RootGuard: Protecting Rooted Android Phones

Yuru Shao, Xiapu Luo, and Chenxiong Qian,
The Hong Kong Polytechnic University

Though popular for achieving full operation functionality, rooting Android phones opens these devices to significant security threats. RootGuard offers protection from malware with root privileges while providing user flexibility and control.

Valued for its openness and wide application array, Android is the world's most popular smartphone operating system, accounting for 78 percent of global market share in 2013 (http://phys.org/news/2014-02-android-gains-apple-windows.html). Still, its security model prevents users and apps from exploiting full system functionality. In particular, root privilege is strictly limited: by default, Android Open Source Project (AOSP) releases and stock Android phones allow only the kernel and a small subset of core services to run with root permissions. This root inaccessibility constrains how people can use their devices and how apps realize Android phones' potential. For example, users cannot remove preinstalled but rarely used bloatware, and security software products do not have privileges to monitor and defeat malware in real time. To overcome such limitations, users must gain root access, or "root" their phones.

Interest in rooting Androids extends well beyond technology enthusiasts. According to Google, users install non-malicious apps for rooting by a ratio of 494 per million total installs (https://docs.google.com/presentation/d/1YDYUrD22Xq12nKkhBfwoJBfw2Q-OReMr0BrDfHyfyPw). Of Google Play's 10 best-selling paid apps, two require root privilege—Titanium Backup at number 5 and Root Explorer at number 8 (https://play.google.com/store/apps/collection/topselling_paid). In addition, big IT companies like Tencent have invested millions of dollars to develop robust tools for rooting Android phones (http://technews.cn/2014/02/15/tencent-invest-mgyun), while community-built ROMs that provide root access have enjoyed large-scale adoption: CyanogenMod has over 10 million installs (http://stats.cyanogenmod.com), and in China 80 percent of Android phones retailing for less than US$660 include customized ROMs with root access (http://it.21cn.com/focus/a/2013/0803/09/23210909.shtml).

While attractive, rooting involves significant security threats. After gaining root privilege, an app has access to the entire system and to low-level hardware. Benign apps exploit such access to provide desirable features for users, but malicious apps could abuse it to make themselves irremovable, bypass Android's security measures,[1,2] and infect phones systemwide.

Independent developers have created apps to manage root privilege, but the root-management model underlying these apps remains vulnerable. Security Enhanced Android[3] and its extensions[4] might offer some protection, but they do not give users root access and complicated policies make them difficult to manage.

To address these issues, we propose RootGuard, a lightweight, practical tool designed to protect rooted Android phones. Providing fine-grain control, RootGuard lets users grant an app the required operational privileges according to its invoked system calls and parameters, while maintaining default policies to guard itself and the Android system against malware attacks. In designing RootGuard, we addressed various challenges—determining which functions it should monitor, placing appropriate function hooks, using kernel memory to store policies and exposing them through a virtual device driver, among others. Still, performance evaluations including both real-world and proof-of-concept malware and employing the benchmark app AnTuTu show that RootGuard can successfully mitigate attacks by malware having root access, and impose low overhead.

