USB connection vulnerabilities on Android smartphones: default and vendors' customizations

André Pereira[1], Manuel Correia[1], Pedro Brandão[2]

[1] Center for Research in Advanced Computing Systems (CRACS-INESC LA);
[2] Instituto de Telecomunicações, FCUP/UP;
Portugal

{apereira, mcc, pbrandao}@dcc.fc.up.pt

Abstract. We expose an USB vulnerability in some vendors' customization of the android system, where the serial AT commands processed by the cellular modem are extended to allow other functionalities. We target that vulnerability for the specific vendor system and present a proof of concept of the attack in a realistic scenario environment. For this we use an apparently inoffensive smartphone charging station like the one that is now common at public places like airports. We unveil the implications of such vulnerability that culminate in flashing a compromised boot partition, root access, enable adb and install a surveillance application that is impossible to uninstall without re-flashing the android boot partition. All these attacks are done without user consent or knowledge on the attacked mobile phone.

Keywords: Android, Security, USB vulnerability, privileges escalation, vendor vulnerabilities

1 Introduction

Nowadays the extended features that smartphones possess are crucial to explaining its success, we see a yearly increase of its market share over traditional phones. The extra features like phone banking, e-mail, GPS, together with the traditional features of phone calling and SMS make the smartphone essential to our daily lives and ease our existence in this day and age. These benefits lead us to expose our personal data increasingly more, as such, the security associated with these systems is essential.

The Android system composes 80% [1] of the worldwide market share, making it a big player on the smartphone business. Since it is an open source Operating System (OS), the research of vulnerabilities on the system is in the best interest of the community.

Vendor customization is one of the advantages of the Android ecosystem, but this is a double-edged sword, since it can introduce security breaches. Attackers could exploit different attack vectors on many of the different ROMs, as vendors add software, such as applications and system owned processes, for dealing with things like USB pairing. According to a recent study [2], vendor customization accounts for 60%

of the vulnerabilities found in the Android ecosystem. We researched Samsung's Android customization and discovered the possibilities introduced to exploit the system.

In this paper we present a newly found vector to exploit the USB connection, more precisely to a vendor customization that extends the AT commands[1]. The system understands and lets these commands be sent by USB. We also describe a proof of concept of the attack and a scenario where this attack could be used.

In the proof of concept, we were able to effectively flash a compromised boot partition without the user consent. This enabled the three main objectives of the attack: gain root access, enable `adb`[2] and install a surveillance application that is impossible to uninstall without re-flashing the android boot partition.

2 Attack Scenario

The main purpose of the attack is to explore the vulnerabilities found on the Android OS, namely the vulnerabilities found in its USB connection. This mandates that the attacker must possess a physical USB connection linking the computer of the attacker to the victim's device.

A practical scenario would be the installation of a public kiosk for charging devices' batteries. However, the true purpose would be to inject malicious code into the devices. Unbeknownst to this malicious purpose, the victim would willingly place its device in a compromised situation, hoping that it would charge the phone's batteries.

Such scenario could be very fruitful, as we expect easy acceptance by the victim to place the phone in such a manner. There are a couple of reasons for this. First the lack of knowledge of the dangers of an exposed USB connection. As this is such an uncommon practice, even an IT experienced user could possibly lack this knowledge. The second reason is the emergency state in which the victim is in. Nowadays our cellphone is an extension of ourselves, it is completely implanted in our daily life and the lack of it is unthinkable. This is even truer for smartphones, since you can perform additional tasks on it, like phone banking and e-mails. So a situation where the cellphone battery is empty or almost empty, would easily lead the victim to expose its device to the charging kiosk.

Given the nature of such an attack, a script is necessary on the computer holding the other end of the USB cable. A script capable of accurately detecting the smartphone, match its vulnerabilities and proceed with the attack. For example we would execute a different type of attack for different Android versions, for different firmware versions, as well as different brands and different products of those brands. As an example, in the Samsung smartphone family, we could have an attack for the Galaxy S2 and another attack using different vulnerabilities found for the Galaxy S4.

