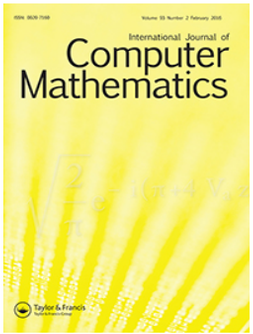

International Journal of Computer Mathematics

ISSN: 0020-7160 (Print) 1029-0265 (Online) Journal homepage: http://www.tandfonline.com/loi/gcom20

Protecting data on android platform against privilege escalation attack

Hwan-Taek Lee, Dongjin Kim, Minkyu Park & Seong-je Cho

To cite this article: Hwan-Taek Lee, Dongjin Kim, Minkyu Park & Seong-je Cho (2016) Protecting data on android platform against privilege escalation attack, International Journal of Computer Mathematics, 93:2, 401-414, DOI: 10.1080/00207160.2014.986113

To link to this article: http://dx.doi.org/10.1080/00207160.2014.986113

Accepted author version posted online: 11 Nov 2014.
Published online: 10 Dec 2014.

International Journal of Computer Mathematics, 2016
Vol. 93, No. 2, 401–414, http://dx.doi.org/10.1080/00207160.2014.986113

Taylor & Francis
Taylor & Francis Group

Protecting data on android platform against privilege escalation attack

Hwan-Taek Lee[a], Dongjin Kim[a], Minkyu Park[b]* and Seong-je Cho[a]

[a]*Department of Computer Science and Engineering, Dankook University, Yongin-si, Gyeonggi-do, Korea;*
[b]*Department of Computer Engineering, Konkuk University, Chungju-si, Chungcheongbuk-do, Korea*

(*Received 19 May 2014; revised version received 24 August 2014; second revision received 26 October 2014; accepted 29 October 2014*)

The users of smartphones are rapidly expanding worldwide. These devices have user's security-sensitive data and are ready to communicate with the outside world. Various kinds of malware are attacking smartphones, especially Android phones, but the existing Android security measure does not work satisfactorily. One-third of the current Android malware were privilege escalation attacks, which try to obtain root-privilege to fully compromise the Android security. We propose a detection and prevention scheme that protects Android against such privilege escalation attack that tries to get full access to all data. The proposed scheme monitors important system calls from an application process. If the system call must be called by privileged Android system components in normal operation, the scheme prevent it from executing. The scheme can detect and prevent new and unknown malware as well as currently known one.

Keywords: Android; privilege escalation; malware; mount; system call hooking; threats

2010 AMS Subject Classifications: 97P30; 68N25

1. Introduction

Smartphones now have computing power comparable to that of a laptop computer. These devices are almost always online and connected to the Internet. They also store security-sensitive information of a user such as contacts, photos, GPS information, and credentials. Various kinds of malware are more and more attacking smartphones, especially Android phones. Android is a mobile platform for mobile devices and runs on more than half of the smartphones sold worldwide [7].

Android is designed to control access to resources by a permission-based policy. The application declares what permission it needs to operate properly in its manifest file. When the application is installed on the device, users are asked to grant or deny declared permissions of the application to resources. This approach can provide users with a chance to protect their privacy. However, it is likely for developers to routinely request more permissions than they require. These excessive requests expose users to malicious attacks [15].

Many Android users utilize root exploits (a.k.a rooting) because they want to customize their phones. Rooting can make a user attain the root privilege, and he or she can alter or replace

*Corresponding author. Email: minkyup@kku.ac.kr

system applications and settings or perform operations unavailable to a normal user. In addition, the user can modify the kernel and install custom firmware [4].

Malware authors also use root exploits to write more sophisticated malware. Malware wants to circumvent Android security mechanisms and eventually escalates its privileges. If malware gets superuser privileges, it performs any operation on the phone. For example, malware can access to files or directories to which access is restricted by the Android permission policy [4].

We propose a new approach which can effectively detect and prevent privilege escalation attacks including root exploits. The approach utilizes the attack pattern of these attacks and characteristics of Android framework. The approach hooks the `mount()` system call and prevents the event that should not occur in a normal operation by monitoring `mount()` system call. Because privilege escalation attacks need to remount/system with *read* and *write* access permissions, the approach detects attacks by telling a unauthorized mount from a authorized one. We identify an unauthorized mount request by checking where a mount point is and who requests it.

This paper is organized as follows. Section 2 describes related work. In Section 3, we show how much dangerous root exploits are and how they work using GingerMaster as an example. Then, we explain the environmental features of Android in Section 4, and propose a new approach to prevent root privilege escalation attacks on Android in Section 5. Section 6 present experimental and statistical analysis results. In Section 7, we conclude and give possible future work.

2. Related work

Felt *et al.* have surveyed that 15–20% of Android phones are rooted, and 6% of iPhones are jailbroken [4]. According to their observation, smartphone users as well as malware authors are encouraged to use root exploits. Smartphone users take advantage of root exploits to customize their phones whereas malware authors use the same root exploits to bypass smartphone security measures. They proposed that mobile phone manufacturers support customization of smartphones so that the users do not need to seek root exploits.

