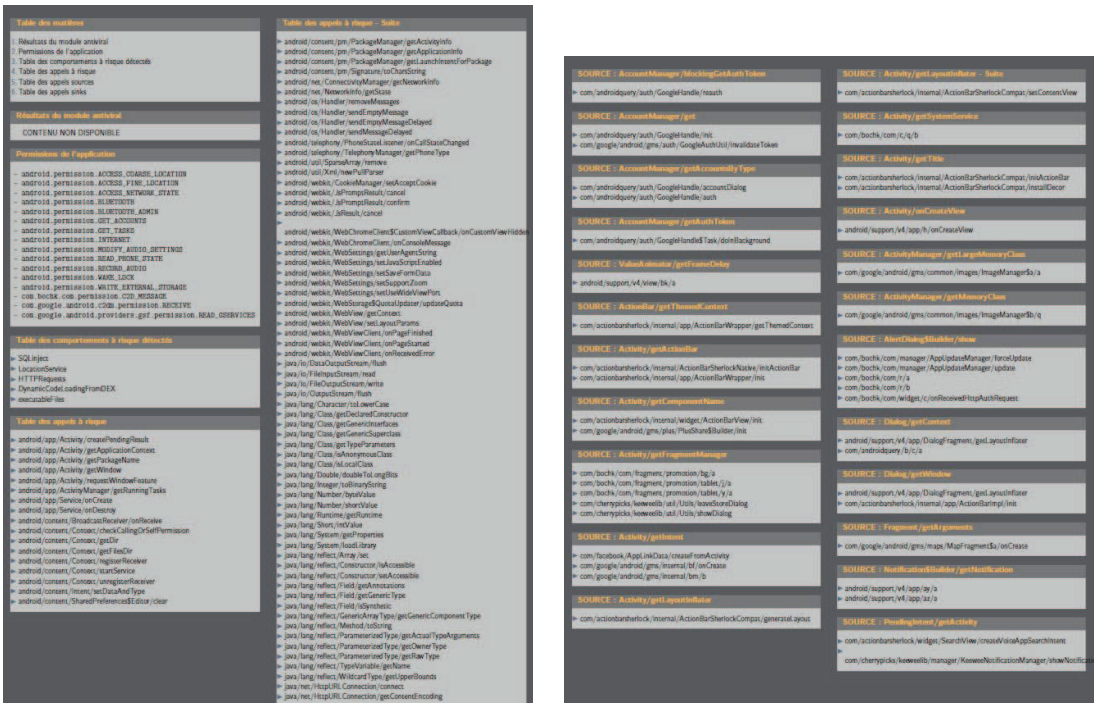

Figure 1: Egide Reports

The usual way to detect malware relies on a more or less manual construction of characteristic signatures (as a sequence or patterns of bytes). It is a deterministic detection method whose drawbacks are first to detect only already known malware and second is prone to false positives. In order to deal with these ever-growing limitations, antiviral research is moving toward heuristic detection (as algorithms aiming at finding a sub-optimal solution to a naturally untractable problem with regards to the complexity theory). *Egide* precisely works in this context by developping a fully heuristic detection based on behaviour similarities with known malware.

This approach has two strong points:

- It enables an efficient detection of unknown malware (they often exhibit common behaviours with known malware).

- The manual and time-consuming step to update signature database is no longer necessary. Consequently, the delay to adapt to the attacker is far quicker.

However the accepted hypothesis – which constitues the basis of the proposed solution – is that there is not necessarily common characteristics between all malware. This is why the traditional malware detection classifies malware into families. It is in fact these families that share common behaviour characteristics. From the existing classification of malware, we wanted to develop a system that learns by experience what those characteristics are.

How is this achieved? Applications of the database are reversed in order to obtain an equivalent source code. As Android applications are written primarily in Java, reverse engineering step is fairly simple. Several open source tools exist for this task.

Once the resources and code are obtained, the following information is retrieved. They are the "characteristics" of the application and consequentely they define our basis for statistical and calculations of similarities on malware families

- Permissions.
- Java paquets/libraries names.
- Java classes and methods names.
- Character strings.
- The entry points in the application.
- Behaviours.
- Calls to the Andoid API.
- Calls to third-party APIs.
- Hash values for each ressource file.
- Structure signature for each method.

Malware behaviors are viral patterns that we often encounter in Android malware such as shell commands to remount the system read and write mode, or dynamic loading code from a resource file.

The last feature is the signature of the structure of each method in the application. Each character of the signature for this method is a structural opcode of the method. The idea is that some opcodes are more important than others such as "`if`", "`goto`" and "`call`" primitives. These are structural opcodes. If two methods have similar structures then they are likely to be similar. Moreover since the structure is represented by a character string, then it is possible to apply efficient text similarities computation algorithms to it.

The statistics of all of these features are computed for all legit applications as well (goodware) as well as for each malware family.

