Practical Attacks on Security and Privacy Through a Low-Cost Android Device

Greig Paul and James Irvine

University of Strathclyde Department of Electronic & Electrical Engineering
Glasgow, United Kingdom
Email: {greig.paul; j.m.irvine}@strath.ac.uk

Received 3 November 2015; Accepted 1 December 2015;
Publication 22 January 2016

Abstract

As adoption of smartphones and tablets increases, and budget device offerings become increasingly affordable, the vision of bringing universal connectivity to the developing world is becoming more and more viable. Nonetheless, it is important to consider the diverse use-cases for smartphones and tablets today, particularly where a user may only have access to a single connected device. In many regions, banking and other important services can be accessed from mobile connected devices, expanding the reach of these services.

This paper highlights the practical risks of one such low-cost computing device, highlighting the ease with which a very recent (manufactured September 2015) Android-based internet tablet, designed for the developing world, can be completely compromised by an attacker. The weaknesses identified allow an attacker to gain full root access and persistent malicious code execution capabilities. We consider the implications of these attacks, and the ease with which these attacks may be carried out, and highlight the difficulty in effectively mitigating these weaknesses as a user, even on a recently manufactured device.

Keywords: Security, Privacy, Android, Exploit, Physical Access.

Journal of Cyber Security, Vol. 4, 33–52.
doi: 10.13052/jcsm2245-1439.422

1 Introduction

The smartphone and tablet markets in developing regions are predicted to see significant growth in the coming years. Time magazine reported on the current low market penetration of smartphones in India, and the likelihood for rapid growth in sales of lower price-point handsets [1]. Cisco's Visual Networking Index predicts that by 2019, mobile data traffic usage in India will grow 13-fold from 2014, and that 51% of devices will be smart devices (i.e. smartphones and tablets) rather than feature-phones [2].

In Africa, there has been a widely documented rise in the use of mobile phones by individuals, as both a means of communication, as well as a means of access to services from governments and banks [3] [4, pp. 80–90]. The rise of mobile banking in Africa is also especially significant, as it has reached significant market penetration and day-to-day usage, with mobile payments in Africa exceeding those from both Europe and North America [5, 6].

There is therefore a motivation for technology-based attacks against users in these regions, leading towards crimes such as theft. This paper aims to explore some of the risks posed by consumer electronic devices targeted specifically towards developing markets. We highlight the risks posed to users as a result of some of the security weaknesses in these devices, and consider the challenges of providing secure computing environments in these situations.

Conventionally in the study of security, attacks involving compromise of a local device are generally neglected, on account of the adage that physical access to a device grants the attacker control over the device. Despite this, the threat model for consumer hardware is rapidly evolving, with Secure Boot in UEFI on desktop and laptop computers able to protect against malicious modifications to the computer's operating system during offline attack [7]. For this reason, we consider physical attacks from non-invasive attacks (i.e. not requiring physical disassembly of the product) to be highly relevant and significant to users of these devices, and shall illustrate the significance of these attacks in Section 4.

2 Overview of Android Security Model

The Android operating system is derived from an open source project, referred to as the Android Open Source Project (AOSP) [8]. Android is a customised userland, built on top of the Linux kernel, providing APIs for sandboxed applications to run, and access hardware in a standardised manner, irrespective

of the manufacturer of the device. This means that many devices run the Android operating system, from different manufacturers, and application software may run on any of these, using the Android platform's hardware abstraction to simplify the development of software.

Android devices, in contrast to more general purpose Linux-based computers, are designed around a security model whereby each user application runs within a separate Unix user account (UID). By controlling access to protected resources through a combination of a permissions layer provided by the Android APIs, and regular filesystem permissions, applications are sandboxed from each other, and are not normally able to access data from other applications.

This isolation offers a different security model from the desktop. The Android model assumes that an application may safely store data privately within its own dedicated area, protected by the filesystem access control permissions. This prevents other applications, or indeed the user themselves, from having direct access to these files.

Android devices, in common with most Linux-based embedded systems, feature a number of separate partitions, in order to separately hold the kernel image, ramdisk, and main operating system. There is additionally a partition to hold the user's data, which is able to be erased in the process of carrying out a factory reset. Ordinarily, no partition other than the user data partition should be mounted read-write — the operating system partition is, by default, mounted read-only, and other partitions are not generally mounted, since they do not contain regular filesystems.