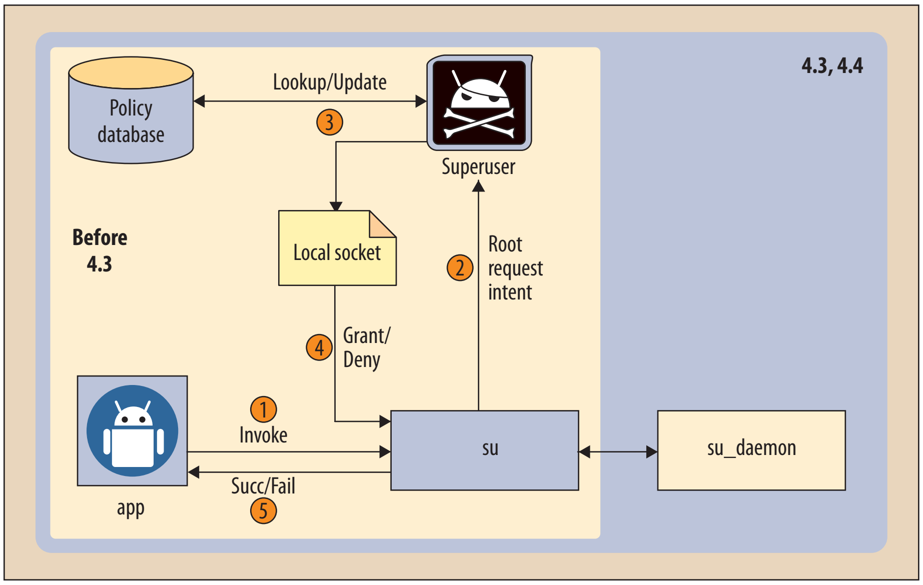

Figure.1. The root-privilege management model. The basic procedure, shown on the left, applies to pre-Android 4.3 versions: when an app invokes su, a root request will be sent to the Superuser app, which grants or denies root access. Android 4.3 and later versions have additional security features; commands must be forwarded to an additional component, *su_daemon*, for execution as shown on the right.

ROOTING ANDROID AND MANAGING ROOT PRIVILEGE

In Linux, the su command—referring alternately to “substitute user,” “super user,” or “switch user”—allows a device operator to switch from current user to root (functionally the administrator). In Android, the su command enables only processes belonging to *root* or *shell* to become root. The goal of rooting Android is to install a customized and unrestricted su that allows any app process to become root. While the mechanics vary depending on device model, existing rooting methods fall into two categories:

- *Exploiting system vulnerabilities.* Using this approach, a potentially exploiting program is deployed on an Android device and then executed to perform privilege escalation. When successful, the procedure opens a root shell, and the system partition (that is, `/system`) can then be remounted as readable/writable, allowing the customized su to be copied there. Most Android devices can be rooted in this way.
- *Flashing fastboot image.* Fastboot allows users to flash a complete file set or, alternately, a file system bundled into a single file known as an “image” to different locations of the file system, such as `/boot` or `/system`. In fastboot mode, users can flash a customized su into the system partition using the command `fastboot flash`. Relatively few devices support this rooting method.

For inexperienced users, many automated tools are available that make both types of rooting easier.

Once the customized su has been created, the next step is installing an app to manage root privilege—that is, to grant or deny other apps’ requests for root access. Such apps, often designated Superuser, can be downloaded from various sites (http://androidsu.com/superuser; www.clockworkmod.com; www.chainfire.eu). Although some differences exist among these apps, they all operate under the basic root-privilege management model illustrated in Figure 1. The original model, as shown on the left, changed very little until Android 4.3 was introduced; more recently, that model has required some modification to accommodate later Android versions’ added security features, as shown on the right.

An app requesting root privilege first invokes su. Then, su sends a privilege-elevation request—or, more simply, a root request—via an Intent messaging object to Superuser. Next, Superuser consults its policy database to decide whether to grant the request. If no corresponding policy exists in its database, Superuser pops up a window asking the user to make this decision. Superuser then sends the decision either granting or denying the initial request for root privilege back to su through a local socket, and su grants the requesting app root privilege or denies it based on the Superuser message.

