Analyising Use of High Privileges in Android Applications

Huasong Meng
School of Computing
National University of Singapore
huasong.meng@u.nus.edu

Abstract—The number of Android smartphone and tablet users has experienced a rapid growth in the past few years and it raises users’ awareness on the privacy and security of their mobile devices. In this project, the features of openness and extensibility make Android unique, attractive and competitive but meanwhile vulnerable to malicious attacks. There are lots of users rooting their Android devices for some useful functions, which are not originally provided to developers and users, such as backup and taking screenshots. However, after observing the danger of rooting devices, the developers begin to look for other non-root alternatives to implement those functions. ADB workaround is one of the best known non-root alternatives to help app gain higher privilege on Android. It used to be considered as a secure practice until some cases of ADB privilege leakage have been found. In this project, we design an approach and implement a couple of tools to detect the privilege leakage in Android apps. We apply them to analyse three real-world apps with millions of users, and successfully identify three ADB privilege leaks from them. Moreover, we also conduct an exploitation of the ADB privilege in one app, and therefore we prove the existence of vulnerabilities in ADB workaround. Based on our study, we propose some suggestions to help developers create their apps that could not only satisfy users’ needs but also protect users’ privacy from similar attacks in future.

I. INTRODUCTION

The rise of mobile devices has greatly enriched people’s lives in this digital era. As the dominator of current mobile device market, Android reserves over 82.8% of the entire smartphone market share [1]. Andy Rubin, the VP of Google stated on his twitter that the number of Android devices activated per day has reached 1.3 millions in past few years [2]. The global gross shipments of Android devices is expected to grow from 1.2 billion in 2015 to 1.54 billions in 2019 [3]. The over-reliance on mobile devices makes people save all the data regardless of personal or business purpose onto their smartphone or tablet, which may lead their privacy under exposure if no proper protection has been enforced.

Android is well-known by its rich functionality and customisation, but there are still some requirements which cannot be implemented by using the official APIs. Google creates a collection of permission labels to define the privilege of apps running on Android OS. Some actions like reading the content displaying on the screen, in another word taking screenshot, was marked as signature level permission, hence are not allowed to be realised by the third party application. As long as the requirement of users exists, the developers would never stop to push the boundary. For that reason, developers are all motivated and successfully come up with two approaches to solve the permission dilemma, namely “rooting the phone” and “ADB workaround”.

Rooting the devices could enable users to gain the admin-istration privileges to do anything they want such as removing pre-installed apps, unlocking more functionalities, or sometimes just simply changing the theme of UI. According to a statistic done by Kristijan Lucic in 2014 [4], there are over 27.44% users indicating that they have rooted their smartphones to remove redundant and useless pre-installed applications. However, there are several security issues behind the “rooting” because it disables the permission mechanism on Android system.

The good news says there is an increasing number of people who have realised the risk and danger of rooting their devices, and has started seeking non-root approaches. Gaining system privilege through Android Debug Bridge (ADB) is one of the best known and widely used workarounds. Users can connect their devices to a PC via either USB or wireless network, launch the ADB and then invoke a service with system level privilege running in the background. After that, an application could communicate with that service, send command to it, and thereby trigger it to work for the application with system privilege. In this manner, that app can do the job even without APIs provided by Android, any time and anywhere unless being powered off.

There are plenty of apps on Google Play Store adopting this ADB workaround to satisfy users’ specific needs which are not provided with APIs by Android OS, including, but not limited to, performing backup and restoration, taking screenshot, recording screen, etc. Those apps using ADB workaround to achieve high privilege are very popular and some of them has millions users.

The security of ADB workaround has been raised up after some exploitation being successfully conducted. It is critical because it refers to the privacy of millions of Android users. In this project, we design an approach to discover the vulnerabilities of ADB workaround and implement a solution to detect the privilege leakage. We apply this approach to three real-world apps downloaded from Google Play Store, analyse them and eventually discover the privilege leakage on each
of three apps by using the tool we implemented. In addition, we conduct an exploitation on one of these three apps named “No Root Screenshot It” and successfully prove the existence of vulnerabilities that we have recently found. Last but not least, we provide some advice to the developers to help them achieve users’ requirement and meanwhile protect users’ data and privacy.

II. BACKGROUND

A. Privilege & Permission on Android

Privilege is a security attribute required for certain operations. In Unix-like operation system, the process privileges are organised in shape of a flat tree, where the users’ privilege is presented as the leaves and the superuser is described as root [5]. Android, as a mobile operation system built based on Unix/Linux, takes advantage of the user-based protection to identify and isolate the resources used by applications. All applications on Android run within the application sandbox and only have limited permissions to access resources which have declared in advance [6].

Similar to other Unix-like operating systems, Android also has the superuser-level privilege called root that has full access to all apps’ data. By default, only a small subset of the core applications run with system or signature permissions, and most of the apps downloaded from the Google Play Store only request the user-level permissions such as access to the Internet, hardware sensor and camera. Android OS does not prevent users or applications with the root permission from accessing and even modifying the other applications even it is not encouraged [6].

