# mPOS Android App Security

Arnaud Barisain Monrose  
Mobeewave Inc

Introduction 2

About security in applications 2

Differences and similarities between our mobile POS and a banking application 2

Android’s openness and it’s implication on security 3

Rooting 3

Self-signed applications installation and permissions 3

Configuration locking 3

State of the art of Android’s security 4

How to make Android more closed 4

Known security holes 6

Application tampering/interception and detection 6

Software 6

Input 6

Backend output (communication) 6

Internal logic 6

Hardware 7

Hardware security and trust 7

Secure Elements 8

Secure boot and chain of trust 8

TPM 9

Trusted Execution Environment 9

Access control 9

Detection of post-release breaches and reaction plan 11

Effort needed to hack the mobile mPos 11

Samsung’s example of TEE, Secureboot and Tamper detection implementation 11

Quick comparison of iOS, BlackBerry, Windows Phone/RT and Android 12

Conclusion 13

References 13

# Introduction

This paper presents:

* The state of the art of Android’s security
* How the mobile POS (mPOS) application can be secured
* The risks that are taken and how to minimize them
* The Embedded Secure Element’s (eSE) role in security
* How to react to vulnerabilities
* The effort needed to “hack” the solution.

Since Android is a fast moving platform, some parts of this paper may become outdated with new versions of it.

# About security in applications

Applications that handle sensitive data on end-user devices have different challenges to overcome than ones running on controlled servers. They run alongside other apps that can be hostile and compromise our application.

Since the devices applications are running on are not directly under our control and monitoring, we need to be able to trust the system, or at least part of it.

Multiple solutions exist, both software and hardware based.

From secure elements to secure boot, the goal is to limit as much as possible the impact of any discovered vulnerability. Their implementation, both in theory and as already implemented by manufacturers, will be detailed in this document.

# Differences and similarities between our mobile POS and a banking application

Banking applications are already available for smartphones. They also handle sensitive data, but do not have the same security requirements as a mobile POS.

If they are victims of a hack, only their data will be compromised.

To prevent this, some banks implement two-factor authentication with a device similar to RSA SecurID, or simply a card with codes on them. The client would be required to input the second code, something the hacker would not be able to easily access.

Wallets share more security concerns and solutions with an mPOS. Google Wallet and ISIS rely on a SE to operate. They also delegate as much logic as they can: Credit Card information is stored on secure remote servers, the SE is responsible for communicating with the POS and they can be remotely disabled.

Some risks are there (keylogging the initial credit card input, losing your device); but they are deemed acceptable.

A mobile POS is different in multiple ways.

If it is compromised, for example, hundreds of credit cards can be stolen without both parties knowing or some parts of the transaction data could be tampered.

The difference with a consumer-owned device is that the supplied hardware and its Operating System are under our control.

This allows us to rely on hardware security (TSM, TEE, and SEs like other banking applications), and tweak the OS for limiting attack vectors: code signature, adding device-wide remote deactivation, remove the ability for the user to install applications or debug the ones running.

# Android’s openness and it’s implication on security

## Rooting

« Rooting » an Android device consists in gaining root privileges. They are the highest you can get on Linux-based systems. Being root allows you to remount the system partition as read-write, write on it (allowing you to patch the kernel), read/write RAM, read/write applications’ private sandboxes, etc …

Since the Android permission scheme is based on UNIX groups, any application with root privileges can access every permission. It also can modify the permissions protection level, which we’ll talk about later in this document.

The recently released Android 4.3 fights rooting by disallowing Zygote-spawned processes (Every Android Java application is a Zygote fork) from executing suid binaries. This doesn’t mean that they can’t become root at all (kernel exploits can bypass it), but it makes it harder. For example, they cannot execute /bin/su.

Rooting has a disastrous effect on security. Therefore, preventing it must not be overlooked, even if it’s not the only way to defeat a device’s security.

## Self-signed applications installation and permissions

Even in an unrooted state, because of its open nature, Android allows users to install self-signed applications (as opposed to Windows Phone/iOS).

It uses a permission system, backed by unix groups as said earlier, to define what applications can and can not do. Applications ask for permissions, and the user has to allow them before the system installs the application (except if they are installed over ADB (android debug bridge)).

Android permissions have multiple protection levels: normal, dangerous, signature, signatureOrSystem.