Zhou and Jiang systematically characterized Android malware using 1260 malware samples [16]. They showed among 1260 samples in their data set, 463 samples (36.7%) embed at least one root exploit and so root exploit is the most dangerous threats to users' security and privacy. They also showed the detection results of the existing mobile security software are rather disappointing. The detection rate ranges from 20.2% to 79.7%.

Android's privilege escalation attack can occur in an application level and kernel level. *Permission delegation attack* [5] is a kind of privilege escalation attack and occurs in the application level. The attack allows an application access functions and resources that it has no privileges or permissions to use. An application, for example, has no permissions to access Camera service but the delegation attacks make it use by binding it to the service of an authorized application. To prevent such attacks, many studies have been conducted, including *Kirin* [3], *Saint* [10], *QUIRE* [2], and *Xmandroid* [1]. These approaches almost do not touch the root privilege escalation problem.

Kirin [3] uses security rules to mitigate malware at install time. Kirin validates the permissions requested by apps are consistent with system's security properties. These properties alone do not necessarily indicate malicious potential, but with other data allow detection. Kirin's rules are too conservative, thus Kirin may reject installation of legitimate apps. Kirin does not consider run-time policies and defines special Kirin Security Language to encode security rules.

Saint [10] regulates the granting of permissions defined by an application at install time, similar to Kirin. During the installation, the installer acquired the requiring permission from the manifest file in the application package. For each permission, it queries the database storing

all policies. If the policy confirms the permission, the installation proceeds. Otherwise, it is aborted. In addition to the install-time policy, it conducts the run-time policy that controls the communication between applications.

Quire [2] extends the Android interprocess communication (IPC) mechanisms that helps developers avoid permission delegation attacks. Quire annotates IPCs so that an application can check the full chain of applications responsible for an IPC call. This addresses the same problem as IPC Inspection but does not force developer compliance.

XManDroid [1] presents a solution for privilege escalation by restricting communication at runtime between applications where the communication could open a path leading to dangerous information flows For example, it forbids communication between an application with GPS privileges and an application with Internet access.

Felt *et al.* [5] present a solution to 'permission re-delegation' attacks against deputies on the Android system. With their IPC inspection system, apps that receive IPC requests are polyinstantiated based on the privileges of their callers, ensuring that the callee has no greater privileges than the caller. IPC inspection addresses the same confused deputy attack as Quires security passing IPC annotations

A kernel level privilege escalation attacks exploit vulnerabilities in Android kernel. By exploiting these vulnerabilities, malware take full control of a smartphone. Usually after getting the root privilege, malware make the smartphone ready to receive and execute commands from a designated server known as a command and control server. In other words, smartphone becomes a bot. This type of attack is frequently used in order to take critical security information of a user.

RGBDroid system prevents a malicious act that occurs after obtaining the root privileges [11]. *RGBDroid* system added two mechanisms to the existing Android security measures. The first mechanism maintains a list of processes called *pWhiteList*. *pWhiteList* lists all processes that can have the root privilege on the system. Processes not in this list must not be allowed to get the root privilege. The second mechanism maintains a list called *CriticalList*. This list enumerates files that should not be deleted or changed in the `/system` folder. Only if it meets these two conditions, a process can update the files in the `/system` folder. However, two lists must be modified according to the version of Android and modules added by smartphone manufacturers. The proposed approach need no such an overhead.

Private data protection (*PDP*) is augmented scheme to *RGBDroid* above mentioned [12]. By hooking `open()` system call, *PDP* checks the user ID of the file to open and the user ID of the calling process. If a root privileged process is trying to access the resources that the normal user owns, PDP prevents the `open()` system call from executing. This scheme does not allow processes with root privileges to access user resources, but you must already have root privileges because hooking the `open()` system call is implemented using loadable kernel module.

Zhou *et al.* proposed a detection technique of malicious apps in official and alternative Android markets and implemented a system called DroidRanger [17]. DroidRanger employs a permission-based behavioural footprinting scheme to detect new samples of known Android malware families, and applies a heuristics-based filtering scheme to identify certain inherent behaviours of unknown malware families. To analyse suspicious code and support the heuristic-based filtering, dynamic execution monitoring inspects runtime behaviours of the suspicious code. The dynamic monitor log system calls made by the code. For efficiency, they focused on system calls used by existing Android root exploits and/or made with the root privilege. For example, the `sys_mount` is an interesting system call because it can be used to remount the Android system partition for subverting the system integrity gaining the root privilege. They manually validate whether a suspicious code is indeed a zero-day malware by looking for abnormal runtime behaviours (e.g., executing specific system calls with root privilege, etc.). DroidRanger performs offline analysis to detect malware in Android Markets.

Our work is different from DroidRanger with a focus on online detecting and blocking malicious behaviours such as illegal modification of system objects and escalation of privilege. We employ process identifier (PID)/parent process identifier (PPID) and pathname information to determine whether any system call request is valid or not. Our technique does not collect system events either.