In line with other Unix-based systems, root access is the highest level of privilege available to code running under the kernel. Code executed as the root user may load modules into the kernel, altering its behaviour, and generally access any resource provided by the kernel, with full access permissions.

2.1 Main Requirements

The Android compatibility definition document, as published by Google [9], is intended as a guide as to good practice when making a device which runs the Android operating system. In order to receive approval for the shipping of Google services on a device (which we note the device investigated in this paper did not include), it is necessary for these guidelines to be followed.

On Android devices, the root user account is not meant for normal use, and there is not meant to be a means for the end user (or applications) to

access it [9]. This is because the root user is inherently able to access the entire filesystem, thus bypassing the access controls of the Android security model, and this would expose protected application data to any code running as root.

Additionally, the Android compatibility definition now requires the use of SELinux in enforcing mode [9]. SELinux is a mandatory access control framework, designed to enforce system-wide security policies across a system. It is capable of constraining applications which run as the root user, and therefore offers some protection against the use of kernel root exploits and other privilege escalation attacks, since such an exploit will not bypass the constraints of the SELinux policies [10]. This naturally requires well-designed and implemented policies, since SELinux merely enforces those policies.

3 Overview of Platform Hardware

While the Android operating system is shipped on a wide variety of devices, at a wide variety of price points, we focus exclusively on low-cost, consumer hardware. The tablet in question we investigated was manufactured in September 2015, according to the information label on the box, shipped with Android 4.4 (KitKat), and retailed for 3499 Rupees in India [11] (approximately 50 USD). It was distributed to attendees of meeting 35 of the Wireless World Research Forum, as a conference gift holding the proceedings.

Android devices feature a bootloader, which initialises the hardware, loads the kernel from storage into memory, and executes the kernel, having set the kernel commandline parameters to specify where to locate the remainder of the operating system.

In addition to loading the kernel, the bootloader is also used to control boot modes, based upon the boot control block, which is implemented to allow a device to boot into a minimal recovery environment. This environment normally does not require the Android userland to be present to run, and ships with its own kernel image and ramdisk, from which the main operating system and kernel can be updated. This environment also carries out factory reset operations, erasing the user data partition. The bootloader also typically offers a means of reprogramming a device's internal memory, to allow for the recovery image to be upgraded, and to allow the device to be programmed as part of the manufacturing process.

4 Vulnerability Identification

In this section, the process through which vulnerabilities were identified on the device is discussed, as well as the capabilities of each exploit. These vulnerabilities are reported in the order they were originally identified during our research. Finally, following our exploration of these identified issues, we carried out a security scan of the underlying Android operating system (since our identified vulnerabilities mostly lie below the Android operating system), and highlight the risks identified there.

4.1 Root ADB Access by Default

Initial exploration of the device indicated that ADB (Android Debugging Bridge) was enabled by default, which is a property of engineering and userdebug builds (referred to as "eng" or "userdebug" builds) of Android. These builds feature reduced security, compared with release firmware (referred to as "user" builds). This was confirmed by carrying out a factory reset of the device, and verifying that ADB remained enabled by default. The build fingerprint was also checked using `getprop ro.build.type`, indicating the build was a "userdebug" build.

One of the relevant security features disabled in non-release builds is ADB host verification, which requires the user accept a public key presented by the computer opening a connection with the ADB daemon on the phone. This meant that it was possible for an ADB session to be established without either a prompt being shown to the user, or confirmation being given by the user.

The ADB connection available over USB offered a standard Unix shell on the device, from which commands may be executed by any device connected to the USB port. One of the binaries available on the device was the `su` binary, designed to escalate the current user to root. On this device, it was possible to carry out an escalation from the ADB shell user, to the root user, without any prompt of input. This escalation to root access was confirmed using the Unix `id` command, indicating the shell was running as UID 0, that of the root user.

It was not possible for user applications to carry out a root escalation using this approach directly, since root access was only granted to the shell user. Nonetheless, this poses a risk for users charging or otherwise plugging their device into an untrusted charger, or where others may have even momentary physical access to the device. In many developing countries, where access to grid-supplied electricity is not practical, users charge devices in shops or public charging stations [12], putting them at risk of a rogue charger connecting over ADB and gaining root access to the device.

