Android rooting methods detection and evasion

Hi all his introduction t the goal of this is to understand how I bypass root detection while pen testing Android apps. They used rootbeer libraries to protect the app to run on a rooted device. In this, we will look at a simple technique to bypass root recognition. There are many modules available to bypass root detection. I used the xposed module to bypass the root detection but it failed so I decided to manually try. What is root recognition? Root detection is techniques that allow the developer to control the app to only work on the non-rooted layout. Many applications are rooted in the scheme because security is not implemented. What's the xposed module? Xposed is a framework that works on rooted devices. It will provide various modules that allow the user to control Android devices. There are also few modules that help escape the logic of root detection that is written in the app. Step by step approach to bypass root detection in Android software. 1. Reverse engineering .apk dex2jar. dex2jar: This is a small tool that allows to convert .apk to jug command file: dex2jar android .apk 2. Search for Logic Root Detection. When we reverse the file we need to open the file in JD-GUI we need to find the file where Logic detects the defined root. 3. Reverse apk with apktool. apktool: a tool for reverse party engineering 3, closed, binary android apps. It can decode resources into an almost original form and rebuild them after doing some corrections. It also makes it easier to work with an app because projects like file and automation have some repetitive tasks like making apk, etc. Command: apktool d -r android.apk 4. Code change that includes Logic's rooted device detection since we have to reverse the code using the full code apktool is available for modification in smali language. We can find all the files that contain Logic Root Detection. Open the file and change the name such as su, superuser, com.rootcloak to something random. See screenshot below for reference. 5. Rebuild the program with the modified code. We can find the rebuilding code to the dist folder in the current directory. command apktool b folder_name 6. Sign the apk file. After rebuilding all folders with apktool we need to sign apk using .jar. Register .jar help us sign the app with an Android test certificate. Command: java -jar sign android.apk Regards, Sanket Solanki (aka m0nkeyshell) Special Thanks to Sachit Bhatia (aka Thunder) & Chirag Savla (aka 3xpl01tc0d3r) for motivating me. In this article, we will look at the techniques used by Android developers to determine if the device on which the app is running is rooted. There are a good number of benefits for an app in diagnosis if it is running on a rooted device or not. Many of the techniques we use for pentest Android software require root permission to install various tools and therefore compromise the security of Many Android applications do not run on rooted devices for security reasons. I have seen some banking applications check for root access and stop running if the device is rooted. In this article, I will explain the most common ways implemented by developers to determine whether the app is rooted or not, and some circumvention techniques to run the app on a rooted device. [Download] common techniques for detecting whether the device is rooted let's start with the most common techniques that are in the most popular applications to detect whether the device is rooted. Once a device has been rooted, some new files may be placed on the device. Checking those files and packages installed on the device is one way to find out if a device is rooted or not. Some developers may run commands that are only accessible to root users, and some may be looking for directories with top B permissions. I have mentioned a few of these techniques below. Superuser.apk the most common package many apps are looking at root detection. The program allows other applications to run as root on the device. Many applications follow applications with specific package names. An example is shown below. The image above shows a package called eu.chainfire.supersu. I have seen some apps from the Android market approved if any apps are running by chainfire on the device. There are some specific applications that only run on rooted devices. Checking for those applications would also be a good idea to detect if the device would be rooted. Example: Busybox. Busybox is an application that provides a way to run the most common Linux commands on your Android device. Run su and id commands and look at the UID to see if the root is rooted or not. Check build tag for test keys: This check is generally used to see if the device is running with a custom ROM. By default, Google gives a 'release key' as its label for stock ROMs. If something is similar to key testing, then it is likely to be made with a custom ROM. Looking at the shape above, it's pretty clear that I'm using android shares of Google. The techniques listed above are just a few examples of what a developer might be looking for in order to identify if a device is rooted. But there could be other ways around to detect the root that is not mentioned here in this article. Bypass Root Detection – A demo in this section, we'll see how to bypass one of the root detection mechanisms listed above. Before we begin, let's figure out the ground. Download the application from the download section of this article and run the app on your device and click the Button Is My Device Rooted? Click. Since I'm running it on a rooted device, it says the device is rooted. Now our job is to trick an app that doesn't have this device rooted. Now, let's get started. Understand the performance of the program let's first understand that This app is checking root access. The code inside the program is doing a simple check for Superuser.apk is.apk As below screenshot shown. As we can see in the figure above, the app is checking whether there is a file called Superuser.apk inside the /system/app/ directory. If it exists, then we display the Device Rooted message. There are several ways to bypass this diagnosis. Let us first come back manually if the device is run by su and id root commands. Let's see if the Superuser .apk on track being reviewed by our target app. To bypass this check, let's rename the Superuser.apk app to Superuser0.apk as below screenshot shown. As we can see in the figure above, we may receive an error message that says it is a read-only file system. We can change it to read the writing as shown in the next step. Change permissions from Read-Only to Read-Write: The following command will change the file system permissions to Read-Write. Now, if we rename the file to superuser0.apk, it works fine as shown in the figure below. If we are already back to the app and click the Button Do my device root? , it shows a message as shown in the figure below. This is just one example of how to bypass root recognition if not implemented properly. Applications may use some sophisticated techniques to prevent attackers from bypassing root detection. Techniques will change depending on how the developer reviews to access the root. For example, if a developer uses the following code to check the root, the method of circumventing the diagnosis is different: Runtime.getRuntime().exec(su); In this case, we need our own binary sue and superuser apk. Developers concluding that preventing users from running their apps on rooted devices can be a good idea from a security perspective, but it's really annoying for a single user who isn't able to run the app just because their device is rooted. Rooting techniques can be easily bypassed most of the time, and it is highly recommended that developers should use sophisticated techniques to prevent attackers from bypassing their validation controls. We offer an arms race between root detection and rooting escape. We examine different methods for detecting the rooted device at both Java and native levels and counter-attack assessment of the major hook tool. For this purpose, an extensive study of Android rooting has been conducted which includes device rooting techniques and invisibility for detection of mobile antimalware product. We then analyze volatile loopholes and in turn enhance our root detection tools. We also apply escape techniques on the rooted device and compare our work with 92 popular root checking applications and 18 banking and financial applications. The results show that most of them are insufficient and can escape through API hooks or static file renamings. In addition, more than 28,000 Android applications have been analyzed and evaluated in order to detect rooting features in recent years. Our study shows More and more rooting has become common as an inevitable trend, and it raises huge security concerns about detection and evasion. As a proof of concept, we have released our root detection app to the Google Play Store to show the work presented in this article.1. IntroductionAndroid has dominated the mobile market for 7 consecutive years and this operating system will continue to do so. According to an IDC report in August 2016 [1], Android has led the smartphone market share over the past 4 quarters with the largest share of 87.6% in Q2. While Google's code change won't be immediately available to the public, the release of the new version still includes most of what the community needs: factory images, source code, OTA distribution channels, and APIs. The unreality of openness is the main reason why android os gain popularity so quickly. Android is regarded as a very complex ecosystem; Each device is a combination of different software (open source and closed source), different hardware (screen size, manufacturers), and various distributors. That makes exploiting globally impossible but also makes auditing work difficult as checking every device and its software is a huge amount of work. The security update process for a specific Android device can be summarized as follows: (1) security flaws found in the operating system, (2) Google patch release, and (3) OEMs (original equipment manufacturers) adopt the patch and including it on your custom made devices. The whole process can take months to finally fix the vulnerability. This fragmentation nature of Android makes this operating system a fruitful land for invaders [2]. Even if Android devices come with a lot of customization, they are still limited in some set of APIs and custom makes up of OEMs [3, 4]. Many advanced users often tweak their devices with access to the highest rating, also known as root or super user, and are able to manipulate the operating system into any custom build of their choices. There are many benefits that encourage users to root their devices [5], some important reasons can be mentioned as follows: (1) Get rid of bloating (the software is pre-installed by vendors), (2) do full system backups (rooting devices only allow backup of user data) , (3) update the operating system in the pre-release of OEMs (including new features and especially security patches), and (4) obtain new functionality brought by custom ROMs, as well as the use of root-related applications. According to Google's report on Android security since 2014 [6–8], the term rooting has been frequently cited as one of the most important factors in the threat of escalation of points. Rooting applications were added to PHAs (potentially harmful applications). Despite being classified as harmful, these applications are not considered malicious until their behaviors are sufficiently disclosed to the user. As shown in the reports, various tools have been used by Google to Root. In a 2014 report, Google Verify Apps detected rooting apps installed on 0.29% of devices (outside of Google Play), and a fraction of devices with PHA installed with rooting applications included double. In a 2015 report, Google used the anomeal correlation engine to detect rooting applications and other PHAs, resulting in 0.32% more than 1 billion devices that had rooted applications installed (page 36). In 2016, Google's certificate served 25% over 2015, which had nearly 200 million requests a day; the intended user rooted the facility to include 0.346% of all installations. As confirmed applications, anomaly correlation engines, and certificates all owned by Google's SafetyNet APIs, we discussed google's internal product in subsequent sections, as well as some studies on rooting in Android.In this work, we use different terms to refer to rooting-related applications. (1) One click Rooting