We presented a scheme to detect privilege escalation attacks and protect smartphones against them [9]. In our previous work, we monitored the `sys_mount` system call for remounting a system partition. This attempt is highly suspicious because remounting a system partition can break system integrity. Therefore, it is necessary to determine whether an attempt for remounting a system partition is requested by authorized processes or not, and filter the unauthorized attempt. This paper follows our previous work with a more in-depth consideration of root exploits and other malware families.

3. Security threats on Android

3.1 *Root exploits*

Android OS security mechanisms can be bypassed using root exploits (also known as jailbreaks). Various root exploits are available on smartphones. According to the survey of root exploits for six Android phones investigated by Felt *et al.* [4], root exploits were publicly available more than 74% of the time, rendering phones vulnerable to sophisticated malware. Root exploits for smartphones are used by both smartphone users and malware authors. Smartphone users covet root exploits to perform complete system backups, to install customized versions of operating systems that contain additional features, and to remove pre-installed certain applications on phones. A root exploit allows smartphone owners to gain complete control of the phones software stack [4]. On the other hand, malware authors use root exploits to write sophisticated malware, or to get extra privileges and perform any operation on the phone. Especially, the existing Android malware uses root exploits given in Table 1.

Zhou and Jiangs study has said that there were 389, 440, 4, and 8 malware samples that contained exploit, RATC, GingerBreak, and asroot, respectively [16]. Felt *et al.* and Zhou *et al.* presented that four of Android malware families, i.e., DroidDream, zHash, DroidKungFu, and Basebridge, have at least one root exploit that were published by smartphone users [4, 17].

Zhou *et al.* found some apps that tried to remount the system partition (with the `sys_mount` system call) to make it writable [17]. This behaviour is really suspicious because remounting a system partition can only be executed by processes with the root privilege. For third-party apps, this may mean that the apps have successfully launched a root exploit. Actually, DroidKungFu launches root exploits, and then elevate its privilege to root so that it can arbitrarily access and modify any resources on the smartphone. As a result, we can summarize rooting steps by

Table 1. Root exploits and their uses in Android Malware [16, 17].

Root exploits	Malware with the exploit	Vulnerable program
Asroot	Asroot	Linux kernel
Exploit	DroidDream, zHash, DroidKung Fu	init(<= 2.2)
RageAgainstTheCage	DroidDream, BaseBridge,	adbd(<= 2.2.1)
Zimperlich	DroidKungFu, DroidDeluxe, DroidCoupon	zygote(<= 2.2.1)
KillingInTheNameOf		ashmem(<= 2.2.1)
GingerBreak	GingerMaster	void(<= 2.3.3)
zegRush		libsysutils(<= 2.3.6)

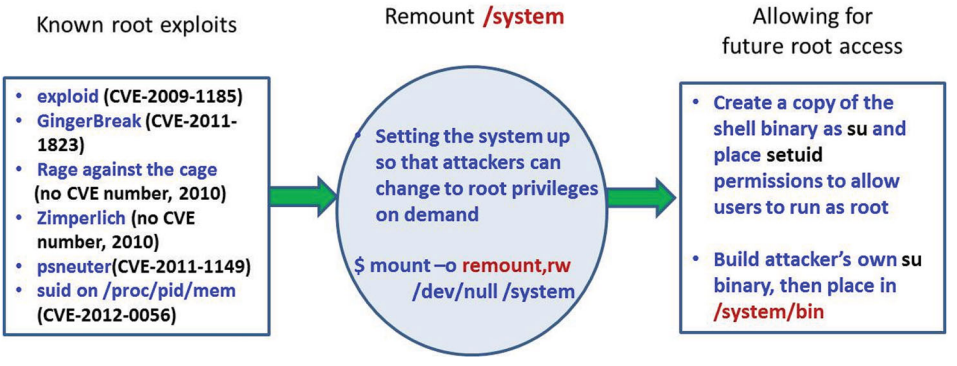

Figure 1. Rooting steps by attackers.

attackers as shown in Figure 1. Root exploits try to remount a system partition. Therefore, we can protect a smartphone against malicious root exploits by disallowing malware or unauthorized apps to remount a system partition.

3.2 *Root Exploit example*

GingerBreak is a root-exploit to attack the vulnerable program `vold` [8, 18]. *GingerMaster* is malware using *GingerBreak*. *GingerMaster* is the first malware that makes use of a root exploit capable of rooting Android 2.3 devices. *GingerMaster* pretends to be a normal application to deceive a user. *GingerMaster* uses *GingerBreak* to get the root privilege.

`vold` is an Android volume daemon, `Mountd` in the previous versions. `vold` does the same things `Udev` does in Linux distribution. It primarily manages device nodes in the /dev directory. `Udev` also handles all user space events raised while hardware devices are added into the system or removed from it. `Udev` uses Netlink socket to receive such events.

