A Feasible No-Root Approach on Android

Yao Cheng(✉), Yingjiu Li, and Robert H. Deng

School of Information Systems, Singapore Management University, Singapore, Singapore
{ycheng,yjli,robertdeng}@smu.edu.sg

Abstract. Root is the administrative privilege on Android, which is however inaccessible on stock Android devices. Due to the desire for privileged functionalities and the reluctance of rooting their devices, Android users seek for no-root approaches, which provide users with part of root privileges without rooting their devices. In this paper, we newly discover a feasible no-root approach based on the ADB loopback. To ensure such no-root approach is not misused proactively, we examine its dark side, including privacy leakage via logs and user input inference. Finally, we discuss the solutions and suggestions from different perspectives.

Keywords: No-Root approach · Android Debug Bridge (ADB) · Privacy leakage · Exploit analysis

1 Introduction

Android is a Linux based system with discretionary access control enforcement. Root access, which is part of traditional Linux systems, is blocked on stock Android devices for security reasons. If users would like to gain complete control over their Android devices with administrative permissions, they could root their devices at their own risks, such as device bricking and warranty turning void.

To avoid the risks of rooting their Android devices, users turn to no-root approaches which enable them to attain their desired permissions but without rooting their devices. The motivation of using no-root approaches might be strong since Android do not always provide all easy-to-use but desperately needed features. Some popular no-root applications [1,2], even paid ones [3], have achieved millions of downloads and high reputations in Google Play.

The existing no-root applications ("apps") primarily use Android Debug Bridge (ADB) [4] to launch a separate privileged executable program as background service on the target device. The background service is designed to respond to user's requests made from the no-root app and perform certain privileged tasks which the no-root app is not authorized to perform.

In this paper, we newly discover a feasible no-root approach leveraging the new ADB functionality provided on Android versions 4.x and 5.x which take up to 95.7 % in the distribution of Android devices according to the official statistics [5]. To our best knowledge, we are the first to discover such no-root

J.K. Liu and R. Steinfeld (Eds.): ACISP 2016, Part II, LNCS 9723, pp. 481–489, 2016.
DOI: 10.1007/978-3-319-40367-0_32

approach. This no-root approach has its advantage compared to the other no-root approaches in that it creates an ADB *loopback* instead of introducing a separate service. After the ADB loopback is created, a no-root app on the target device can run as a debugger to execute ADB commands to accomplish the privileged operations.

Though, we have not found any wild samples using this no-root approach yet, they may appear in the market at any time in any form, e.g., malicious apps pretending to be no-root apps. To ensure that such no-root approach is not misused in a proactive instead of reactive manner, we examine the dark side of this approach and reveal that the attacks leveraging this no-root approach can be launched from an app on a standalone victim device instead of on a development computer connected to the victim device. We reported the issue to Android. The latest Android 6.0 takes action to remove ADB client and ADB server on the latest Android 6.0 to avoid the attacks.

2 A Feasible No-Root Approach

2.1 ADB

ADB [4] is a debug system for Android that allows developers to connect development computers and Android devices/emulators. Developers can debug Android devices on separate development computers via ADB. ADB includes three components as shown in Fig. 1 (the components not in red), i.e., ADB client, ADB server[1], and ADB daemon. A developer issues an ADB command via an ADB client on a development computer. An ADB server on the development computer, passes the command from ADB client to an ADB daemon which runs on a target Android device. The response to the command is passed back to the developer along the same route. Before debugging, there is a switch to be enabled in the Settings→Developer options. Since Android 4.2.2, at the first time a development commuter connects the target Android device, a confirmation dialog showing the MD5 hash of an RSA public key of the development computer is prompted to obtain the explicit confirmation from the device owner.

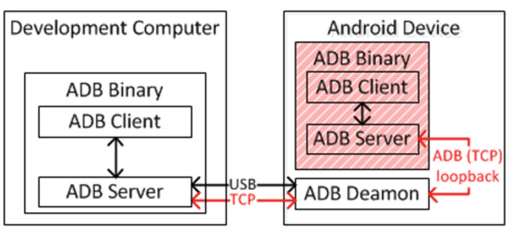

Fig. 1. ADB architecture.

[1] In practice, ADB server is implemented in the same binary as ADB client.

After the connection established, two categories of commands can be issued from the developer computer to the connected Android device, i.e., ADB commands and shell commands. ADB commands fulfil the functionalities for debugging purpose, such as device connection, app (un)installing, data transfer, and shell starting. Shell commands can be used after the shell starting, when a shell user, whose UID is 2000 on Android, is born with shell permissions. The majority of shell permissions [6] have protection levels equal to or higher than "dangerous". Note that any permissions higher than "dangerous" level are either hidden or not for use by third-party apps.