programs (or just rooting programs). They are applications used for rooting purposes. By exploiting the disadvantages of the operating system to perform concessions, these programs are strictly banned by Google in the Play Store and classified as PHAs (potentially harmful applications). (2) Root Manager Apps. They are apps that act as safe gateways for root points. One of superSU's best-known root managers is by chainfire [9]. (3) Root Apps. This term is not confused with one-click rooting programs. (4) Root Checking Apps. In general, they can have modules embedded within financial applications or anti-malware products. Due to the historical reason, users and developers continue to be interchangeable using the terminology of rooted and rooted apps meant to be (1), (3), and even (2). We do our best to avoid these terms and different programs throughout the article by providing explanations in the field. Rooting since 2009 has been a controversial issue in the mobile world and a return to the birth of the Unix operating system [10]. The rooting action stands at the boundary between user experience and device security. Rooting escape allows applications to run on rooted devices, despite not having to do so because of security. While banking and security companies are trying to harden their root checking modules, a large community, however, is doing photos [11]. Strong support from both parties makes it difficult to end the controversy. Inspired by an arms race between rooting and anti-rooting, we did several studies on the Android operating system. Starting with how rooting works and the rooting risks to why Google and banking, financial companies often raise concerns about this issue, we try to understand the current state of volatile and detection techniques and then re-look at root detection work on Android 4.4 and later versions. We also do a large-scale analysis Different data sets of Android applications in order to better understand the features of Android applications in recent years. Our contributions to this study can be listed as follows. (1) We review current rooting methods for Android devices (Part 2.2) and techniques to avoid being detected by banking applications or anti-software product, targeting android version 4.4 and later ones. We conducted a review of 110 root checking applications (92 root checking applications and 18 banking/financial applications) and show their current implementation against rooting. While 89 root checking programs can be escaped through java API hooks, we discovered 18 banking/financial programs that implement native code checks and are therefore immune to this volatile technique. Then we try to customize the current system and successfully build 10 of the 18 apps that run in the rooted environment. This effort helps us to confirm that with the same approach, any root checking app can be escaped because it is just a matter of time. (2) Based on the features of the Android operating system and the rooting behaviors of applications, we collect a set of possible APIs and embedded commands and implement an Android app to do the rooting work of detection and build it as an Android app, which we have to google play store recently released (S4URC Root Checker in Figure 5) [12]. (3) We conducted large-scale analysis of over 28,000 Android applications to evaluate other aspects of the study, which included 7200 apps from the Poly Store and more than 21,000 types of malware we collected from different websites. These programs are separated and the source code was analyzed to reveal their behaviors about rooting. This will help us better understand malware behaviors as well as the ability to measure the prevalence of score-increases by rooting on Android. Detailed analysis will be discussed in section 6.The rest of this article can be summarized as follows: Section 2 briefly provides some context on rooting and increasing points. Section 3 presents works related to Android rooting as well as security concerns and solutions in paperwork. Sections 4 and 5 discuss the techniques of bypassing the root checks of popular applications and implementing our root check program [12]. The results of the analysis will be presented in section 6, and we will finish our work in section 7.2. Background in this section, we offer some areas of Android security architecture, rootings, and hook techniques APIs Android.2.1. Android Security ArchitectureAndroid OS is designed by stacking various software components on top of each other and securing them from the ground up; Each component assumes that the following components are properly secured. Most codes and dimons are running as normal points with the program's sandbox limitation, versus root points (or supermodel points) that only go to the initial processes for over the machine . The accumulation of security components corresponding to each layer is described as figure 1. A further reference on the design of the Android operating system can be found on the official Android page [13].2.2. RootingRooting's principles, in the smartphone world, are a process to get the highest rating on the operating system [10]. The user rooted in Android (and general Linux distributions) that have UID 0 is disabled in the user to limit malicious programs from gaining absolute control over the system. An application with a root rating can do practically anything, such as disembarking file systems, killing processes, or executing any arbitrary order [14]. The common binary file found in the system partition of each rooted device is the su file, which stands for super user or switch user. Basically su has a uid set flag set and always runs as UID 0. Applications that use roots often try to run this binary file that comes with a goalkeeper Android app (SuperSU.apk or SuperUser.apk) to increase their points to the highest. Rooting can be categorized into two main types: (1) soft root, which is gaining root on a booted system: this technique is completely software-based and can be achieved by exploiting vulnerabilities in the core, in a process that is running as a root or even through a symbolic link [2]. Based on these vulnerabilities, many applications were created to provide users with one-click rooting tools. Rooting obtained by soft rooting is usually temporary, especially on devices that lock the bootloader, as it checkes the integrity of the system every time the device restarts. This requires additional work to remain in root points, namely, unlock bootloader and manipulate recovery mode. Attack vectors that support soft root will be discussed in the next section. (2) On the other hand, hard root requires physical interaction with the device. The main idea is to boot into recovery mode that is separated from the user and perform su installation or replace the entire operating system with the new one that su is installed. On some devices, bootloaders are locked, which makes the rooting process more difficult. Hard root like soft root does not suffer from temporary rooting problem, as the bootloader already opens which turns it into a permanent rooting technique. We show different types of rooting methods in Figure 2. All applications designed to use root privilege, regardless of benign or malicious intent, require running su files either