`vold` runs with the root privilege. Its primary role is to manage mounting of disk volumes. It also receives Netlink messages from the kernel like `Udev` [14]. But it has two security vulnerabilities. First, it does not verify messages came from the kernel. Second, it uses signed integer in a message as an array index, but it does not check whether index is less than zero. *GingerBreak* exploits these two vulnerabilities.

The *GingerBreak* exploits the vulnerability through the following steps. For the first step, The *GingerBreak* collects information needed to exploit such as identification of the `vold` process, addresses and values of interest. For collecting information, the *GingerBreak* searches and inspects `/proc/net/netlink` to find netlink socket users and `/proc/pid/cmdline` to identify the `vold` PID.

It then analyses `/system/bin/vold` to obtain global offset table (GOT) address range (Figure 2). GOT is the table where all addresses of library functions are recorded. *GingerBreak* get the address of `system()` function by inspecting `/system/lib/libc.so`. It uses `logcat` to obtain the fault address and machine state information in the `vold` (Figure 3).

The second step is to trigger execution of exploit binary and create a `setuid`-root shell. *GingerBreak* carefully crafts a net link message and send it to `vold`. Finally, the GingerBreak got a root privilege by executing the setuid-root shell. The *GingerBreak* exploit code, once launched, will be triggered to root the phone and then elevate it to the root privilege. After obtaining root privilege, *GingerBreak* will connect to the remote command-and-control (C&C) server and wait for instructions.

```
[*] vold: 0070 GOT start: 0x00014344 GOT end: 0x00014384
[*] vold: 0070 idx: -3072 fault addr: 0x00013290
[+] fault address in range (0x00013290,idx=-3072)
[+] Calculated idx: -2003
[*] vold: 0559 idx: -0002003
[*] vold: 0559 idx: -0002002
[*] vold: 0559 idx: -0002001
[*] vold: 0559 idx: -0002000
[*] vold: 0559 idx: -0001999
[*] vold: 0559 idx: -0001998
[*] vold: 0559 idx: -0001997
[*] vold: 0559 idx: -0001996
[*] vold: 0559 idx: -0001995
[*] vold: 0559 idx: -0001994


[!] dance forever my only one
```

Figure 2. The GingerBreak tries to find the address range of GOT.

```
I/DEBUG   (   72): *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
I/DEBUG   (   72): Build fingerprint: 'google/soju/crespo:2.3.2/GRH78C/93600:user/release-keys'
I/DEBUG   (   72): pid: 70, tid: 90  >>> /system/bin/vold <<<
I/DEBUG   (   72): signal 11 (SIGSEGV), code 2 (SEGV_ACCERR), fault addr 00013290
I/DEBUG   (   72):  r0 fffff400  r1 00000001  r2 fffff400  r3 000000b3
I/DEBUG   (   72):  r4 00016268  r5 afd17ef9  r6 00015420  r7 000000b3
I/DEBUG   (   72):  r8 00015360  r9 00000004  10 000000b3  fp afd17ef9
I/DEBUG   (   72):  ip fffffff6  sp 100ffc60  lr 00013268  pc 0000e5ba  cpsr 20000030
```

Figure 3. The GingerBreak uses `logcat` to get needed information.

3.3 *Privilege Escalation attack example using root exploit*

Similar to many others we detected in this study, GingerMaster repackaged its exploit code into a popular legitimate app (in this case, one that displays photographs of models). The embedded exploit code, once installed, will be triggered to root the phone and subsequently leverage the elevated privilege to download and install other apps from a remote server without the users knowledge.

GingerMaster is a malware that attacks an Android phone. It claims to be an application that displays the 'beauty of the day' photo. A photo is downloaded from the website and is not included in the application package.

GingerMaster uses 'GingerBreak/Honeybomb' root exploit in the background. Once rooted, it installs additional applications without users awareness. In addition, *GingerMaster* creates a service that steals information from your device, sending it out to a remote website. The information sent includes the following: user identifier, IMEI (international mobile station equipment identity), IMSI (international mobile subscriber identity), phone number, SIM (subscriber identity module) card number, and screen resolution.

Three ARM executables and one shell script are stored in the asset folder of *GingerMaster* APK (android application package) file. These files are copied to `/data/data/PACKAGENAME/files` on the device. All files have .png file extension to deceive the user. Their names are `gbfm.png`, `install.png`, `installsoft.png`, and `runme.png`. *GingerBreak* exploit codes are stored in the file, `gbfm.sh`, which is created by the malware and is executed in s separate thread later.

This malware can install additional utilities to enhance its functionality, or to prevent itself from being removed. In addition, the vulnerability can be patched very fast. The malware needs a way to get the root privilege whenever it needs such as making a backdoor, and so on.

Once rooting is carried out successfully, GingerMaster will subsequently attempt to install a root shell (with file permission mode 4755) into `/system` partition for later use (Figure 4).

```
lrwxrwxrwx root     root              2014-10-07 11:10 uptime -> toolbox
lrwxrwxrwx root     root              2014-10-07 11:10 vmstat -> toolbox
lrwxrwxrwx root     root              2014-10-07 11:10 watchprops -> toolbox
lrwxrwxrwx root     root              2014-10-07 11:10 wipe -> toolbox
-rwsr-sr-x root     root        26264 2014-10-07 11:19 su
# pwd
/system/bin
```

Figure 4. `/system/bin` has new root-privileged files after rooting.

