Once Root Always a Threat: Analyzing the Security Threats of Android Permission System*

Zhongwen Zhang[1,2,3], Yuewu Wang[1,2], Jiwu Jing[1,2], Qiongxiao Wang[1,2], and Lingguang Lei[1,2]

[1] Data Assurance and Communication Security Research Center, Beijing, China
[2] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
{zwzhang,ywwang,jing,qxwang,lglei}@lois.cn

Abstract. Android permission system enforces access control to those privacy-related resources in Android phones. Unfortunately, the permission system could be bypassed when the phone is rooted. On a rooted phone, processes can run with root privilege and can arbitrarily access any resources without permission. Many people are willing to root their Android phones to uninstall pre-installed applications, flash third party ROMs, backup their phones and so on. People use rootkit tools to root their phones. The mainstream rootkit tools in China are provided by some well-known security vendors. Besides root, these vendors also provide the *one-click-unroot* function to unroot a phone. The unroot process gives users a feeling that their phones will roll back to the original safe state. In this paper, we present the security threats analysis of permission system on phones rooted once and unrooted later. On these phones, two categories of attacks: tampering data files attack and tampering code files attack are carried out. Also, the attacks' detection rate, damage degree, influence range, and survivability in the real word are analyzed. Analysis result shows even under Antivirus' monitoring, these attacks towards permission system can still be carried out and survive after the phone is unrooted. Therefore, the permission system faces a long-term compromise. The potential defense solutions are also discussed.

Keywords: Android, Permission System, Rooted Time Window.

1 Introduction

Android permission system is one of the most important security mechanism for protecting critical system resources on Android phones. Android phones contain numerous privacy-related system resources, such as various sensors, sensitive data, and important communication modules. Abusing these resources will

* The work is supported by a grant from the National Basic Research Program of China (973 Program, No. 2013CB338001).

W. Susilo and Y. Mu (Eds.): ACISP 2014, LNCS 8544, pp. 354–369, 2014.

result in serious leakage of users' private data and even financial loss. To prevent the privacy-related resources from being abused, permission system ensures only the applications (apps) granted certain permissions can access corresponding resources. Otherwise, the access request will be rejected. In Android system, permissions that an app possesses represent the app's capability to access resources.

Besides permissions, root privilege is another resource access capability. Nowadays, it is common for users to root Android phones. According to [23], 23% Android phones are rooted at least one time in China mainland by the first half of 2012. After a phone is rooted, users can fully control it. For example, they can remove the disliked pre-installed apps, custom personalized system, backup their phones, and flash third party ROMs.

Getting root privilege is a big challenge to permission system. With root privilege, the highest privilege in user mode, malware can access sensitive database files (e.g. SMS and Contact databases files) and hardware interfaces (e.g. Camera, Microphone) without getting corresponding permissions beforehand. In this kind of resource accessing, permission system does not play any role. That is to say, the permission system can be bypassed. We call this kind of attacks as bypass attacks. Moreover, YaJin Zhou et al.'s work [33] reveals 36.7% malware leveraged root-level exploits to fully compromise the Android security. As the threats brought by root become more and more serious, users are motivated to unroot their Android phones. Besides, the mainstream rootkits in China mainland provide the *one-click-unroot* function, which makes unroot process easy to be carried out. For example, the unroot process of *Baidu, Tencent*, and *360* takes less than 1 minute. For convenience of description, we call the time between root and unroot as a rooted time window.

Will unroot prevent Android system from suffering security threat? Unrooting a phone will make malware lose root privilege. For the bypass attacks, without root privilege, malware cannot access system resources any more. Also, that the permissions system is bypassed not means it is compromised. When malware lose root privilege, they subject to the permission system's access control again. Therefore, unroot can defend the bypass attack towards permission system. However, we wonder if there are other attacks happened in rooted time window and cannot be defended by unroot. To answer this question, we analyzed the implementation of permission system.

Permission system is composed of data files and code files. Data files include a metadata file named *packages.xml* and installation files (*apk* files). Both of them contain app's permissions, and they are protected by nothing but the UNIX file system access control mechanism. By tampering the *packages.xml* file or the *apk* files with root privilege, we carry out 3 kinds of attacks to escalate permissions. By doing this, even after a phone is unrooted, the escalated permissions still open a backdoor for malware to abuse system resources. What is more, we also explore the feasibility of removing access control of permission system by tampering its code files. The feasibility of this attack is verified on Galaxy Nexus,

which is a phone supporting Android Open Source Project. This way could fully compromise the permission system and open a backdoor for all apps.