4.2 SELinux Bypass

Although root access was obtained from the ADB shell, this access was still potentially subject to SELinux policy constraints. While no denials were encountered in the process of carrying out this work, it was found to be possible to easily disable SELinux. SELinux was firstly ascertained to be enabled through the use of the `getenforce` command, which indicated the policies were in "Enforcing" mode (rather than merely in permissive fault logging mode). By running the command `setenforce 0` from the rooted shell, it was possible to disable the SELinux access controls, as verified by the "Permissive" response from the `getenforce` command.

4.3 Bootloader Root Shell

Access to the device bootloader was gained by holding down both the volume down, and power buttons, to turn on the device. In this mode, the device presented a menu of options, selectable using the volume keys and power button. In bootloader mode, an ADB device was again presented over the USB interface, once again without ADB authentication. Upon opening the shell, it was possible to escalate to root access using the `su` command. The presence of this vulnerability in the bootloader means that even in the event of the regular firmware being patched or upgraded, it would also be necessary to make significant modifications to the lower level boot stack, which may or may not be practical to carry out, given the risks of carrying out an upgrade of device bootloaders in-the-field.

While accessing this shell, it was observed that the user data and operating system filesystems were both mounted in read-write mode. This meant that it was possible to easily make persistent modifications to the operating system image, or access user data, directly from the bootloader shell.

We also observed that the bootloader installed on the device to the partition `mmcblk0boot0` did not feature a cryptographic signature at its footer, indicating it is likely that this bootloader is unsigned, and therefore potentially vulnerable to tampering or modification by a suitably determined adversary.

4.4 Recovery ZIP Signing Keys

The recovery environment, used to install operating system updates, has the capability for a ZIP file containing new firmware to be loaded into memory and installed. This ZIP file should be signed with a private key corresponding

to a certificate stored on the device itself, in order to verify that the firmware being installed has not been modified or corrupted in the process of reaching the device.

This signature check relies upon the confidentiality of the firmware zip signing keys — if a third party is able to generate signed firmware images, they may replace any component of the device operating system, including installed applications or even the device kernel, simply through creation of a custom firmware ZIP file. In the case of this device, the recovery image accepted standard ZIP signing keys, which are publicly available within the AOSP source code repositories [13]. As a proof of concept, a ZIP was created to display a message to the screen, and add a new file to the device filesystem. It was then signed using the AOSP testkey, and successfully installed onto the device, using the "Install from ADB" feature of the recovery environment.

The file `input.zip` was signed using the command `java -jar signapk.jar -w testkey.x509.pem testkey.pk8 input.zip output-signed.zip`, and installed using the command `adb sideload output-signed.zip`

The execution of the ZIP file was confirmed through the output of the command to display a message to the screen, and the new file added to the filesystem being observed following a reboot. This illustrated it was possible to make arbitrary modifications to the device operating system, such as adding new files, or modifying existing files, which would not be reverted following a device factory reset.

4.5 Application Package Signing Keys

Android applications (APKs) are signed in a similar manner to ZIP firmware upgrades. To prevent application replacement attacks, where a user is encouraged or coerced into installing a false update to an application, the Android platform will not allow an update to an application to be installed if its signing key does not match the existing signing key for the application. This ensures that the party signing the APK holds the same key as originally used by the developer. Likewise, applications are protected against downgrading, by ensuring that the version code has been incremented since the previous update, which could be used to install an old version of an application with vulnerabilities, for exploitation.

The signing keys used for the platform applications (which have privileged access to system APIs) were found to, again, be the default AOSP signing keys,

which are publicly available. It was therefore possible to replace core system applications on the device, including for example, the dialer, settings interface, keyboard, and overall firmware user interface (referred to as SystemUI).

A modified version of these applications could then be used to upgrade an existing version of the application, without the user being made aware of any extra risks. This is of particular significance since platform level applications have full access to the entire device and permissions. Having a publicly available platform key significantly violates the Android security model, which assumes the platform key is not available to attackers [14].

