An Android Security Extension to Protect Personal Information against Illegal Accesses and Privilege Escalation Attacks

Yeongung Park
The Attached Institute of ETRI
Yuseong, Daejeon, Korea
santapark@ensec.re.kr

Chanhee Lee, Jonghwa Kim, Seong-Je Cho*, and Jongmoo Choi
Dankook University
Yongin-si, Gyeonggi-do, Korea
lchan12@nate.com, {zcbm4321, sjcho, choijm}@dankook.ac.kr

Abstract

Recently, it is widespread for malware to collect sensitive information owned by third-party applications as well as to escalate its privilege to the system level (the highest level) on the Android platform. An attack of obtaining root-level privilege in an Android environment can form a serious threat to users from the viewpoint of breaking down the whole security system. This paper proposes a new scheme that effectively prevents privilege escalation attacks and protects users' personal information in Android. Our proposed scheme can detect and respond to malware that illegally acquires root-level privilege using pWhitelist, a list of trusted programs with root-level permission. Moreover, the scheme does not permit even a privileged program to access users' personal information based on the principle of least privilege. As a result, it protects personal information against illegal accesses by malicious applications even though they illegally obtain root-level permissions by exploiting vulnerabilities of trusted programs.

Keywords: Android, permission model, personal information, privilege escalation attack, private data protection

1 Introduction

The trustworthiness of third-party applications in Android vary widely, so Android treats all applications as potentially malicious. Each application runs in a process with a low-privilege user ID, and applications can access only their own files by default. Each application also runs in its own virtual machine. To be more secure operating system (OS) for mobile platforms, Android provides some security features: robust security at the OS level through the Linux kernel, mandatory application sandbox for all applications, and application-defined and user-granted permissions [1, 3, 4, 9]. The Linux kernel provides Android with a user-based permissions model, process isolation, and the ability to remove unnecessary and potentially insecure parts of the kernel.

In the permission-based security model, applications are explicitly given permissions for operations that they need and each application runs as its own user. Unless the developer explicitly exposes files to other applications, files owned by one application cannot be read or written by another application. This model allows users to limit the impact of malicious applications on their systems and on their personal privacy. However, Android's security model has been shown to be still vulnerable to application-level privilege escalation attacks [3, 4, 9, 15, 16] including confused deputy attacks and collusion attacks. Basically, Android does not deal with transitive privilege usage, which allows applications to bypass restrictions imposed by the permission model. Therefore, the permission-based security model cannot fully protect user's personal information under privilege escalation attacks and a variety of privilege escalation attacks have been reported on Android. Many studies have shown that malicious applications can be installed

Journal of Internet Services and Information Security (JISIS), volume: 2, number: 3/4, pp. 29-42

*Corresponding author: Tel: +82-31-8005-3239, Fax: +82+31-8021-7398

on Android devices while stealing users' private data through traditional attacks or privilege escalation attacks (see, for example, the BaseBridge [16], DroidKungFu [9, 16, 11], DroidDream [9, 16, 2], and GingerMaster [10]).

In Android, if a process obtains root privilege, the process has full access to the file system (we refer to it as privileged). Since a privileged process has full access permission to the system, privilege escalation attacks grant malware or unauthorized applications with full access permission to the system. If an attacker can illegally elevate himself/herself to root privileges, he/she can gain access to any sensitive data on the user's smartphone.

To deal with privilege escalation attacks and to guard personal information, various security extensions and enhancements to Android's framework have been proposed such as XManDroid [3, 4], RGBDroid [15], Kirin [7], Saint [14], Apex [13], TaintDroid [6], QUIRE [5], so on. However, as we will elaborate in further detail on related work in Section 2, none of the existing solutions sufficiently and efficiently address users' private information leakage via privilege escalation attacks. Some approaches cannot detect transitive permission usage attacks or fail to protect the system against malware and sophisticated runtime attacks. Others rely on the users to take security decision or suffer from inefficiency.