To evaluate the practical effectiveness of these attacks, we run our demo malware under Antivirus' (AV) monitoring at the rooted time window. The result shows these attacks have a 100% rate to evade detection of AVs in China and 80% abroad. To permission escalation attacks, more than one half of escalated permissions can hide from permission list provided by AVs. Besides this, damage degree, influence range, and survivability after unroot are also analyzed.

The main contributions of this paper are listed as follows:

- To the best of our knowledge, we primarily analyzed the implementation of permission system and illustrated 4 attack models from the perspective of tampering data files or tampering code files of permission system.
- We evaluated the attacks in the aspects of evasion rate, damage degree, influence range, and survivability. The analysis result indicates that even under AVs' monitoring, attacks can be carried out at rooted time window and survive after the phone is unrooted.

The remaining part of this paper is organized as follows. Section 2 describes the problem statement; Section 3 shows tampering data files attacks; Section 4 describes tampering code files attacks; Section 5 evaluates the attacks; Section 6 discusses potential defenses solutions, Section 7 discusses the related work, and Section 8 shows our conclusion.

2 Problem Statement

Android gives each app a distinct Linux user ID (UID) at installation time. Normally, the UID given to each app is bigger than 10000. Once the UID is given, it cannot be changed on the same phone. Each app is regarded as an unique Linux user and runs in its own process space. The *systemserver* is the most important process in Android system. Many significant system services, such as *PackageManager* service, *ActivityManager* service, are running as a thread of the *systemserver* process. Many system resources, such as GPS, Bluetooth, are managed by the *systemserver* process. The *systemserver* also has an independent process space, whose UID is set as 1000. When the phone is rooted, an app could change its UID to 0, which is the UID of root. The UID of the *systemserver* is not affected, which is still 1000.

Users can use rootkit tools to root their phones. The mainstream rootkit tools in China are provided by *Baidu, Tencent, LBE*, and *360* etc. These tools provide not only the *one-click-root* function but also the *one-click-unroot* function. When a user wants to root his phone, he only needs to push the *one-click-root* button. After several minutes, the rooting process is finished and his phone is rooted. During the rooting process, the rootkit tool first exploits Linux vulnerabilities [10] to temporarily get root privilege. The Linux vulnerabilities could be: *ASHMEM*, *Exploid*, *Gingerbeak*, *Levitator, Mempodroid, RageAgainstTheCage, Wunderbar*, *ZergRush*, and *Zimperlich*. Next, the tool places a customized *"su"*

binary file into */system/bin* or */system/xbin* directory. At last, the tool sets the *s* attribute to the *"su"* binary file. With *s* attribute, the *"su"* binary file could run with root privilege. If the user pushes the *one-click-unroot* button, the added *"su"* file will be deleted, which means the phone is unrooted. The unrooting process takes less than 1 minute.

Several reasons attract people to root their phones. After the phone is rooted, they can uninstall the disliked system apps, which cannot be uninstalled before. Besides this, the user also could flash a third party ROM to his phone, backup his phone, customize Android system and so on. The website [20] lists 10 reasons attracting users to root their phones, let alone those mobile forums. According to [23], 23% Android phones had been rooted at least once in China mainland by the first half of 2012.

Taking advantage of the rooted phone vulnerability, many attacks can be carried out. After a phone is rooted, attackers could do anything using a malware, which is in the form of an app. For example, with root privilege, the malware could directly call *libgps.so* to access GPS location without using the *system-server* process. Moreover, YaJin Zhou et al.'s work [33] reveals 36.7% malware leveraged root-level exploits to fully compromise the Android security. The malware *DroidKungFu* [22] is a typical example. As the security threats on a rooted phone become more and more serious, users want to unroot their phones as soon as possible.

We assume that the attacker's malware can get root privilege on a rooted phone. Since users want to unroot their phones as soon as possible, the malware will quickly lose root privilege. We suppose the attacker's goal is stealing users' private data such as GPS location, photos, and SMS. These data are frequently updated. Hence, the attacker wants to steal the private data all the time regardless the phone is unrooted or not. For example, the attacker would keep an eye on users' GPS location all the time. Moreover, some private data may be created after the phone is unrooted, such as a newly applied credit card account. The attacker may want to steal these new data as well. Therefore, the malware should keep working even the phone is unrooted. The attacker knows after a phone is unrooted, his malware will lose root privilege. So, the malware should not rely on root privilege all the time.

An effective way to make the malware keep working after the phone is unrooted is opening a backdoor during the rooted time window. In this way, two kinds of attacks can be carried out. The first kind is tampering data files to escalate required permissions. For example, tampering the *packages.xml* file to escalate the *"android.permission.ACCESS_FINE_LOCATION"* permission to get GPS location. This way enables malware to pass the access control checks of permission system all the time. The other one is tampering the code files of permission system to remove the access control. This way makes private data freely accessible to all apps. Therefore, even if the phone is unrooted, the above attacks open a backdoor for malware.