Besides employing underlying Unix-like privilege system, Android also has its own privilege management mechanism, which is also known as permission levels. On Android platform, permissions are classified into several protection levels. Most of the Android developers are made available to either the “normal” level permissions or the “dangerous” level permissions in their development. The normal level permissions, such as Internet, vibration, NFC or setting alarm, are considered as having no great risk to the user’s privacy or security. Those permissions will not prompt the user to grant permission if properly declared in manifest during the development. The dangerous level permissions indicate that the application needs access to private data or control over the device that may potentially have a negative impact to user. Unlike the normal level, all the operations classified in dangerous level will not be executed until obtaining user consent. In addition to normal level and dangerous level, there are two more protection levels namely signature level and signature or system level defining risky permissions. The former is only granted to the applications signed by a trusted party like Android development group. The latter could only be granted to the apps that are embedded in Android system image or signed by vendors of the system image [7]. The grant of these two permission levels is not to be approved by users, instead, it is conducted by signature validation mechanism of Android system during installation [8]. Many functions that users require but not provided as public APIs by Android OS, like backing up, taking screenshot and screen recording, belong to the signature level permission.

| Permission             | Description                                      |
|------------------------|--------------------------------------------------|
| CALL_PRIVILEGED        | Initiate a call without user confirmation         |
| CAPTURE_VIDEO_OUTPUT   | Capture video output stream                      |
| DELETE_CACHE_FILES     | Delete cache files                                |
| DELETE_PACKAGE         | Uninstall package                                |
| INSTALL_PACKAGES       | Install packages                                 |
| READ_FRAME_BUFFER      | Access to the frame buffer data (e.g. screenshot) |
| READ_LOGS              | Read system log files                            |
| REBOOT                 | Reboot the system                                |
| SET_TIME               | Set system time                                  |
| WRITE_APN_SETTINGS     | Overwrite APN setting                            |

B. Privilege Escalation

In order to implement the function like backup, taking screenshot or screen recording, developers have to find a way to escalate the privilege of their apps to the signature level or higher. There are two privilege escalation approaches on Android, namely rooting and non-root workarounds.

1) Rooting: The word “root” may not be a strange term to an Android player. Rooting is the process of allowing users of Android devices to attain privileged control, which is also known as the root access [9]. The rooting process grants users with superuser level permissions to use their devices, which could provide users a ton of customizations and opportunities to exhaust the functionality of Android OS. However, besides the risks of voiding warranty and bricking the devices, rooting an Android device could also expose all the data and program to the adversary and bring severe security vulnerabilities [10]. The rooting process varies with hardware manufacture and system version, but generally speaking, there are three steps to root an Android device, including unlocking Bootloader, flashing custom Recovery ROM, and lastly, installing SuperSU [11]. Once an Android device is rooted, users can access and modify the system resource. Furthermore, user can customize privilege and assign to any application installed on the rooted device.

2) Non-root Alternative: Rooting Android device is a risky practice because it may void the warranty, brick the device and bring with numerous security vulnerabilities. Therefore, developers start to seek non-root alternatives to escalate privilege. There is an alternative approach called ADB workaround to attain high level privilege without rooting the device, and it becomes popular whilst the growth of users’ concern to their device security.

Take the programmatic screenshot as an example. An app needs to have a signature-level permission from the system to take screenshot, which is impossible for normal developers to obtain through normal level permission request in interface. However, there are still two workarounds even without the
concession from Android development team: (1) taking screenshot on rooted devices; or (2) making use of a process with higher privilege to escalate the privilege of the app. The latter approach does not require the holistic change to the Android devices like “rooting”, and has better security and reliability [12].

ADB is a development tool provided by Google to allow developers to debug apps shell in the Android devices from their PCs. A process requiring signature-level permissions, such as taking screenshot or backup, is not allowed to be implemented in app by third party developers, but could be started from a ADB shell window [12]. That is the reason why ADB workaround could achieve higher privilege. Using ADB workaround, developers could implement all methods requiring signature-level permissions, pack all of them into a executive that could be started on ADB and run them in the background of Android OS as a service until being killed (e.g. power-off, restart). Thus the unprivileged App could communicate with the privileged proxy to indirectly achieve the functionality that may not able to be done on unprivileged App only.

C. Reverse Engineering on Android

Reverse engineering, also known as back engineering, is defined by Wikipedia as “the term describing processes of extracting knowledge or design information from an existing product and reproducing based on the extracted knowledge or design information with personal modification” [13].

The Apps on Android are mainly written in Java and HTML5/CSS, packaged in format of APK and executed on JD-GUI which could be easily further into Java codes by using Java dex2jar to modify an app and repackage it [15]. Another tool called assembler and disassembler for APK files, thereby facilitate decompilation and reverse engineering on Android are more complicated and more challenging than doing the same things on Java code [14].

There are some tools designed for easy decompilation of Android Apps being made available to the public for free. For example, Apktool [3] is a very useful tool providing small assembler and disassembler for APK files, thereby facilitate to modify an app and repackage it [15]. Another tool called dex2jar [4] allows users to convert APK back into a jar file, which could be easily further into Java codes by using Java decompiler such as JD-GUI [4] [14].

Most developers do not want the reverse engineering happened onto their products. Techniques like obfuscation could help developers to make the code unreadable before it being released to the market to prevent reverse engineering. A tool named ProGuard has been integrated into the Android build system by Google to help users shrinks, optimizes, and obfuscates the code [16]. Another tool DexGuard, which is a premium version of ProGuard, are widely used by commercial organisations and enterprises because it offers more advanced features to protect Android Apps from both static and dynamic analysis [17].

D. Instrumentation

There are two famous alternatives of frameworks that achieve hooking on Android. One of them is an open-source frameworks called Xposed. Another well-known framework is Substrate which is not open-sourced but has better documented API. Both of them have similar functions and work based on a similar mechanism as well [18]. Xposed is more prevalent in the research field and developers community. There are over 800 modules implemented based on Xposed framework available to the public for free on Xposed Module Repository [19].