---

1 Serial commands for modem configuration and management.

2 Android Debug Bridge [10]

3 Vulnerabilities

The following vulnerabilities are used in the proof of concept described in the next section. We will first elaborate on said weaknesses and then describe the overall attack. Some vulnerabilities are documented commands, like the standard AT commands and others were discovered in our work. AT commands by themselves are not the vulnerability, but the vendors' implementation of them make them so.

3.1 AT COMMANDS

The AT commands (ATC) define a command language that was initially developed to communicate with Hayes Smartmodem. Nowadays it stands as the standard language to communicate with some types of modems. For example, protocols like GSM and 3GPP use this type of commands as a standard way of communicating with the modem. With ability to issue these commands to the modem, we are able to [3]:

- Issue calls;
- Send SMSs;
- Obtain contacts stored inside the SIM card;
- Alter the PIN.

In order to understand this attack vector, we need to comprehend how a modern smartphone works. A modern smartphone is built with two OSs, running in two very different environments [4]. On one hand we have the AP, Application Processor, where the android OS and all the applications with which the user interacts run. On the other we have the Baseband/Cellular Processor (BP/CP), where the entire cellular (ex.: GSM) communication is done and where the modem lies. Issuing AT commands is not the only way to communicate with the modem, but it is the most popular way, together with RPC (Remote Procedural Calls).

The RIL, Radio Interface Layer [5], is the layer on the OS responsible for establishing a communication between the android OS and the modem. If a component in the application framework needs to send messages or make calls, it uses the RIL in the application framework to do so.

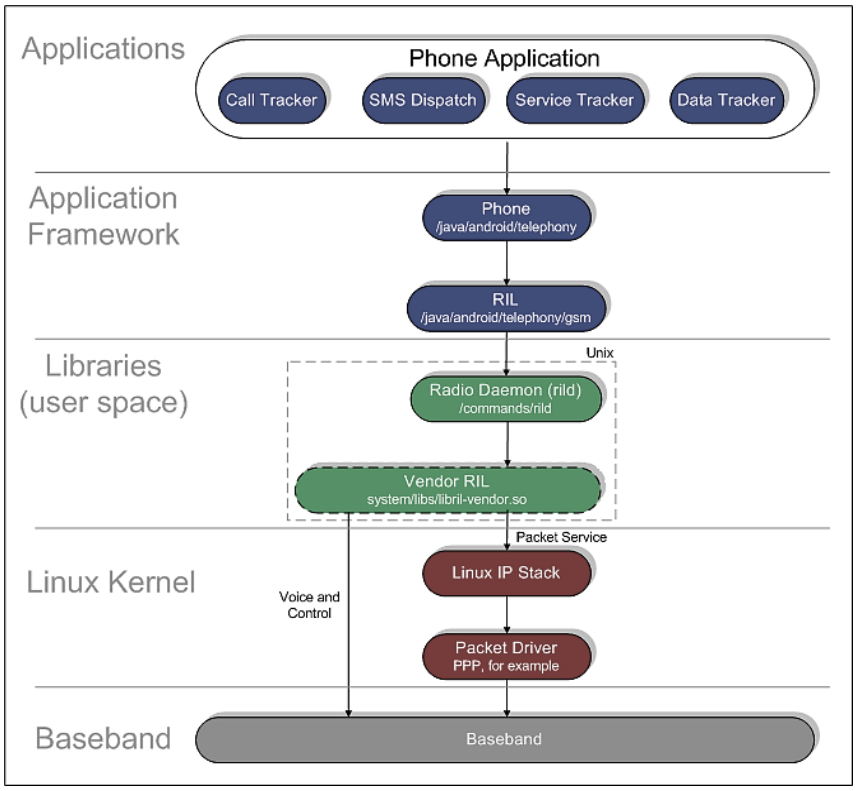

Fig. 1. Android Telephony system architecture from [6]

Fig. 1 details the communication stack, from the applications to the baseband. Where in the application framework, the RIL uses the Linux IP stack as a way of communicating with the baseband, establishing the communication channel.