In this paper, we propose a new kernel-level security model to efficiently prevent privilege escalation attacks and protect users' private data against illegally privileged programs in an Android environment. Our proposed model can protect an Android system against privilege escalation attacks by introducing two new concepts, pWhitelist and PDP (Private Data Protection). pWhitelist is a list of trusted programs with root privileges. Any program not on pWhitelist cannot run with root privileges. If any malware or application not on the list tries to open, read, or write files owned by applications after gaining root privileges via privilege escalation attacks, our model detect and protect the trial using pWhitelist.

To secure personal information including private data in smartphones, our PDP mechanism disallows trusted programs to access sensitive data owned by applications through enforcing the least privilege principle in permission-based access control. An attacker or malware can evade the pWhitelist mechanism by directly exploiting vulnerabilities in trusted programs with root privileges and escalating privileges. The PDP mechanism can defeat the pWhitelist evasion technique prohibiting uncontrolled access to application data by trusted and privileged programs.

Our model uses system-centric approaches by allowing the kernel to enforce security decisions based on user ID of applications and files. So it does not require application developers to add security features to their applications. We demonstrate that the proposed model efficiently detects privilege escalation attacks and mitigates the effects of the attacks by carrying out real implementation based experiments.

The remainder of this paper is organized as follows. After discussing related work in Section 2, we propose design and implementation of our security model to protect users' private information against privilege escalation attacks in Section 3. In Section 4, we present the effectiveness of our proposed model through some experiments according to realistic scenarios in which malicious program tries to access illegally to private information. Section 5 evaluates performance of the proposed model. Finally we draw conclusions on this work and describe future work in Section 6.

2 Related Work

2.1 Privilege Escalation Attacks

Android's security framework (enforcing sandboxing and permission checks) is not sufficient for transitive policy enforcement allowing privilege escalation attacks [3, 4, 15, 16]. The privilege escalation attacks including confused deputy and collusion attacks have been reported on Android showing the vulnerabilities of Android's security framework.*Confused deputy attacks* concern scenarios where a malicious application exploits the vulnerable interfaces of another privileged (but confused) application.

Collusion attacks concern malicious applications that collude to combine their permissions, allowing them to perform actions beyond their individual privileges.
Malicious codes for the Android platform have appeared to steal personal data or escalate privileges. Examples are BaseBridge [16], DroidKungFu [9, 16, 11], DroidDream [9, 16, 2],and GingerMaster [16, 10]. *Android.BaseBridge* [16] performs privilege escalation attack to elevate its privileges so that it can download and install additional applications onto user's smartphone. When an infected application is installed, it attempts to exploit the udev Netlink Message Validation Privilege Escalation Vulnerability in order to obtain root privileges. The BaseBridge malware uses HTTP to communicate with a central server and transmit potentially identifiable personal information. BaseBridge can also send premium-rate SMS messages to predetermined numbers. BaseBridge has another name AdSMS, and has a number of variants.
DroidDream [16, 2] has appeared in Google's market. DroidDream application steals personal information and is very much like traditional trojans seen on the desktop. Some researchers have referred to the DroidDream series of trojans as a mobile botnet. More advanced malware has appeared in third-party markets. One example is DroidKungFu [16, 11], which takes advantage of an escalation of privilege exploit called "Rage Against The Cage". At least four other well-known escalations of privilege could have been used. The DroidKungFu malware collects personal information of the phone's owner such as IMEI(International Mobile Equipment Identity), Device ID and SDK version and sends to a remote server, then tries to obtain a root shell. The root shell receives commands from a C&C(Command and Control) server, and installs a hidden backdoor application. The smartphone infected with DroidKungFu has thus become a bot or a zombie.
GingerMaster [16, 10] is similar to DroidKungFu. It infects normal Android applications. Once an infected application is installed, it registers a service, collects personal data on the user's device, sends the information to a remote server and tries to obtain a root shell. The root shell installs another malicious application that receives commands from a C&C server.