4.6 Android OS Security Status

To conclude our analysis of the device, we carried out an analysis of the device's resistance to a variety of standard, well-known attacks and exploits against the Android operating system, using the Bluebox's "Trustable" security scanner [15]. The results of this highlighted that the device was protected against only 3 of the 12 vulnerabilities scanned for. This scanner was selected, as it is capable of detecting all of the recent high-profile security vulnerabilities of the Android operating system, including "StageFright", the multiple variants of the "MasterKey" attack, and a number of kernel root exploits, including the futex attack. Figure 1 highlights the results of this scan. We also note that the device was vulnerable to the CVE-2015-3636 local privilege escalation attack via kernel ping sockets. An open source implementation of this exploit is available [16], and was used to verify that the device was vulnerable. Any application capable of executing a binary on this device (or indeed a user with access to a shell) was able to gain local root access, as shown in Figure 2, where a shell running as the root user was obtained.

Of these vulnerabilities, FakeID, Futex, ObjectInputStream and PendingIntent were reported in 2014, yet remained un-patched in this device, with a manufacture date of September 2015. This was due, in part, to some of these fixes being withheld until the release of future major versions of Android, rather than immediate security patches being released and backported to older software versions. While Google has begun to issue security backport patches and notifications [17], this is a very recent change, and requires the vendor to apply these patches. In the case of this device, the presence of serious weaknesses like the futex root exploit, suggest this is not the case, and that patches are not being applied prior to the launch of devices.

Indeed, by checking the build date of the software on the device from the command `getprop ro.build.date`, the software was found to have

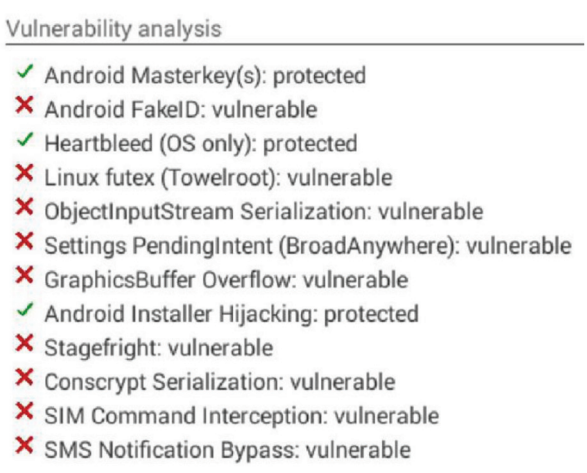

Figure 1 Result of Bluebox security scan for Android vulnerabilties.

```
shell@lc1913padv3wsl706:/data/local/tmp $ ./poc
Creating target socket.............. OK
9 + 39205 sockets created
2097152 bytes allocated
4194304 bytes allocated
4194304 bytes allocated
Done!
shell@lc1913padv3wsl706:/data/local/tmp # id
uid=0(root) gid=0(root) groups=1003(graphics),1004(inp
ut),1007(log),1009(mount),1011(adb),1015(sdcard_rw),10
28(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet
),3006(net_bw_stats) context=u:r:shell:s0
shell@lc1913padv3wsl706:/data/local/tmp #
```

Figure 2 Successful execution of local root exploit CVE-2015-3036.

been built in April 2015, several months before the product's release date. The software therefore appeared to not have been rebuilt by the manufacturer in the 5 months prior to launch, thus explaining the lack of many security patches. Since the device in question had no over-the-air update capabilities, we suggest a device shipping with 5 months' of disclosed vulnerabilities present puts users at risk from the moment the device is removed from its box.

4.7 Summary of Attack Vectors

A number of vulnerabilities have been identified on this device. All of these were in the default configuration, in an out-of-box setup as experienced by users. They are listed below in summary form.

- Privilege escalation to root possible from ADB shell on device
- ADB shell accessible without authentication, enabled by default
- SELinux can be disabled via a root shell
- ADB shell allowing privilege escalation to root available in bootloader
- Recovery image uses well-known signing keys intended only for testing
- Platform applications are signed using well-known signing keys intended only for testing
- Multiple previously disclosed vulnerabilities, including root exploits, unpatched on released device.

5 Potential Attacks

This section shall consider some of the potential attacks which could be carried out against users, as a result of the attack vectors described previously. These attacks should be considered in the context of a user in the developing world, who may be using this device for a variety of tasks, including banking or accessing government services, as discussed previously.