The last two are interesting because it only grants permissions based on the application’s signature (Google Wallet uses this to access the NFC and SE), and the last one also grants the permission if the app is installed in the system partition (read-only on unrooted devices).

An improvement would be to restrict many permissions to Mobeewave’s signature.

## Configuration locking

Android has a built-in policy enforcement mechanism. It’s called the “Device Administration API” and is currently used by some applications like the Exchange email client.

Its scope is limited, but it can force the user to have a passcode (with complexity requirement), force device encryption, and set the idle auto-lock time.

The application can also request a device lockdown, data wipe or passcode reset.

The user can disable the policy, but the application will be notified, allowing it to revoke access to its data until the user enables the policy again.

This is not enough for an mPOS, where we need the configuration to be locked and modifiable only by administrators. For our needs, we can modify the Settings application and add passcode verification to it.

Simply removing it would make device maintenance too hard and inflexible.

# State of the art of Android’s security

## How to make Android more closed

In order to secure our mPOS as much as possible, we need to close Android’s open doors. This section includes some details of the implementation of these security measures. A proof of concept patchset is done separately.

All of these protections might seem worthless, overkill or easily avoidable, but it’s their combination that matter. The goal is to have redundancy, so that if one security is flawed and defeated, others will limit the scope of the exploit

Rooting must be as hard to achieve as possible. Until the boot process is secure, rooting the device will allow the attacker to disable other security measures. As said before, the ground work for rooting prevention has been done by Google in Android 4.3 with the inclusion of SELinux and suid binaries execution disabled.

In order to reduce the exploitability of potential security holes, we must make third party code execution impossible.

There are three ways to install packages on retail Android phones:

* Google Play Store
* ADB
* Manual installation on the phone from a downloaded APK

Our software will not come with the Play Store, and ADB disabling will be covered later.

The only thing left to disable is the installation of non-mobeewave signed applications. This can be achieved by either patching the PackageManager so it does not install applications that do not conform to our requirements, or disable the “Allow installation of software from untrusted sources” setting.

The PackageInstallerActivity in the PackageManager enforces this setting. While easier to patch and less chances to break the package manager, it’s also easier to work around, since ADB will still be able to install applications.

Adding a watchdog, monitoring the /data/app folder and checking for unauthorized applications might be a useful addition.

If an application manages to be installed, we can prevent it from doing bad things, like keylogging, by removing it’s ability to obtain the required permissions.

As said before, Android allows restricting permission access. We can protect dangerous permissions, like “SYSTEM\_ALERT\_WINDOW” (which allows any window to stay on top of another one) by setting their protectionLevel to “signatureOrSystem” in “frameworks/base/core/res/AndroidManifest.xml”.

This will disallow applications installed in the /data partition from acquiring the dangerous permissions, until they are signed by Mobeewave’s key. It allows us to protect the OS while keeping the ability to update our own applications, without updating the whole system.

Now, we must ensure that ADB is disabled. Since it is by default, and can only be enabled in the settings, it will be protected by the Settings passcode.

Still, if we want to make sure that ADB is not running as root and disabled by default, we need to edit /system/build.prop and ensure that the following values are set :

“ro.secure=0

ro.kernel.qemu=0

persist.service.adb.enable=0”

The first setting tells ADB not to run as root. The second one too, because if it is set, Android will consider that it is running in an emulator and will grant administrative privileges.

“ro” values are read only, which means that after they’ve been set once they will not be modifiable. Since /system/build.prop is read-only, it cannot be edited unless you gain root access.

The last setting simply disables ADB, since we do not want it even in non-privileged mode.

We don’t want to totally disallow ADB, since we might need it for debugging. It will already be protected by our security pin code, and in a hidden developer menu, so it can’t be enabled by mistake.

There is room for one improvement. Since Android 4.2.2, you must authorize a computer on the unlocked device before ADB allows any action. We can modify this message and add an authentication process, only known by Mobeewave.

Another improvement to the kernel would be to restrict the scope of what can be done, even using root. For example, HTC devices will not persist any changes made to /system. It can still be defeated, but it could be worth looking into similar solutions.

The last step is to secure the bootloader and the recovery.

For the recovery, we just need to use the stock recovery, which only can install signed updates. The booloader locking will be done in collaboration with the device manufacturer.