Generally any Android malware look likes a healthy/legit application in which a piece of malicious code has been inserted. There are therefore many healthy behaviors and only a few malicious behaviors. In order to separate the wheat from the chaff, the inverse frequency of the characteristics of healthy applications is multiplied to the frequency of features found in malware families. This is an adaptation of the TF-IDF [11] algorithm used by search engines to give a relevance index to words. For example, the words "`the, of, then`" ... are not worth anything in a Google search. They are present in all documents which makes them irrelevant.

The result thus obtained for each feature is a weight representing its importance/relevance in the malware family. This will be used for similarity computation.

It becomes possible to calculate series of similarity factors between an application and each malware family. The problem is that we need a single factor in determining whether an application is malicious or healthy. This is where an learning system based on neural networks is used. Its role is to determine beyond which values or similarity factors combination values, we can decide whether the application is malicious or not.

The similarity vectors are used to train the neural network. Once this training phase is completed, the antivirus is ready to use. The effectiveness of such an antivirus has been tested on real conditions and has been proved very efficient, especially on new malware that were targeting specifically targeting organizations of vital importance. They all were detected successfully without any false positive.

Finally, for each analyzed application, *Egide* generates a static analysis report (see Figure 1). It shows for which line in the source code, behaviors and risky method calls are detected. This provided thus a starting point to the expert in case of an aditional manual static analysis.

Figure 2: Egide Final Report

2.2 The *Panoptes* Tool

This tool performs an advanced dynamic analysis by anayzing network communications at runtime. It can monitor all communications based on `HTTP`, `HTTPS`, `POP`, `IMAP`, `SMTP` between the application and the Internet. Network communications are the bottleneck through which pass almost all the risky behaviors and vulnerability exploitations. This is why it is essential to be able to analyze it.

The system is quite simple: the phone is connected during the dynamic analysis to a WiFi access point which is running an interception program. Plaintext communications can be intercepted directly. For encrypted communications based on SSL/TLS, a fake certificate from a certificate authority was added to the native list contained in the phone. This means that recipients can be authenticated by that authority. Whenever the phone starts an encrypted request, the transaction is intercepted. The Wifi access point forwards the request to the

recipient as if it were the legitimate issuer and transmits the response from the server, which is certified by the false Certificate Authority in the phone. The access point is considered at this present time as the legitimate recipient to the phone and as the legitimate sender contacted by the server: it is able to decrypt communications. It is in fact a simple man's middle attack on the SSL/TLS protocol.

As soon as the tested application is closed, the `HTTP` and `HTTPS` sessions are reconstructed. An interactive graph of communications in the form of html document (Figure 3) is generated in order to navigate through this huge mass of information. This document is used for manual analysis and allows in particular to detect potential vulnerabilities and personal information leakage.

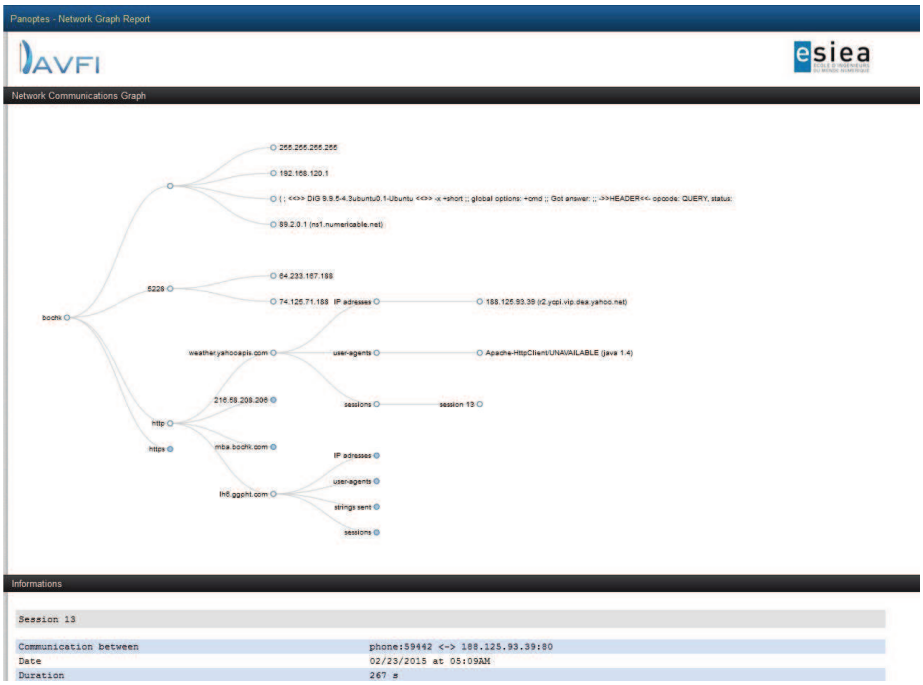

Figure 3: Panoptes Graph

The test platform is pretty simple. The required material is

- A Wifi card with Master mode available.
- An Ethernet connection.
- Rooted Android phone.