Previous works have examined user attitudes towards the perceived sensitivity and value of personal data on smart-phones, and what concerned them most about potential actions of software. A survey of these actions highlighted that, asides from permanently disabling or breaking a handset, the top ten concerns from users related to actions which would cost the user money (such as making premium rate calls or sending premium rate SMS messages), or were destructive (such as deleting user data like contacts) [18]. Other concerns raised included the public sharing of data which users felt was confidential, such as their text messages, emails, or photographs. Users were also concerned about the sharing of their data with advertisers, or the abuse of their contacts for spamming, and the risk of theft of passwords or other credentials such as credit card details.

In a survey of perceived value and sensitivity of their data, the sensitivity of location data and passwords was highlighted, as well as that of other types of data, such as photos and videos, or of messages [19]. It is therefore clear that users are concerned both about the theft of their data, and of the risk of the loss of such data, and the consequence of its loss.

5.1 Lockscreen Bypass

The first attack we identified allows an unauthenticated attacker to bypass the device's lockscreen, if the user made use of a PIN, password or pattern lock

for security. Since the lockscreen on Android is designed to fail insecure, in the event of corrupted (or missing) settings, simply removing the pattern or PIN data is sufficient to completely bypass the lockscreen. By removing the file `/data/system/password.key`, or `/data/system/gesture.key`, for PIN/password or pattern locks respectively, the lockscreen security was completely removed. Alternatively, the cryptographic hash of the password may be obtained from this file, and brute-forced, as described in [20, p. 268–275], in order to establish the plaintext password, PIN, or pattern lock, as set by the user.

This attack is possible, since the device exposes an unauthenticated ADB shell by default, with privilege escalation to root available through use of the su command. In the event that ADB is disabled on the device, it may also be carried out directly from the root ADB shell available through the bootloader. Finally, a firmware update ZIP file could be crafted (and signed) to remove this file via recovery, thus removing lockscreen security, and providing the user with full access to the device as though a password had been entered.

The ability to bypass the lockscreen poses a clear privacy risk to users, as it serves as an authentication bypass to carry out operations such as accessing user data, sending messages, and similar. It also allows access to messages and potentially sensitive files, including (for example), one-time passwords sent over SMS to the device. The ability to bypass the lockscreen therefore aids an attacker in carrying out interactive exploration of the device.

5.2 Theft of User Data

With ADB access available to the device by default, the `adb pull` command may be used to extract files from the shared storage area of the device, or the SD card. These areas of storage are not protected by per-application filesystem permissions, and are designed for the storage of data which a user may wish to transfer to a computer. The `adb pull` command does not require root access to succeed, and may be carried out either from within the regular operating system, or from the bootloader. In addition, due to the lack of authentication on ADB connection attempts, it is also possible to use ADB from the lockscreen to access data.

This attack makes it possible for a malicious attacker to extract all of a user's photographs, or documents, from the device. By expanding this attack to utilise root permissions (through privilege escalation via the `su` command), it is possible to further extend it to result in the theft of private per-application user data. This may include passwords for user accounts, as well as user

messages, photographs, cryptographic keys, and other sensitive data which should only be accessible to a single application, such as tokens. If this were carried out against a banking app, there is potential for sensitive user data to be obtained, depending upon the design of the application in question.

It was also possible to gain a full image of the device's data partition, using root access to recursively select all files found on the user data partition, and store them in a single compressed archive, which could easily be extracted from the device over ADB.

5.3 Full Access to Device Partitions

With full read-write access to the device, including operating system partitions, it was possible to make modifications to the installed operating system. This was achieved from within the operating system itself, by re-mounting the system partition as root using the command mount `-o remount, rw /system`. It was also possible to carry this out from the bootloader shell, as well as through the installation of a custom ZIP from the recovery environment. These changes are persistent through a factory reset, increasing the severity of this attack, since rectifying the modifications will be beyond the abilities of most users, if they were able to detect the modifications in the first place.

5.4 Installation of a Keylogger

A malicious attacker may wish to capture user credentials, in order to gain access to user accounts, or financial credentials such as card details. Alternatively, they may simply wish to know what a user is writing on their device, especially if it is used to carry out sensitive tasks. It was possible to install a keylogger through a variety of methods on this device, without the user being aware. Firstly, from a rooted shell, it was possible to replace the default keyboard application on the system partition (this change will persist between installs). It was also possible to remotely socially engineer a user to install an updated version of the keyboard application, using the publicly known signing keys to create an apparent update to the keyboard. This update could be distributed on the internet, or indeed installed through the USB interface of the device while charging. An updated version of an application could be installed over ADB using the command `adb install -r filename.apk`, or directly from the device filesystem by downloading it to the shared storage, and selecting the application from the file manager.