## Known security holes

Android, like all complex systems, has known security holes. Since they will be the first tested against our system, it is important to know them in order to test if our system is still vulnerable or not.

In the past, they have been multiple exploits (Android 4.0.x log symlink vulnerability, Gingerbreak) that allow rooting, but they will not be described in this document, as their explanation would be too long.

# Application tampering/interception and detection

## Software

### Input

Android applications can’t intercept touch events easily. Hooking into the event subsystem would require rooting and patching.

The screen can be partially obscured by another application. An example would be replacing the PIN code input and intercepting it, telling the user that the PIN is wrong and hide the malicious view, allowing the user to finish his purchase, not noticing that his PIN has been stolen.

Overlays can’t forward touch events to the obscured view or even emulate them without rooting, so they will not be able to control the mPOS directly.

### Backend output (communication)

Communications can be easily logged if the attacker controls the network on which the device is connected, so we should trust them the least.

The mPOS app only accepts secure connections to the server, and enforces the use of Mobeewave’s certificates. This makes sniffing impossible, even with a man in the middle attack, since it is impossible to sign the data with Mobeewave’s keys.

If the application is tampered, it can accept other certificates, or even log the communications itself. This means that the attacker can log purchase data (cart, amount, name, etc …), but that can already be done by the tampered app, so it is not a concern.

Credit Card information is pre-encrypted by the reader/secure element, and is unreadable by anybody except Mobeewave’s server, making it secure to process and transmit over untrusted software and networks.

### Internal logic

Once the device has been modified to run unauthorized code, the mPOS application can be tampered. Android apps, like any other Java application, can easily be decompiled and repackaged with modified code.

Internal logic modification can lead to card data/PIN logging, modification of the displayed transaction amount/cart items, etc …

Since the data has been modified, the package will need to be resigned with another key than ours, allowing us to detect a tampered application.

One problem is that if the system is already compromised enough to accept the replacement of our app with one with a different signature, any other signature check can also be patched.

## Hardware

All of these modifications are software, which means that if we manage to keep the chain of trust strong, we have a good chance of preventing them from being exploited, or at least making it really hard to.

Hardware modifications, on the other side, can be undetectable and alter what happens in the device.

For example, the NFC antenna and the SE input/output could be spied, the RAM frozen then read elsewhere …

Since hardware tampering is hard to protect against, we need to make it hard to do and easy for us to detect. The phone could, for example, have its motherboard protected by a layer of resin that prevents modifications.

Detection would be monitoring the hardware for abnormal changes (high temperature, opened case, …) and react accordingly. For example, some hardware will shut itself off, or even destroy itself if attacked. This can be a solution for us, even if we still can’t be sure that we can detect every hardware change.

# Hardware security and trust

Since software based security is regularly defeated, especially in complex systems, hardware based solutions have been researched.

The idea is to delegate the handling of sensitive confidential data and its encryption to a tamper-resistant secure chip. It can accept input directly from the user or other hardware components (like a NFC antenna), or simply compute data using data sent by the complex system. That way, if used to do so, it can prevent data from being exposed to the normal OS, ensuring it only leaves the secure chip in an encrypted form.

Since the chip has a limited function set, the attack surface is greatly reduced compared to rich and complex operating systems, like Linux, and allows us to ensure the safety of our data even if the normal OS has been compromised.

The internal logic of the applications running on the chip can be made non upgradable or upgraded only with cryptographically signed programs, we can trust the code running on it much more than anything running in the complex OS.

## Secure Elements

They come in three forms: UICC (Universal Integrated Circuit Card), embedded SE and microSD.

A very widespread secure element is the SIM card (which is an UICC).

It’s currently used (and has been for years) to authenticate mobile phone users on the GSM/UMTS/LTE networks. These chips have a microcontroller with ROM (containing the operating system), RAM and EEPROM, allowing them to run small upgradable applets.

The inclusion of an Embedded Secure Element (eSE) can be used to enhance the security of the application even more by delegating it some responsibilities. We can trust an eSE due to its isolation, embedded software and limited feature set. Compared to UICCs, it has the advantage of being hard to remove or to swap.

This eSE solution is already used by Google Wallet and ISIS. Payment applications are stored in an eSE (running JavaCard OS).

If a secure element can be used by our mPOS, it would greatly increase its security.