In our scenario we can use the USB connection to issue such commands. This vulnerability is only made possible due to the fact that some Android smartphone manufacturers, like Samsung and HTC, enable this through the USB channel. This means that it is possible to send, using the USB connection, AT commands directly to the baseband. We stress the fact that this is not a default feature of the Android OS, but manufacture added.

In a practical scenario, upon the detection of a new USB device, the driver responsible for that device recognizes that the device has a modem type of interface through the USB, notifying the OS of such. Hence forward the OS has a way of communicating with the modem, which can be used in our attack scenario. We can thus send AT commands to make phone calls or send SMS to premium cost numbers. This way making the attacker earn money, or more accurately stealing it.

Complementing this, some manufacturers extend this list, adding some proprietary commands, with the purpose of enabling control and capabilities that they want to have on the system. These commands are private and manufacturers do not publish

them. But without the use of some form of encryption, anyone is able to eavesdrop the communication channel and try to understand, what lies under the system.

3.2 Vulnerabilities discovered and AT Samsung proprietary commands

In the case of Samsung, a vast list of its family of smartphones has this vulnerability, where it is possible to communicate with the modem through the USB channel, without any previous configuration on the smartphone. This is something that does not happen with `adb`.

Samsung extends the standard AT command set that comes with the 3GPP and GSM standards adding their proprietary commands. This extends the capabilities of interaction that their computer software (Kies) has on the device.

In fact Kies for Windows uses both the standard set and the extended proprietary AT command set, to at least achieve the following operations:

- Obtain the contact book;
- Obtain the files in the SD card;
- Update the firmware on the cellphone.

Through eavesdropping methods[3], on the USB channel and watching the android `logcat`[4], it was possible to discover the way in which the Kies software interacts with the device to accomplish those tasks.

As mentioned, the Kies software uses a proprietary set of AT commands. To achieve this, several processes inside the smartphone parse the commands received, to know if it is just an ordinary AT command or a proprietary Samsung command. According to the result, it issues or not the command to the modem. The smartphone must be a Samsung phone and contain a valid Samsung ROM. The ROM has a chain of processes that intercept the commands sent by USB and send them or not to the modem.

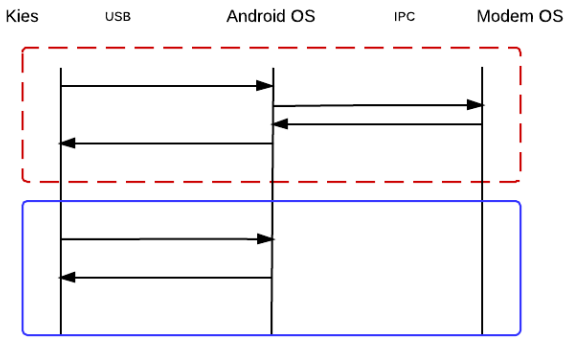

Fig. 2. Chain of communication when AT commands is sent through USB.

[3] http://www.sysnucleus.com/, USBTrace: USB Protocol Analyzer
[4] Android Logging System [19]

In **Fig. 2** we are able to see in the red dashed box a non-proprietary AT command that the OS intercepts and sends to the modem. In the blue box we have an example of a flow for a proprietary command that is not sent to the modem.

If we use a program to communicate using the serial port (eg.: Realterm[5]), we can send commands to the modem through a virtual COM type, instantiated by the driver. When we issue the proprietary command `AT+DEVCONINFO` and analyze the Androids' `logcat`, we can see that there is a process with a tag `dun_mgr` that is listening to the USB channel. Initially it receives the command and delegates it to the process with the AT tag. This process executes it and responds to the original process, which forwards the response through the USB channel.

dun_mgr	AT command: at+devconinfo
at	ATcmd:0 at+devconinfo 14
at	AT_ProcessCharParserDataInd at cmd sent state=4
dun_mgr	atx response: ...