It remounts the `/system` partition a read/write mode and silently download and install other apps.

4. The Android environment

Each application in Android runs in its own process. In most cases, the security of the system and applications is enforced at the process level using the facilities of a standard Linux such as the user ID and group ID assigned to these applications. Furthermore +, it is possible to run a specific application as the access control provides the authorization mechanisms to enforce restrictions on specific tasks.

Both application and system files in Android can be given access permission of Linux. The access entity is divided into three group: owner, group, and others. Whichever group your belong to, you can have permission to read, write, and execute (`rwx`). In general, an owner of system files is either 'system' or 'root' user. Application files, created by an application, are owned application-specific users and assigned to the user ID of the application. An application cannot access files owned by other applications. Due to different user IDs, system files are also protected from applications.

One of key enforcement of Android security is to mount the system image read-only. All configuration files and executables are located in this system image. Therefore, even though attackers get permission to write to any files on this image, they cannot modify the files. To overcome this limitation, an attacker may try to remount the system image in read/write mode. However, remounting requires an attacker have root privilege [13]. Android grants root privilege to the kernel and a few core system programs [6].

The data partition stores all user data and applications. This partition are different from the system partition, it can be used by the application freely. However, malicious attackers can install many applications and create a large number of user files. Therefore, Android sets a quota on the amount of user data that can be input or loaded into the device.

4.1 *Remounting /system folder*

`/system` folder is a mount point to which `/system` partition is mounted. By default, this folder stores the system files for running Android. Also, in this folder, the user interface of Android and pre-installed system applications on the device is stored. The `/system` folder should not be freely changed by users. Android mounts `/system` folder in read-only mode at boot time and prevents it from being changed. Therefore, to change a file in the system folder, you will need to remount it in read/write mode. To remount the `/system` partition, a process needs the root privilege.

4.2 *Zygote*

Android application is implemented in Java language and executed on the *Dalvik Virtual Machine (DalvikVM)*. Each time you start an Android app, the process called `Zygote` forks

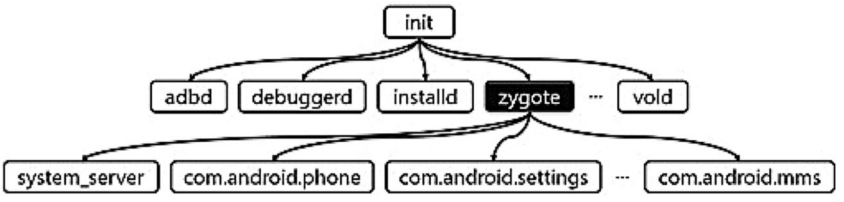

Figure 5. Process hierarchy in Android.

a child process and let the child process execute the requested the app, instead of starting a DalvikVM each time. `Zygote` waits in a state where all the core libraries linked in. The real speedup is achieved by not copying the shared libraries. Because the child process inherits all of them, app startup time is significantly reduced. Also, `Zygote` uses Copy-On-Write policy and this memory will only be copied if the child process tries to modify it. So multiple processes can share unmodified memory pages after fork(). That means although many apps are running at the same time, the amount of actual memory page can be very small.

All user processes are children of `Zygote` process and `Zygote` is the child of `init` process. Actually, all system processes have `init` process as their ancestors. Especially `system_server` is the child of `Zygote` process and cannot have the root privilege. Process hierarchy in Android is shown in Figure 5.

5. The proposed approach

After obtaining the root privilege, *GingerMaster* remounts the `/system` partition and creates some new files to prepare malicious acts in the future. Because Android mounts `/system` partition in read-only mode at the boot time, all malware must remount it in read/write mode in order to modify `/system` folder.

In Android, a process must have appropriate permissions and privileges to remount the `/system` partition. However, almost user applications do not have such permissions and privileges. Malware claims to be normal applications and belongs to this category. Therefore, the root shell executed by malware must not be allowed to remount `/system`.

Many approaches to prevent these acts are possible from the user level to the kernel level. In order to prevent the user application to remount the `/system` partition, we have added new functionality that checks the mount requesting process has permissions and privileges using the Linux loadable kernel module.

We must know whether the requesting process is a user application or not. All Android user applications have one common ancestor, `init`, in the process hierarchy. If we can determine the requesting process has `init` as an ancestor, we can tell the process is from a user application or not.

5.1 *Distinguishing a user process*

Android is based on the Linux kernel. Therefore, we have `/proc` filesystem in Android. `/proc` file system stores information about all processes currently running. All processes currently running has the directory named as its PID and information is stored in the form of subdirectories in it. /proc filesystem has the following information: the name of the process, PID of themselves, PID of the parent process, UID (user identifier) and GID (group identifier) of the owner of the process, etc.

```
[*] ---------- Opened mount system call ----------
[*] /system folder remounted!
[*] PPID: 692
[*] PPID: 592
[*] PPID: 75
[*] PPID: 1
[-] Detected that the system folder had been remounted
[-] Process name is 'eu.chainfire.gingerbreak'(PID: 592)
[*] ---------- Closed mount system call ----------
```

Figure 6. Detecting illegal remounting of *GingerBreak*.