2.3 The *Tarentula* Tool

Tarentula is complex web crawling framework to collect malicious and legit apps throughout the world.

We started the project with a DAVFI Android malware database built with malware coming from the University of North Carolina [1] and from the *Contagio* website [6]. Other sources have been added later on [7, 8]. Our database contains around 1800 samples distributed over around 40 different malware families. Having malware is not enough, you also need clean applications (goodware). But how to determine whether an application is malicious or not? There is no perfect answer here. The best approach consists in gathering open source applications. This minimizes the risk of introducing a malicious content into the database.

So we crawled the alternative library of open source Android applications *Ddroid* as well as all Google Android code projects, with a web bot. In all about 1,800 open source applications have been collected. Healthy and malicious databases thus have the same size.

Malware detection heuristic methods are generally methods coming from the data mining domain. So they need to be based on sound statistical to generalize on actual data effectively. In other words, we needed more sample to have an excellent statistical reference set. So we worked on collecting of Android malware massively. This is an interesting topic because it is rarely discussed in scientific articles on malware detection or just flown over. However this is a central topic in the context of heuristic detection, especially if we intend to build a heuristic antivirus which is competitive with other commercial antivirus.

The question that arises from this observation is "how AV companies collect their antiviral malware?" Their main sources are most likely sample submission by customers/users/Virus Total and database sharing with other antiviral companies.

One can see that the databases of the tool Andrubis were fed 70% through the exchange of samples with the antiviral companies [2, Table II, page 15]. The second largest source is labeled"*Source Unknown*" and is actually made of applications uploaded by individuals and companies. These are ways that we did not have at our disposal, so we turned to another public sources: the web.

We designed a web crawler (web bot) called *the Tarantula* to download applications on all types of media and sources. Unofficial media such as `ftp`, alternative Android markets and torrents are preferred for two main reasons:

- The probability of finding malware is higher.
- Crawling Google Play, which is the official source, is a complex task because you must have a Google Account linked to a unique identifier for the phone and it is not possible to download continuously with this account. Indeed Google blocks requests with a fairly low limit of successive downloads (30). To avoid account blocking, it is necessary to manage a large group of accounts in paralell. It is also necessary that all these accounts are linked to phones configurations representing a wide range of

possibilities because it is not possible to download an application for a non-compatible configuration.

Tarantula has allowed the collection of 280,000 applications to date. The identifcation of malware among these applications is a work in progress. We reasonably expect to find between 10,000 and 20,000 malware (rough estimate).

The architecture of *the Tarantula* is described in Figure 4.

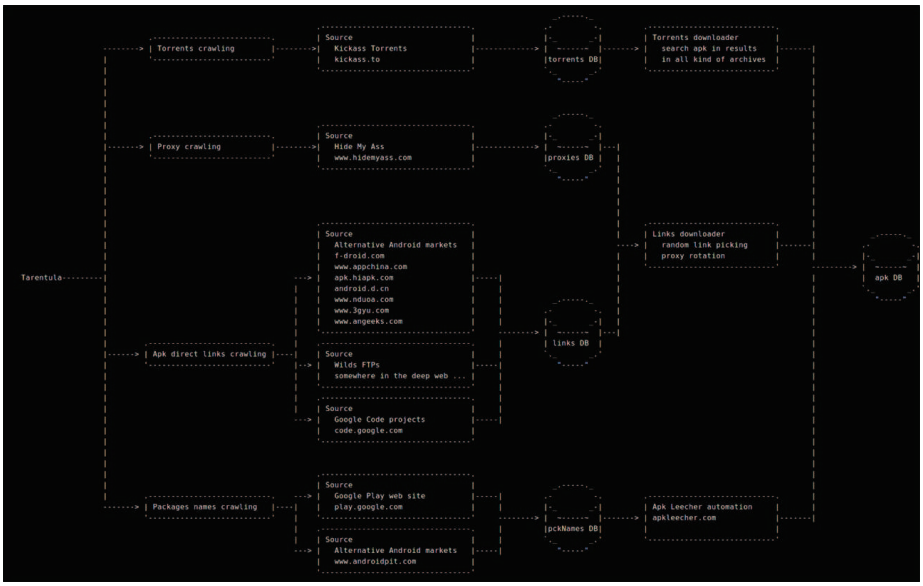

Figure 4: Panoptes Graph

3 The Paradigm of Insecurity: The Facebook App

In order to illustrate how an application may undermine our privacy, the worst example is likely to be the *Facebook* application. This application is one of the most used app in the world and Facebook by nature contains a lot of users' private information. In other words, *Facebook* is some sort of voluntary STASI since *Facebook* spies us and people not only contribute but love that. However they are totally unaware of how their privacy is undermined. In the *Facebook* world there is no such things as (virtual) friends.