3 Tampering Data Files Attack

3.1 Inserting Permissions into the *packages.xml* File

The heart of permission system is a system service named *PackageManager* service (PMS). After an app is installed, PMS writes its permissions into the *packages.xml* file. At the next system boot time, PMS directly grants the permissions preserved in the *packages.xml* file to each app. Therefore, adding permissions into the *packages.xml* file is one way to escalate apps' permission.

The *packages.xml* file is only protected by the user-based UNIX file system access control mechanism. Normally, the file belongs to the *systemserver*, and other users (processes) cannot access it. However, as long as a malware gets root privilege, it becomes a superuser. Superuser can read and write any files in file system. Furthermore, Android supports the Java APIs of executing external command. Through the Java APIs, malware could execute the *"su"* command to gain root privilege on a rooted phone. The attack flow is shown as Figure 1.

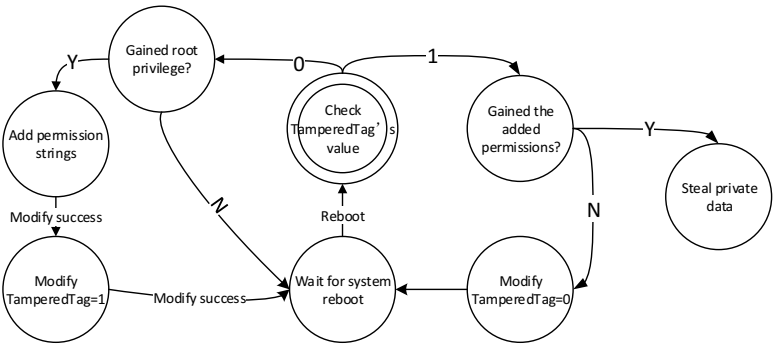

Fig. 1. The attack flow of escalating permissions by modifying the *packages.xml* file

The attack flow can be divided into two parts: tampering the *packages.xml* file to add permission and starting attack based on the added permission. To control the attack flow, we introduce a flag *TamperedTag*, whose value 1 or 0 indicates that the *packages.xml* file has or has not been tampered.

The demo app first checks the *TamperedTag*'s value. If its value is 0, which means that the *packages.xml* file has not been tampered, the demo app will take a further step to check whether the phone has been rooted. One way of doing this is executing the *"su"* command, and then see the execution result. If the command is successfully executed, the demo app can get root privilege and do the following steps of the attack. Otherwise, the demo app will do nothing but wait for the system reboot (see Figure 1). With root privilege, the demo app adds, for example, the *android.permission.READ_SMS* permission into the *packages.xml* file for itself. Once added, the value of *TamperedTag* should be changed to 1.

If the *TamperedTag*'s value is 1, which means that the *packages.xml* file has successfully been modified, the demo app checks if it has indeed got the READ_SMS permission. If yes, the demo app could read and steal SMS. Otherwise, the demo app changes the *TamperedTag*'s value to 0 to restart the attack. This step increases the robustness of the attack.

The demo app has less than 430 lines of code and can successfully run on any rooted Android phones. Furthermore, the demo app can be extended to escalate any permission in the real world.

3.2 Bypassing Signature Verification to Share UID with High Privileged Apps

By default, each app has a distinct UID and runs as an independent Linux user. For these independently running apps, Android has no restriction on their certificates. Android also allows different apps to share the same UID. For the apps running with the same UID, Android enforces that they must be signed by the same certificate.

Apps with the same UID can run with the same permission set and access each other's data and, if desired, run in the same process. Some system apps share the same high privileged UID. For example, *SystemUI* and *SettingsProvider* share the UID: *android.uid.system*. This UID has a bunch of privacy-related permissions such as READ_CONTACTS, CALL_PHONE; and many hardware-related permissions such as INTERNET, BLUETOOTH.

Normally, malware cannot share the same UID with other apps because their signatures are mismatched. However, if an app satisfies the following optimization terms, PMS will not verify the app's signature.

1) The package setting info of the app, denoted as a <*package*> node, exists in the *packages.xml* file;
2) The path of the app is consistent with that preserved in the <*package*> node;
3) The last modification time of the app is the same as the timestamp preserved in the <*package*> node;
4) The <*sig*> sub-node of the <*package*> node exists;
5) The certificate value exists in the <*sig*> sub-node.

By satisfying the above terms, malware can escape the signature verification step enforced by the permission system. As a result, the malware can break the certificate restriction of sharing UID, and shares UID with any privileged app. A demo attack flow can be described as follows.