2.2 The Existing No-Root Approach

The existing no-root apps adopt ADB to launch a separate service in their preprocessing, and delegate the privileged tasks to this service during runtime. The preprocessing usually includes two manual operations. The first is to connect a mobile device to a development computer and switch on the debug mode. The second is to run a provisioned enabler on the development computer which has been downloaded separately from a no-root app's website. To understand the purpose of using an enabler, we introduce a typical enabler script as shown in Listing 1. In Listing 1, "svc" denotes the native service that performs a target task which requires certain high-level permissions. The executable service is pushed to the device (Line 1) and started by ADB shell (Line 3) so that it inherits the shell permissions for exercising some privileged functionalities. After that, the no-root app which directly interacts with users, is able to work by delegating some of its tasks to the running service through sockets. The service needs to be restarted once the Android device is rebooted, i.e., to run the enabler again.

2.3 A Feasible No-Root Approach Based on ADB Loopback

Different from the existing approach, whose privilege resides in a separate service, we newly discover a feasible no-root approach based on ADB loopback and requiring no separate service.

An Android device of versions from 4.x to 5.x can debug another Android device, because these new versions have introduced the ADB components, which are originally on development computers, to Android systems, i.e., ADB client and ADB server (the dashed components in Fig. 1). In addition, the connection mode is not limited to USB cable. A new TCP mode allows a development computer using TCP links to connect to the target Android device. However, an inconspicuous side consequence is that an Android device gains the capability of debugging itself by connecting its ADB server to its local ADB daemon via TCP mode (the loopback in Fig. 1). Based on such ADB loopback, we discover a new feasible no-root approach.

Listing 1. The existing no-root script.

```
1 adb push ./svc /data/local/svc
2 adb shell chmod 777 /data/local/svc
3 adb shell /data/local/svc &
```

Listing 2. The core snippet of Looper.

```
1 adb tcpip 5555
2 adb shell adb kill-server
3 adb shell HOME=/sdcard adb start-
    ↪ server &
```

It takes a simple preprocessing to establish the ADB loopback. What a user needs to do in this preprocessing is to run a script, which we name as "*Looper*", on a development computer connected to the target Android device. Listing 2 shows the core snippet of Looper. Looper turns on the TCP mode at port 5555 on the target Android device (Line 1). Then, it restarts the ADB server setting "/sdcard" as HOME folder (Line 2 and Line 3). The purpose of changing HOME folder is to guarantee that Looper could work as well on Android 4.2.2 and higher. This is because since Android 4.2.2, ADB introduces the RSA authentication that stores its key pair in the HOME folder. Looper changes the HOME folder to a shell-user-accessible folder, so that the RSA key pair of the ADB server can be stored successfully for later authentication. After confirming the dialogs requiring the explicit confirmation from the device owner, the ADB loopback is established, and its effect lasts till the Android device is rebooted.

After ADB loopback is established, a no-root app with the permission to connect to local TCP ports can play the role of a debugger. The permissions of ADB that are intended for remote development computers are now available on stand-alone Android devices. As a result, by using ADB loopback, no-root apps can perform privileged tasks as intended.

3 Exploits on the Dark Side

No-root has always been a double-edged sword[2]. It is important to explore its dark side proactively. In this section, we demonstrate two typical exploits on such no-root approach.

3.1 Adversary Model

The scenario of our investigation is that a user has an Android device which is *not* rooted. (S)he has installed a no-root app that adopts the newly-discovered no-root approach on his/her device for the purpose of enjoying privileged functions without rooting the device. We investigate the potential threats causing by a malicious app only with the internet permission, which can be the no-root app itself or other apps on the same device.

[2] The existing no-root approach could lead to privacy leakage due to the insecure socket communication between the no-root app and its native service [7].

3.2 Privacy Leakage via Application Logs

Android provides a logging system for inspecting debugging outputs. The access to log messages is regulated by callers' UIDs. Normal users, i.e., third-party apps without root privilege, can only access the logs related to themselves. However, an app, leveraging the no-root approach we discover, can get system-wide logs using "logcat" which is the official tool for dumping log messages.

If there is no sensitive information being logged, there should be no information leakage via logs. Android documents have suggested that logs should be managed, e.g., removed in release versions, according to their types [8]. Even though, it happened that some informative data is logged [9]. We are interested in whether developers manage sensitive logs properly nowadays, since private log may become readable to other apps in this scenario.

The sensitive information on mobile devices is classified into four categories. The device parameters reflect the characteristics of devices, including Android version, device model, manufacturer, root status, and phone service information (phone number, IMEI, and IMSI). The app account information is on the application level, which includes account ID, account credential, and personal profile. The user interaction indications indicate the operations a user performs, such as opening an activity and inputting a password. Finally, geographic data, network information, and others are classified into the last category.