The core of Android runtime is the process called Zygote. Each application is executed in form of a copy of Zygote, which is also known as “fork”. When an Android device is booted, a script named “init.rc” is started, followed by loading all necessary classes and methods by “/system/bin/app_process”. When Xposed comes into play, an extended startup app_process is copied to the “/system/bin” and then be started during the booting. In this way, Xposed can inject that extended startup process into the DVM instance, before each main method being called and invoked. Xposed provides a private and native method named hookMethodNative which is implemented within extended app_process. It could change the method type to “native”, and then onload its own implementation to that “native” method. Each method being hooked by Xposed will be converted into a “native” one, and the actual method being called could be treated as a brand new method with original implementation and overrided hook handling code written by Xposed framework developer [20].

Hooking usually happened before or after the execution of target methods. A hook module could not only record and log

1 Apktool is available on http://ibotpeaches.github.io/apktool/
2 dex2jar is available on https://sourceforge.net/projects.dex2jar/
3 JD-GUI is available on http://jd.benow.ca/
4 Xposed is available on http://repo.xposed.info/
5 Cydia Substrate is available on http://www.cydiashubstrate.com/
all the data sending in and passing out the target methods, but also alter the behaviour of target methods by modifying its parameters or return value [21]. Therefore a dynamic analysis by Xposed hooking could help us to sketch out the data flow of a program. Figure 1 shows an example of hooking process by Xposed hooking could help us to sketch out the data flow parameters or return value [21]. Therefore a dynamic analysis also alter the behaviour of target methods by modifying its all the data sending in and passing out the target methods, but also alter its behaviour by modifying return value.

III. APPROACH & CASE STUDIES

A. Approach

The apps using ADB workaround is usually a combination of a normal app with all functions being restricted at normal or dangerous level permission, and a proxy started by ADB and therefore has signature level permission on Android. In Android, most of apps communicates with proxies through socket channel, which has no strong protection and generic access control. A malicious app could easily obtain the control of proxy if it knows the protocol of communication between app and it. Whether the interface of the proxy is protected to prevent from third party access is the first potential issue.

Some apps implement password authentication into the protocol to strengthen protection to the proxy. However, because of the inconsistency of app and proxy’s life cycles, there must be a mechanism to temporarily save the password. By this means, a malicious app could still have chance to know the password if proper analysis has been done. Therefore, whether the protection is effective and secure enough is the second potential issue.

The approach to analyse the app and find the vulnerabilities of ADB workaround is initiated based on two potential issues we have mentioned recently, and targeted to exploit if any vulnerability has been found. We summarised this approach into four step:

1. **an analysis on the proxy activation;** This analysis could be done on reading proxy activation script if exists. The script could be either in batch file or bash script depends on which OS, Windows or Linux, to be run. Some apps do not provide script file to the user for Windows OS, for instead, a desktop application with UI is provided to achieve better user experience. In this circumstance, the Linux version of activation package is recommended to be download because script file is more widely used and possibly given on Linux OS. A simple and clear script file could disclose some details of the protocol of communication between app and proxy, such as the name of service proxy, the native executable file of proxy if any, and how the service being activated. Besides the analysis onto the script file, the name, process ID and permission group of service proxy running in the background could also be found by typing command “adb shell ps” in ADB through USB to the device. Moreover, the port opened for the communication between service proxy and app could be found in similar way by typing ADB command “adb shell netstat” to retrieve all active network usage on the device. However, in this step, the pairing of process and specific port listening may not be able to be observed if multiple proxies had been activated, like the scenario during the case study of app III.

2. **an analysis on the APK file;** Concept of reverse engineering will be involved in this step like APK decompilation. Once the service proxy and port number have been identified, the next step is to discover the implementation of the communication between proxy and app in APK file. The APK file could be unpackaged and decompiled into smali/Java source code or assembly code, by using tools like Apktool or dex2jar. The smali/Java code has supreme readability which may help us to look through different classes to locate the code of protocol’s implementation. In fact, the decompilation analysis may not always to be proved as a smooth and easy process because most of developers obfuscated their code before releasing the APK file to the app Store. In this situation, the disassembly will be helpful and a supplement to the smali/Java code reading. Reading assembly code could help us recognise the constant strings and numbers defined within same class. For example, the magic number used for authentication in app II, which is 89234820, was found in this manner.

3. **an dynamic analysis;** Only reading the script and source code may not be sufficient to sketch out the entire protocol between proxy and app. The objective of dynamic analysis is to find both control flow and data flow occurred when the app interacts with service proxy. Reading logs through logcat is a simple but effective way to gain a brief understanding to the protocol. However, hooking by Xposed framework will be one of the best solution to complete the analysis when the source code has been enforced with strict obfuscation or an authentication has been applied onto the socket channel between app and proxy server. Hooking method could be forced onto the key methods in class(es) in charge of communication between app and proxy which has been discovered in last step, then sniff and extract the arguments passed in and return value through the system logs. According to the case studies in this project, the methods to be hooked are mostly used to handle the action trigger (e.g. takeScreenshot) and socket channel I/O (e.g. write). Hooking on the prior method(s) by printing logs could show us the control flow of the protocol, and hooking on the latter method(s) by extracting arguments’ value could help us understand the data flow between service proxy and app itself. By now with both control flow and data flow confirmed, the protocol between service proxy and app, which used to escalate privilege on Android OS, has been unveiled.
authentication analysis if any; There is very possibly an authentication process if any series of numbers or a random string was found in the data flow of the protocol. In that case, it is encouraged to clarify if the password a constant (like app II) or dynamic (like app III); then followed by determining relationship between the password and life cycle of app or/and proxy(s) if the password is found to be dynamic. For the dynamic password, if the password change is triggered by proxy and independent of app itself, the password is generally stored at somewhere that the app has permission to read; otherwise the password itself or the mechanism of its changing should be able to be found in app’s source code.