Fig. 3. `Logcat` of issuing AT command

This led us to conclude that there is an interception by the OS, before deciding whether to send the command to the modem or to the Android OS.

Command "AT+DEVCONINFO".

The execution of the proprietary command `AT+DEVCONINFO`, in addition to obtaining all the information displayed in **Fig. 4**, also triggers the smartphone to mount the SD card on the computer as a mass storage device. When we eavesdropped the USB connection, we could confirm that this was one of the firsts commands sent to the phone by the Kies software. This assists the program in gathering all the necessary information in order to work properly. As it can be seen in **Fig. 4**, the information includes the smartphones' firmware version, the phone model and IMEI.

Fig. 4. Response message from issuing `AT+DEVCONINFO`

With this functionality, it is possible to make an attack to steal all the information inside the SD card.

Command "AT+FUS?".

[5] Realterm is a terminal program specially designed for capturing, controlling and debugging binary and other data streams. [20]

With the execution of `AT+FUS?` the smartphone is placed in download mode, which is the mode where all Samsung smartphones need to be in order to flash their internal partitions and thus update the smartphones' firmware. Normally putting the phone in download mode involves mechanical key pressing, i.e., input from the user, which implies acceptance to flash the phone. This is not the case when it is triggered by the AT command, no user intervention is needed, which enables an automation of this process. The discovery of this command, led us to conclude that this is the way that the Kies software uses to update the phone firmware. Download mode is similar to `fastboot` mode that you find in other Android smartphones, but in download mode we cannot use the `fastboot` tool included inside the android SDK, only Odin [7] and Heimdall [8] serve that purpose. They are both popular flashing software for Samsung devices. Odin is closed-source free flashing tool for windows and Heimdall is a cross-platform open-source tool suite.

This is the most damaging of the vulnerabilities, since we can alter the partitions inside the phone, explicitly injecting code on the partition that we want. The novelty of this attack, like the `AT+DEVCONINFO` command, is that no prior configuration is needed on the phone. We can do this, just right after the phone is bought. Placing the phone in download mode enables us to use Odin to flash the phone with malicious partitions.

3.3 ADB enabled

It is important to mention `adb`, the android debug bridge, since later we use it to gain further capabilities on the system.

The `adb` serves as a means of communication between a computer (host) and a smartphone (device). The communication is done via USB, but it is also possible to configure the device so that the connection is made through WiFi.

An USB connection and an `adb` enabled Android, could pose a serious security threat to the smartphone, so serious that since Android version 4.2.2 [9] Google made a security enhancement to the `adb` connection. Making sure that every USB connection has an accepted RSA key pair with the host computer the android is connected to. So every new USB host the android smartphone tries to connected, needs to be previously accepted by the user.

With `adb` enabled we can [10]: **a)** get shell access to the device; **b)** install applications that were not verified by the Google app store bouncer[6]; **c)** uninstall applications; **d)** mount the SD card; **e)** read the contents of `logcat`; and **f)** start an application.

Shell access through `adb` could also unveil new attack vectors has shown in [11], were it is possible to gain privileged access, with rooting techniques like Super One-Click root [12] and also Cyndia impactor [13].

In fact Kyle Osborn presented in Derbycon 2012, a shell script suite[7] that uses `adb` to injected several rootkit malwares and tools, to help in the extraction of the screen

[6] Dynamic heuristic malware detection system, for the Google app store.

[7] https://github.com/kosborn/p2p-adb: p2p-adb Framework

pattern, user information and other data. Prior to that, in Defcon 2011, Joseph Mlodzianowski and Robert Rowley built a public charging kiosk, to raise awareness about the dangers with USB connections. The users would plug the device, and the kiosk would prompt the device id of the user, with no other malicious intent.