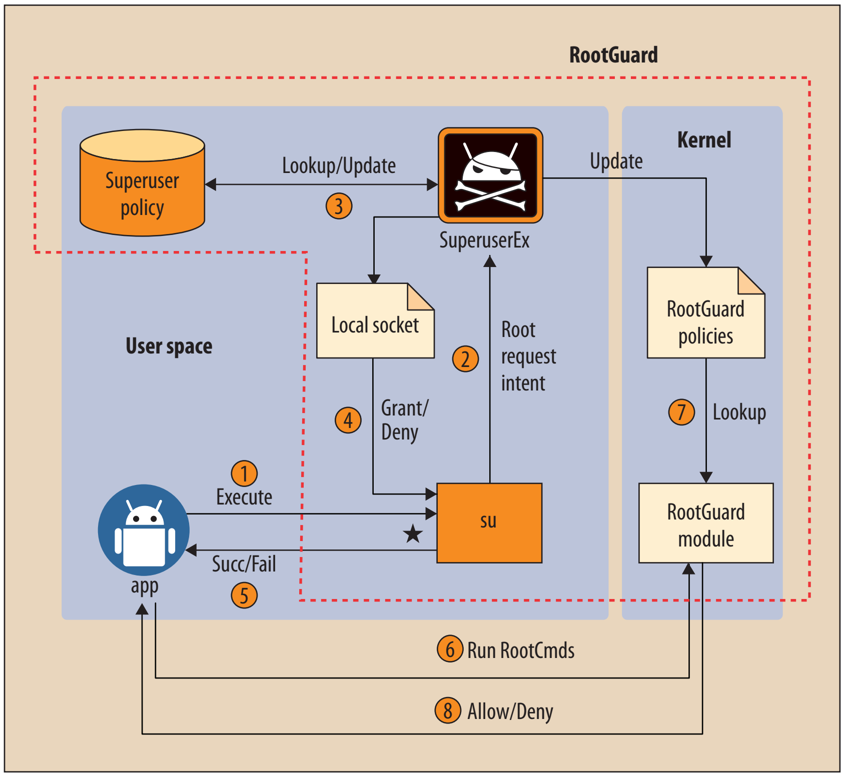

Figure 2. RootGuard-enhanced root-management model. All components within the dashed line are protected by RootGuard; on top of basic root-privilege procedures, the RootGuard module monitors apps executing root commands and looks up policies to determine whether the operation is allowed.

To this basic root-management model an additional component, *su_daemon*, has been incorporated to handle Linux capability bounding (LCB), a set of security features introduced beginning with Android 4.3. LCB, in part, prevents an app from obtaining root privilege even if it can switch to root; *su_daemon* is a root-privileged daemon process started during device booting, without capability bounding. When Superuser grants an app permission to run commands as root, those commands are forwarded to *su_daemon* for execution. This proxy-daemon model requires only minimal revision of Superuser.

SECURITY FLAWS IN AVAILABLE ROOT-MANAGEMENT TOOLS

Most available root-management tools raise two fundamental security issues.

First, in querying whether an app request for root privilege should be granted or denied, they provide users with only the app's name. Malicious apps can easily circumvent this process by behaving like legitimate apps and gaining user trust before conducting malicious activities; once an app is granted root access, existing tools will not monitor its privileged behaviors. Moreover, they provide only coarse-grain control; users who grant an app root privilege might in fact only want specific app functions that do not require root privilege.

Second, currently available root-management tools cannot defend themselves against attacks from malware with root access; malware that obtains root privilege can render such systems useless.

Our examination of available root-management mechanisms' underlying security model revealed four attack surface vulnerabilities. While apps without root privilege can launch the first two attacks, the second two result from a more serious concern: malware gaining root privilege.

Attacking the root request Intent

The fact that su sends root requests through Intent renders the Intent object vulnerable to spoofing and hijacking.[5] We found the following vulnerabilities in two popular root-management tools and reported them to the developers.

Intent spoofing. The Superuser app pre-installed in MIUI V2.3 (http://en.miui.com), a popular third-party Android ROM, fails to verify the source and the encapsulated data of root request Intents. This creates two problems. First, attackers can forge a root request from an app that does not require root privilege and then fraudulently spoof Superuser into warning the user that an app wants to access root. Second, attackers can craft a root request to crash Superuser because it cannot properly handle malformed data.

Intent hijacking and eavesdropping. Generally, su sends root requests via a broadcast generated by `/system/bin/am`, and Superuser registers a broadcast receiver for receiving the requests. However, Superuser 3.1.4 from developer ChainsDD (http://androidsu.com/superuser) fails to protect the broadcast receiver it registers with permissions. Consequently, another app can register an alternate broadcast receiver that eavesdrops on the same broadcast. Moreover, this fake receiver might have higher priority and hijack the root request, aborting it without Superuser's awareness.

Attacking su

Written in C, su is vulnerable to common software attacks. For example, one recent report (http://forum.xda

-developers.com/showthread.php?t=2525552) identified the unsanitized environment vulnerability, the shell character escape vulnerability, and *su_daemon* compromises, which could cause su to let any app execute commands as root without a user's permission.

Attacking Superuser's policy storage

Superuser records a user's decision to grant an app root privilege in its policy database. However, malware with root privileges can access and then modify this database to identify itself and colluding malware as trusted. Colluding malware can thus obtain root privilege without user notification, rendering the root-management system invalid.