2.2 Security Extensions to Android

Various solutions have been proposed in the last few years for defending an Android system against privilege escalation attacks. Android provides third-party applications with an extensive API that includes access to phone hardware, settings, and user data. Access to privacy- and security-relevant parts of Android's rich API is controlled by an install-time application permission system. The **Kirin** system [7] is an extension to Android's application installer that focuses on enforcing install-time application permissions. It denies the installation of applications that may encompass a set of permissions that violates a given system policy at install-time. The Kirin uses predefined security rules templates to match dangerous combinations of permissions requested by applications. As Kirin analyzes individual applications and employs static policy, it does not protect the user from applications that collaborate to leak data or protect applications from one another.
Saint [14] extends the functionality of the Kirin system to allow for runtime inspection of the full system permission state before launching a given application. It adopts a fine-grained access control model and governs install-time permission assignment and run-time use. In order to prevent privilege escalation attacks, Saint requires application developers to assign appropriate security policies on their application's interfaces and add security features to their applications. Since application developers might fail to consider all security threats, developer-defined permission systems are more likely to be error-prone than system-centric approaches. **Apex** [13] presents another solution for the same problem where the user is responsible for defining run-time constraints on the top of the existing Android permission system. Apex does not address privilege escalation attacks where permissions are split over multiple applications. Like Saint, it unfortunately relies on the user to take security decisions. These Saint and Apex approaches

allow users to specify static policies to shield themselves from malicious applications, but do not allow applications to make dynamic policy decisions.

TaintDroid [6], a sophisticated framework, presents dynamic taint analysis technique to prevent unauthorized leakage of sensitive data and runtime attacks. The framework attempts to tag objects with metadata in order to track information flow and enable policies based on the path that data has taken through the system. TaintDroid enforces its taint propagation semantics by instrumenting an application's DEX bytecode to tag every variable, pointer, and IPC message that flows through the system with a taint value. It is to restrict the transmission of tainted data to a remote server by monitoring the outbound network connections made from the device and disallowing tainted data to flow along the outbound channels. TaintDroid has the shortcoming that it mainly addresses data flows, whereas privilege escalation attacks also involve control flows. Tracking the control flow with TaintDroid will likely result in much higher performance penalties. Besides, it cannot detect attacks that exploit covert channels to leak sensitive information [3, 4].

QUIRE [5] is a lightweight provenance system that prevents privilege escalation attacks via confused deputy attacks. It is focused on providing provenance information and preventing the access of sensitive data, rather than in restricting where data may flow. When there is an Inter Process Communication (IPC) request between Android applications, it forces the applications to operate with a reduced privilege of its caller by tracking the call chain of IPCs. QUIRE's approach requires only the IPC subsystem be modified with no reliance on instrumented code, therefore QUIRE can work with applications that use native libraries and avoid the overhead imparted by instrumenting code to propagate taint value. However, this approach is application-centric not system-centric. QUIRE does not address privilege escalation attacks that are based on maliciously colluding applications. Since the Inter-Component Communication (ICC) call chain is forwarded and propagated by the applications themselves, colluding applications may force the ICC call chain to obscure the originating application, and hence, circumvent QUIRE's defense mechanism. Furthermore, the unexpected denial of access by the receiver of the call chain might lead to application dysfunction/crash on the caller's side [3, 4].

XManDroid (eXtended Monitoring on Android) [3] is a middleware-level security framework that extends the monitoring mechanism of Android to detect and prevent application-level privilege escalation attacks at runtime based on a system-centric system policy. It provided a solution that addressed the recent privilege escalation attack that exploited covert channels of the Android system. It protects against some privilege escalation attacks, and allows for enforcing a more flexible range of policies, applications may launch denial of service attacks another applications (e.g., connecting to an application and thus preventing it from using its full set of permissions). However, XManDroid does not allow the flexibility for an application to regain privileges which they lost due to communicating with other applications [QUIRE].