Once these four steps listed above have been fully understood, attackers are theoretically able to exploit an app that uses ADB workaround in a programmatic manner. In the later part of this section, case studies on three apps using ADB workaround will be analysed and one of them will be exploited by apply this approach.

B. Case Studies

A group of screenshot apps with the highest popularity and the most recent updates were chosen from the Google Play Store. There are altogether 13 screenshot apps being downloaded from Google Play Store and installed on testing Android devices. We have run all of them by following their instruction, observed the outcomes, and recorded all key information. The statistics of those 13 screenshot apps could be found in Table V in appendix of this report. After reviewing all 13 screenshot apps, as shown in Figure 3, a simple classification were been made according to their functionality. These apps were firstly be separated according to their compatibility with un-rooted devices. 5 apps has instruction saying that they only support rooted devices. Among the 8 apps which works on un-rooted devices, 6 apps could only take screenshot by pressing hard-key combination (usually power key + volume down key), which may vary with configuration of manufactures, This method almost works on Android OS version 4.0 and later versions only [22]. Two apps left, Screenshot Ultimate and No Root Screenshot It develop by “iccecoldapps” (noted as app I) and “edwardkim” (noted as app II), has been found using ADB workaround to take screenshot and selected in case studies.

Unlike taking screenshot, screen recording is the function that users could not obtain by either third party apps without signature level permission, or any solution provided by Android OS. Keyword “screen recording” has been searched on Google Play Store. The number of apps found is much less than the searching result of “screenshot”. However, most of screen recording apps are using ADB workaround to achieve its functionality. In this project, one recording app, FREE screen recorder NO ROOT developed by “Invisibility Ltd” (Noted as app III), has also been chosen and analysed in case studies.

1) App I – “Screenshot Ultimate”: “Screenshot Ultimate” is one typical screenshot app that does not require a rooted device. It supports screenshot taking through ADB workaround. However, it is not like the deal above board since too obvious instruction may bring Google’s restriction. The ADB workaround is mentioned in a paragraph of “Help” instruction, and the URL to download the script and other necessary files are given in another place and could only be found in “Setting” → “Capture methods” → “Manual”. Overall, it is still kind of good experience because of detailed step-by-step instruction and troubleshooting notes.

Analysis of ADB Channel

The native executable file, named “screenshortultimatemenative1”, and scripts for both Linux and Windows OS have been downloaded as a zipped file from the URL given in the help instruction. After reading through the script file, the flow of service activation has been summarized and shown in Figure 5. With the process name of service running in the background, what we are going to do is to analyse the APK file and unveil the protocol of screenshot taking process between app and that service.

Decompile of APK
### Table II
**APPLICATIONS HAVE BEEN STUDIED IN THIS PROJECT**

| Name                        | Description            | Size   | Identity Package Name                       |
|-----------------------------|------------------------|--------|---------------------------------------------|
| Screenshot Ultimate        | Screenshot             | 3.2M   | com.icecoldapps.screenshotultimate          |
| No Root Screenshot It       | Screenshot             | 838k   | com.edwardkim.android.screenshotitfullnoroot|
| FREE screen recorder NO ROOT| Recording              | 7.4M   | uk.org.invisibility.recordablefree          |

The APK file is generated from Java code through a systematic routine which might be easy to be analysed when compared with the native executable. APK file could be downloaded by searching the mirror site of Google Play Store\(^6\) or extracted by using specific apps like “APK Extractor”\(^7\). Once the APK file has been obtained, the decompilation could be done for analysis purpose.

The reverse engineering tool “dex2jar” has been used to decompile the APK file to the jar format. Then further Java decompilation has been done by “JD-GUI”. So far, the original source code should be perfectly reversed if there is no exception happened due to code obfuscation or other protection mechanism. Unfortunately, the class organisation of the code obtained from the decompilation of “Screenshot Ultimate” is not quite readable because the obfuscation is believed to be applied. Some core methods which control the logic flow of screenshot taking are missing. Clues could only be found by analysing package structure, libraries imported and source code from the remaining classes.

**ASL Library & Static Analysis of Socket Channel**

Obfuscation cannot perfectly hide everything in the decompiled source code. After carefully reading through the source code of “Screenshot Ultimate”, we found some hints to shape the mechanism of screenshot taking. For example, there was an address in Android OS partition “/system/bin/fbread” appearing more than once in the obfuscated classes, just like the code snippet (com.icecoldapps.screenshotultimate.android) shown below:

```
for (;;) {
    if (!$this.k) {
        if (!$this.l) {
            break label470;
        }
        throw new Exception($this.m);
    }
    t = "system/bin/fbread";
    break;
}
try {
    Thread.sleep(200L);
}
catch (Exception paramContext) {} 
```

The occurrence of “/system/bin/fbread” suggests that this app was very likely taking screenshot by reading image data from framebuffer, which is commonly expressed as short writing “fb” in Android development.

Many apps take screenshots by using a library called Android screenshot library (known as “ASL”\(^8\)). A comparison has been done between the native executive provided by both ASL and “Screenshot Ultimate” to validate that presumption. The exact equivalence in file checksum value among two native executives as shown in Figure\(^6\) indicates that the “Screenshot Ultimate” is one of the third party apps that take screenshot by invoking ASL APIs on unrooted devices.