The ideal case would be to be able to take advantage of a secure element which can communicate with the CLF modem directly over a secure channel.

This would allow us to move the complete EMV Level 2 logic to the SE. It would then only send part of the credit card data to the application unencrypted (like, for example, the last four digits of the card), and then give the full information encrypted, and only readable by Mobeewave’s servers.

This will not fully protect the mPOS app if it’s compromised (especially from PIN entry interception), but no credit card information will be leaked, giving us a minimum amount a security on an untrusted platform.

By making the secure element required, we can also prevent the application from being used on unauthorized devices or compromised ones using techniques similar to the ones Samsung uses for KNOX with the TrustZone (described later in this document).

## Secure boot and chain of trust

A solution to trust the software is to introduce the concept of a “Chain Of Trust”, which the secure boot process implements.

This is what Apple does with iOS and Microsoft with Windows RT.

When an iOS device starts, it executes code stored on a trusted ROM chip. This code, using a cryptographic digital signature check, ensures that the bootloader was not tempered with. If the signature is valid, the next step is executed. Each step verifies and runs the next one.

Windows RT devices work on a similar principle: UEFI checks if the bootloader integrity can be verified using one of the public keys in its list. If it can, the next steps are the same than on iOS.

Microsoft does make the distinction between Secure Boot and Trusted Boot. Trusted Boot is anything after the bootloader has been loaded (kernel, drivers and system file checks), and Secure Boot is when the UEFI checks if the bootloader is trusted.

Secure Boot requires a TPM, while Trusted Boot doesn’t. Obviously, the security is weakened when Secure Boot is not used, since a patched bootloader will not guarantee the integrity of the trusted boot security.

One problem with this solution is that it is well known that a chain is only as strong as its weakest link is. Breaking that chain in iOS is called “Jailbreaking”. It shows that such a system has already been defeated in the past.

ROM exploits have been found. Since it is the first step in the chain of trust (and not writable) and implicitly trusted, devices affected by this exploit are not patchable with a software update.

## TPM

A TPM is a small chip placed on the motherboard with very specific tasks.

It can store secure data (typically certificates and keys), provide cryptographic functions to the system and do a system health check.

It is used by Microsoft’s BitLocker for full drive encryption (where the TPM holds the drive key and will only release it when the system has been verified), or by Windows RT tablets for Secure Boot (where it holds the root certificate used to check the authenticity of the boot loader).

The TPM can also be used to compute a hash of the system configuration and sign it. A remote server can then check if the system is healthy without trusting it to do the health check, which can be easily patched by rootkits.

## Trusted Execution Environment

A TEE (Trusted Execution Environment) is a secure area of the device. It introduces the notion of a “Secure World” and “Normal World”.

ARM’s TrustZone is a TEE implementation. Since mPOS devices are likely to run on ARM devices and since ARM has donated its TrustZone API to GlobalPlatform (leading to the TEE Client API), we are going to talk specifically about it.

The device’s resources (such as the CPU and the RAM) are basically partitioned into two parts (secure and normal) by the TrustZone, acting like they are two separate chips. They can communicate via a secure bus and APIs.

The goal of the TrustZone is to have as little as possible running in its limited OS so that the attack vector is reduced. It exposes functions to the normal world, for which it guarantees security. It is similar to a secure element in that way, but it does not require additional hardware costs.

The secure world can simply process some data, but it can also accept user input (A touchscreen PIN input for example).

It is interesting to note that the TEE allows the device to have a software TPM/SE implementation.

# Access control

In order to prevent the use of stolen or lost devices, Access Control mechanisms can be used.

There are multiple ways already implemented in Android, and other ones implemented in other access control systems that we could build into our solution.

1. PIN

* Simple multiple digit code (typically 4).
* Pros:
  + Easy to remember
  + Can be used as fallback for more complicated access control systems
* Cons:
  + Easy to brute force if repeated attempts are not delayed
  + Can easily be stolen (seeing it over a shoulder, look at the fingerprints on the screen, overhearing it during a conversation, etc.)