Bugiel et al. recently extend their previous XManDroid framework with a kernel-level module [4]. They propose another security framework for Android that monitors application communication channels in Android's middleware and in the underlying Linux Kernel and ensures that they comply with a system-centric security policy. They conduct a heuristic analysis of Android's system behavior (with popular applications) to identify attack patterns, classify different adversary models, and point out the challenges to be tackled. They establish semantic links between IPCs and enable the reference monitor to verify the call-chain at the middleware level, and realize mandatory access control on the file system and local Internet sockets at the kernel level. They also provide a callback channel between the kernel and the middleware.

RGBDroid(Rooting Good Bye on Droid)[15], our previous work, is a very recent Android security extension which detects and responds to the attacks associated with escalation or abuse of privileges. RGBDroid has introduced pWhitelist and Criticallist. The concept and role of pWhitelist are the same as those described in this paper. Criticallist has a list of critical resources that can affect operations and

behaviors of Android framework and user applications. The Criticallist presents the core resources that even privileged process cannot modify. If the resources are modified, consistency of Android system is broken. It is efficient to protect privilege escalation attacks of malicious applications and to maintain the integrity of a system. However, pWhitelist can be evaded by attacks which exploit vulnerabilities of trusted programs with root privileges. Therefore, RGBDroid cannot protect users' private data owned by applications against the attacks that bypass the pWhitelist. In this paper, we extend RGBDroid with a kernel-level PDP (Private Data Protection) mechanism to overcome the limitations of the RGBDroid. In the following sections, we shall describe the mechanism in our security model and their interaction.

3 Design and Implementation

Android provides several basic security mechanisms such as sandbox and file access control using UID (User ID) and GID (Group ID). However, it is based on the UNIX-like access control model so that a root-privileged application can access any resources including privacy-sensitive information. Hence, attackers naturally focus on obtaining the root privilege. Once obtaining it, they conduct the privilege escalation attacks such as acquiring the root shell or accessing information through other legitimate applications to dominate the target system and to steal private information.

In this paper, we propose a novel scheme to protect the privilege escalation attacks and users' sensitive information. The privilege escalation attack is defined as a status where attackers (or malicious applications) exploit vulnerabilities of a trusted program with the root privilege so that they controls the program, or a type of intrusion that takes advantage of vulnerabilities of system programs to grant the attackers elevated access to the system. For instance, the *vold* (Volume manager daemon), that is a process executed with the root privilege in the Android system, has been exploited and performed payload codes given by attackers.