\(^6\)http://www.apkmirror.com/
\(^7\)Apk Extractor could be downloaded at https://play.google.com/store/apps/details?id=com.ext.ui&hl=en
\(^8\)Android screenshot library is available at https://code.google.com/archive/p/android-screenshot-library/downloads
The ASL enables Android developer to write screenshot app without root requirement. Once the user follows the instruction and executed the native executable file by running the given scripts, proxy with shell permission could help user take screenshot which the application has no privilege to do so. Take “Screenshot Ultimate” as example, user could just click the “Screenshot” button when user want to take screenshot of his/her device, then the app send the screenshot command to the proxy running in background via socket channel, following by the proxy as a process named “screenshotultimatenative1” reading the current hardware framebuffer, converting to the image format and saving to the specific location.

Hence the protocol of communication between app and proxy won’t be difficult to be sketched out. After writing a demo app using ASL API and running on the test devices, the result of screenshot taking was proved to be same as “Screenshot Ultimate”. The communication protocol to take a screenshot by invoking ASL library was shown in Figure 7.

2) App II – “No Root Screentshot It”:
Similar with the “Screenshot Ultimate”, “No Root Screenshot It” is another screenshot app does not require rooting the devices. However, what makes “No Root Screenshot It” unique is the strong security and protection enforcement have been done during the development. Obfuscation has been conducted onto both service activator and APK. Meanwhile, the communication channel between app and proxy has also been protected by using some identification trick like a password, which will be pointed out later.

Analysis of ADB Channel
The service activation has been done by executing a .Net application named “Screenshot It Enabler” rather than simply running a batch script. Therefore the analysis of the service activation will be more complicated because it is involved with the decompilation of .Net application. Moreover, the script file was not found in the enabler’s package, which means it has been packaged into the APK and the purpose of the enabler is just to run the “shell” command to execute it.

In this project, a .Net decompilation tool named “JetBrains dotPeek” has been used to do the reverse engineering of the activation tool. Even though the enabler application has been obfuscated, some variables and C# code logics could still be recovered after the decompilation. The scripts to enable the proxy has been unveiled by observing the C# code from the decompilation result. In addition, the file name of the script could be confirmed as “screenshoot” in this way. Then the script file’s location could be easily found by browsing the file manager on a rooted phone, or by decompiling the APK file to search the file name. The code snippet providing key clue is indicated below.

```
startInfo2.FileName = "adb.exe";
startInfo2.Arguments = "shell /data/data/" + this.b + "/screenshot daemon";
```

Static Analysis of Socket Channel
After clarifying the ADB communication to activate the proxy, the following steps will focus on discovering the communication between app and the proxy, thereby obtain the command(s) to control proxy to take screenshot at any occasion. On the APK side, obfuscation has been done down very strongly onto both class names and variable names, which makes it impossible to observe the entire protocol by just reading the decompiled Java code. It is cleared that the class named ScreenshotService is in charge of the communication with the proxy but the code is not as readable as the previous one Screenshot Ultimate. What makes things worse is that one magic number, 89234820, being found and being referenced multiple times by reading through the assembly code of ScreenshotService class. For this reason, it is proved that there is a great possibility that the app having (1) multiple communication session with proxy to take a screenshot; and/or (2) an authentication trick to indicate the app’s identity, which might be the reason of the existence of the magic number 89234820.
Dynamic Analysis of Socket Channel

Unlike what we have done in case study of app I, only static analysis is not adequate to find the protocol that used for communication between “No Root Screentshot It” and proxy. In order to unveil what kind of command that the app has sent to the proxy to take screenshot and how they interact with each other, a dynamic analysis technique called “hooking” was brought in this project.

Hooking system APIs on Android could be enabled by using a framework known as Xposed on a rooted device. By reading the decompiled code during the static analysis, the communication between proxy and app has been found conducted through socket channel of Android. Therefore, the monitoring of the socket channel during the communication between app and proxy could be done by writing a module based on Xposed framework which hook all the socket channel related packet data IO functions.

| Class Name            | Method Name     |
|-----------------------|-----------------|
| screenShotServiceClass | takeScreenshot  |
| screenShotServiceClass | read            |
| screenShotServiceClass | write           |

1 The takeScreenshot was hooked to indicate the beginning of communication.

After deploying the Xposed module named “hookNoRootScreenShotIt”, all the necessary data have been logged and printed out during the IO operation of the socket channel. The control flow of the app and proxy communication has been finally unveiled and shown in Figure 9.

Proxy Service Identification

Similar with those two screenshot apps we have analysed previously, this app III also requires users to complete the proxy activation before the app unlocking the record function. However, this activation process on Windows OS is executed by running an exe file, and the same process on Linux OS is also started by a well compiled and packaged jar file, which makes the analysis on the proxy activation very difficult.

TABLE III

| Methods Hooked in App II Dynamic Analysis |
|------------------------------------------|
| Class Name                          | Method Name         |
| screenShotServiceClass               | takeScreenshot†     |
| screenShotServiceClass               | read                |
| screenShotServiceClass               | write               |

† The takeScreenshot was hooked to indicate the beginning of communication.

Decomposition

By now we have still yet to connect all the clues found in that last step because the port(s) listening did not tell us which services they are belonging to. For that reason, the decomplation is needed for the further analysis.

Firstly, the APK file has been extracted out of the device, and then been decomplied into samli code. The clues that recently found, two port numbers 7938 and 7940, could be searched within the source code to locate the key classes we will analyse on. As the searching result of keyword 7938 shown below, we found the variable name which bearing that port number, namely “video_port”. Similarly, the port 7940 has been found in variable name “input_port” recorded in same xml file.
Next, we continued searching the occurrence of two variables “video_port” and “audio_port”. After filtering from the search result, we preliminarily confirmed that the code reflecting the control flow and data flow was located in class “RecordService” and “Projection” separately.