4 Anatomy of the attack (Script)

As mentioned in section 3.3.2 we developed a script to automate the attack process in our proof of concept development. We will describe it in this section.

4.1 Architecture

The script has to be fast, fully automated, effective and able to perform operations on numerous levels of the OS stack.

We had the need to make use of the functionalities of two different OSs, Windows and Linux. For that we deployed the script in a guest virtual machine containing Xubuntu that is able to communicate with its Windows7 host machine. We use a Xubuntu virtual machine so that the script can take advantage of the Linux OS scripting environment. Linux comes with libusb[8] as a generic driver to handle USB devices, which in turn is used by the usbutils, making it more practical for scripts like this to be developed. We will detail its functionalities further down.

As virtualization software we used Virtual Box[9]. This program enables us to create guest virtual machines and at running time exchange the control of the USB device from guest to host and vice-versa. In order to give the guest control over the USB device, it presents a virtual USB controller to the guest, and as soon as the guest starts using the USB virtual device, the USB device will appear as unavailable to the host. So only one OS has access to, and thus can control, the USB connection at a time.

It is essential that the guest machine is Linux and the host Windows and not the other way around. This guarantees that the Samsung device drivers have direct access to the USB device, which would not work inside a guest machine, because the devices are emulated by Virtual Box. The type of attacks done by the guest machine cope with this emulation process.

We have a host with Windows 7 so that we can have access to Odin that is used to flash Samsung devices. This tool is unable to work on Linux, because of the dependence that it has on the Windows' Samsung drivers. Other tools are able to flash firmware on Samsung phones on Linux, like Heimdall, but it is only able to target a limited number of Samsung smartphones, whereas in the case of Odin we are able to target all Samsung smartphones.

A communication channel between the host and the guest is needed, in order that the guest, which is doing most of the work, can tell the host when and which smartphone to flash. For that we use a shared file between the guest and the host OS,

[8] http://www.libusb.org/

[9] https://www.virtualbox.org/

so when the guest needs to communicate it writes the file. The host is polling the file for changes. The roles are exchanged when the communication is in the opposite direction.

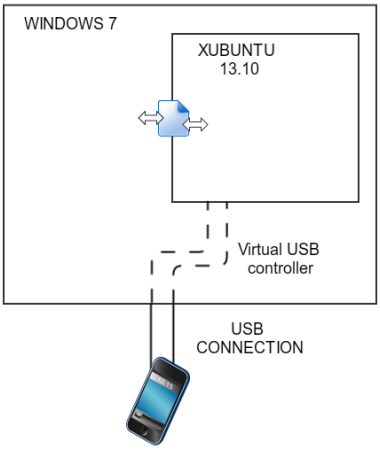

Fig. 5. Architecture of communication from Guest to Host.

In the guest machine, a full list of the USB devices that we want to attack is needed, in order to tell the guest machine which devices to filter, so that they can be controlled inside the guest and not on the host. This is done by identifying the product id and the vendor id, which together identify all USB devices. The vendor id identifies the brand, for example Sony, and the product id identifies the product, for example Xperia.

The script running on the Xubuntu (guest) is responsible for:

- Detecting plugged USB devices;
- Identifying the type of device;
- Identifying the vulnerabilities known for that device;
- Attacking using the known vulnerabilities;
- Communicating with the host, in case the vulnerabilities require the use of the Odin tool;
- Identifying the mounted external cards of USB devices.

The Windows 7 (host) is responsible for:

- Communicating with the guest, to know which device to flash;
- Identifying the flash image that matches the device and its firmware ;
- Identifying the correct version of Odin for flashing ;

It uses GUI automation libraries, namely Pywinauto [14], to control Odin without human intervention.