1. Passphrase

* Just like the password fields used on almost every login page. The user defines a phrase for unlocking his phone. A certain complexity can be required.
* Pros:
  + Like a PIN, can be used as a fallback
  + More secure than a PIN (more complex, cannot be read using fingerprints)
* Cons:
  + Tedious to use, especially on a small screen
  + Can be forgotten

1. Face unlock

* Uses the camera of the device to unlock the device.
* Pros:
  + Nothing to remember or enter
* Cons:
  + Slow
  + Heavily dependent on lightning conditions
  + Can be spoofed with a simple photo

1. Fingerprint

* Scans the user’s fingerprint.
* Pros:
  + High level of security
  + Fast and easy to use
  + Already supported by some Android device makers
* Cons:
  + A fingerprint can be cloned without the user knowing, but it’s difficult and expensive
  + May require multiple attempts to work

1. Badge

* Swipable/tappable badges (like the HID ones) are given to each user
* Pros:
  + Fast and easy to use
  + Widespread (cheap to buy)
  + Acceptable level of security
* Cons:
  + Clonable using hardware easily buyable on websites like eBay
  + Easy to steal at the same time as the device

Devices should be also remotely administrated. Android allows applications to become device administrators, which gives them the ability to change the access control method and remotely wipe the device.

This feature is already used by the Android Device Manager, which allows any user to remotely locate or wipe their phone, and by Exchange e-mail, allowing administrators to enforce a minimal PIN strength and also remotely wipe the devices.

Mobeewave needs a system like this in order to lock and wipe compromised or lost devices. This could easily be enforced by the mPOS app itself and would not rely on another firmware modification.

# Detection of post-release breaches and reaction plan

Since security breaches cannot be fully prevented and will eventually happen, Mobeewave needs to be able to limit the scale of the problem.

This is why the mPOS should be able to update itself over the air, since the patches need to be deployed to the devices as soon as possible after testing.

Android has a built-in OTA (over-the-air) update system that Google uses for the Nexus and Google Play Edition devices. Custom firmwares implement their own updaters, which act just like Google’s with automated installation, requiring no other user action than pressing “Install”.

Minimal user interaction should be required: otherwise they might be tempted to skip the update. Some updates may also be made mandatory for connecting to the middleware and activation of new devices running outdated software denied.

# Effort needed to hack the mPOS

In order to measure the effort needed to hack the mPOS, we will consider the best case scenario: The app runs on a trusted OS which uses Secure Boot and a TrustZone program continuously checking the health of the system. Android is configured not to accept app sideloading and permissions are restricted to Mobeewave’s signature.

An eSE is present and handles all of the EMV Level 2 logic.

If the attacker wants to compromise the mPOS app, he will need to:

* Find a way to install and run arbitrary code on the device
* Escape the application’s sandbox (typically gaining root privileges and work around SELinux)
* Get the mPOS app, decompile it and find what to patch, write the patch and recompile it.
* Patch the operating system so that it allows the installation of a patched mPOS app
* Execute the mPOS

All of this without the secure world program not noticing anything wrong with the system’s health.

This is non-trivial work and requires a combination of multiple security flaws.

Even if the attacker did all of that, credit card information would still be safe since encrypted by the SE.

Since this relies on unknown (or undisclosed) exploits, the time and financial investment that hacking represents is hard to quantify, but it is significantly higher than on an untrusted OS and hardware.

Of course, the SE could be hacked, but this would require much more work due to the nature of the SE and its limited operating system.

Another attack would be to sniff the secure connection between the CLF and the SE, but this requires tools and can be really hard if the motherboard is protected against tampering. It would be easier to listen for the contactless traffic using an external sniffer, but then it’s not our app/device’s concern.

Even then, the impact is limited thanks to EMV (with technologies like iCVV).

# Samsung’s example of TEE, Secureboot and Tamper detection implementation

Samsung, starting with the Note 3 and Galaxy S4 has based their KNOX security solution on it. It is a great example

KNOX relies on Secureboot, a Chain Of Trust, SE Linux and ARM TrustZone.

The KNOX container is an environment isolated from other Android apps on the phone. They cannot communicate using IPC (Inter-Process communication), via Android APIs or using the filesystem. Only apps certified and signed by Samsung can be ran in it.

It is implemented using SELinux containers. The problem with them is that they assume that the kernel is trustable and not compromised.

That’s where the program running in the TrustZone secure world is used.

As said in the TPM section, the TrustZone program acts like a virtual TPM.