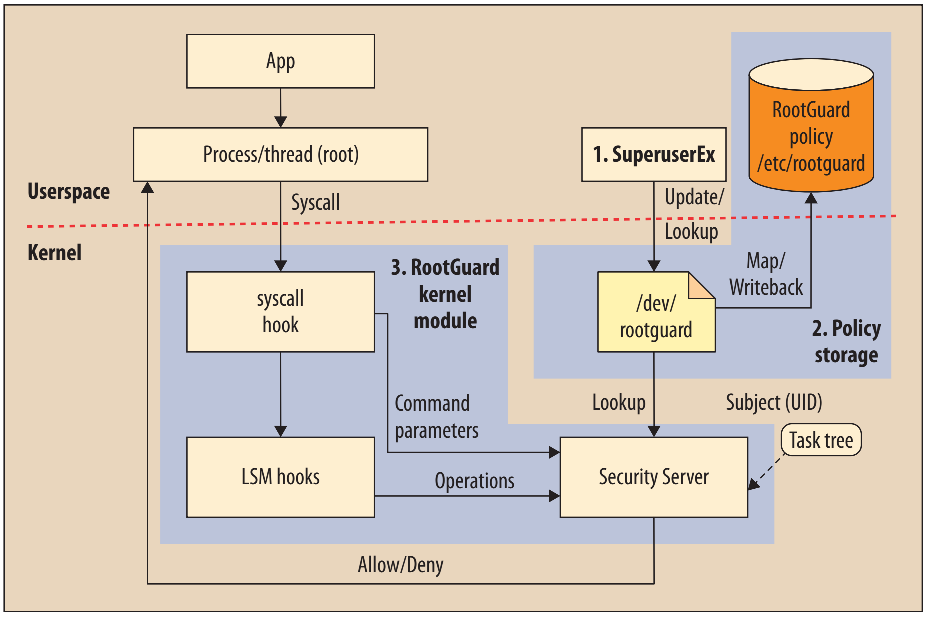

Figure 3. RootGuard's three main components consist of SuperuserEx, which allows users to review and update policies; a policy storage database; and the kernel module, which grants or denies root access by means of Linux Security Module (LSM) hooks, a system call hook, and a Security Server.

Attacking the local socket file

Superuser uses Local Socket to send decisions granting or denying root access back to su, which creates a temporary socket file in its private data directory. Then, su changes the socket file's ownership to Superuser and sets its access mode as private readable/writable—meaning that apps other than Superuser cannot access this socket file. However, attackers that know its location can manipulate the socket file to grant or deny any other apps' root requests.

ROOTGUARD

RootGuard is an enhanced root-management system that both protects rooted Android phones and grants root privileges to apps flexibly and robustly. Two features distinguish it from existing tools. First, RootGuard provides fine-grain control that gives more up-front information, allowing users to grant root privilege to an app's operations based on its invoked system calls and parameters, as well as default policies that protect lay users. Second, RootGuard defends itself against attacks from malware that has gained root privileges. Figures 2 and 3 show the system's architecture and design.

Overview

As Figure 2 suggests, RootGuard provides compatibility by allowing apps to request root privileges using the same basic procedure as the current root-management model in Figure 1 (steps 1–5). However, RootGuard protects all components within the dashed line. When an app executes root commands (step 6), the RootGuard kernel module interposes to monitor the operation, and then looks up policies (step 7) to determine whether the operation is allowed (step 8). In addition, it sanitizes environmental variables and parameters in su to defend against path manipulation and attacks on shell escape characters.

Design and implementation

Figure 3 depicts RootGuard's three major components.

SuperuserEx. This component is built on top of the open source Superuser (https://github.com/koush/Superuser) adopted by CyanogenMod. We add new modules that offer users a GUI for reviewing and updating RootGuard policies, but keep other parts of the original Superuser intact to achieve compatibility.

Policy storage database. RootGuard policies are permanently stored in the file `/etc/rootguard`. To speed up policy querying and updating, RootGuard maintains policy copies in kernel memory, and exposes them to the user space as a virtual device file, `/dev/rootguard`. In the device driver, we export a function `lookup()` to our Security Server, which resides in the kernel module discussed in the following section, to look up existing policies.

Meanwhile, the driver defines file operations in its `file_operations` structure to support SuperuserEx access. When the user adds a new policy into `/etc/`

rootguard, the driver will be notified by SuperuserEx through Netlink (http://man7.org/linux/man-pages/man7/netlink.7.html) to synchronize the new policy into kernel memory. We also add a command in Android's initialization script /init.rc to map data of /etc/rootguard into kernel memory at boot-up. Moreover, to avoid overloading kernel memory, we design a compact policy format that minimizes RootGuard policies' size.