4.2 Using the vulnerabilities found

As we mentioned in the previous sub-section, the guest machine detects plugged devices, identifies them, matches them to the vulnerabilities found and executes the attacks that target those vulnerabilities. We will now cover the vulnerabilities and the attacks.

Device identification.

The purpose of this attack is firstly to identify the firmware version of the smartphone with the command `AT+DEVCONINFO`, as it was shown in **Fig. 4**. Enabling us to identify the firmware version of the smartphone in question. In **Fig. 4** it is identifiable by `VER (S5839iBULE2/ EUR_CSC /S5839iBULC1/ S5839iBULE2)` that shows the following details as per the format PDA CSC Modem Boot:

- **PDA**: The build version, which includes the system partition.
- **CSC**: (Country Sales Code): Language and country parameters.
- **Modem**: Identifying the modem firmware version.
- **Boot**: For the version of the Boot partition.

Changing the Boot Image.

As we described in 3.3.2 the AT command `AT+FUS?` places the phone in download mode and allows flashing a new boot partition.The primary functions of the boot partition are:

- Mount the primary partitions necessary for the system to boot;
- Start the first process of all OSs based on Linux, i.e. the `init` process;
- Read and execute the `init.rc` boot configuration file;
- Load the kernel and the ramdisk.

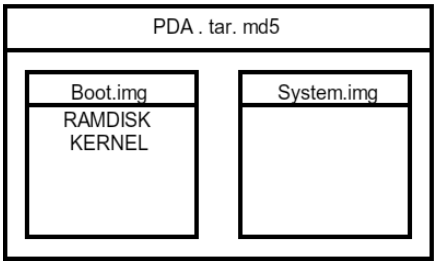

Fig. 6. Structure of a typical PDA file.

Fig. 6 illustrates a firmware file to be flashed on a device. The boot partition is placed in a `boot.img` type file, which is inside of the PDA file. This file in turn is a tar file with a checksum, usually a .tar.md5. Usually inside the PDA file there is also the System partition. In order to flash we only need the `boot.img` file. For our proof of concept we re-constructed a stock boot partition.

The bootloader [15] is very different from the boot image. The bootloader is the very first piece of software that gets executed on a device. One of its roles is to check the integrity and signatures of the partitions before it starts them. This prevents any modification of the partitions, such as the boot or the system, by an un-authorized third party. When a bootloader is unlocked, like in some Samsung devices', it does not check the signatures of the partitions, which means a further modification of the partitions is possible, thus enabling our attack.

Normally the `boot.img` file is divided in two parts, the Kernel and the ramdisk. The ramdisk is the first file system mounted on the root of the system. There we can find files like the `init` file and the `init.rc`, the image that is used in the booting process of the smartphone and various other folders needed for good performance of the system.

In the `init.rc` [11] file we can find several shell type instructions for the initial configuration of the system. Instructions that will be executed with root access by the `init` process. It is in this file that we will alter the ramdisk of the boot partition, adding malicious instructions.

In this scenario we want to show three different things that can be done by altering just the boot partition, first make the `adb` debug over USB option always enabled, second obtain root access, third install a surveillance application, in this case Androrat [12].

For the first objective, we have to alter the `on property:persist.service.adb.enable=0`. This system property tells the system, what operations to do when disabling `adb`. In the original file, when `adb` was disabled, we had that it would `stop adbd`, effectively stopping the `adb` daemon on the smartphone. We changed this from `stop adbd` to `start adbd`, rendering it impossible to disable the `adb` from configuration, even thought it might appear disabled in the systemss' options.

For the second objective, we want to obtain root access on the smartphone using the binary tool `su` [18]. The binary configured with permissions of execution for everyone, and with the owner as root, enables every user to have root access. We put the `su` file inside the ramdisk, being placed on the root of the system and then copying it to `/sytem/xbin/`. This is done adding these lines to the `init.rc` file.