through libraries or directly after obtaining permission from the Goalie program (SuperUser.apk or SuperSU.apk [15].2.3. Android HookingWhen Android OS starts booting, the Zygote process will be started by init.rc script, or more specifically/system/bin/app_process native Damon [16]. Zygote is the parent of all application processes, making it responsible for all your forks. After all classes are loaded, Xposed Framework [17] Expand app_process to add an extra jug to the classpath and call the method to act in the Zygote field. It allows fake method calls and custom code injections to manipulate the program's behaviors. For example, Gajarani et al. [18] use this framework to hide their emulator from detection by advanced malware. ARTDroid [19] is another work targeting art run time that enables sandboxes to analyze apps without escaping. Similar to Xposed, this technique also requires root points for action. In this study, we mainly focus on Java hooking techniques that use Xposed as a tool for rooting escape. Work on the Cydia platform for native code hooks has been abandoned since 2013 [20]; So it's only applicable to Android 4.3 and older versions, which occupy less than 2% of the distribution of Android versions [21].2.4. The risk of rootingAs we discussed one click root applications, they are usually transparent to users. In other words, users have no idea what's under the GUI (graphical user interface) other than what they interact with. So malicious programs can exploit CVEs (vulnerabilities and shared exposures), especially on older devices, to make a severe impact on the system. Dan Rosenberg has discovered the exploitation of the symbolic link attack with the /data/to data/local/tmp (read-write mode). By changing the /data/local.prop file and changing ro.kernel.qemu to 1, it makes the ADB shell (Android Debug Bridge) user switch to root mode [22]. This vulnerability was fixed on Android 4.2.In about apps that use update-attack, when super-user ratings were obtained, these apps were able to update themselves and run arbitrary commands to re-mount the system partition, make a copy, or even destroy almost any data. After saying that, when malicious applications are rooted rated, the system is now compromised. For example, we point to previous studies to prove and show that the proposed security mechanisms do not gain based on the assumption of another secure system when malware gained root power [23–25]. In this section, we will provide some work related to rooting check/escape and increasing points on your Android device.3.1 Articles and studies on the rooting of the exponential rating of et al. [14] conducted a study of 182 programs selected to identify current rooting detection methods. The authors present a comprehensive effect of the classification of rooting methods and suggest different techniques for rooting detection and escape. Seven categories have been carefully studied for root detection, namely checking installed packages and files related to rooting, tag construction, system features, directory permissions, running processes/services/tasks, and shell commands. To conclude the work, the authors say the arms race between root detectors and fugitives is an asymmetric rivalry that favors fugitives. In other words, it doesn't matter how Detectors are, there will always be techniques to evade root detection. The authors then suggest a reliable approach should come from the core territory, which is fiercely immune to most attacks originating from higher layers of the ecosystem. While we agree with the conclusion that rooting always benefits the escapeer compared to the detector, we still believe that there is a big demand for effective root checking applications. Most banking applications that we have reviewed so far employ at least some of the above checking techniques, in order to protect their applications from running in an unsafe environment, which can be abused for further attacks. We re-examine all the root detection techniques listed in the article and propose new methods that have been proven to work on a larger scale than sample data. The finding we come across is that the Cydia platform [20] only supports Android 4.3 and older ones. That means authors hook techniques have implemented other with 98.8% android devices working [21]. Also, as the authors noted in their study, RDAanalyzer is revised to evade the newly discovered detection technique(s). We believe, without the Cydia function, the only way to discover detection techniques in native code is to reverse shared object files, which is time-over and really difficult to achieve completely. In Table 6, we had to do memory analysis to collect library files and keywords, and still we were able to escape some banking/financial programs as described in Table 5. This finding encourages us to develop a root checking program at both Java and native levels in Section 4, which is employed by RootCloak [26] and other similar tools that employ the Xposed Framework [17]. Kim and his colleagues provide an analysis of 76 popular financial programs in Korea to address their self-defense mechanisms that are used to protect application from a deep-rooted or manipulated environment. The authors develop a tool called MERCIDroid to identify the reason between environmental scrutiny and termination of executions. MERCIDroid first seeks environmental information providers, for example, getPackageInfo() and File.exists(), and uses review terms such as command name or su file name or apk file related to the root installed to conclude that the device is rooted. Based on the SDMG (self-defense mechanism graph) made by the contact graph, the tool then detects the execution terminator, which is the API that stops the program (or displays the alarm dialog). The results show that there are 67 out of 73 programs that successfully bypassed root checks. The results, in our opinion, can be greatly enhanced using the techniques mentioned in section 4 of this article. For example, MERCIDroid only depends on little environmental information providers to break their defense mechanisms; su binary file names can be easily changed to avoid detection. System properties and making tags must Add checks to complete. Finally, in the case of native code modification work, the authors claimed to use IDA Pro [28] to bypass lib4 checks but did not provide detailed techniques. To the best of our knowledge, one of the IDA plugins is to be able to patch the binary file IDA-Patcher [29], which is no longer active in development. Nevertheless, the app secured by native code is hard to break; this shows us developing