Take the communication between video server and app as example, the core function in charge of the communication flow is supposed to be “videoWrite”, which located in line 1038 in the smali code of RecordService. This videoWrite method has been called many time once after the occurrence of the constant string with all letter being capitalised, which is suspected to be the command sending to the server. Moreover, by browsing through the smali code, a method named “openSocket” has been called within the class RecordService, which helps us to come up with a presumption that the protocol we are going to discover was achieved through socket channel.

```
.line 1038
const-string v1, "AUTH 
invoke-virtual (p0, v1), Luk/org/invisibility/recorder/service/
  ->RecordService;
  ->videoWrite(Ljava/lang/String;)V
```

```
.line 1036
const-string v1, "127.0.0.1"
invoke-virtual (p0, v1), Luk/org/invisibility/recorder/service/
  ->RecordService;
  ->getResources();
  ->getInteger(I)I
```

Dynamic Analysis on Socket Channel

Like what we have done onto the app II, hooking by exposed is always considered the most convenient and effective way to sketch out the complete control flow and data flow of the protocol. The target function to be hooked has been confirmed during the previous analysis, which is “videoWrite” located in class named “RecordService”. In order to find as many details about the protocol as possible, some other methods located in the same class of “videoWrite” have also been hooked and shown in Table IV. With the information obtained from the output logs of methods’ hooking, the protocol of the communication between video server and app to start screen recording has been found and displayed as Figure 10.

| Class Name | Method Name |
|------------|-------------|
| RecordService | videoWrite |
| RecordService | writeCaptureOption |
| RecordService | startListen |
| RecordService | stopListen |
| RecordService | startCountdown |
| RecordService | startRecord |
| RecordService | stopRecord |

Poaching the Passcode

A sixteen-digit-long string ce2757a06d455af2 as showed in Figure 10 grabbed our attentions because it was presumed to be the authentication code, or password for short, according to the location of its occurrence. Nevertheless, it has not yet been confirmed to be a string constant or a dynamic changing string so far. In order to clarify the nature of that password, a series of experiments has been conducted.

Firstly, we close the app after taking a screen recording video clips, then re-opened it and took another screen recording. Hooking logs shown in logcat console showed that the password didn’t change. In that means, the password is independent of app’s life-cycle. We have repeated the above steps many times and all the results proved the conclusion is correct. Next, we killed all proxies related to this app and did the service activation again. The purpose of this experiment is to determine if the password is independent of the life cycle of proxies. If answer is yes, that means this password is generated by one of the proxies, and will keep updating each time re-activating the proxies, such as rebooting the device. On the contrary, the password would be confirmed to be a constant, which is same as the magic number in the case study of app II, if there is no evidence indicating that the password itself varies with different instance of proxy. After many times repeating the above steps, the password was found changed each time we re-activate the proxies.

Since the password is proved to be generated by proxy, there must be a place that the proxy stored the code at somewhere that the app with user privilege could access and read. This idea was generated by recalling the analysis on a backup app called “Helium” in paper of Bai et al. [23]. After a series of searching initiated by this idea, we finally located the password in a log file named videoserv.log under the directory data/local/tmp, and luckily found the first occurrence of the current password was always after the word “AUTH” in the log file. For that reason, the attackers could all along hold the current password by writing a simple code snippet on Android to read the log file and then extract the string at given location.

```
12618: ready (1408)
12618: client_read_fd: 4
```
IV. EXPLOITATION

The processes of privilege escalation in three apps have been completely unveiled during the analysis and described in last section. Theoretically speaking, if there is a malicious app being installed on your device while the proxy running in background of OS, behaving exactly same as the original screenshot app, it can unperceivably steal the screenshot of your device at any time on any occasion. In this project, the app II is chosen to be conducted an exploitation to prove the privacy vulnerabilities of ADB workaround.

Unlike the back-up app named Helium mentioned in the paper of Bai, et al. [23], app II doesn’t implement a dynamic identity schemes. That means the exploitation of app II in this project, could be conducted in a very straight forward manner. Besides that, the malicious app could also co-exist with the genuine App. In this section, the exploitation processes of app II will be introduced and the outcome of exploitation will be shown at the end of this section.

An app named “exploitNoRootScreenshotIt” simulating the malicious exploitation of the app II had been implemented for the demonstration purpose. In that exploitation app, there are in total four messages being organized into two lots and sent out to the localhost on port 6003 through Android’s socket channel. The first two messages are used for the configuration purpose, and the last two messages are sent out as screenshot taking command once the acknowledgement of first batch messages have been received from the proxy, which is “screenshotService” running in the background. The entire process was displayed in Figure 11.

The screenshot obtained is converted to a bmp file under the sub-directory named “temp” 10. The access permission of that folder was set as read-only to the user group. Therefore, once the screenshot has been taken by the proxy, the exploitation app could access to the newly captured screenshot located in the “temp” folder and make a copy to the target location such as folder under external storage “/sdcard/hack_screenshots/”.

The screenshot image is renamed according to the capture time to avoid being overwritten and convenient maintenance.

10The full directory path is /data/data/com.edwardkim.android.screenshot/itfullnoroot/temp
at the same time. The exploitation has been tested on two devices (one Nexus 7 and one Xiaomi Rednote 3G) and worked well just like the genuine app “No Root Screenshot It”. This exploitation could even be further designed and programmed to take screenshot automatically with specific frequency without any notice of user, the user’s privacy could be consequently exposed to attacker.

V. RELATED WORK

There are some previous studies unveiling the security risk of ADB workaround despite it is considered much safer than device rooting. Security concern of ADB workaround mainly comes from the difference between roles of proxy and application on Android OS. In this project, these risks could be summarised into two types –

(1) whether other apps could send command to the opening proxy; and –
(2) whether the communication between app and proxy is properly protected if the scenario of (1) is possible to happen.