- `COPY /SU /SYSTEM/XBIN/SU`
- `CHMOD 06755 /SYSTEM/XBIN/SU`
- `CHOWN ROOT /SYSTEM/XBIN/SU`

This will allow root access to any operation, for example when `adb` is enabled it will have root access.

For the third objective we want to install a surveillance application, in this case Androrat which stands for Android RAT (Remote Access Tool), in a way that the user does not know of its existence. First we place the application `.apk` file in the ramdisk directory, so that once it boots, it places the `.apk` in the root of the system,

similar to what we did for `su`. Then we add again to the `init.rc` script, the following code:

- `COPY /ANDRORAT.APK /SYSTEM/APPS/ANDRORAT.APK`

Once the phone boots, the application is installed as a system app. This makes the removal of the application extremely difficult for regular users even with root access. As described the application gets installed each time the phone boots.

The AndroRat application enables several remote surveillance capabilities on the phone, such as get contacts, get call logs, get all messages, GPS location.

AndroRat is composed of a client and a server, the client is the application that gets installed on the phone, the server runs on a PC, regardless if it is a Windows, Linux or Mac, since it runs in a java virtual machine. The client communicates with the server by TCP/IP. We altered the AndroidManifest.xml of the original application, deleting the following lines:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

This way making the icon is not visible in the app section of the phone, so an inexperienced user would not detect the malicious application.

Installing the new boot image.

For a successful attack we need to create a dictionary of previously altered boot partitions, to encompass all the different smartphones that we want to attack. For example, for the bundle version presented in `VER(S5839iBULE2/ EUR_CSC/ S5839iBULC1/ S5839iBULE2)`, we have to specifically map this bundle version to a boot previously altered that matches it.

After an initial identification of the smartphone version from the Linux guest machine, it is necessary that we place the smartphone in download and notify the host machine, handing over the control of the USB device to the host. When putting the device in download mode, its product Id and vendor Id gets altered, to a non-filtered combination of vendor Id and product id by Virtual Box, rendering the process of handing over the control of the USB device automatic. The guest machine saves the IMEI of the smartphone, so that once the phone reboots, it already knows that it is a compromised smartphone, which would enable the guest to do other types of attacks, since the `adb` is on and we have root access.

After the host is notified to flash the device with the firmware version, it maps the given version to a previously altered boot image that is ready to be flashed. It also maps the correct version of Odin.

Using a GUI automation tool for Windows, the host commands Odin to choose the correct image file folder and name and then to flash the phone. The smartphone proceeds with the flashing of the partitions and reboots normally. After rebooting, its product id and vendor id changes once more to the previous ones, handing over again the control of the USB connection to the guest machine, so that it can proceed with the attack.

First the Guest Linux does:

1. Detection of plugged USB devices.
2. Matching of its vulnerability.
3. Checks if it has a compromised boot partition.
4. Notifies the boot image to flash the device and saves the IMEI.
5. Places the phone in download mode.

Then the Host Windows does:

6. Identification of which file matches give version.
7. Makes use of GUI automation tools to control Odin and flash the phone

And again the guest Linux finishes with:

8. Proceeds with the rest of the attack, now that it possesses root access and `adb` enabled.

List of phones tested with the vulnerabilities found.
We successfully verified the attack on the following phones:

- Samsung GT-S5839i
- Samsung GT-I5500

And it was also possible to confirm that the attack was possible, by issuing AT commands that the fallowing phones had the vulnerability:

- Samsung GT-S7500
- Samsung GT-S5830
- Samsung I9100
- Samsung S7560M

It is necessary that the smartphone has an original ROM from Samsung. We expect that the span of vulnerable versions of Samsung smartphones be much more wide than this, since in our assumption, having the vulnerability or not is implicitly related with the ability that the smartphone has on communicating with Kies software. So as far as we now, most (if not all) Samsung smartphones are supported by Kies.