Kernel module. The RootGuard kernel module identifies operations from apps that have root privilege and decides whether they can be executed based on existing policies. It is made up of Linux Security Module (LSM) hooks, a system call hook, and Security Server.

LSM hooks. LSM supports use of its API to mediate kernel object access by placing hooks in the kernel code immediately prior to access.[6] We use such hooks to hold off apps' root operations, first querying RootGuard policies through the Security Server. For example, when an app tries to mount the system partition as writable, the system call sys_mount is invoked. Before the mounting operation is performed, the execution path enters our LSM hook, rg_mount ("rg" is short for RootGuard), which queries the Security Server to determine whether or not the operation is allowed. Because we focus on functions mostly related to root operations, we only use a small number of LSM hooks.

Security Server is designed specifically to enforce RootGuard policies, providing an interface through which LSM hooks can query Security Server for RootGuard decisions.

System call hook. Collecting information in LSM hooks is sometimes insufficient because the low-level operations involved in manipulating kernel objects might not provide thorough semantic knowledge regarding an app's high-level behaviors. To overcome this problem, we install a hook to interpose the system call sys_execve, which helps identify operations more accurately by inspecting the parameters of each shell command.

The traditional method for hooking a system call is to substitute the corresponding syscall table entry with the a customized function's address. However, unlike other syscalls, the actual address of sys_execve is not stored in the syscall table. Instead, it is invoked indirectly via a wrapper written by the ARM assembly.

A method for hooking Apple's iOS functions has been reported,[7] but it is not compatible with Android system calls. The wrapper code utilizes a relative jump instruction, which enables RootGuard to use an inline hooking technique to modify this jump's target location.

Security Server. Security Server is designed specifically to enforce RootGuard policies, providing an interface, as previously described, through which LSM hooks can query Security Server for RootGuard decisions. In addition, Security Server uses the information collected in the sys_execve hook to identify app operations, determine an operation's subject, and grant or deny root access according to RootGuard policies.

We use a unique identifier (UID) to distinguish app and nonapp operations because each Android app is assigned a UID number greater than 10,000, while UID numbers lower than 10,000 are reserved for system usage. Because apps can execute a single root command every time or, alternately, open a root shell to run a set of commands, we identity the subject (or UID) in each case by tracing ancestors in the privileging process, following the parent pointers (parent and real_parent) in its task_struct structure until an app process is found.

Default policies

RootGuard has a set of default policies to protect inexperienced users and defend against attacks from malware with root privileges.

We analyzed the most popular root-required apps in Google Play and defined four major groups:

- apps for browsing the entire file system and editing files,
- apps for backing up files,
- security apps providing real-time detection and protection, and
- apps for accessing and configuring hardware settings.

When installing an app, RootGuard asks the user to specify one of these groups according to the app's function. The app will be subject to policies corresponding to that group; otherwise, a more general set of conservative policies apply. We assume that a legitimate app will use root privileges only to accomplish what its description advertises and nothing else. For example, an app requesting root access for browsing files will not alter hardware configurations.

To help users monitor and customize how apps use root privileges, RootGuard records every operation in detail—including time, target, and so forth—and presents these in SuperuserEx. Users can inspect this log and, if they wish, modify an app's policies.

RootGuard's overall default policies cover seven subject areas: partitions, devices, system files, private data, processes, apps, and app components. These vary depending on the requesting app's assigned group, and provide three options:

- allow the operation without user interaction,
- deny the operation without user interaction, or
- ask for user permission.

Some examples, focusing on file browsing tools, illustrate how these default policies work in practice.

Mounting system partitions. The `/system` partition is mounted as read-only by default: allowing apps to mount system partitions as writable is dangerous because malware can leave back doors there. Only file-browsing apps are automatically granted permission to mount system partitions because when users want to edit files there, `/system` must be writable.

Accessing hardware devices. Devices are exposed in the user space as files located in the directory `/dev`. Once malware gains root privileges, it can steal sensitive data by accessing hardware devices—the GPS, for example—directly. Moreover, unexpected writings to these devices could crash the whole system. Therefore, apps not designed to manipulate hardware, such as file-browsing apps, are not allowed to read and write hardware devices.