Facebook collects user's submitted informations but what about information collection without the users' knowledge? What about informations stored in plaintext on the phone (unencrypted)? We have experienced in our lab that any plaintext data stored in the phone can be stolen in less than a minute by means of a "digital serynge" that can inject, collect data into the phone whatever may be the security settings.

After the *Facebook* application is launched we then connect it to our personal account and after some basic navigation, we get all local data created during the process:

```
adb shell su -c 'cat /data/data/com.facebook.katana/**/*'> facebook-data.dump
```

With this command you dump all local data on your computer. You get one file containing all of them. After looking it in binary mode, we have ve seen recuring patterns of interesting data. The next thing to do is to build some regex to parse it.

Here is a simple regex to get the personal contact list.

```
/"displayName":"([^"]*?)"ng.*?"friendshipStatus":"([^"]*?) ".*?"contactType":"([^"]*?)".*?"
                cityName":"([^"]*?)"/
```

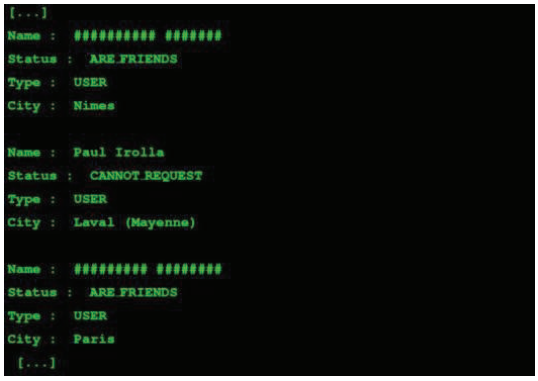

Figure 5: Facebook Contact List

We can then know the name, the friendship status, where they live or at least the last city they provide... We also can get the phone number if your contacts provided it, a score about how often you discuss with this person (a number between 0 and 1).

But it is not over and worse things are coming. You can get a huge amount of personal data that are stored unencrypted in the phone:

- Private messages.
- Private photos.
- Private wall content.
- Many other private and non private data...

So an attacker does not need to know your account credentials, it only have to get an access on your phone for less than a minute. And this can be a much easier task with a digital serynge like ours.

Let us now expose what kind of information, the *Facebook* application is leaking by performing the dynamic analysis. The *Facebook* app make a one-kilometer POST request (due to lack of space, we will not give it here but it is available on request; since it is commercial code we cannot publish all the technical details unfortunately). Since it is under a cryptic form (only apparently) here is the reverse procedure to apply to determine what is going on:

1. Unescape url codes recursively.
2. Parse the output string as a JSON object.
3. Until the data super-structure is entirely reversed
   (a) Try to parse each string in the JSON object as a JSON object.
   (b) Try to decode each strings which seems to be in a base64 format, then
      i. Try to unzip the result with gzip if the magic number is "`1F8B`",
      ii. and finally read the result string with a WINDOWS minidump reader like WinDBG (it is not a joke!).

Let us now summarize what information goes is leaking towards the *Facebook* servers:

- Bootloader used.
- Device model/manufacturer/serial/hardware/ROM.
- CPU model/architecture/version + Kernel version.
- Screen settings.
- The complete list of system applications.
- All environnement variables.
- Open file descriptors count.
- Software and hardware file descriptors limits.
- Locations settings, developper settings, lockpattern settings:

```
1  LOCK PATTERN ENABLED=1
2  LOCK PATTERN SIZE=3
3  LOCK PATTERN VISIBLE=1
```

- Application settings.
- **Security settings**.
- Sound used for alarm alert.

- Spell checker settings and Screensaver settings.
- Notification settings (including used sound).
- Battery settings (including current energy level).
- Sounds/music settings, camera settings, Wifi connection settings.
- Sdcard and memory size/free space/used space.
- Usual user tracking info (timestamp for each user action):

```
connection = WIFI
connection class = POOR
network extra info = Panoptes-AP
```

In conclusion, *Facebook* knows everything on you but also on your phone/tablets.

4 The Analysis of Banking Apps

We are now presenting the analysis results for a lot of bank apps throughout the world. Up to now, we have analyzed around 50 apps.

Bank	Country	Bank	Country
BNP Paribas	France	LCL	France
Crédit Agricole	France	Sofinco	France
Société Générale	France	BforBank	France
Finaref	France	Bradesco	Brazil
BMCE	Morocco	Barclay	UK
UBS	Switzerland	JP Morgan	USA
Wells Fargo	USA	Bank of America	USA
Burke and Herbert	USA	PNC Financial Service	USA
Commerzbank	Germany	Deutsche Bank AG	Germany
HSBC	UK	Santander Group	Spain
Sberbank	Russia	Hapoalim Bank	Israel
Shahr Bank	Iran	VTB	Russia
LandKredit	Norway	Nordea Mobilbank	Norway
Oversea-Chinese Banking Corporation	Singapore	DBS Bank	Singapore
United Overseas Bank	Singapore	Bank of China	Hong Kong
Bank Negara	Indonesia	Commonwealth Bank of Australia	Australia
National Australia Bank Limited	Australia	Bank of Communications	China
Mitsubishi UFJ Financial Group	Japan	Advanced Bank Of Asia	Cambodia
Public Bank Berhad	Cambodia	Bangkok Bank	Thailand
State Bank of Mongolia	Mongolia	HanaNBank	Korea
Agricultural Bank of China	China	Industrial Bank of Korea	Korea
Mizohobank	Japan	State Bank of India	India

4.1 A Few Statistics

Before exposing a few case studies to illustrate the different cases of lack of security in banking apps, let us give a summary on the different key points.