We use a cover app and shadow app model to illustrate how to start an attack. The cover app does not contain malicious code of stealing private data, but contains the code of loading the shadow app into Android system. The shadow app contains malicious code to steal private data as well as necessary permissions to access the private data. To start an attack, the attacker should develop a cover app first. The cover app is not over privileged that can escape detection tools based on permission analysis like Kirin [13], Stowaway [14], and

Pscout [2]. Then, following the cover app, the attacker develops a shadow app. For concealment considerations, they should have the similar appearance. The shadow app can be loaded into Android system as the cover app's payload or by remote downloading. These ways of loading malicious code are widely used [33].

In this attack, the attacker should set the value of the shadow app's *android:sharedUserId* attribute to the UID that it wants to share. The UID could be an app's package name or a shared user's name defined by Android system such as *android.uid.system*. The *android:sharedUserId* attribute is defined in the shadow app's *AndroidManifest.xml* file (manifest file). Every Android app has a manifest file to declare the desired permissions, assign the desired UIDs, and list its components, etc. To meet the optimization requirement, the cover app can modify either the attribute of the shadow app's *apk* file or the content of the *packages.xml* file to make the value of *path* and *ft* matched. The *ft* is the timestamp of the last modified time of the *apk* file.

Then, the cover app traverses the *packages.xml* file to get the shared UID's certificate info (preserved in <*sig*> sub-node), and uses that info to replace the cover app's <*sig*> sub-node. At last, the cover app uses the shadow app's *apk* file to replace the cover app's *apk* file (stays in the */data/app directory*). After reboot, the shadow app can turn into a shared user running with a higher privileged permission set and can access other shared users' data.

This kind of attack not only extends malware's permissions but also extends the influence from one app to multiple apps. The feasibility of the attack is verified on Nexus S, Nexus 4g, Galaxy Nexus, and Sony LT29i.

3.3 Escalating Permission by Silent Update

The permissions escalated using the way described in Section 3.1 and 3.2 are fully depend on the *packages.xml* file. When the file is deleted, the escalated permissions will be gone. In this section, we will discuss another way of permission escalating that is not depend on the *packages.xml* file.

All the installed apps including system and non-system apps will be reinstalled at system boot time. During the re-installation, the permissions will be re-parsed from each app's manifest file. After the re-installation is done, PMS will update each app's permissions based on the following policy. For system apps, PMS updates each system app's permissions to those declared in its manifest file. While for non-system apps, PMS will not update any permission, except the permissions updated by Android platform, which is rarely happened. Therefore, the actual permissions of a system app is a union of the permissions declared in the manifest file and the permissions stored in the *packages.xml* file, while the actual permissions of a non-system app are only the permissions stored in the *packages.xml* file.

Attacks towards Non-system Apps. Normally, non-system apps cannot be given more permissions than those stored in the *packages.xml* file. However, when the <*package*> node of an app is removed from the file, PMS will grant all permissions declared in the app's manifest file to this app. Malware can use this way to escalate permissions, which could be called as silent update.

The attack flow is described as follows. With root privilege, the cover app remounts the *data* directory to be writable. Then, it moves the shadow app into */data/app* directory to replace the cover app's *apk* file. Next, the cover app should delete its <*package*> node or delete the entire *packages.xml* file. At the next system boot time, PMS will recover the cover app's permissions from its manifest file, which is contained in the *apk* file. As the cover app's *apk* file is replaced with the shadow app's, the permissions are actually recovered from the shadow app's manifest file. And, the permission info will be written into the *packages.xml* file. Finally, the cover app is updated to a malicious one silently. A typical example in the real world is the *BaseBridge* [28], which exploits root privilege to silently install apps without user intervention [33].

The attacker can use the same way to replace any installed apps with malicious ones as long as he can remove the target app's <*package*> node. After the node is removed, the target app is actually regarded as uninstalled, and the malicious one will be installed as a new app. Android adopts self-signed mechanism, and the signature of the shadow app is verified by self-signed certificate, so there is no need for the shadow app to be signed with the same certificate as the target app.

We validated the feasibility of the attack on Nexus S, Nexus 4g, Galaxy Nexus, and Sony LT29i.

Attacks towards System Apps. Android regards apps in the */system/app* directory as system apps. Also, Android pre-installs several system apps to provide the basic phone functions. For example, the pre-installed *Phone.apk* is used to place phone call, the *Mms.apk* is used to send multimedia message, and the *Contact.apk* is used to manage contacts. Compared to non-system apps, system apps play a more important role in Android system. However, there is no special requirement for the system apps' certificate. The system apps also apply to the self-signed certificate mechanism. Therefore, malware can also replace system apps with malicious ones.