Accessing system files or other apps' private data. Linux's discretionary access control cannot prevent apps with root privileges from accessing system files and the private data of other apps. RootGuard's default policies only allow file-browsing and backup tools to access system files and apps' private data.

Manipulating process memory. Behaviors of a process can be completely changed if its memory is altered. An app with root access can inject code into another process's memory and thereby interpose its functions; malware with such access can inject a malicious payload into a legitimate process and then conceal its behaviors. Therefore, RootGuard prevents apps other than security tools from manipulating the memory of other running processes; moreover, it prohibits any attempt to access SuperuserEx's memory in case of code-injection attacks.

EVALUATION

For evaluation purposes, we implemented RootGuard in Android 4.2 with Linux kernel 3.4.0. The tool has 4,762 lines of C/C++ code and 1,327 lines of Java code, measured by cloc 1.6, and a few XML files. We first describe potential threats posed by malware with root privileges and present four case studies that demonstrate how RootGuard mitigates such threats. We then discuss RootGuard's performance for users running popular apps and evaluate its overhead using a well-known benchmark app.

Threats posed by malware with root privileges

Android's application sandbox restricts apps' ability to access the file system and other system resources. Thus, while malware with user-granted permissions could create problems, the damage is generally limited because the operations do not have sufficient privilege to hide and inflict permanent infection. Malware granted root privileges, however, can circumvent all Android security measures, in addition to those of existing root-management tools. Six malware threats stand out.

Threat 1: Silent installation and uninstallation. Prior to installing an app, users are shown all permissions that app is requesting[8] and can cancel the installation if any are suspicious. However, after gaining root privilege, malware can install or uninstall any nonsystem apps directly by running `pm install` or `pm uninstall` commands. Consequently, new malware can be installed and legitimate apps might be uninstalled without user awareness.

RootGuard prevents apps other than security tools from manipulating the memory of other running processes; moreover, it prohibits any attempt to access SuperuserEx's memory in case of code-injection attacks.

Threat 2: Antimalware tool termination. By default, apps do not have the privileges necessary to terminate other apps. However, any malware with root privilege can run a `kill` command to terminate antimalware apps and make itself undetectable. Moreover, malware can disable key antimalware app components, such as services and broadcast receivers, thereby invalidating detection functionality. Even worse, some antimalware apps will continue to run even after key components have been disabled, because the main thread remains untouched; users have no idea the security function is compromised.

Threat 3: Irremovability. System apps in the `/system/app` directory cannot be uninstalled because the system partition they reside in is not writable. After gaining root privileges, however, malware can temporarily remount the system partition as writable, delete those apps, and insert itself in `/system/app`. Users cannot remove malware system apps in `/system/app`.

Threat 4: Access to other apps' private data. Generally, an app's private data cannot be read or altered by other

Table 1. AnTuTu benchmark results measuring RootGuard's runtime performance against the basic Android Operating System Project (AOSP) build.

AnTuTu test case	AOSP		RootGuard	
	Average	Standard deviation	Average	Standard deviation
Multitask	525.05	7.04	519.28	17.21
Dalvik	228.47	2.76	223.33	8.18
CPU integer	346.84	2.36	349.38	1.62
CPU floating point	81.47	0.84	79.47	2.60
RAM operation	269.21	1.51	261.05	11.09
RAM speed	584.84	13.71	574.43	32.87
Storage I/O	765.05	74.44	742.48	70.92
Database I/O	420.84	50.50	399.52	40.71
Total	**3,221.79**	**76.74**	**3,148.95**	**143.51**

apps, or is permission-protected and available only to apps with corresponding permissions. For example, to read or write contacts data, an app has to request *READ_CONTACTS* or *WRITE_CONTACTS* permissions. However, no such restriction applies to malware with root privileges, which can alter the access mode of the contacts database file to "global readable" and "global writable" and so steal the other app's data. In the most severe cases, malware directly modifies the signature database of security software to bypass any scanning, making itself undetectable.

Threat 5: Back doors. Malware with root privileges can create a back door in the infected system that allows it to bypass normal authentication. When it needs root permissions, the malware can leverage this back door to elevate privileges directly, without running exploit programs or requesting permission from su again.