PERMISSIONS	Western Banks	ASIAN BANKS
INTERNET	100%	93%
ACCESS_NETWORK_STATE	96%	87%
ACCESS_FINE_LOCATION	71%	80%
WRITE_EXTERNAL_STORAGE	68%	73%
READ_PHONE_STATE	61%	60%
CAMERA	54%	53%
ACCESS_COARSE_LOCATION	54%	73%
c2dm.permission.RECEIVE	46%	40%
CALL_PHONE	39%	47%
ACCESS_WIFI_STATE	39%	47%
READ_CONTACTS	32%	33%
gsf.permission.READ_GSERVICES	29%	33%
GET_ACCOUNTS	29%	27%
ACCESS_MOCK_LOCATION	14%	7%
READ_EXTERNAL_STORAGE	14%	13%
RECEIVE_BOOT_COMPLETED	14%	13%
WRITE_CONTACTS	11%	13%
NFC	11%	20%
RECEIVE_SMS	11%	20%
WRITE_SETTINGS	11%	7%
CHANGE_WIFI_STATE	11%	20%

PERMISSIONS	Western Banks	ASIAN BANKS
SEND_SMS	7%	7%
RESTART_PACKAGES	7%	13%
CHANGE_NETWORK_STATE	7%	7%
READ_SMS	7%	7%
RECORD_AUDIO	7%	20%
READ_LOGS	7%	13%
ACCESS_LOCATION_EXTRA_COMMANDS	7%	13%
KILL_BACKGROUND_PROCESSES	7%	0%
ACCESS_NETWORK	4%	0%
GET_TASKS	4%	47%
RECEIVE_MMS	4%	0%
MOUNT_UNMOUNT_FILESYSTEMS	4%	13%
DISABLE_KEYGUARD	4%	7%
READ_OWNER_DATA	4%	13%
READ_CALENDAR	4%	0%
WRITE_CALENDAR	4%	0%
BROADCAST_STICKY	4%	7%
SMARTCARD	4%	7%
NFC_TRANSACTION	4%	0%
ACCESS_DOWNLOAD_MANAGER	4%	0%
READ_CALL_LOG	4%	0%

Figure 6: Bank Apps Permissions Granted

As far as permissions are concerned (how far and deep, apps can access the different resources in the phone), Figure 6 summarizes our overall observations (percentages are rounded down). From a general point of view, while permissions can be considered as legitimate to enable the app to operate properly, a few permissions must be seen as the direct violation of our privacy at least potentially: `ACCESS_FINE_LOCATION, CAMERA, CALL_PHONE, READ_CONTACTS...`. Our interpretation is that in most cases, the developers do not want to complicate things and gives root access to the app to facilitate the development. We want thus consider that in most case this comes from a lack of seriousness that an intended will to access our data. But banking applications have generally a large power of action on the Android system.

Regarding behaviours, the situation is not better (see Figure 7).

BEHAVIORS	Western Banks	ASIAN BANKS
Load app content from web	96%	87%
Can use clear text communications	89%	87%
Get OS name	75%	73%
Get android unique id	71%	20%
Get IMEI	61%	73%
Use addJavascriptInterface	54%	73%
Get OS version	50%	100%
User tracking capabilities	25%	47%
Get MAC address	18%	40%
Get MSISDN (Phone number)	11%	27%
Get IMSI	7%	20%
Get CID	4%	7%
Get LAC	4%	7%
Get SIM serial number	4%	20%
Get access point MAC address	4%	20%

Figure 7: Bank Apps Behaviours

Let us stress on a few key points.

- Almost all applications dynamically load the graphical content of their pages from a remote server. The possibility offered by mobile development (both Android, Apple) to execute `html` and `javascript` is pushing companies to outsource the application content: what has been done for a website can be copied almost unchanged for an application. Moreover, as the content is loaded dynamically, there is no need to update the application to change its graphics. The problem is that this makes the vulnerability exploitation much easier in case of the content is loaded through the `HTTP` protocol since an attacker can then inject code, including JavaScript *via* man-in-the-middle attacks. The use of `HTTPS` is not always systematic, especially in third-party development frameworks, particularly those used for ads.