root detection mechanisms using NDK (Part 4). In an attempt to limit the increased scoring attack, a number of works have been proposed. Dvey et al. [30] indicated that the attack can be mounted at the run through the access control mechanism, which allows the use of passing permission. Bugill et al [31–33] conducted heuristic analysis of the Android system to address collusion attacks and solve the wrong vice problem, as well as provide a broad monitoring mechanism for Android. Park et al [34] offer a solution called RGBDroid to protect the system even after root points have been granted to malware. Xing et al [35] shows an increase in points called Pileup in Logic Update that makes malware gain higher points even than signature licenses. Zhongyang et al [36] suggests DroidAlarm to analyze leak capabilities in Android malware. All these studies so far have proven that there is still much work to be done to increase the security of Android.3.2. Google SafetyNet AttestationTraditional rooting techniques are all about modifying the /system partition to install custom script and gain root access. Chainfire, a senior developer of XDA, provides a different approach by modifying only the boot image in order to achieve root points; Systemless rooting is intended as a newer solution for advanced users who want to use services that check their devices without having to root. As the newest update, the technique no longer works for apps that adopt Google SafetyNet (as such, Android Pay) [38]. Google SafetyNet (or SafetyNet Attestation specifically), however, doesn't really target the deep-rooted environment. The tool provides a set of APIs that allow app developers to use them in their apps to check the device's safe mode in general. Google uses a neutral term to describe it as CTS (compatibility test suite). The results returned from the basicIntegrity API and ctsProfileMatch are the most important, because they show the integrity of the device. Another work has shown SafetyNet also aims to root out symptoms like binary sue, and with proper modification, it can be tricked at API level [39].3.3. Rooting OEMs escape reviews nowadays begins to release their devices with open bootloaders or open bootloaders. Restrictions on the use of only major platforms have been loosened over the years as the process of openness and free for mobile customers. Some OEMs like Google even give Training users safely lock bootloader without bricking their phones. By doing so, users can now install the operating system of their choice for a variety of purposes: expanding features, installing new incompatible applications, and speeding up the device with unlocking; rooting plays an important role most of all. Unfortunately, rooting is also the target of malware, which is why some applications from banking to entertainment at least try to recognize it and avoid running if the result is positive to rooting. Escape techniques were born as users need to bypass these checks, and they also happen to help hide malware. One of the most important works on this issue is RootCloak [26]. This framework makes use of the Xposed framework [17] and the Cydia platform for Android [20] to bypass shared checks. As described in Figure 3, Xposed modifies app_process and hooks access apps from the Xposed framework using de.robv.android.xposed.installer/bin/XposedBridge.jar. XposedBridge .jar API in turn gets an object-class Java loader. Because ClassLoader contains a class that needs to be called, it hooks the target method in ClassLoader using the class name and the name of the method the user wants. RootCloak can virtually defeat any root check program in java code, if the class and method name are given. With the prevalence of various reversal tools such as enjarify [40], information can be obtained without problems. RootCloak has been developed as a module of the Xposed framework; For example in the list below, RootCloak tool is installed to hook to PackageManger for the list of applications: findAndHookMethod (android.app.ApplicationPackageManager, lpparam.classLoader, getInstalledApplications, int.class, new XC_MethodHook()By using this technique, RootCloak is able to bypass applications that check for rooting by intercepting ActivityManager, PackageManager, Runtime, and ProcessBuilder. For applications that check specific keywords, filepath can be escaped by returning bogus results instead of sensitive strings such as SuperSU, Root, Fire Chain, and the like, some of which are described in Figure 6. Cydia, which only works on Android 4.3 and older, RootCloak can no longer escape root checking techniques that originate from the native level, embedded in Android applications as shared object files (.so files). Be able to bypass checks from different root checkers, especially advanced modules of banking and financial applications. Explaining how these modules work and analyzing banking and financial applications are discussed in sections 5 and 6.4. Root Checking MethodsDespite the possibility of escaping by the same approach we will soon present in Part 5, we are still seeing the existence of root checking In critical applications, which have been proven to be required by Google Report and several banking/financial applications that we have analyzed. While many applications employ native development kits to bypass API hooks, they are short of outposts and fall into this escape. As we mentioned earlier, 10 of the 18 banking and financial applications only check the existence of the Sue binary file by its name, so escaping is as simple as renaming this binary file by rebuilding it. To encounter this technique, we recommend adapting regular expression for file names and process and service names, as well as checking the amount of the aerated each file (if readable) against our database, which is a set of multiple rooting programs and binary etch values. We apply this type of check almost everywhere in any category below to ensure the effectiveness of diagnosis. In this section, we re-work-checking the rooted device listed in [14] and 92 root check applications we collected from the Google Play Store, as well as suggesting some improvements and directions to enhance the effectiveness of this work. For the sake of completeness, we offer all the existing techniques collected from our study. Native Development Kit (NDK) is a set of tools provided by Google that allows us to use code C and C++ with Android and provides platform libraries that you can use to manage native activities and access physical device components such as sensors and touch input. NDK may not be suitable for many novice Android