5.5 Deletion of Data

With root access to the device, it was possible to gain full access to the filesystems, and to erase all user data. By carrying out a backup prior to this process, and potentially encrypting the backup, it would be possible for a malicious party to ransom a user's data, requiring them to make a payment to gain access to it again. As one of the main concerns of users was the deletion of their data, this may cause significant inconvenience to users [18]. The ability to scale this attack to many devices, through the use of communal charging areas [12], would also make the ability to spread malicious software like this particularly harmful.

5.6 SMS Interception

With root access to the device, it is possible for the device operating system to be modified, such that SMS messages may be intercepted or redirected. This could take place without the user being notified or aware, and could be used to relay seemingly secure one-time passwords, which are commonly sent via SMS. This modification would be made in the messaging app or system frameworks, and could be carried out either using root access from a plugged-in device, or by socially engineering a user to install a rogue "updated" version of the messaging app, using the publicly known signing keys to sign the application package.

5.7 Premium Rate Abuse

One of the main concerns identified in [18] was the risk that rogue software could run up a bill by making premium rate calls or text messages. With root-level access to Android devices, it is possible to modify the dialer to make calls when the device is not in use, or to modify the messaging app to silently (without a record kept in the sent items) subscribe the user to premium rate SMS services, and send premium rate messages. While the Android operating system contains software to warn the user before premium rate SMS messages are sent, these warnings are easily removed or bypassed on a rooted device, where the operating system may be modified without user intervention or knowledge. A financially motivated attacker with the ability to carry out this attack on many devices, such as by creating a fake mobile phone charging station, could potentially compromise a large number of devices, and make significant quantities of money by making use of premium rate network services on behalf of unwitting users. To hinder discovery of such an

attack, the modification may be configured to only operate when the device is charging, such that no abnormal power use would be noticed by a user.

5.8 Random Number Generation Compromise

Another attack which we found to be possible was the effective "short-circuiting" of the kernel's random number generation. This was made possible through the root access exposed by the device. By renaming `/dev/urandom`, and creating a new symbolic link towards `/dev/zero` using the command `ln -s /dev/zero /dev/urandom`, all random number generation using the kernel APIs was compromised, after the depletion of the existing random pool by running `cat /dev/random`. A long series of random numbers was generated, and found to return all-zeroes, as expected. The ability for an attacker to do this allows for the compromise of security of encrypted communications, since randomly generated keys would be entirely predictable. Additionally, numbers expected to be random (such as nonces or initialisation vectors), may be re-used and predictable, potentially compromising the security of the protocol. In particular, for elliptic-curve based signature algorithms such as ECDSA, re-use of a nonce causes a catastrophic failure resulting in the ability for the signing key to be determined from 2 signatures using the same nonce [21, p. 68–72]. There is therefore potential for significant harm, if malicious software, or indeed a malicious attacker, were to abuse root access in order to render ineffective random number generation on the device. This may also expose a user to further attacks, since private components generated for Diffie-Hellman key exchange (as seen in TLS), would be predictable and repeated, potentially resulting in the ability for interception of HTTPS-based traffic. On a device where sensitive tasks, such as mobile banking, were being carried out, it would then be possible for a remote attacker with ability to observe network traffic (such as over an insecure Wi-Fi network) to break an HTTPS connection through knowledge of the compromised output of the random number generator.

5.9 Physical Damage to Device

Finally, with root access, it would be possible for a malicious attacker to abuse root access to render the device inoperable, and useless, effectively depriving the user from their computing device. By using the Unix `dd` command, it is possible to carry out raw read and write operations to block devices on the device. By overwriting the kernel image and recovery image, the device

would be beyond the repair capabilities of most users, since there would be no standard means through which to restore the firmware. As we noticed the device bootloader was unsigned, it is likely that an attacker could overwrite this with an empty partition, to permanently prevent the device from working. While there may be no clear or strong motive to cause physical damage to user devices, the ability to carry out this attack is a concern — potentially an unscrupulous retailer could try to drive more sales of new devices by having an attacker abuse this access to cause damage to devices. For a user reliant upon their mobile device as their main means of accessing electronic services, this would be potentially disruptive, and also cause them to lose data, reducing their confidence and trust in the technology.