All Android user applications are children processes of `zygote` process. However, a system process with root privileges or pre-installed core applications by the manufacturer do not have `zygote` as their parents. Therefore, if we find `zygote` process while tracking the parent process of the process that requested the mount, we can see the process is from a user application. In such a case, we do not allow the mount operation.

Because the legal processes with the root privilege and core system applications are not user applications, `zygote` cannot be their parent process. Only user applications have the `zygote` process as a parent (see Section 4.2). If we meet the `zygote` process while tracking the parent, we can know the requesting process is a user application. Figure 6). shows the example of this tracking. In the figure, the process with PID 75 is `zygote`, PID 592 *GingerBreak*, and PID 692 the root shell.

For most malware including *GingerMaster*, we can detect the process malicious only if we check whether `zygote` process is the parent of it or not. But some malware such as LeNa.b deceive users by changing the parent of `system_server` from `zygote` to `init`. The mount requesting process is `system_server` and the parent of it is `init`, the `system_server` is a malicious process.

5.2 *Detecting malware exploiting the vulnerability of* `udev`

Malware such as DroidKungFu exploits the vulnerability of the `udev` of `init` process and can create the process with the root privilege. The method of finding the ancestor of the process is not enough to detect this kind of malware. In order to prevent the type of malware, we must check the pathname of the mount requesting process. If the process is stored in `/data/data` folder, we can be sure the process from a user application. Are the algorithm considering all cases described above in Algorithm 1. The Table 2 shows the detection result of the checking PID only approach (Original) and the checking the pathname one described above (Augmented).

Table 2. Detection results of malware targeting rooted smartphones.

Malware	Kaspersky diagnosis	Original	Augmented
DroidDream	Exploit.Linux.Lotoor.l	O	O
	Backdoor.AndroidOS.KungFu.a	O	O
	Backdoor.AndroidOS.KungFu.bw	O	O
	Backdoor.AndroidOS.KungFu.eh	X	O
DroidKungFu	Backdoor.AndroidOS.KungFu.ey	X	O
	Backdoor.AndroidOS.KungFu.hb	O	O
	Backdoor.AndroidOS.KungFu.z	O	O
	HEUR:Backdoor.AndroidOS.KungFu.a	X	O
GingerMaster	Exploit.Linux.Lotoor.z	O	O

Algorithm 1 The algorithm of the `hooked_mount()` system call

```
1:  function HOOKED_MOUNT(source, target, filesystemtype, mountflags, data)
2:      if target is /system then
3:          PID = getpid()
4:          while true do
5:              PPID ← read /proc/PID/stat
6:              if PPID is 1 then
7:                  get the name of the PID process
8:                  if processName is zygote or system_server? then
9:                      Terminate the malware process
10:                 else
11:                     if process is from the program in the /data/data then
12:                         Terminate the malware process
13:                     end if
14:                 end if
15:             end if
16:             PID = PPID
17:         end while
18:     end if
19:     return orig_mount();
20: end function
```

6. Performance evaluation

We measured the execution time of the `mount() system call` before and after adding our method to the kernel. The module implements the checking PID only algorithm. We repeated the measurement 100, 1000, 10,000, 100,000 times and calculate the average. The measured execution time of the `mount()` system call is given in Table 3. The checking pathname `mount()` system call takes twice the time than the checking PID only one.

We also measured the time taken to tracing up to `init`. These times are measured on the following environment: Android 2.2 emulator with 2GB main memory on Ubuntu 12.04 with Core i5 and about 8 GB RAM. We measured 1000 times and it takes about 0.41 µs in four upward searching to `init` process.

To analyse the overhead cost of the proposed approach, we assume the distribution of the number of nodes is based on the statistical population of the processes. Our analysis and our process management techniques assume that processes join according to a Poisson process with rate μ and leave according to an exponential distribution with the rate parameter t. We let new nodes arrive and leave according to a Poisson process with the same rate to keep the number of nodes in the system roughly constant. The probability of being a malware at each node, P_n on

Table 3. The execution time of the `mount()` system call (msec.).