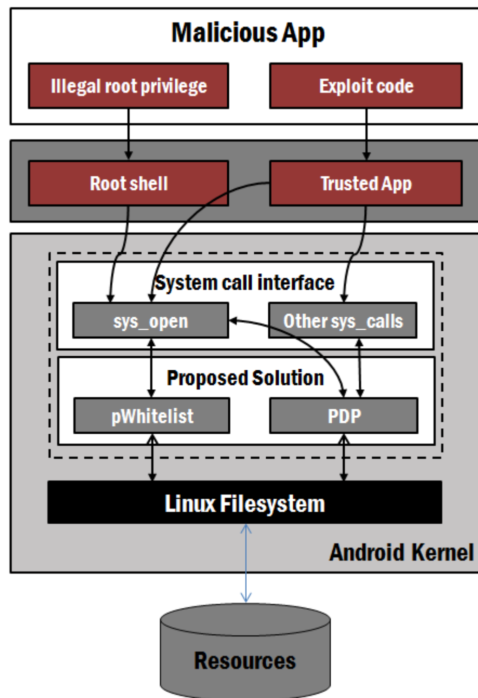

Figure 1: Protection mechanisms for the privilege escalation attacks

The proposed scheme consists of two mechanisms, namely *pWhitelist* and *PDP* (Private Data Protection), as shown in Figure 1. The scheme is a kind of response-centric access control, suitable to mobile environments. We first explore the characteristics of the root privilege usage scenarios in the Android system and find out that the root privilege is used in a limited manner from only some predefined authorized processes. This exploration triggers us to design the *pWhitelist* mechanism that can prevent malicious applications and a shell from being executed with the root privilege.
However, the *pWhitelist* mechanism cannot restrict inappropriate accesses to user's private information through authorized and trusted processes. In other word, an attacker exploits vulnerabilities of the existing authorized processes and disguises himself as an authorized one while accessing resources. To overcome this problem, we design a second countermeasure, that is the PDP (Private Data Protection) mechanism.
For designing PDP, we investigate the interrelation between the processes and resources in the Android system and classify resources into two groups according to the access level, as presented in Figure 2. One group is the system-level resources that contain Linux kernel, runtime and system libraries such as core.jar and framework.jar used for Android framework. The other is the user-level resources including ContactsDB (database for contacts applications), messages, browser data and multimedia data. PDP makes use of the fact that, in the Android system, user-level resources are hardly accessed by the root-privileged processes, only accessed under in a well-defined path.

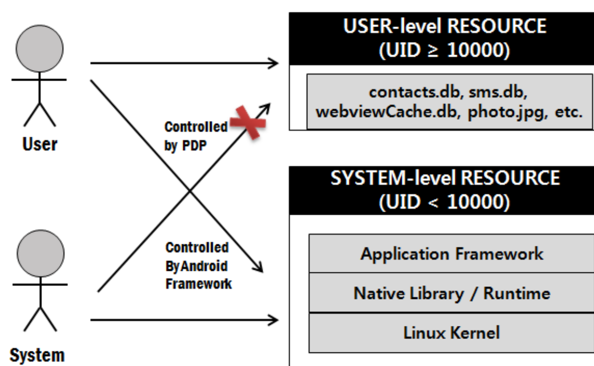

Figure 2: Access Control to User and System Resources

According to the resource classification, the proposed scheme monitors the accesses of the root privileged processes and controls their behaviors. When an access request is out of the rules of the pWhitelist or PDP mechanisms, the request is denied. For the monitoring and control purpose, we hook the system calls in the Linux kernel using the LKM (Loadable Kernel Module) programming interfaces. The system call hooking does not require any modifications of the Android platform, enhancing portability to various Android versions. Also, LKM allows dynamic insertion/deletion of software components into/from the Linux kernel without the troublesome kernel rebuilding and rebooting.

3.1 pWhitelist

pWhitelist is defined as a set of trusted and authorized programs that can make use of the root privilege. A program that is not included in pWhitelist cannot access resources with the root privilege. Therefore, if a malicious program has obtained the root privilege using the privilege escalation attacks, it is not allowed to access resources when it is not in pWhitelist.
In the Android system, most resources are manipulated as files. So, we can implement the pWhitelist mechanism by hooking the sys_open() and sys_creat() system calls. Figure 3 presents the pseudo code

for the pWhitelist implementation.

```
unsigned shortuid;
unsigned shorteuid;

if program_uid == 0 OR program_euid == 0
if !(procname  ∈ { prognames_in_whitelist} )
        return deny;
call original_sys_call;
```

Figure 3: Pseudo code for pWhitelist mechanism

In the PC environments, any user can become the super-user if he/she knows the root password. Also, any program created by the super-user can utilize the root privilege. In other word, there are a lot of programs that make use of the root privilege, which makes the interrelations between the root-privileged programs and resources quite complex. Hence, employing the pWhitelist mechanism proposed in this paper is not easy in the PC environments. Actually, LIDS (Linux Intrusion Detection System) [8], which takes the similar approach of this paper in the PC environments, requires considerable configuration overheads.

On the contrary, in the Android-based smartphones, users cannot make use of the root privilege directly. Also there are limited numbers of programs that can use the root privilege. Our investigation uncovers that there are 26 authorized programs who has the root privilege. So, in the current implementation, we make pWhitelist with the 26 programs. Any program except the 26 programs is denied to open or create resources with the root privilege, which eventually make it impossible to read or write user information. It also prevents a shell from being executed with the root privilege since any shell program is not in pWhitelist.