When creating a KNOX container, the server sends a challenge to the device. This challenge is passed to the secure world program, which computes an answer after checking some system health status (such as hashes of the kernel’s pages, which must correspond with the precomputed ones). If the system is not compromised, the KNOX container will be created.

To fight future breaches, the secure world program periodically checks the system’s health using various methods. If a compromised system is detected, it will react accordingly to how it was configured, allowing the securisation of the device, even post-attack.

The program also ensures that the bootloaders (multiple ones are used on a Samsung device) are all signed by Samsung. If a non Samsung bootloader is detected, an (speculated) e-fuse will be written , and the device will display “KNOX Warranty Void”. This cannot be reverted, and can prevent the device from using KNOX containers.

# Quick comparison of iOS, BlackBerry, Windows Phone/RT and Android

Compared to Android and Blackberry 10, which are open platforms, the competition has closed their mobile OSes and only distributes applications through their stores.

Obviously, they let developers install their applications on their devices without having to pass the store validation.

* Blackberry lets you manually install applications to any phone (BB10 devices need to be in developer mode), but you must sign it using the debug key that is provided to you by Blackberry after you register on their portal.
* iOS Devices accept applications signed by developers only if they have been provisioned (their UUID must be entered into the Apple Dev Center, and a file must be downloaded and installed on the phone). The debug key lifetime is limited (it lasts for some months), requiring the app to be re-signed after it expires. They can be installed from Safari (over the air) or using Xcode.

An enterprise distribution certificate allows anybody in it’s possession to install any app they want on any iDevice without Apple’s consent. It is made for app deployment outside of the app store, and requires to be enrolled in the enterprise distribution program.

* Windows Phone devices can be unlocked if you have a Developer account. After unlocking, the device will accept a maximum of 2 non-store applications for an unlimited time. Any app can be installed, and there is no signature to take care of.

Regarding firmware modifications, many Android devices are sold with an unlocked bootloader, allowing users to flash custom firmwares without having to unlock it themselves by exploiting a vulnerability. Usually, cell phones subsidized by carriers are locked.

iOS, Blackberry and Windows Phones/RT devices always come locked. Vulnerabilities are required to let it boot kernels signed with another key than the manufacturer’s. They rely on the “chain of trust” and secure boot mechanisms described earlier.

# Conclusion

This paper exposes the necessity to take extra caution and secure the environment around the application as much as possible. This requires collaboration of the system up to the Boot ROM level.

Hardware security, when handling sensitive data like we do, can and should be leveraged, especially on untrusted and untrustable environments.

Naturally, every solution presented here (even hardware based) is not unbreakable and might end up being hacked one day or another by somebody with enough incentive, knowledge and resources.

The goal is to make this hack as hard and expensive as we possibly can in order to discourage it: if our platform is hard to break into, hackers will be more tempted look into easier and cheaper ways to achieve their goal.

We also need to react fast in what could be described as a “cat and mouse game”, just like the one Apple plays with jailbreakers.

# References

1. Apple, “iOS Security”, 2012 <http://images.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf>
2. Google, “Android Security Overview” http://source.android.com/devices/tech/security/index.html
3. Trusted Computing Group website <http://www.trustedcomputinggroup.org>
4. GlobalPlatform, “GlobalPlatform made simple guide: Secure Element” http://www.globalplatform.org/mediaguideSE.asp
5. SANS Analyst Program, “Implementing Hardware Roots of Trust : The Trusted Platform Module Comes of Age” http://www.sans.org/reading-room/analysts-program/hardware-trust
6. ARM, “Securing the System with TrustZone Ready Program”, <http://www.arm.com/files/pdf/Tech_seminar_TrustZone_v7_PUBLIC.pdf>
7. Hervé Sibert, “Le TEE, nouvelle ligne de defense dans les mobiles”, https://www.sstic.org/media/SSTIC2013/SSTIC-actes/conf\_invit1\_j3\_2013/SSTIC2013-Slides-conf\_invit1\_j3\_2013-sibert.pdf
8. Samsung, “Samsung KNOX Whitepaper”, 2013 <http://www.samsung.com/global/business/business-images/resource/white-paper/2013/06/Samsung_KNOX_whitepaper_June-0.pdf>
9. Samsung, “KNOX Technical Details”, 2013 https://www.samsungknox.com/overview/technical-details