List of tested Anti-virus Apps.
AVG, **Avast** and **Virus Scanner** were the anti-virus chosen for testing. First we examined if any of them detected/prevented the attack, and later, after the attack if any of them detected malicious software on the phone.

The results are that none of them prevented the attack at first. After the attack, and after a scan had been performed, AVG detected that `androrat` was installed and informed the user that it could be malicious. However, upon trying to uninstall the

threat it states that it cannot. Nothing more has been detected by AVG, or by the other two anti-viruses. They did not detect alterations to the `init.rc` file, or that `su` binary was added when compared to the previous state.

5 Conclusion

We exposed a serious vulnerability on some vendor customization of the android OS. We described our proof-of-concept with which we were able to explore the implications of that vulnerability, such as gaining root access by flashing a compromised boot partition. As the extended functionalities are intended to be used by the computer application of the vendor to configure and manage the smartphone, they were developed knowingly and with the mentioned intent. In our view, implementation of such "features" should be at least disclosed to users, in order that they understand the risks of an exposed USB connection. Our future work will involve developing a smartphone application to warn and advise the user regarding these possibilities. Depending on the smartphone's root access it can also be possible to allow the charging by possible hibernating the process responsible for handling the AT commands. Another future investigation will be to confirm that the "features" are still present in the latest version of the Samsung Kies software and the latest version of Android.

6 Acknowledgments

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project *«FCOMP-01-0124-FEDER-037281»*

References

1. Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, According to IDC - prUS24442013, http://www.idc.com/getdoc.jsp?containerId=prUS24442013.
2. Wu, L., Grace, M., Zhou, Y., Wu, C., Jiang, X.: The impact of vendor customizations on android security. Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Secur. - CCS '13. 623–634 (2013).
3. Specification, T., Terminals, G.: AT command set for 3GPP User Equipment (UE) (3G TS 27.007 version 2.0.0). 4, 17–18 (1999).
4. Mulliner, C., Liebergeld, S., Lange, M., Seifert, J.-P.: Taming Mr Hayes: Mitigating signaling based attacks on smartphones. IEEE/IFIP Int. Conf. Dependable Syst. Networks (DSN 2012). 1–12 (2012).

5. Singh, A.J., Bhardwaj, A.: Android Internals and Telephony. Int. J. Emerg. Technol. Adv. Eng. 4, 51–59 (2014).
6. Module, H.: Android RIL Integration Guide- Huawei. (2014).
7. Odin 3.09 - Odin download with Samsung ROM Flashing Tool, http://odindownload.com/.
8. Heimdall | Glass Echidna, http://glassechidna.com.au/heimdall/.
9. Security Enhancements in Android 4.3 | Android Developers, http://source.android.com/devices/tech/security/enhancements43.html.
10. Android Debug Bridge | Android Developers, http://developer.android.com/tools/help/adb.html.
11. Vidas, T., Cylab, E.C.E., Votipka, D., Cylab, I.N.I., Christin, N.: All Your Droid Are Belong To Us : A Survey of Current Android Attacks. WOOT. (2011).
12. SuperOneClick Root v2.3.3, http://www.superoneclickdownload.com/.
13. Cydia Impactor, http://www.cydiaimpactor.com/.
14. pywinauto - Windows GUI automation using Python - Google Project Hosting, https://code.google.com/p/pywinauto/.
15. Hoog, A.: Android Forensics: Investigation, Analysis and Mobile Security for Google Android. Elsevier (2011).
16. Android Init Language Google Git, https://android.googlesource.com/platform/system/core/+/master/init/readme.txt.
17. VRT: Androrat - Android Remote Access Tool, http://vrt-blog.snort.org/2013/07/androrat-android-remote-access-tool.html.
18. Superuser, http://androidsu.com/superuser/.
19. Gargenta, M.: Learning Android. O'Reilly Media, Inc. (2011).
20. Terminal Software, http://realterm.sourceforge.net/.