6 Potential Mitigations

While the fundamental vulnerabilities identified here cannot necessarily be resolved easily by end-users, it is possible to identify some potential countermeasures to take. Firstly, users should disable ADB on the device, from the "Developer options" menu. This setting is ordinarily disabled by default, but was enabled on the device due to an engineering build of firmware being shipped on the finished device. Secondly, users should consider making use of a USB cable with the data lines shorted together, to prevent the transfer of data across the USB port when charging. Since the bootloader may be attacked separately from the operating system, users should ensure the device is always charged using this cable, and that the device is kept in sight at all times while charging (to prevent the cable being swapped). While there is no easy way to prevent abuse of the publicly-known signing keys used for the recovery image and platform applications, vigilance against installing any third party software, and avoiding the installation of any software would help to alleviate this. This would be a considerable trade-off for users, however, to forego the installation of software in order to avoid such attacks.

A technically confident user may attempt to remove the su binary from the device by remounting the system as read-write, as discussed previously, and using the command `rm /system/xbin/su`, although this will not protect the device from other attacks such as the abuse of known recovery ZIP and platform application signing keys, or indeed the use of the CVE-2015-3036 root exploit, as demonstrated in Section 4.6. This procedure is also somewhat risky in that removing the wrong file may result in the device being unbootable.

On future devices, it would be beneficial for over-the-air firmware updates to be possible as well — on this device, there was no facility inbuilt for

network-based updates that we could identify. Instead, there was a menu option inviting the user to place a file named `update.zip` on the SD card, and select a menu option to install it. In addition to the problem of the signing key being known for these update files, making it easy for a malicious party to distribute fake updates containing malicious software, the manufacturer is unable to issue prompt and regular security updates directly to devices, to address issues identified.

7 Conclusions

In this paper, we have highlighted some major security weaknesses in a recent, low-cost Android device, intended for developing markets. We identified that insufficient measures were in place to protect user data. We showed how a root shell could be gained on the device, out of the box, through both the device firmware and the bootloader. We also demonstrated the device shipped with a kernel root vulnerability, and that this is exploitable by any locally running software (such as an app). We also showed how persistent modifications to the firmware could be made, which would persist through factory resets, allowing for highly pervasive malicious software to be installed and target user data. Furthermore, we highlighted the risks of the device using the default firmware signing keys, and application signing keys, the private components of which are publicly available. We demonstrated that SELinux mandatory access control could easily be disabled by the root user.

The implications of vulnerabilities such as this are particularly significant for users of devices in the developing world, often the recipients and buyers of such low-cost devices as their main computing device. Merely plugging this device into a public charging station would be sufficient for a malicious party to gain full control over the device, extract all of a user's personal data, and install pervasive malicious software onto the device. This software could act as a keylogger, recording passwords and financial information, or could serve to silently gather sensitive data (such as two-factor authentication SMS messages) and forward them to the attacker. Malicious software could also erase all of a user's data, and demand a ransom for its return, or even simply destroy the device. We demonstrated these changes will persist through a factory reset, and that they are not visible to the end user. As the changes persist, it is not possible for a user to remove such malicious changes without advanced technical knowledge, and a known-clean firmware image to replace their device's software with. While we have presented some mitigations against these attacks, these require the user to be highly vigilant,

and would not allow them to make use of many of the functions of the device (such as installing applications), in order to defend against some of these attacks.

With the rise in adoption of smartphones and tablets in developing markets, the security and privacy of their users should be considered a priority when developing their software. Guidelines from the Android Compatibility Definition documents should be followed to avoid known security weaknesses. Finally, low level device firmware should be audited to ensure that bootloaders and other interfaces do not expose low-level root access to a device by default, which would undermine the security model of an otherwise-secure operating system.

Acknowledgment

This work was funded by EPSRC Doctoral Training Grant EP/K503174/1.