The attack flow is similar to that of the non-system apps', while this one is much easier. The only thing that the cover app needs to do is replacing the target system app with a shadow app. There is no need to remove the target app's <*package*> node. That is because, when the shadow app is put into the */system/app* directory, it will be regarded as a system app. According to the update policy, system apps could gain full permissions declared in the manifest file directly. The shadow app can be developed by re-packing or re-coding since Android is open-source.

Besides replacing original system apps, malware can also disguise as a system app and gain full permissions declared in the manifest file. Simply, the cover app places the shadow app into */system/app* directory and deletes the original *apk* file from the */data/app* directory.

We verified the feasibility and effectiveness of this attack on Nexus S, Nexus 4g, Galaxy Nexus, but not Sony LT29i. That is because, this phone has blocked modification to the *system* partition. The */system/app* directory belongs to this

partition. When the *system* partition is modified, the phone will reboot automatically, which makes all modifications fail to go into effect.

Compared to the above two attacks described in Section 3.1 and 3.2, the permissions escalated by silence update do not depend on the *packages.xml* file. No matter what happened to the *packages.xml* file, the permission escalated in this way will not disappear.

4 Tampering Code Files Attack

All system code are stored as files in the file system. With root privilege, attackers can also tamper the code file of permission system to fully compromise it.

Permission system carries out the access control by checking whether an app has certain permission strings. The check function is provided by Android APIs such as *checkPermission, checkCallingPermission*, and *checkCallingUriPermission*. The difference between them is they taking different parameters. Taking *checkPermission* as an example, it takes a permission name and a package name of an app as the parameters. When the method is called, it first checks whether the app (identified by the package name) is a shared user. If yes, the method checks the shared user's *grantedPermissions* field. Otherwise, it checks the app's *grantedPermissions* field. The *grantedPermissions* field exists in a data structure (*PackageSetting*), which stores all info of an app needed to run. If the field contains the required permission string, the *checkPermission* API will return *PERMISSION_GRANTED*, or else it will return *PERMISSION_DENIED*.

An available way to compromise permission system is tampering the return value of those APIs. A typical compromise example is tampering the *checkpermission* API to return *PERMISSION_GRANTED* always.

The source code of permission system are compiled into a file named *services.jar*, which is located in the */system/framework* directory. Provide that the malware is designed as adaptive, it needs to analyze the construction of the *jar* file and identify the exact point to tamper. However, we doubt the feasibility of automatic analysis without manual intervention. Therefore, we assume that the attacker would temper the code files in the way of replacing the original *services.jar* with a malicious one prepared ahead.

Two ways can be used to create a malicious *services.jar* file. The first one is decompiling and recompiling the target phone's *services.jar* or *services.odex* file using tools such as *baksmali, smali*, and *dex2jar*. The *services.odex* file is an optimized version of the *services.jar* file. Some vendors shipped their phones with *odex* version of code files in order to increase the phones' performance. The other one is exploiting Android Open Source Project (AOSP). Several phones, such as the Nexus series phones published by Google, support AOSP. By fetching the source code of these phones, the attacker can customize his own permission system, let alone modify the return value of an Android API.

Even the attacker successfully gets a customized *services.jar* file, he cannot directly use it to replace the original one. Some issues should be overcome. The first one is optimization. In the real world, most vendors such as Samsung, Sony,

HTC optimize the *jar* file to the *odex* file in their factory image. The optimization is hardware-related. Even the source code is identical, the *odex* file generated on a phone cannot run on a different versions of phone. Only the same *odex* file running on the same version of phone is allowed. The other issue that the attacker should overcome is signature. During the system boot time, the Dalvik Virtual Machine (DVM) will load all Java classes into memory to create the runtime for Android system. In this process, the DVM verifies the *dex* or *odex* files' signature and the dependencies with other *dex* or *odex* files.

There are two ways can be used to overcome the signature issue. The first way is tampering the code file of the DVM to remove the piece of code doing the verification work. The DVM is compiled into a file called *libdvm.so*. Android has no signature restriction to *so* files. Therefore, replacing a *so* file does not face the signature issue. However, decompiling a *so* file to modify the code is difficult. An optional way to remove the verification code is getting the source code of the target phone. This way only fits attacking phones supporting AOSP.

The second way is extracting signature from the original *odex* file and using it to replace the malicious one's signature. After analyzing the *odex* file's construction, we find that the signature has a 20-bytes length and has a 52-bytes offset from the file header. To obtain the signature value, we use the *dd* command: *dd if=services.odex of=myservices.odex bs=1 count=20 skip=52 seek=52* The whole command means that reads 20 bytes (signature) from the original *service.odex* file, and writes the 20 bytes (signature) to the *myservices.odex* file. Both reading and writing should skip 52 bytes from the header.

Based on the second way to overcome the signature issue, a demo attack is carried out on a Galaxy Nexus phone running 4.1.2 code version. This phone supports AOSP and runs *odex* version of code files. Using this phone, we successfully replace the original *odex* file provided by Google's factory image with the one we generated from Android source code.