- Almost all applications have the ability to communicate in plaintext (unencrypted form) and the majority of them use the *addJavascriptInterface* function. This feature entitles the javascript code to call java predefined functions. Only on older versions of Android, there exist a vulnerability induced by this feature, which allows JavaScript to call any Java function java through a reflection mechanism. This gives the opportunity for man-in-the-middle attacks to get a remote shell on the phone whenever

the content is loaded via `HTTP`.

- We see that a large number of private information is retrieved such as IMEI, MAC address, phone number...

Lastly, it is worth mentioning that the permissions used and the detected behaviors are not all displayed. It means that the user is far from being aware of all what the app is actually doing.

4.2 Cases Studies

Let us now present a few cases studies.

4.2.1 JP Morgan Bank (USA)

The first application is JPMorgan Access app. This application is used for managing JPMorgan accounts on mobile.

What is interesting is a `json` file which is sent by JPMorgan servers whenever the app is started (Figure 8). This *json* file contains a field "*signature*". It is actually a long string of base64 characters. This string seems a bit too long if we suppose that the goal is to authenticate the phone/user, for example. Therefore it is possible that this is an encrypted message.

Figure 8: JP Morgan Access App - JSON File sent to the phone

We used an open source tool *APImonitor* which is able to exploit an application. This tool decompiles the application then adds a monitoring function on the target function parameters and finally recompile it. This enable to display the arguments of these target functions in the *logcat* file (the centralized Android log system) dynamically.

The manipulated implementation reveals the reception of the signature chain and subsequently the execution of a decryption function, supporting our hypothesis. We also get the return value of the function that actually is the deciphered string. Due of a limitation on the number of characters that can be written in the *logcat* file, it is however not possible to recover all of this chain.

Here is what one obtains:

```
1 ‘‘oxrohccRtI/m1w9NC/7nqwANljaa8fORRXcJ2S1EiThNdeuW6GEr
2 L7NQogAnOFtPdYlwP1Gh2+0aNqsnrKeGbw==
3 ##########
4 MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAMx6N9b4yaIFC60o
5 f8YWU1eO8sh4KRoldfJRKmtVazOKg2p3UUwMT5oUwBYYEhWsSl
6 +bTD6DMCIQrwr2iSWO9DkCAwEAAQ==
7 ##########
8 Root Call Blocker,LBE Privacy Guard,Dual Mount SD Widget,
9 Hexamob Recovery Pro,Total Commander,Boot Man’’
```

So there are several strings separated by delimiters. These delimiters allow us to find the place in the code where the strings are processed:

```
1 ‘‘String[] arrayOfString = str2.split(’’##########’’
2 ...
3 RootTools.log(’’Executing sh Commands : ’’ +
4 arrayOfString5[0] + arrayOfString5[1]);
5
6 arrayOfString6[0] = arrayOfString5[0] ;
7 arrayOfString6[1] = arrayOfString5[1] ;
8 ...
9 List localList = runcmd(arrayOfString6);’’
```

The chain is split and a portion is executed by the shell interpreter of the phone. In other words, this is the remote execution of arbitrary shell commands. In addition, the *runcmd* function executes the command as *root* if the phone has the ability to do so (*id est* on rooted phones).

This behavior comes from the action of security framework that is in charge to check a few given parameters that may indicate that the phone has been infected. Part of this check is done through the remote execution of shell commands whenever you start the application.

With the current behavior in place, JPMorgan operators have the capacity, if they wanted to, to control their clients' phone remotely. This should be seen as a backdoor. All the technical details have sent to the bank in January 2014 (they contacted us in fact). We do not have news since. The vulnerability is still active after two months. Nothing has been corrected yet

4.2.2 BNP Paribas (France)

The second case is that of BNP Paribas mobile app to manage personal accounts. Dynamic analysis here has not fully worked because the `HTTPS` connection from the Panoptes access point have been rejected. Some banking applications do not trust the certification authority list contained in the smartphone and embed their own list. This is in fact a strong key point from a security perspective.

Nevertheless we have indentified a vulnerability that is involved in an advertising framework used to promote additional services of the bank, and these connections are unencrypted.

The framework uses an interesting javascript code. The `A4STRK` function does not appear to be a normal javascript function. This seems to be the name of a Javascript to Java interface. Parsing the app source code gives the answer to us: this is actually the name of a *javascriptInterface* interface.

The problem is that this javascript code is transmitted unencrypted and therefore can be modified to exploit vulnerability which affects either the *javascriptInterface* or its container *webView* through a man-in-the-middle attack. There exist several well-known vulnerabilities (CVE-2013-4710 and CVE-2012-6636) which can be exploited in two cases and that would enable to get a remote shell on the phone:

- The phone contains the Android API before version 4.2. This case still relates to many users.

- The application targets an Android API before version 4.2, even if the Android API is itself not vulnerable. Many applications, including the BNP Paribas app, target obsolete versions (API v8) in order to be compatible with a maximum number of different configurations.