The top-ranked 10 account-sensitive free apps from Google Play and Anzhi Market are examined, respectively. The observation shows that 11 of the 20 top-ranked apps log some sensitive data in Table 1.

Table 1. The sensitive information collected from log messages.

Applications	Device params	Account info	User interaction	Others
org.mozilla.firefox (G)	✓	✓	-	-
com.tencent.mtt (A)	-	✓	-	-
com.taobao.taobao (A)	✓	✓	-	-
com.sinovatech.unicom.ui (A)	✓	✓	✓	✓ Location
com.skype.polaris (G)	✓	-	-	✓ User agent string, country code
com.tencent.mobileqq (A)	-	-	-	✓ Gateway IP, SQL statement, established connections, network info and quality test
com.google.android.youtube (G)	✓	-	-	✓ Country code, network info
com.facebook.katana (G)	-	✓	-	✓ Gateway IP, GPS data
com.cleanmaster.mguard (A)	✓	-	✓	-
com.snapchat.android (G)	-	-	✓	-
co.vine.android (G)	-	-	✓	-

One interesting example is due to the improper use of third-party SDK. Snapchat [10] uses Flurry [11] SDK to help its developers obtain the usage analytics. Flurry defines log APIs for developers to monitor the runtime behaviours of apps during developing and debugging. It is observed that some real-time user operations are logged using Flurry APIs in the release version. One of such cases, which happens during registration, is demonstrated in Listing 3. It can be inferred that Snapchat first focuses on the email field (Line 2), and then the edit on this filed begins (Line 3). After that, it focuses on the password filed waiting for inputs (Line 4). Once a user starts inputting his/her password, it immediately outputs the corresponding log (Line 5). Even there is no direct leakage of email or password, the information about focusing and editing can be used maliciously to launch other attacks such as keylogger attacks.

Listing 3. FlurryAgent logs in Snapchat showing user interactions during registration.

```
1  W/FlurryAgent(20495): Event count started: R01_BEGIN_REGISTRATION
2  W/FlurryAgent(20495): Event count started: R01_FOCUS_ON_EMAIL
3  W/FlurryAgent(20495): Event count started: R01_EDITED_EMAIL
4  W/FlurryAgent(20495): Event count started: R01_FOCUS_ON_PASSWORD
5  W/FlurryAgent(20495): Event count started: R01_EDITED_PASSWORD
```

3.3 User Input Inference

User input inference is a way to obtain users' private information such as account credential by capturing their input. An attacker can apply the input inference to surmise the credential at the time of user inputting. Unfortunately, if the no-root approach is misused, both input timing and input characters are available.

Good Timing of Credential Input. Normally, when a login activity is shown on screen, if the keyboard is invoked at the same time, there is a higher chance that a user is going to input account credential to this activity.

Login activities usually share a common pattern which can be used to detect them. A login activity normally consists of at least two EditText fields for inputting the username and password, respectively. Among the two, the second EditText field conceals the password by representing each input character in a black dot or asterisk. This pattern is reflected in the activity layout which can be obtained in XML format using the shell command "uiautomator". And the keyboard appearance can be captured using the shell command "dumpsys".

We test the good timing detection algorithm with the top 20 finance apps in Google Play. Experiments show that the algorithm can capture all the login activities in 15 apps. The other 5 apps are verified to have no login activities.

Inference of Input Characters. The characters that a user inputs on a touchscreen can be inferred from knowing both of the touch position on screen and the software keyboard layout.

First, let us consider the touch positions. The dispatch destination of each click position is supposed to be the app running on screen only. However, with the dark side of the no-root approach, a malicious app on the same device can access directly the touch coordinates using the shell command "getevent" no

matter it is running on screen or not. In this way, the accurate touch position is known by parsing these raw events [12] returned by this command straightly.

Second, let us consider the keyboard layout. The position of each key varies according to different layouts, e.g., "QWERTY" layout. Even for the same layout, the position might be different due to the adjustment by vendors. The information about the input method, e.g., its vendor name and whether it is invoked, is available using "dumpsys". As a result, the combination of touch positions and the keyboard layout can further surmise the input characters.

4 Discussion

After we verify that the no-root approach can work on Android versions from 4.x to 5.x, we reported it to Android in August 2015. Android admitted soon that the no-root approach can work as intended, and so do the exploits on its dark side. Later in October 2015, Android adopted a straightforward solution by removing the ADB client and ADB server, i.e., the ADB binary from the newly released Android 6.0. These two components are responsible for accepting debugging commands and communicating with the ADB daemon, respectively. As a consequence, an Android device can no long be used to debug other Android devices. While it is a simple solution to remove the debug functionality, it is not ideal due to sacrificing much benefit/convenience provided by ADB debugging and no-root apps. A preferred solution should mitigate the ADB loopback exploits while still make it work for benign no-root apps, such as extending the existing permission-based mechanism. We leave this to the future work.