programmers who need to use only Java code and APIs framework to develop their apps. Even some Android apps can be escaped by api (school) hooks; We realize that RootCloak is still not ready to hook up to native code at the time of this writing. While most of the root checking features in Java can be simplified again in native code, we decide to discuss both at the same time and point out the differences if necessary. To clarify the work, we refer to keywords, processes, damons, closed names and directs in tables 1, 2, and 3.Keywords use daemonsu superuser supersu kinguser kingroot busyboxRooting related to binary and daemons: these are common keywords in xposed chainfirewordKeys rooted devices related to escape programs: to escape the root of the check. In Java code, RootCloak uses the Xposed framework to perform an interception with api callstitanium greenify Strickson kerneladiutorKeywords related to programs that require the name of the rooted devicePackage use case com.jrummy.root.browserfree directory browsingrootcloak xposed inner mountain chainfire to external storage.oasisfeng.greenifiy Android tool toolscom.jrummy.apps.. build.prop.editorcom.grarak.kerneladiutor org.namelessrom.devicecontrolcom.jumobile.manager.systemapp Modifying settings, System Properties, scanning device partitionstericson.busybox Linux commands utilityde.robv.adnroid.xposed.installercom.devadvance.rootcloakcom.devadvance.rootcloakplus Root root Use case/system/bin/system/xbin/sbin/data/data/system/usr/system/bin/.ext likely includes binary su/system/program/data/program/data/dalvik-cache apk files and cacheWithin domain this article we only provide some of the most popular keywords for the sake of intuitive demonstrations. While we still apply static conditions to check rooting, different tactics will be discussed in detail for each technique to enhance detection accuracy. Check for installed packages. PackageManager is installed on Android responsible for manipulating all applications. By querying all installed apps, we are able to get a list of closed names that can be useful for detection. Different types of apps are used to check. (1) One click Rooting Programs. Such applications are not allowed in the Google Play Store due to the risk of exploitation. (2) Root Manager Apps. For name but there are a few, SuperSU, SuperUser, Kingroot, and Kingo apps. (3) Root programs. It is most popular with hundreds of applications in the Google Play Store and in the wild, according to users' demand. We have collected these applications, as well as the amounts of MD5 etch and their package names. The devices that have installed one are obviously prone to rooting. This can also be achieved by running the pm list package command. Check the existence of files. Rooting on different files works correctly: binary su and root related APK files (SuperSU.apk or SuperUser.apk and applications that require root points to work). These APK files won't be required after the installation process, but will be kept by default unless users deliberately delete them. Investigating the existence of certain files is considered as an important factor for root recognition. Our tool tries to check the existence of those files in different directories: /system/bin/and/system/xbin/for binary su; Android TV and NDK provide files (at Java level) and (files) popen (native code) to do the work. One common weakness of root checking applications is that they tend to use static keywords, which makes search less flexible. To improve accuracy, we apply regex (regular expression) to maximize the search range. For example, regex filename_._[] \ \.apk will match any file that looks like these: filename_aa_1.apk or filename_bb_3.apk. In native code, this can be found by JNI (Native Java Interface) file code = fopen(file_path). Check processes, services and tasks. As of Lollipop (API Level 21), getRunningTasks API was no longer available for third-party applications due to leaks of personal information to the caller. So we only evaluate the processes and services running on ActivityManager. We use a list of keywords collected related to the root in order to perform a matching with the process and service name. Kingroot, SuperSU, and Kingo are the most popular tools (Windows and Android applications) appearing on rooted The ones having such keywords are likely to be rooted. We use getSystemService to recover an ActivityManager and interact with system mode and get a list of running processes (RunningAppProcessInfo) and services (RunningServiceInfo). Just as Linux-based Android is, the ps-aux command can also list existing processes and services. Check the execution of shell commands. Android OS has some inherited legacy commands from Linux; Some of them can be run as a normal user; Others can only be executed as roots. We try different commands and check return values to see if the device is rooted based on different factors. Determine the existence of some favorite files or text: by running su, pm grep list packages, ls, cats, ps-x, and the like, ouput can be a complete path leading to binary su, suspicious process (es), text strings containing keywords in Table 1, or the package name in Table 2.In native code, can be system <a_keyword > <a_directory> >full_path_to_a_file>(command_to_be_run) use. It also covers previous cases of process/service reviews and package name lists. Check tag construction and system properties. The rooted device contains different amounts of Build Tag and System Properties when the device has custom built-in operating system. If android.os.Build.TAGS is test keys instead of publishing keys, this indicates an informal build. Using system properties, we can either use APIs to check rooting, readSystem (ro.debuggable) and readSystem Allton (ro.secure), or run getprop grep-wE ro.debuggable ro.secure as a normal user. If the returned values 1 and 0 are sequential, that means an ADB shell will be able to run as the root user on the device's shell. Directory permissions. Rooting sometimes changes permissions (reading, writing, executing, and listing) in certain directories. Table 3 can be used in combination with Java or native code to compare differences with non-native devices. For example, /data/ and its subcategory are immutable by default to read, write, and run by default. If some rooting applications accidentally or deliberately change permissions in these locations to global reading/writing, the user and regular programs will be able to perform operations in them. So far, we have not found any rooting marks based on this check. Simple commands such as ls-l may reveal the current