5 Attack Evaluation

5.1 Evasion Rate Evaluation

We implement all attacks on one demo malware. To understand the attacks' efficacy in the real world, we evaluate the demo malware's evading detection rate and permission hidden rate under Antivirus' (AVs) monitoring. We select the top 5 AVs in China and abroad, respectively. In China, we download the top 5 popular AVs on Google Play, which are *360, LBE, Tencent, Kingsoft*, and *Anguanjia*. At abroad, according to [29], we select the top 5 **free** AVs, which are *Lookout, McAfee, Kaspersky, ESET*, and *Trend*.

As AVs do a scanning when apps are installed, we test whether the malware can evade detection at installation time. We first install 10 AVs on our test phones, then install the demo malware and waiting for the 10 AVs' scanning result. The result shows that 9 of 10 AVs assert it is clean, only 1 AV detects it out but mistakenly classify the threat as *"Exploit.Android.RageAgainstTheCage.a"*, which we have not exploited. Then, we run the demo malware under the 10

AVs' monitoring to see if the malware could evade detection at runtime. The result is, none of the AVs detect the malware out when it is running. Therefore, these attacks can evade 100% AV's detection in China and 80% AV's detection at abroad at installation time and evade 100% AV's detection at runtime. Since rooting a phone mainly happens in China mainland, evading AVs in this area is the main goal of our attacks, which has been achieved.

Towards attacks of tampering data files to escalate permissions, we test whether the escalated permission can hide from the permission list provided by the 10 AVs. Permissions escalated by silence update are not necessary to test, as they exist in both the manifest file and the *packages.xml* file. What worth testing are those escalated by the attacks of tampering the *packages.xml* file, which are described in section 3.1 and 3.2. We only checked permissions listed by AVs in China except *360*, as *360* and the AVs abroad do not offer the function of listing permissions when this paper is written. The demo malware used to launch the two attacks do not apply any permission in its manifest file. For testing the attacks described in Section 3.1 (denoted as InsertPerm attack), the demo malware inserts 5 different permissions into the *packages.xml* file. While for the attacks described in Section 3.2 (denoted as ShareUID attack), the demo malware shares UID with an app possesses 24 permissions. The test result is shown in table 1. It should be mentioned that *Kingsoft* relies on Android system to list permissions.

Table 1. Permission detection rate

AVs	LBE	Tencent	Kingsoft (Android system)	Anguanjia	Android system	Avg. Dtc. Rate
InsertPerm	3/5	0/5	0/5	3/5	0/5	24%
ShareUID	4/24	0/24	24/24	6/24	24/24	48.3%

The result shows that the 4 AVs have an average detection rate of 24% and 48.3% towards the two attacks, respectively. Most escalated permissions are not detected out. The InsertPerm attack completely cheated 3 in 5 AVs, while the ShareUID attack completely cheated 1 due to the effectiveness of Android system in listing shared user's permissions.

5.2 Damage Degree Analysis

The two kinds of attacks have a different damage degree.

Tampering data files attack only makes the malware itself pass permission checking process, while other apps' permissions are still constrained by the permission system. Therefore, this type of attack only opens a backdoor for the malware itself, and the extent of damage is limited in the malware.

Tampering code files is an extreme damage to the permission system. Section 4 shows a demo attack that removes the resource access control of permission

system, which makes apps freely access private data such as *SMS, Contact*. While the hardware resources cannot be freely accessed, as the access control towards these resources are based on UNIX groups rather than permission strings. However, in the real world, attackers can modify any part of the permission system. To address the hardware access issue, they can modify permission system to automatically set an app as a member of all UNIX groups. For example, the attacker could tamper permission system to set an app as a member of groups like *camera, radio, net_admin*. Then, all apps could take pictures, place phone calls, and access the Internet without applying permissions. By doing this, permission system will comprehensively loss its efficacy and all resources could be freely accessed. This type of attack opens a backdoor for all apps. Obviously, attacks of tampering code files are a disaster to Android security. However, this kind of attack has a quite limited range, which will be discussed in the next section.

5.3 Influence Range Analysis

Another dimension to analyze an attack is the influence range. The two kinds of attacks have a different damage degree as well as a different influence range.

Regarding the tampering data files attack, malware only require the read and write ability towards data files. The *package.xml* file and non-system *apk* files are placed in the *userdata* partition. This partition will be frequently written by Android system. For example, installing and uninstalling an app should write the *userdata* partition. Therefore, this partition is designed as modifiable. Attacks writing this partition can be carried out on all rooted devices. However, system *apk* files stay in */system/app* directory, which belongs to the *system* partition. This partition is used to store system code, which should never be modified; therefore it is designed as read only. Some vendors such as *Sony* restrict modifying this partition even with root privilege. On these phones, the attacks of tampering system apps cannot be carried out. The influence range of replacing or disguising as a system app depends on the phone's version.