On the other hand, the ignorance of developers and markets is another important cause of the exploits. On the app developers' side, proper coding and configuration would help to protect apps against some malicious exploits. It is important for app developers to clean up sensitive logs when producing release versions. On the app markets' side, it is suggested that app markets enforce effective and specific vetting processes. Google Play has set up an example of using its bouncer [13], which checks for malicious operations and certain vulnerabilities in each app submitted to Google Play and suggests whether or not accept the app in the market. We suggest that Android markets, both official and third-party ones, should check for the usage of logging code, e.g., debug or verbose level log, so as to avoid leaking sensitive information in logs.

5 Related Works

Several ADB based attacks have been identified before. Vidas et al. [14] mentioned in their survey that an untrusted ADB connection via USB could result in security breaches when an attacker is physically close to the target device. Recently, Symantec detected a Windows malware which may infect Android devices with ADB [15] via USB connections. Hwang et al. [16] presented some feasible stealthy attacks which can be performed with ADB capabilities. In this

paper, we firstly discover a feasible no-root approach that based on ADB loopback to achieve extra privilege in Android system without root. The dark-side exploits of this no-root approach and the evaluation on real-world apps are complementary to the ADB based attacks identified before in terms of providing a better understanding on how ADB can be misused.

Previous research has shown that some existing no-root applications can be misused. Lin et al. [7] attacked some existing no-root screenshot apps and abused their screenshot functionalities. It was shown that user input can be inferred by analysing the screenshots taken by these apps. In order to prevent the newly-discovered no-root approach from being misused or attacked, we proactively explore its dark side.

Developers' negligence in code regulation was pointed out that a malicious app can read SMS, obtain contacts and access location by selectively reading the system logs in earlier versions of Android [9]. However, since Android 4.1, an app is restricted to read its own logs only. Nonetheless, it is still not a secure way to log sensitive information. Because like one of the dark-side exploits in this paper, the system-wide logs may become available to an installed malicious app. The evaluation on the top-ranked real-world apps shows that many of them still log informative data, which leads to severe privacy leakage.

6 Conclusions

In this paper, we discover a feasible no-root approach leveraging ADB loopback working on Android devices of versions 4.x and 5.x for the first time. To ensure that this no-root approach is not misused in a proactive manner, we investigate its typical dark-side exploits and evaluate them with real-world apps. Finally, we discuss the mitigation that could be adopted by different parties.

Acknowledgement. This material is based on research work supported by the Singapore National Research Foundation under NCR Award Number NRF2014NCR-NCR001-012. We thank Professor Lingyun Ying from Chinese Academy of Sciences for his helpful discussion at the early stage of this work.

References

1. Helium - app sync and backup. https://play.google.com/store/apps/details?id=com.koushikdutta.backup
2. Clockworkmod tether (no root). https://play.google.com/store/apps/details?id=com.koushikdutta.tether
3. No root screenshot it. https://play.google.com/store/apps/details?id=com.edwardkim.android.screenshotitfullnoroot
4. Android debug bridge. http://developer.android.com/tools/help/adb.html
5. Platform versions distrubution. http://developer.android.com/about/dashboards/index.html
6. Shell permissions on android. https://android.googlesource.com/platform/frameworks/base/+/android-5.1.0_r5/packages/Shell/AndroidManifest.xml

7. Lin, C.-C., Li, H., Zhou, X., Wang, X.F.: Screenmilker: How to milk your android screen for secrets. In: NDSS (2014)
8. Log. http://developer.android.com/reference/android/util/Log.html
9. Lineberry, A., Richardson, D.L., Wyatt, T.: These aren't the permissions you're looking for (2010). https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf
10. Snapchat. https://play.google.com/store/apps/details?id=com.snapchat.android
11. Flurry. http://www.flurry.com/
12. Getevent. https://source.android.com/devices/input/getevent.html
13. Android and security. http://googlemobile.blogspot.sg/2012/02/android-and-security.html
14. Vidas, T., Votipka, D., Christin, N.: All your droid are belong to us: a survey of current android attacks. In: WOOT, pp. 81–90 (2011)
15. Windows malware attempts to infect android devices. http://www.symantec.com/connect/blogs/windows-malware-attempts-infect-android-devices
16. Hwang, S., Lee, S., Kim, Y., Ryu, S.: Bittersweet adb: Attacks and defenses. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, pp. 579–584 (2015)