The number of repetition	Before	After	overhead
100	0.176	79.252	79.076
1000	0.348	409.401	409.353
10,000	2.830	4,092.189	4,089.359
100,000	27.219	40,858.549	40,831.33

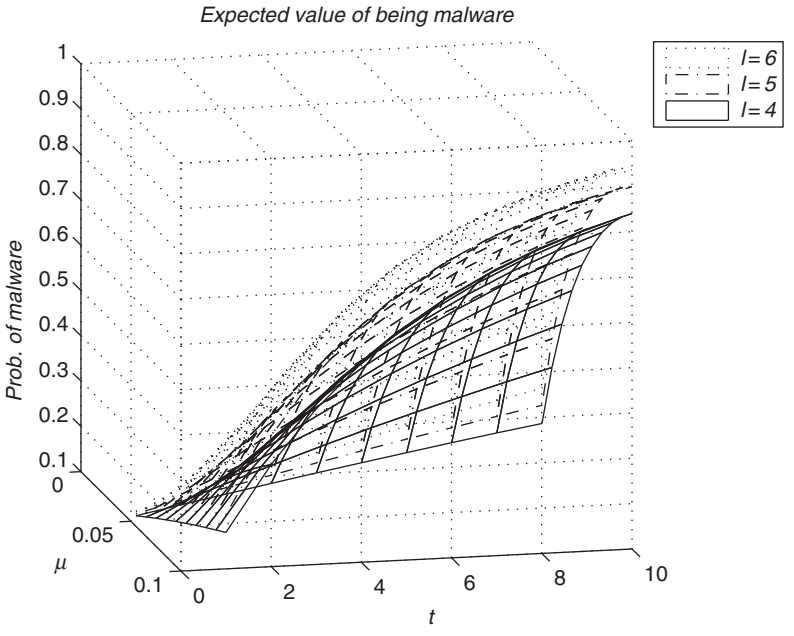

Figure 7. The rate of finding malware, L, when nodes are in the level (l) 4, 5, and , 6.

each level is as shown in the following equation:

$$P_n = 1 - (1 - e^{-\mu t}) \cdot \frac{1}{\mu} \cdot \frac{1}{t} \tag{1}$$

$(1 - e^{-\mu t})$ is the cumulative distribution function of an exponential distribution. It represents the probability of finding malware in time t. In the process tree, l is the level distance corresponding to the number of parent nodes visited to access the init process. The level of root node is 0. The rate of finding malware, L, is as shown in Equation (2). Figure 7 shows the rate when nodes are in level 4, 5, and 6.

$$L = 1 - (1 - P_n)^l = 1 - \left(\frac{1 - e^{-\mu t}}{\mu t}\right)^l. \tag{2}$$

To derive an equation for the overhead cost in the process tree structure, the maintenance cost, C, becomes in the following equation:

$$C = 1 - L(1 - L(1 - L \cdots)) = \frac{1}{1 + L}. \tag{3}$$

The malware process will be terminated and the cost will be compensated by $1/(1 + L)$. We can see the overhead cost decreases as the probability of being malware increases (Figure 8).

7. Conclusion and future work

Many Android malware samples actively collect or steal personal information on the infected phones, including phone numbers, SMS/MMS messages, email addresses, GPS data, and user accounts. Moreover, lots of malware samples embed at least one root exploit, and some malware

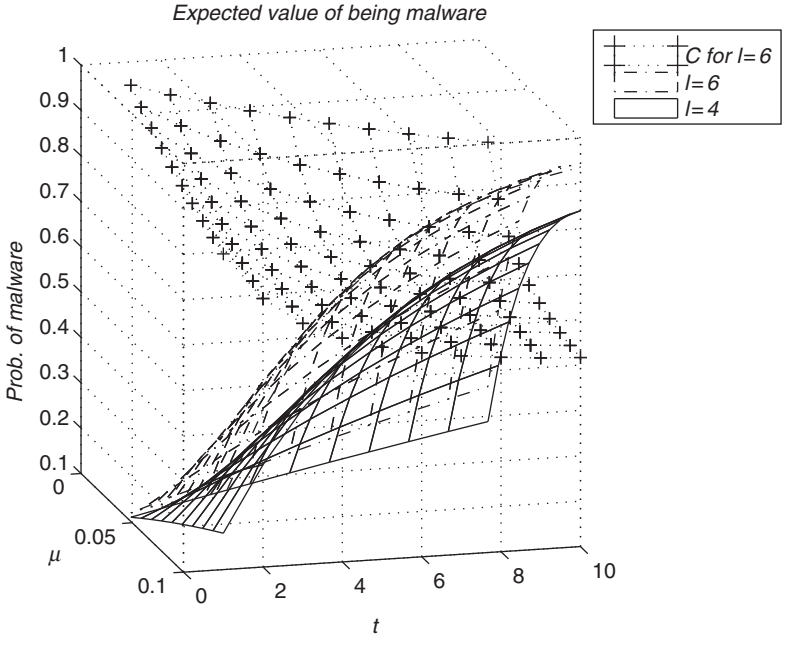

Figure 8. The process tree maintenance cost, C, when $l = 6$.