Furthermore there is at least one vulnerability affecting *webView*. This vulnerability has not been disclosed by Google and consequently Google will not publish any security patch to correct it for version 4.3 and prior versions. It is therefore a major vulnerability which allows a third party to take control over the phone.

We have sent all technical details to the security officer of the bank who promised to correct everything (december 2014). Three months later, the bank has corrected nothing. The vulnerability is still exploitable.

4.2.3 Bradesco (Brazil)

Regarding this Brazilian bank, here are the main results:

- The dynamic analysis shows that a private key is received unencrypted from Bradesco servers. This is a private key used to identify the phone by Bradesco servers and is used for additional services that Bradesco offers (*InfoMoney*). This key can be stolen by a man-in-the-middle attack to impersonate.
- Furthermore the application embeds an obsolete version of *jquery* JavaScript lib that contains multiple vulnerabilities. We will let Bradesco assessing the potential risks of these vulnerabilities.

4.2.4 SBerbank (Russia)

The Russian bank Sberbank app is an interesting case in that it shows for all phones – regardless of the OS – and their relationship with the surrounding wifi access points.

Interesting information are sent unencrypted to the Yandex servers (the Russian equivalent of Google). The *wifinetworks* variable contains what appears to be a list of MAC addresses and signal strength (in decibels). The first address is actually the MAC address of the wireless access point, while the others are those of surrounding access points.

```
1 wifinetworks=001122334400:-45,0060B3E268C8:-66,4018B1CF2655:-77,4018B1CF2255:-77,4018
                B1CF6455:-79,C8D3A352B1B0:-78,4018B1CF6
2 515:-83,586D8F747EC7:-85,4018B1CF2654:-76,4018B1CF2254:-83,4018B1CF6454:-84,4018B1CF6514
                :-88,4018B1CF23D4:-90,4018B1CF23D5:-83,4018B1CF63D4:-92,D8C7C8138A92:-90
3 (001122334400 is the MAC address of our wifi access point used for interception)
```

This information is sent by the Yandex maps API for geolocation purposes. Why does the app need the wifi access points? GSM geolocation is not accurate indoor. However with a list of wifi surrounding access points and signal reception strength of these points from a device, it is possible to locate by triangulation. This involves having a precise location of WiFi access points.

Looking at the answer from Yandex server, we have identified a XML document with a tag "*foundByWifi*" and coordinates that are liely to be the location of the nearest bank in Russia. The question that naturally follows is: how is it possible that Yandex has the ESIEA Wifi acess points (the academy which hosts our lab) in their MAC address database?

Yandex, like Google and any geolocation operators do, has a global wifi hotspots database in order to make the precise location indoors. It is the only explanation which comes in mind. How do they build such a database? Some of these access points has been manually collected and are public. It is also possible that open-source wardriving databases as well as data collected and stored by Google cars have been used additionally. One can also think that there have been agreements with telephone/Internet operators to access to a list of public wifi routers. But it would provide a rather partial mapping of global access points only. In reality, each of our phones is contributing unconsciously to this huge database. Here is what the Google or Yandex maps services are doing while running continuously:

- GSM position is sent at regular time intervals to their servers. When this position cannot be obtained precisely, it means that it is an indoor location then MAC addresses, SSID (Google) and the signal reception strength of surrounding access points are mapped and sent to the last known GSM location. The unknown access points are added to the database. Just three different GSM precise positions are sufficient to triangulate and detrmine the position of a new wireless access point precisely. This is the way through which they are able to have a fairly complete mapping of world global access points.

- Considering the amount of information collected by operators like Google and Yandex, like many others, this is even no longer surprising. This information could be misused. Being aware that open source projects like *snoopy-ng* [10] or cheap drones can profile users through wireless access points they connect to, we can easily imagine what could be done by companies which have nearly unlimited information, resources and money.

4.2.5 Bank of China (China)

The application can check for available updates. A link on the official market is sent whenever a new update is available. Then the app downloads and installs the file. Moreover other navigation links (loaded by the app) are received. The main issues are

- Security issue: this process is done entirely in *HTTP*.

- Potential risks with a MitM attack:

  - Installation of an arbitrary app by social engineering
  - Loading of arbitrary web pages.

– Exploiting the confidence of using a bank app, social engineering could be devastating.

4.2.6 Miscellanous Banks

National Australia Bank Limited The client ID is written in the *logcat* (the centralized Android log system). On vulnerable Android version, each application can read logs of any other application. Moreover any attacker that have a physical access to the device could obtain this information. It could be used as a base information for social engineering attack. The client contact list could be written in the *logcat*, as an unconditional `'Log.v("ContactListAdapter", "Phone contact: " [...] )'` is called. However the dynamic analysis could not reach this code section, so this behavior could not be formally verified.