status of a specific file or directory. The work flow of our root check app is provided targeting Android 4.4 and later ones in Figure 4.5. The current root circumvention techniques of AppsBased reviews in the formative study in Part 2 on Android architecture and rooting methods, we notice the correlation between methods (APIs) and objects can be intercepted. The reason is that Android API calls are made from the app layer to the framework (see Figure 1), which can be corrected in when the operating system is compromised. For example, a review program </full_path_to_a_file> </a_directory> </a_keyword> root may be using PackageManager for Package name of all installed Using getInstalledApplications API. By analyzing the source code analyzed from the various root check programs described in section 6, we have a sufficient list of related APIs that are checked for rooting signs (Table 4). Also, studying the rooting applications checked in part 4 will give us a hint of targeting several root-related objects, in the form of files (su, SuperSU.apk, root programs high in Google Play, etc.) or in the form of distinct keywords (chainfire, superuser, runtime.exec(), etc.). As a result, we created a fake tool that targets current root checking applications. To this end, we realize that most of them are poorly designed and vulnerable to volatile techniques such as hooking Java APIs. For example, we manually tested the top 92 applications in the Play Store by providing the root checker keyword (1-a data set in section 6). 89 applications have escaped, 2 applications unexpectedly crashed due to incompatible APIs, and only 1 application failed. Class name Number of apps using these APIs Fraction android.app.ApplicationPackageManager.getInstalledPackages10.61%android.app.ApplicationPackageManager.getInstalledApplications00%android.app.ApplicationPackageManager.getPackageInfo84.85%android.app.ApplicationPackageManager.getApplicationInfo137.88%android.app.ActivityManager.getRunningServices10.61%android.app.ActivityManager.getRunningTasks00%android.app.ActivityManager.getRunningAppProcesses127.27%java.lang.Runtime.exec4929.70%java.lang.ProcessBuilder2515.15%java.io.File5231.52%java.lang.Class.forName42.42%Index Name Evaded 1 Banking App 1 Yes2 Banking App 2 Yes3 Banking App 3 Yes4 Banking App 4 Yes5 Banking App 5 Yes6 Finance App 1 Yes7 Finance App 2 Yes8 Finance App 3 Yes9 Finance App 4 Yes10 Finance App 5 Yes11 Banking App 6 No12 Banking App 7 No13 Banking App 8 No14 Finance App 6 No15 Government 1 No16 Government 2 No17 Banking App 9 No18 Banking App 10 NoPackage name Library name Root detection Finance App 1 libXAS_jni.so/sbin/su /system/bin/su /system/sbin/su /system/xbin/su /system/xbin/sudo /system/app/superuser.apk /system/app/UnRoot.apk /system/app/Nakup.apk /data/data/com.noshufou.android.su /data/data/com/ajantech.app/UnRoot /data/app/com.noshufou.android.su-2.apkGovernment 1 libsmartmedic.so Store rooting_pattern information into smartmedic1.db fileSecurity App 1 libEngineManager.so native.startRootCheckBanking App 1 libBengine.so Check /proc/%d/cmdline for suBanking App 2 libap1.7.8.soCheck su files Check Telephony status Check Wifi status (including MAC address) Check for noshufou and supersuFinance App 2 libap1.7.2.so Check for noshufou and supersuWe continue to follow the case of 18 banking and finance applications that applied more advanced rooting check techniques in native code [41]. Java APIs Hook fails entirely to evade these programs even though we have tried to customize the hook tool as much as possible. We perform dynamic analysis on the memory of each banking application by /proc/pid/mem while running these apps on the rooted device. By doing so, we are able to collect interested strings related to rooting, as described in Table 6. With trial and error, we managed to evade 10 banking applications by renaming Binary Sue to another name (i.e. Flicker) or setting up apk file names to avoid checking the file name. By running that su file with its new name, the root rating is still granted. In other words, banking and financial applications can still escape.6. Analyzing the result in this section, we continue to masterfully root problems by evaluating the multiple data sets that we have collected over the years, ranging from malware in the wild to rooting related to benign apps and apps in the Google Play Store.We first describe our data sets as follows. (1) Applications in the Google Play Store. All were downloaded from December 2016 to February 2017. (a) Root Checking Apps. They include 92 independent root checking apps (free) and 18 root checking modules from banking/financial apps. (b) Root programs. The top 35 apps that request using the root rating in the Google Play Store each have more than 1,0,000 users. These apps enable users to access rich features that are not available in the original build of OEMs (original equipment manufacturers). c) Others. They include 7,200 benign apps from the Google Play Store that are randomly selected from several categories of Google Play (sports, games, training, fun, etc.) to avoid bias. We want to use these programs to measure a fraction of the rooting-related programs. (2) Malicious programs in the wild. 21061 more malware types than VirusShare (2013-2014) [42] with small part (approx. 1%) gained from Contangio Minidump [43] manually. We use this data set to assess the popularity of malware associated with rooting, as compared to the 1-b and 1-c data sets from the Google Play Store.6.1. Rooting applicationsWe reviews selected high results that restored the keyword Search Search Root Checker. 92 root check apps were downloaded in the Google Play Store and were manually tested with and without RootCloak escape. 89 applications have been escaped by RootCloak, 2 applications crashed at startup, and only 1 escape program escaped from RootCloak. As we continued reversing each application, we found that the program could avoid the escape that worked the check module in your native code, thereby being immune to the spread of the hook API. To gain insight into these applications, we carefully review each of the 89 applications using the Xposed framework to hook up to 11 classes as described in Table 4 and Figure 7. Most of them try to check the existence of rooting-related applications (getApplicationInfo), files (java.io.File), and processes (java.lang.ProcessBuilder) and also try to run shell commands through java.lang.Runtime.exec class. As common objects (developed in native code) analysis is outside the scope of this study and these applications have already been failed by Java Hook, we didn't try to check your source code any more.6.2. Banking and finance applications at KoreaRooting Czech are also embedded as a module within banking and financial programs to prevent the manipulative environment. We selectively test RootCloak with 18 applications; They are able to bypass RootCloak promptly and refuse to run in the root environment. We tried to customize RootCloak with Xposed as much as possible but these banking programs employ their checking modules by native code so it is impossible to escape them by expanding the hook API. With trial and error, we finally found a way to avoid detection just by renaming the su binary file to different names. 