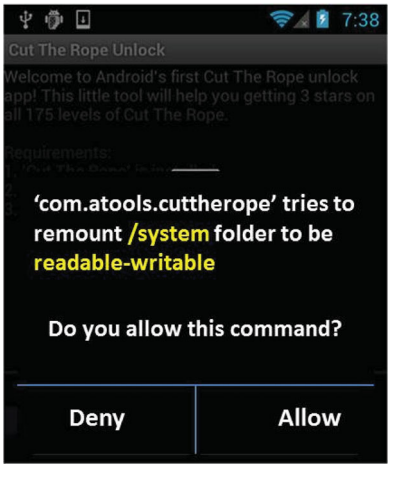

Figure 9. Asking the user if `mount()` is allowed or not.

families contain more than one root exploit. The reason malware does use root exploits is because it can circumvent smartphone security mechanisms using root exploits. If malware gains root privileges, it can completely control the infected smartphone and steal any information. Since certain malware sets use specific system calls to subvert the system integrity after obtaining the root privilege. We observed that the `sys_mount` system call is used by the existing root exploits. To protect smartphone systems against malware and root exploits, it is essential to restrict the privilege of root and maintain smartphone's integrity.

In this paper, we have focused on the `sys_mount` system call invoked at the time of remounting a system partition. The proposed approach uses the information about the process that tries to

remount the `/system` folder to determine whether the mount is illegal or not. If a process tries to remount `/system`, we trace the parent of the requesting process. Although a process has the root privilege, a user application is not allowed to mount `/system` folder. Therefore, even if a user granted root privilege to malware, such as LeNa, unconsciously, no malicious act occurs. We strengthened the approach method by checking where the process's code comes from.

Our approach can detect new and unknown malware of this kind as well as currently known one. However, we must consider the case where the user changes the `/system` folder directly. We can add the functionality of asking the user whether she wants to allow the mount operation (Figure 9). By asking this, we can selectively block the mount operation.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This research was supported by Ministry of Culture, Sports and Tourism(MCST) and from Korea Copyright Commission in 2014, and by the ICT R&D program of MSIP/IITP. [2014(2013-005-016-002), Development of technology for preventing illegal use of android application source code].

References

[1] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, *Xmandroid: A new Android evolution to mitigate privilege escalation attacks*, Technical report, Technische Universität Darmstadt, April, 2011.
[2] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach, *QUIRE: Lightweight Provenance for Smart Phone Operating Systems*, Proceeding of the 20th USENIX Security Symposium, USENIX, San Francisco, CA, August, 2011, pp. 347–362.
[3] W. Enck, M. Ongtang, and P. McDaniel, *On Lightweight Mobile Phone Application Certification*, Proceedings of the 16th ACM conference on Computer and Communications Security (CCS'09), ACM, Chicago, IL, November, 2009, pp. 235–245.
[4] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, A Survey of Mobile Malware in The Wild, in *2011 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2011. Available at http://dl.acm.org/citation.cfm?id = 2046618.
[5] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin, *Permission Re-Delegation: Attacks and Defenses*, Proceedings of the 20th USENIX Security Symposium, USENIX, San Francisco, CA, August, 2011, pp. 331–346.
[6] Google, *Android Security Overview: Rooting of Devices*. Available at http://source.android.com/devices/tech/security/index.html.
[7] Google, *Welcome to the Android Open Source Project*. Available at http://source.android.com/.
[8] X. Jiang, *GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread)*. Available at http://wwwcsc.ncsu.edu/faculty/jiang/GingerMaster.
[9] H.-T. Lee, M. Park, and S.-J. Cho, *Detection and prevention of LeNa Malware on Android*, J. Internet Serv. Inf. Secur. 3(3/4) (2013), pp. 63–71.
[10] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, *Semantically rich application-centric security in Android*, Secur. Commun. Netw. 5(6) (2009), pp. 658–673.
[11] Y. Park, C. Lee, C. Lee, J. Lim, S. Han, M. Park, and S.-j. Cho, *RGBDroid: A Novel Response-based Approach to Android Privilege Escalation Attacks*, Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats (LEET'12), San Jose, CA, April, 2012.
[12] Y. Park, C. Lee, J. Kim, S.-J. Cho, and J. Choi, *An Android security extension to protect personal information against illegal accesses and privilege escalation attacks*, J. Internet Serv. Inf. Secur. 2(3/4) (2012), pp. 29–42.
[13] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, *Google Android: A State-of-the-Art Review of Security Mechanisms*, CoRR, Vol. abs/0912.5101, 2009.
[14] S. Smalley, *The Case for SE Android*. Available at http://dl.packetstormsecurity.net/papers/govt/caseforseandroid.pdf.
[15] T. Vidas, D. Votipka, and N. Christin, *All Your Droid Are Belong To Us: A Survey of Current Android Attacks*, Proceedings of the 5th Workshop on Offensive Technology, USENIX, San Francisco, CA, 2011.
[16] Y. Zhou and X. Jiang, *Dissecting Android Malware: Characterization and Evolution*, IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, 2012, pp. 95–109.

[17] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, *Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets*, Proceedings of the 19th Annual Network and Distributed System Security Symposium, Internet Society, San Diego, CA, 2012.
[18] Y. Zhou, Q. Zhang, S. Zou, and X. Jian, *RiskRanker: Scalable and Accurate Zero-day Android Malware Detection*, in *MobiSys'12*, 2011. Available at http://dl.acm.org/citation.cfm?id = 2307663.