Regarding the tampering code files attack, there are three limitations. First, as different vendors make different modifications to Android source code to customize their own system, the code of permission system may differ from each phone version. Therefore, one tampering way may only fit one phone version. Second, the code of permission system on most phones is optimized into *odex* files. The optimization is hardware-related, which makes the *odex* file hardware-related as well. Even the source code is the same, the *odex* file generated on a phone cannot run on another version of phone, which makes the attack not only limited by source code but also limited by phone version. Third, as described above, some vendors block the *system* partition from being written. Take a rooted Sony LT 29i phone as an example, when we try to write the *system* partition, the phone automatically reboots. Vendor's constrain is another limitation. Therefore, the influence of tampering code files attacks has a rather limited range. Based on our experience, those phones supporting AOSP are particularly vulnerable to this kind of attack.

5.4 Survivability Analysis

Both tampering data files to escalate permissions and tampering code files to compromise permission system cannot be blocked by unroot.

As we illustrated in section 2, unroot is deleting the "su" file. Without "su" file, the "su" command cannot be executed. As a result, the root privilege cannot be obtained. However, the attacks described in this paper do not depend on root privilege all the time. The truth is, once the attacks are carried out at the rooted time window, the root privilege is no longer needed any more. Blocking the usage of root privilege has no effect on the attacks. First, the escalated permissions open a backdoor for malware to abuse system resources. Second, tampering code files attack makes permission system compromised, which is irreversible. Even unroot cannot make permission system roll back to the normal state.

Therefore, although the phone is unrooted, these attacks can survive and make permission system suffer a long-term threat.

6 Potential Solution Discussion

6.1 SEAndroid

Google has officially merged SEAndroid [27] onto the 4.3 version of AOSP to mitigate security issues brought by root exploits. It constrains the read and write ability of some root-privileged processes such as *init* and can confine apps running with root privilege. By introducing SEAndroid, the *packages.xml* file and the *system, userdata* partitions cannot be freely read/written by super users any more. Therefore, malware even with root privilege cannot launch the attacks mentioned in this paper.

However, although SEAndroid [27] does a great job in mitigating root exploits, as the authors mentioned in their paper, it cannot mitigate all kernel root exploits. Further, Pau Oliva shows the other two weakness of SEAndroid and gives out 4 ways to bypass SEAndroid [31]. Since SEAndroid has been introduced into Android not long ago, there are some imperfections, and it still has a long way to go to be widespread.

6.2 Our Proposal

SEAndroid has done a lot of work from the perspective of protecting the existing Android system. Unlike SEAndroid, we provide some proposals from the perspective of improving the implementation of permission system to confront the permission escalation attacks mentioned in this paper.

Firstly, removing the *packages.xml* file from disk immediately after it has been successfully loaded and re-generate the file after all apps have been shut down. Since the *packages.xml* file is only loaded at system boot time and never be loaded till next boot. There is no need to keep the file on disk always. As the file cannot be accessed by malware, attacks exploiting the *packages.xml* file can be blocked. Secondly, for non-system apps, only installing the ones recorded in

the *packages.xml* file at system boot time. If the file does not exist at this time, Android system should warn users there may be an attack and offer users some options such as letting users re-grant permissions to each app. Thirdly, for system apps, using certificate to identify their identification rather than directory. In this way, malware cannot disguise as system apps as they cannot pass identity authentication.

These proposals can work together with SEAndroid to provide a better security protection to Android. Even if SEAndroid is bypassed, these proposals can combat the permission escalation attacks mentioned in this paper.

7 Related Work

7.1 Defence against Privilege Escalation Attack

The tampering data files attack is a type of privilege escalation attack. Privilege escalation attacks have drawn much attention these years. These works [12][18][26] show that the privilege escalation attack is a serious threat to users' privacy.

A number of detection tools [5] [14] [2] [16] [13] have been proposed mainly aiming to detect whether an app has unprotected interface that can induce privilege escalation attacks. These static analysis tools are likely to be incomplete, because they cannot completely predict the confused deputy attack that will occur at runtime. Confused deputy attack is a kind of privilege escalation attack. Some enhanced frameworks [15] [9] [4] [17] [3] have been proposed to confront this kind of attack at runtime. But all these works cannot confront the privilege escalation attacks mentioned in this paper. It is because that, these attacks do not depend on other apps to start attacks. Malware in these attacks have all required permissions themselves.