10 of 18 applications can be escaped with this way (Table 5). Since the disclosure of the reviewed participants may reveal their sensitive information that should be strictly avoided, we decided to hide all 18 real names of 18 bank/financial requests after receiving 2 responses from bank contact points. Parties on the right will be provided with the necessary data if needed. These apps only use native code to check the existence of su files, which is easy to bypass without having to intercept native calls. By renaming the file, we were still able to run the su file successfully and obtained the root rating. But this method will no longer work if the device resets, as all rooting processes will be terminated consequently without being able to recover due to the change of the su file. This issue can still pay off by recompiling su and SuperSU app altogether to update the new binary name. We continued our work by analyzing native libraries of some financial and banking programs to better understand what these programs are doing under those SO files. Their SO files are listed in Table 6. We will not reveal their identities once again for the security issue. It turns out that these programs also check the existence of closed names, which are relatively inefficient because they are all static and can be manipulated to bypass the check. We do not review this case further because it is perfectly possible to customize the rooting process with the name of a new su to evade these banking programs. Custom root apps with new package names and binarys are also able to operate with the new Sue. For what it's worth, each of these banking/financial programs has more than 1 million users, which makes them attracted to attackers.6.3 Other analyses to complete our study, we conducted some basic analysis with the rest of our collected applications. We divided them into 3 groups (i.e. data sets 1-b, 1-c, and 2) such as Table 7 and used predefined keywords to scan through each application. Our goal is to determine the prevalence of rooting-related applications in the Google Play Store, as well as from malware from the wild. The results surprise us with the frequency of using keywords related to the root in benign applications that are much higher (there is 7,200 benign programs, compared to more than 21,0 types of malware). Note that the data set is 1-c (7,200 fresher benign programs) (2016-2017) than data sets 2 (21,061 types of malware collected from 2013 to 2014) but fewer. That means rooting-related applications have become more popular over the years. While the keyword list is not complete and there are many factors to evaluate (as such, some benign applications can also root out check applications by chance, which affects the counting in Table 7), we only consider this analysis as a reference for measuring a fraction of the rooting-related applications and no longer continue. KeywordRoot appsBenign appsMalwarechainfire3913root2837401348libsuperuser020supersu7207superuser1257220shell10367172Runtime342663sudo0431486.4. Cloud-based Root App Download PredictionBased in the generated signatures of malware and root app repository, we extend your work to cloud-based predictions from the downloaded root app to the mobile device. Whenever a user downloads and installs any APK file in the game store or out of the wild, it's our app to send some basic information such as the value of the abrasion and name depending on the cloud system, which in turn matches the database and returns alerts if that is a root app, as in Figure 8. In our opinion, it can help prevent inadvertent rooting from one-click rooting applications and protect the device from any root application in general. In our upcoming work, this will be published with us now rooting an integrated program check (in Figure 5) [12].7. Concluding in this article, we discussed various aspects of Android rooting, including current trends of Android users, escape techniques, and cross measurements. As an open ecosystem, Android attracts a lot of attention from the good and the bad, and may lead to a huge impact due to its large user base that once vulnerabilities are introduced. The rooting process is inevitable because it gives users full control over the devices they have on their own. But, however, a large portion of users are not aware of the potential risk of running in the background. Sooner or later, the detection methods proposed in this study will escape when developers are able to customize their ROM and boot image in Will, which could eliminate the possibility of detailed checks entirely. But in the meantime, we still believe that such detection methods are necessarily required to provide a certain level of secure security to users. Rooting escape and root detection are always in a race that favors the escapeer towards the detector. And in the researchers' view the security review for rooting should be adopted at the core level, which depends heavily on OEMs and Google to effectively prevent such runaway techniques. The authors' conflicts of interest declare that there is no conflict of interest regarding the publication of this article. AcknowledgmentsThis research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC Technology Research Center) support program (IITP-2017-2012-0-00646) supervised by the IITP (Institute for Information & Communications Technology Promotion). The work was supported by a funding from the Institute for the Promotion of Information and Communication Technology (IITP) provided by the Korean Government (MSIP) (No. 2016-0-078, the development of cloud-based security intelligence technology to provide customized security services). MaterialsList supplements from 89 (free) root checking applications in the Google Play Store and the corresponding class name that you check for rooting signs. The Google Play Store has its own sorting algorithm for app rankings. We did not provide any parameters to filter search results. Mentioned.