The description of the first kind of security concern could be found in the paper written in 2014 by Lin, et al. The communication channel between the application and its ADB proxy relies on network sockets without any protection enforced. For that reason, once an ADB proxy has been activated, any application has the privilege to communicate with it and even request service from it at any time without restrictions. This vulnerability gives attackers a chance to analyse the protocol of such communication and build a malicious application to request service from ADB proxy exactly as same as what genuine application does [12].

Some developers have realised the fact that the communication channel between the application and ADB proxy may be risky, and therefore implemented some authentication routines to strengthen security. However in the paper of Bai, et al, it was proved that such authentication was ineffective as long as the reverse engineering and analysis being feasible on given application. What developer can do to secure the communication is only applying some basic authentication since there is no way to enforce strong protection onto the socket network. That authentication is usually very weak in front of analysis [23]. Some application like “Helium”, a backup/restore application mentioned in the paper written by Bai, et al., has been found using protection during the communication between application and ADB proxy. ADB proxy requests a password that sent out from a specific process to provide service. Unfortunately, vulnerability was found in the protocol of password distribution. The password generated each time when ADB proxy being activated, and it is independent of app’s life cycle. In this way, the proxy has to find a place that readable by apps executed with user group privilege, save the password into a file and waiting for app to read from it. This life cycle inconsistency makes adversary possible to find the current using password and thereby exploit the genuine app by writing another app following exactly same protocol in communication with the ADB proxy.

VI. CONCLUSION & FUTURE WORK

In this project, we have come up with an approach to find security vulnerabilities of ADB workaround. By conducting case studies on three different apps, we found most of apps using ADB workaround have risk of being exploited. Followed by case studies, an exploitation has been done successfully, which proves and verifies the approach we have summarised in the analysis.

This project could be treated as not only a support to those previous studies on the similar field, but also a reference to the Android developers. The successful exploitation in this project is sufficient to show that the security and protection enforcement of the Android apps during design and development periods is crucial to users’ privacy and doomed to be an endless and challenging journey.

Some suggestions on Android development have been summarised throughout the study and research in this project. The Android developers are advised to raise security awareness and take some security practices into account when implementing the functionality based on ADB workaround, including:
1) **Access control for socket channel communication.**
An access control protocol is strongly advised to be implemented on both app and proxy sides. It could be like exchanging of passcode, to enable the proxy to validate the identity of app. A good access authentication should not be exploited by analysing on solely one side among app and proxy. In this way, the proxy could reject to provide service when the command received through socket channel failed to validate the identity of app. The objective of this advice is to solve the problem (1) mentioned in Section 5.

2) **Identification for application.** Only the access control is not enough to ensure the security when facing to the second problem described in Section 5. One possible solution for this issue may be writing a *handshake* process in the proxy implementation, to make both app and proxy exchange their authentication. And the ADB proxy will execute the commend only after a successful validation. Thus the proxy service will only accept the command sent from the exactly same app. Once the app is removed and re-installed, regardless of genuine app or malicious app, another handshake validation will be required thereby to ensure the ADB proxy will not be misused.

**References**

[1] IDC, “Smartphone OS Market Share, 2015 Q2,” http://www.idc.com/prodserv/smartphone-os-market-share.jsp 2015, online; Accessed: 2016-03-14.

[2] A. Rubin, “There have been half a billion android activations to date, with over 1.3m added every day,” https://twitter.com/Arubin/status/245663570812100608, September 2012, online; Accessed: 2016-03-14.

[3] IDC, “Worldwide smartphone growth expected to slow to 10.4% in 2015, down from 27.5% growth in 2014, according to IDC,” http://www.idc.com/getdoc.jsp?containerId=prUS25860315 2015, online; Accessed: 2016-03-14.

[4] K. Lucic, “Over 27.44% Users Root Their Phone(s) In Order To Remove Built-In Apps, Are You One Of Them?” http://www.androidheadlines.com/2014/11/50-users-root-phones-order-remove-built-apps-one.html November 2014, online; Accessed: 2016-03-14.

[5] N. Provos, M. Friedl, and P. Honeyman, “Preventing Privilege Escalation.” in *USENIX Security*, vol. 3, 2003.

[6] Google, “System and kernel security,” https://source.android.com/security/overview/kernel-security.html, online; Accessed: 2016-03-18.

[7] Google, “<permission>, Android Developers,” http://developer.android.com/guide/topics/manifest/permission-element.html online; Accessed: 2016-03-18.

[8] A. Shabtai, Y. Fleidel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, “Google android: A comprehensive security assessment,” *IEEE Security & Privacy*, no. 2, pp. 35–44, 2010.

[9] Wikipedia, “Rooting (Android OS) - Wikipedia, the free encyclopedia,” https://en.wikipedia.org/wiki/Rooting_(Android_OS) February 2016, online; Accessed: 2016-03-18.

[10] E. Ravenscroft, “Rooted vs. Unrooted Android: Your Best Arguments,” http://lifehacker.com/rooted-vs-unrooted-android-your-best-arguments-1599101019, July 2014, online; Accessed: 2016-03-18.

[11] M. Casserly, “How to root Android Marshmallow, Lollipop and older versions of Android: The beginner’s guide to rooting, risks and benefits. How to install the latest version of Android, and how to install custom ROMs including CyanogenMod,” http://www.pcadvisor.co.uk/how-to/google-android/how-root-android-phone-tablet-unroot-summary-3342120, January 2016, online; Accessed: 2016-03-20.

[12] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang, “Screen-milk: How to Milk Your Android Screen for Secrets,” in *NDSS*, 2014.