Threat 6: Rootkits and bootkits. Rootkits are particularly surreptitious malware designed to bypass normal detection methods and enable privileged access without a system's awareness; bootkits are a kind of rootkit that first takes control during the boot process. Both user-mode and kernel-mode rootkits and bootkits can be implemented on the Android platform (https://viaforensics.com/mobile-security/dude-droid-sys-call-table-rootedcon-2013.html), but installing them requires root privilege. For example, the first bootkit found on Android, DKFBootKit (www.csc.ncsu.edu/faculty/jiang/DKFBootKit), needs root privileges to install the bootkit payload.

Case studies showing RootGuard's effectiveness

To demonstrate RootGuard's effectiveness in mitigating malware attacks, we chose three real-world malware samples and built a malicious proof-of-concept (PoC) app that, together, represent the six threats described above. We assume here that RootGuard operates only according to its default policies.

RootSmart (Threats 1, 3, and 5). RootSmart (www.csc.ncsu.edu/faculty/jiang/RootSmart) can download other malware from remote servers and install them silently. To install an app without the user's awareness, RootSmart executes the `pm install` command. Similarly, to uninstall an app, such as an antimalware tool, RootSmart executes `pm uninstall`. RootGuard can prevent these operations because its policies do not allow an app to run `pm install/uninstall` without the user's permission. RootGuard also prohibits RootSmart from creating a backdoor (`/system/xbin/smart/sh`) into the system partition because RootGuard does not allow an app to remount the system partition as writable.

AVPass (Threat 4). AVPass (http://contagiominidump.blogspot.hk/2014/01/android-avpass.html) uses root privileges to modify the signature databases of many popular antimalware apps. RootGuard can stop AVPass because an app is not allowed to modify another app's private files without permission.

DKFBootKit (Threat 6). As the first bootkit targeting Android, DKFBootKit mounts the system partition as writable, copies itself into the `/system/lib` directory, replaces several commonly used utility programs (for example, `ifconfig` and `mount`), and alters bootstrap-related daemons, like vold and debuggerd, and scripts. However, one of its primary steps—remounting system partition as writable—cannot be performed according to RootGuard policy, which immediately prevents bootkit installation.

PoC app (Threat 2). We built a PoC malware app specifically to demonstrate Threat 2. This app first enumerates system processes. When it finds antimalware software, it terminates that process by executing the `kill <pid>` command, using root privilege. RootGuard denies this operation because an app is not allowed to kill other processes without the user's permission. In addition, our PoC malware can query key components of an antimalware tool and disable them by executing `pm disable`. RootGuard policy also prohibits this action.

RootGuard-enhanced device user experience

We ran five popular Android apps from Google Play that require root privilege—Titanium Backup, CPU Tuner, Root Explorer, LBE Privacy Guard, and Root App Delete—in a RootGuard-enhanced device. Our results show that when a user specifies the proper group during installation, the app will function normally and fully; the user need not add any additional configurations to the default policy. If an inappropriate group is specified, the app's root payload will not function normally, but the user can inspect its operation log in SuperuserEx and then modify the policy as needed.

Performance overhead

To measure RootGuard's runtime performance overhead, we ran the widely used benchmark app AnTuTu (www.antutu.net) in two Google Nexus S devices, one having the basic AOSP build installed and the other with RootGuard. For both devices, we confirmed that the same number of apps were loaded and running at any time.

Table 1 shows the results for 50 runs of the AnTuTu benchmark for AOSP and for RootGuard. In general, RootGuard introduces only very low added overhead, within one standard deviation of the AOSP result. RootGuard has a number of checks in read and write operations, so the performance loss in the storage I/O and database I/O tests are reasonable. RootGuard's RAM operation and RAM speed scores are a little lower than AOSP's, but within an acceptable range. The CPU integer and CPU floating-point test scores should not be affected because they do not involve operations RootGuard has interests in, so the differences here might result from measurement noise. Multitask and Dalvik scores measure user experience performance and are influenced only slightly because RootGuard monitors operations.

OTHER SECURITY CONSIDERATIONS

Attackers might employ kernel-mode rootkits or exploit kernel vulnerabilities to attack RootGuard. Operating at the same security level as the OS, kernel-mode rootkits can invalidate RootGuard's hooks through direct kernel object modification (DKOM). To mitigate kernel-mode rootkit incursions, we disabled support for the Linux loadable kernel module (LKM) when compiling the kernel. In future work, we will enhance RootGuard's ability to monitor and manage kernel module loading. Note that a few kernel-mode rookits are loaded by manipulating kernel memory via the device file `/dev/mem`; because RootGuard prevents apps from reading or writing `/dev/mem`, these do not pose a problem.