Some framework security extension solutions [21] [34] [25] [6] enforce runtime permission control to restrict app's permission at runtime. However, these works do not consider the situation that malware get root privilege. Although these works can control an app's permission at runtime, all of them rely on policy database files and lack of protection to the file. When malware get root privilege, the policy file could be tampered or deleted. Once the policy file is tampered or deleted, the solution loses its effectiveness.

7.2 Protections towards System Code

Some solutions aim at protecting the integrity of system code, which could be used to confront the tampering code files attack mentioned in this paper.

One way to protect the integrity of system code is making sure that the phone is trustily booted. Some techniques, like Mobile Trusted Module (MTM) [30] and TrustZone [1], can be used to ensure that the system is booted in a trusted manner. Based on MTM, paper [8] and [19] implement two different secure-boot prototypes; patent [24] and [11] give out two different secure-boot architecture

reference designs. In addition, paper [32] outlines an approach to merge MTM with TrustZone technology to build a trusted computing platform, on which the secure-boot process can be provided by TrustZone.

Another way of protecting system code is making sure that the system code is correctly executed at runtime. MoCFI [7] provides a general countermeasure against runtime attacks on smartphone platforms, which can be applied to Android platform and ensures that the Android software stack can be correctly executed.

8 Conclusion

To the best of our knowledge, this paper is the first work focusing on analyzing the security threat faced by permission system at rooted time window. At rooted time window, two kinds of attacks can be carried out. The first kind attack is tampering data files managed by permission system to gain stable resource access capabilities, which still hold even if system is unrooted. Tampering code files is another attack way that can be carried out on rooted time window and influences forever. This way is more destructive than the former one, but its influence range is limited by the phone's version. Even under AV's monitoring, these attacks can be carried out and make permission system suffer a long-term threat.

References

1. Alves, T., Felton, D.: Trustzone: Integrated hardware and software security. ARM White Paper 3(4) (2004)
2. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: ACM CCS (2012)
3. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XMandroid: a new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
4. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: 19th NDSS (2012)
5. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: 9th MobiSys (2011)
6. Conti, M., Nguyen, V.T.N., Crispo, B.: Crepe: Context-related policy enforcement for android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 331–345. Springer, Heidelberg (2011)
7. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: Mocfi: A framework to mitigate control-flow attacks on smartphones. In: NDSS (2012)
8. Dietrich, K., Winter, J.: Secure boot revisited. In: ICYCS (2008)
9. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight provenance for smart phone operating systems. In: USENIX Security (2011)
10. Duo Security: X-ray for Android (2012), `http://www.xray.io/`
11. Ekberg, J.E.: Secure boot with trusted computing group platform registers, US Patent US20120297175 A1 (November 22, 2012)
12. Enck, W., Ongtang, M., McDaniel, P.: Mitigating android software misuse before it happens. Technical Report NAS-TR-0094-2008 (September 2008)

13. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: 16th ACM CCS (2009)
14. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: 18th ACM CCS (2011)
15. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: Attacks and defenses. In: USENIX Security Symposium (2011)
16. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Univ. of Maryland (2009) (manuscript)
17. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the 19th NDSS (2012)
18. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: 18th ACM CCS (2011)
19. Kai, T., Xin, X., Guo, C.: The secure boot of embedded system based on mobile trusted module. In: ISDEA (2012)
20. LifeHacker: Top 10 reasons to root your android phone, http://lifehacker.com/top-10-reasons-to-root-your-android-phone-1079161983
21. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: 5th ACM CCS (2010)
22. NC State University: Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets (2011), http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html
23. NetQin: 2012 moblie phone security report (2012), http://cn.nq.com/neirong/2012shang.pdf
24. Nicolson, K.A.: Secure boot with optional components method, US Patent US20100318781 A1 (December 16, 2010)
25. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in android. In: SCN (2012)
26. Schlegel, R., Zhang, K., Zhou, X.Y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: NDSS (2011)
27. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In: NDSS (2013)
28. Symantec: Android.basebridge (2011), http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99
29. Toptenreviews: 2014 Best Mobile Security Software Comparisons and Reviews (2014), http://mobile-security-software-review.toptenreviews.com/
30. Trusted Computing Group (TCG): Mobile Phone Work Group Mobile Trusted Module Specification (2010), http://www.trustedcomputinggroup.org/developers/mobile/specifications
31. viaForensics: Defeating SEAndroid C DEFCON 21 Presentation, http://viaforensics.com/mobile-security/implementing-seandroid//-defcon-21-presentation.html (March 8, 2013)
32. Winter, J.: Trusted computing building blocks for embedded linux-based arm trustzone platforms. In: 3rd ACM Workshop on Scalable Trusted Computing (2008)
33. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), pp. 95–109. IEEE (2012)
34. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)