[13] Wikipedia, “Reverse Engineering - Wikipedia, the free encyclopedia,” https://en.wikipedia.org/wiki/Reverse_engineering February 2016, online; Accessed: 2016-03-23.

[14] G. Nolan, *Decompiling android*. Apress, 2012.

[15] J. Freke, “smali/baksmali - GitHub,” https://github.com/JesusFreke/smali online; Accessed: 2016-03-23.

[16] Google, “Proguard,” http://developer.android.com/tools/help/proguard.html, online; Accessed: 2016-03-23.

[17] GuardSquare, “Dexguard,” https://www.guardsquare.com/dexguard online; Accessed: 2016-03-23.

[18] A. S. Constantinescu, “Ensuring privacy in the android os by hooking methods in its api,” *Journal of Mobile, Embedded and Distributed Systems*, vol. 7, no. 3, pp. 107–112, 2015.

[19] rovo89, “Xposed Module Repository,” http://repo.xposed.info/ online; Accessed: 2016-03-23.

[20] rovo89, “XposedBridge Development tutorial,” https://github.com/rovo89/XposedBridge/wiki/Development-tutorial online; Accessed: 2016-04-02.

[21] M. Hofmarcher, M. Strauß, and W. Narzt, “Cross-platform end-to-end encryption of contact data for mobile platforms using the example of android.” 2014.

[22] C. Hoffman, “How-To Geek: How to Take Screenshots on Android Devices Since 4.0,” http://www.howtogeek.com/121133/how-to-take-screenshots-on-android-devices-since-4-0, June 2012, online; Accessed: 2016-03-20.

[23] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo, “All Your Sessions are Belong to us: Investigating Authenticator Leakage through Backup Channels on Android,” in *Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015.

Appendices refer to next page...
### APPENDIX A

**ABBREVIATION**

| Abbreviation | Full Word/Phrase | Abbreviation | Full Word/Phrase |
|--------------|------------------|--------------|------------------|
| ADB          | Android Debug Bridge | IPC          | Inter process communication |
| API          | Application Programming Interface | JVM          | Java Virtual Machine |
| APK          | Android Application Package | NFC          | Near Field Communication |
| App          | Application Software | OS           | Operating System |
| ASL          | Android Screenshot Library | PC           | Personal Computer |
| DEX          | Dalvik Executable | ROM          | Read-only Memory |
| DVM          | Dalvik Virtual Machine | UI           | User Interface |
| IO           | Input/Output | USB          | Universal Serial Bus |

### APPENDIX B

**STATISTICS OF SCREENSHOT APPLICATION**

#### TABLE V

**STATISTICS OF SCREENSHOT APP ON GOOGLE PLAY STORE**

| Name                        | Unrooted Method | Size | Identity Package Name                      |
|-----------------------------|-----------------|------|--------------------------------------------|
| Screen Capture - Sigourney  | Hardkey         | 5.2M | com.mobilescreen.capture                   |
| Screenshot Easy             | Hardkey         | 5.2M | com.icecoldapps.screenshoteasy             |
| Screenshot Ultimate         | ADB             | 3.2M | com.icecoldapps.screenshotultimate         |
| Screenshot Capture          | Hardkey         | 3.1M | com.tools.screenshot                      |
| NoRoot Screenshot Lite      | N.A.            | 545k | com.mobikasa.screenshot.lite              |
| Screenshot and Draw         | N.A.            | 1.1M | com.conditionelta.screenshotanddraw.trial   |
| Screenshot                   | Hardkey         | 2.4M | com.enment.screenshot                    |
| Screenshot                   | Hardkey         | 1.2M | com.geekslab.screenshot                   |
| Screenshot                   | Hardkey         | 4.86M| com.ondice.screenshot                     |
| Screenshot                   | N.A.            | 2.3M | com.geeksoft.screenshot                   |
| Screenshot ER Demo          | N.A.            | 3.2M | fahrbot.apps.screenshot                   |
| No Root Screenshot It       | ADB             | 838k | com.edwardkim.android.screenshotitfullnoroot |
| Screenshot It               | N.A.            | 840k | com.edwardkim.android.screenshotitfull     |

1 N.A. indicates that application only work on rooted devices.
2 only compatible with devices made by some fixed manufactures.

### APPENDIX C

**CODE SNIPPETS OF HOOKING APP II**

```java
XposedBridge.log("Loaded app: " + lpparam.packageName);
Log.d("hookEdward", "Loaded app: " + lpparam.packageName);
if (lpparam.packageName.equals("com.edwardkim.android.screenshotitfullnoroot")) {
    XposedBridge.log("Screenshot App Found : " + lpparam.packageName);
    Class<?> screenShotServiceClass = XposedHelpers.findClass("com.edwardkim.android.screenshotit.services.ScreenShotService", lpparam.classLoader);
    XposedBridge.hookAllMethods(screenShotServiceClass, "read", new XC_MethodHook() {
        protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
            XposedBridge.log("Before Hooking (r) -- param.args[0] " + param.args[0]);
            XposedBridge.log("Before Hooking (r) -- param.args[1] " + new String((byte[]) param.args[1]));
            XposedBridge.
            log("Before Hooking (r) -- param.args[1] " + new String((byte[]) param.args[1]).length());
        }
        protected void afterHookedMethod(MethodHookParam param) throws Throwable {
            XposedBridge.log("After Hooking (r) " + param.args[0] " + param.args[1]);
            XposedBridge.log("After Hooking (r) " + param.args[1] " + new String((byte[]) param.args[1]));
            XposedBridge.
            log("After Hooking (r) " + param.args[1] " + new String((byte[]) param.args[1]).length());
        }
    });
```
```