RootGuard has components in the user space—specifically, SuperuserEx and the policy storage database—that adversaries may attack. Although we adopt additional measures to protect them—for example, denying any access to SuperuserEx's memory by other apps to defeat potential code-injection attacks—as a kernel-level mechanism RootGuard cannot mitigate all kernel attacks and so could be disabled if attackers successfully exploit kernel vulnerabilities. In future work, we will explore using virtualization techniques to provide better protection.

In addition, an attacker who knows RootGuard's default policies can design a malicious app specifically so that it fools the user into categorizing it as a file-browsing tool, thus allowing the app to steal sensitive data. Although RootGuard cannot directly defeat such attacks, it can inform the user that root operations are being executed by showing a message in the notification bar. If the user did not trigger the root operation, he or she can check the SuperuserEx records for details. Moreover, RootGuard and antimalware apps complement one another; it is possible to use them in tandem to defeat advanced malware.

Overall, RootGuard improves the security of rooted Android phones, effectively mitigating attacks by malware with root privileges without affecting app performance and at the same time achieving low overhead. Future work will extend RootGuard's ability to collect additional context information, including the sequence and the pattern of system calls, so as to further facilitate user decision making and prevent malicious app behaviors, particularly by those using native code. C

Acknowledgments

This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E); the National Natural Science Foundation of China (No. 61202396); the Open Fund of Key Laboratory of Digital Signal and Image Processing, Guangdong Province; the

Shenzhen City Special Fund for Strategic Emerging Industries (No. JCYJ20120830153030584); and the Hong Kong Innovation and Technology Fund (No. ITS/073/12).

References

1. A. Shabtai et al., "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy*, vol. 8, no. 2, 2010, pp. 35–44.
2. E. William, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security & Privacy*, vol. 7, no. 1, 2009, pp. 50–57.
3. S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," *Proc. 20th Annual Network & Distributed System Security Symp.* (NSDD 13), 2013; www.internetsociety.org/doc/security-enhanced-se-android-bringing-flexible-mac-android.
4. S. Bugiel, S. Heuser, and A. Sadeghi, "Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," *Proc. 22nd Usenix Security Symp.* (Security 13), 2013, pp. 131–146.
5. E. Chin et al., "Analyzing Inter-application Communication in Android," *Proc. 9th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 11), 2011, pp. 239–252.
6. C. Wright et al., "Linux Security Modules: General Security Support for the Linux Kernel." *Proc. 11th Usenix Security Symp.* (Security 02), 2002, pp. 17–31.
7. D. Damopoulos et al., "Exposing Mobile Malware from the Inside (Or What Is Your Mobile App Really Doing?)," *Peer-to-Peer Networking and Applications*, Dec. 2012; doi:10.1007/s12083-012-0179-x.
8. A. Felt et al., "Android Permissions Demystified," *Proc. 18th ACM Conf. Computer and Comm. Security* (CCS 11), 2011, pp. 627–638.

***Yuru Shao** is a research assistant in the Department of Computing at the Hong Kong Polytechnic University. His research focuses on smartphone security. Shao received a BEng in information security from Wuhan University, China. Contact him at csyshao@comp.polyu.edu.hk.*

***Xiapu Luo** is a research assistant professor in the Department of Computing at the Hong Kong Polytechnic University and is affiliated with the Shenzhen Research Institute of the Hong Kong Polytechnic University. His research interests include smartphone security, network security and privacy, and Internet measurement. Luo received a PhD in computer science from the Hong Kong Polytechnic University and was a postdoctoral research fellow at the Georgia Institute of Technology. He is a member of IEEE. Contact him as corresponding author at csxluo@comp.polyu.edu.hk.*

***Chenxiong Qian** is a research assistant in the Department of Computing at the Hong Kong Polytechnic University. His research focuses on security and privacy, particularly in mobile environments. Qian received a BEng in software engineering from Nanjing University, China. He is a member of IEEE. Contact him at cscqian@comp.polyu.edu.hk.*

Call for Articles

IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable information to software developers and managers to help them stay on top of rapid technology change.

IEEE Software

Author guidelines:
www.computer.org/software/author.htm
Further details: software@computer.org
www.computer.org/software