References

[1] B. Bajarin. (December 2014) Why India will be the world's second biggest smartphone market. [Online]. Available: http://time.com/3611863/india-smartphones/

[2] Cisco. (May 2015) VNI mobile forecast highlights, 2014–2019. [Online]. Available: http://www.cisco.com/assets/sol/sp/vni/forecast_highlights_mobile/index.html

[3] S. Etzo and G. Collender, "The mobile phone revolution in Africa: Rhetoric or realty?" *African affairs,* 2010.

[4] K. E. Skouby and W. Idongesit, *The African Mobile Story.* River Publishers, 2014.

[5] D. Porteous, "The enabling environment for mobile banking in Africa," 2006.

[6] B. Warner. (March 2013) What Africa can teach us about the future of banking. [Online]. Available: http://www.bloomberg.com/bw/articles/2013-03-06/what-africa-can-teach-us-about-the-future-of-banking

[7] G. Paul and J. Irvine, "Take control of your PC with UEFI secure boot," *Linux J.*, vol. 2015, no. 257, Sep. 2015.

[8] Google. The android source code. [Online]. Available: http://source.android.com/source/

[9] Google. (October 2015) Android 6.0 compatibility definition. [Online]. Available: http://source.android.com/compatibility/index.html
[10] S. Smalley and T. M. R2X, "The case for SE Android," *Linux Security Summit,* 2011.
[11] (October 2015) Datawind Ubislate 27CZ. [Online]. Available: http://www.pricedealsindia.com/mobiles/Datawind-Ubislate-27CZ-price-in-india-dpi4016.php#gotostore
[12] D. Wogan. (November 2013) Charging a mobile phone in rural Africa is insanely expensive. [Online]. Available: http://blogs.scientificamerican.com/plugged-in/charging-a-mobile-phone-in-rural-africa-is-insanely-expensive/
[13] Google. (October 2008) AOSP platform signing keys. [Online]. Available: https://android.googlesource.com/platform/build/+/master/target/product/security/
[14] D. Hackborn. (May 2011) Restrict system packages to protected storage, android code review. [Online]. Available: https://android-review.googlesource.com/#/c/22694/
[15] J. Forristal. (October 2014) Measuring mobile security & trust: Introducing trustable by bluebox. [Online]. Available: https://bluebox.com/measuring-mobile-security-trust-introducing-trustable-by-bluebox/
[16] CVE-2015-3636. Commit used: 9868289bdb53c. [Online]. Available: https://github.com/fi01/CVE-2015-3636
[17] Google. (August 2015) Android security updates. [Online]. Available: https://groups.google.com/forum/#!forum/android-security-updates
[18] A. P. Felt, S. Egelman, and D. Wagner, "I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices.* ACM, 2012, pp. 33–44.
[19] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov, "Understanding users' requirements for data protection in smartphones," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on.* IEEE, 2012, pp. 228–235.
[20] N. Elenkov, Android Security Internals: *An In-Depth Guide to Androids Security Architecture.* San Francisco: No Starch Press, 2015.
[21] A. Kak, "Elliptic curve cryptography and digital rights management," *Lecture Notes on Computer and Network Security,* 2015. [Online]. Available: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture14.pdf

Biographies

G. Paul received the B.Eng. (Hons.) degree in Electronic & Electrical Engineering from the University of Strathclyde, Glasgow, UK, in 2013. He is currently pursuing the Ph.D. degree in the Mobile Communications Group at the University of Strathclyde. He is a Graduate Student Member of the IEEE, and The Institution of Engineering and Technology, and the Chair of the University of Strathclyde IEEE Student Branch. His research interests include secure data storage and retrieval, practical considerations in the design of secure systems, and the design of privacy-preserving service architectures. Greig is the recipient of an EPSRC Doctoral Training Grant.

J. Irvine received the B.Eng. (Hons.) degree in Electronic and Electrical Engineering and the Ph.D. degree in coding theory from the University of Strathclyde, Glasgow, U.K., in 1989 and 1994, respectively. He is currently a Reader with the Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow, U.K., where he also leads the Mobile

Communications Group. He is a coauthor of seven patents and the books Digital Mobile Communications and the TETRA System (Wiley, 1999) and Data Communications and Networks: An Engineering Approach (Wiley, 2006). His research interests include mobile communication and security, particularly resource allocation and coding theory. Dr. Irvine is an elected member of the Board of Governors of the IEEE Vehicular Technology Society, a member of the IET, a Fellow of the Higher Education Academy, and is a Chartered Engineer.