Bangkok Bank

- All sensitive code sections are dynamically decrypted during application runtime. It make the reverse engineering very tought.
- Panoptes SSL/TLS hacking did not succeed. The application have fake root CA countermeasures.
- No plaintext communication has been observed during dynamical analysis.
- No vulnerabilities have been hightligthed.

State Bank of Mongolia

- The authentication is performed in 2 steps: username check and password check. With this process, an attacker can guess username with much less different possibilities.
- Username can be remembered and is stored unencrypted in the phone.

4.3 Banking Apps Security Comparison: Western World versus Asian World

The overall security awareness of Asian banks seems superior to what we have observed for European/American continents. In particular, the use of custom obfuscation, security routines on the native layer (c libs.), custom trusted SSL root CA... is prevalent and shows a significant care for security. Therefore the analysis was much harder than what we have performed for Western Banks apps But there is always some black sheeps in the flock (like the Bank of China, see further).

Figure 9 summarizes the results on Asian banks analyzed.

Banking application	Vulnerability found	Plaintext communications during runtime	Fake root CA countermeasure	User tracking capabilities	Strong use of crypto/obfuscation
Oversea-Chinese Banking Corporation	No	No	Yes	Yes	Yes
DBS Bank	Potential	Yes	Yes	Yes	No
United Overseas Bank	No	No	Yes	No	Yes
Bank of China	Yes	Yes	No	Yes	No
Bank Negara Indonesia	No	No	No	No	No
Commonwealth Bank of Australia	No	No	Yes	Yes	No
National Australia Bank Limited	Yes	Yes (But harmless)	No	Yes	No
Bank of Communications	No	No	Yes	No	No
Mitsubishi UFJ Financial Group	No	Yes	Partially	Yes	No
Advanced Bank Of Asia	No	No	Yes	No	No
Public Bank Berhad	N/A	N/A	N/A	No	N/A
Bangkok Bank	No	No	Yes	No	Yes
State Bank of Mongolia	Yes	No	No	No	No
Vnechtorgbank	No	No	No	No	No
Industrial Bank of Korea	N/A	N/A	N/A	No	N/A

Figure 9: Appraising of Asian Mobile Banking Security Assessment

5 Conclusion and Future Work

We intend to cover all banking apps throughout the world. WHile many operations can be automated, we however want to perform a final manual analysis systematically in order to get rid of any risk of misinterpretation and false positive. Other kind of apps will be analyzed as well (games, email clients, security tools...). Recent cases have shown that innocent-looking aps may be the vector of dirty motivation and operations [4]. We want also develop our tools further by using more advanced mathematics. The {Egide, Panoptes} reports should also made public once security issues will be corrected by banks. We will take a great care to verify that banks not only correct the existing vulnerabilities, if any, but also whether the users' privacy issues have been solved as well.

The technology of mobile (Banking) apps are far from being totally clean and mature. Beyond a few cases of vulnerabilities, users' privacy is not the priority of developpers or outsourcers (here banks)! We have also observed significant differences of awareness and security vision between Asia and Western world.

This study demonstrates that there is a strong need for pressure on app developpers to take care of users' privacy but also on banks. As consumers but also as citizens we have to use this pressure to force the different IT actors to respect our security and privacy. The bank apps market is not mature and has developped too quickly. Functionalities take precedence over security and users' fundamental rights for privacy and data confidentiality.

Another point which is worth mentioning is the difficulty to identify a visible contact point to report security issues. This is the reason why now we have decided to let banks to contact us. But this issue also holds for any other kind of applications.

Finally, it is important to be aware that all the tested apps are on the Google Play as well. This means that Google does not perform apps' security analysis at all. It does not care about users' privacy either (but we all already know that). Google, as a world IT power has the power to force developers to do a better job.

References

[1] Android Malware Genome Project, `http://www.malgenomeproject.org/`

[2] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, Christian Platzer (2014). ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors, *Vienna University of Economics and Business* White Paper, `https://iseclab.org/papers/andrubis_badgers14.pdf`

[3] Ethem Alpaydin (2010). *Introduction to Machine Learning*, Second Edition, MIT Press.

[4] James Ball (2014). Angry Birds and 'leaky' phone apps targeted by NSA and GCHQ for user data. *The Guardian*, January 28th, 2014, `http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data`

[5] DAVFI Project Website, `http://www.davfi.fr/index_en.html`

[6] Contagio, `http://contagiodump.blogspot.fr/`

[7] The Drebin Dataset, `http://user.informatik.uni-goettingen.de/~darp/drebin/`

[8] Virus Share, `http://virusshare.com/`

[9] OpenDAVFI Project - Free and Open, New Generation Antvirus Engine for Android, Linux and Windows, `http://www.opendavfi.org`. The website will open soon.

[10] The Snoopy-ng Project, `http://research.sensepost.com/tools/footprinting/snoopy`

[11] Graham J. Williams and Simeon J. Simoff (2006). *Data Mining - Theory, Methodology, Techniques and Applications*, Springer Verlag.