The list of the authorized programs is managed in a system file. While the pWhitelist mechanism is initialized, it reads the file and builds pWhitelist. Therefore, when a new program that can use the root privilege is added in the Android system, we can reflect it easily by just updating the file.

To protect the pWhitelist against malicious insertion and modification attacks, our scheme can store it in an encrypted form. A secret key or a private key is needed to encrypt or decrypt the pWhitelist. A detailed study of issues involved with key management is beyond the scope of this paper.

Table 1: Hooked system calls for PDP implementation

System call	Hooking purpose
sys_open()	Check the process that tries to access resources (files) owned by user
sys_chown()	Check the process that tries to modify an owner of resources (files)
sys_symlink()	Check the process that tries to create a symbolic link
sys_unlink()	Check the process that tries to delete a name from file system or to remove a symbolic link
sys_link()	Check the process that tries to link an existing file to a new file
sys_chmod()	Check the process that tries to change the permissions of each given file according to mode
sys_rmmod()	Check the process that tries to remove a module from the Linux kernel

3.2 PDP (Private Data Protection)

In the Android system, a user interacts diverse applications and each application manages a variety of user information. For instance, a contacts application manipulates the ContactsDB (database for contact information), which is owned by the application. Since the ContactsDB includes privacy data, it needs to be protected from malware or malicious applications. Also, since the ContactsDB is included in the user-level resources, as shown in Figure 2, it needs to be protected from the root-privileged programs, as well (or to be accessed under a well-defined and controlled way). For this purpose, we introduce the PDP mechanism.

PDP is a capability that restricts a root-privileged programs (or processes) from accessing resources owned by user-level applications. This mechanism is devised to take care of malicious behaviors that bypass the pWhitelist mechanism. Specifically, a malicious application may exploit vulnerabilities of an authorized process in pWhitelist and delegate its requests to the process. Then, it can bypass the pWhitelist mechanism. However, the PDP mechanism can identify that a root-privileged program tries to access a user-level resource and can deny the request.

To implement the PDP mechanism, we hook seven system calls summarized in Table 1, using LKM programming. The hooking algorithm is presented in Figure 4. It first identifies whether a requesting process has the root privilege using UID or EUID. Then, it checks the type of requested resource and denies the request if it is the user-level resource. Hence, it can prevent attackers and malicious applications from accessing user-level files via authorized processes using the privilege escalation attacks. Note that, in the Android system, the UID of root is 0 and UIDs of user-level applications are larger than 10,000. Eventually, the TPA mechanism limits any root-privileged process to access any user-level resource.

```
unsigned short uid;
unsigned short euid;

if program_uid == 0 OR program_euid == 0
if file_owner_uid >= 10000
        return deny;
call original_sys_call;
```

Figure 4: System call hooking for implementing PDP mechanism

4 Experiments

To verify the effectiveness of our proposed scheme, we conduct two experiments based on two malicious behaviors that occur frequently in real Android systems. To this end, we first survey several suspicious codes in the existing Android market and analyze their behaviors especially focusing on the privilege escalation attacks. Then, we choose two malicious behaviors, illegal root shell acquisition and information leakage, and investigate how our proposed scheme reacts such attacks.

The experiments have been carried out on the Android Emulator 2.3, which run on the 64-bit Ubuntu version 10.10 on our experimental system consisting of Intel i5 processor and 8GB DRAM.

4.1 Root Shell Acquisition

Once obtaining the root privilege, most illegal codes try to execute the root shell to retain the root privilege and to exploit the privilege escalation attacks. However, according to the security model of Android,

smartphone users cannot make use of the root privilege at first hand. In other word, the shell program that takes charge of user interfaces does not need to be executed as the root privilege. Therefore, preventing a shell from being invoked as the root privilege is one of the effective ways to protect the privilege escalation attacks.
Figure 5 shows a test case where, using *adb* (Android debugging bridge), an attacker tries to execute the root shell in the traditional Android system that does not employ our scheme. It reveals that the root shell, unauthorized by the Android security model, is initiated successfully. Then, it serves user requests, not only a normal request but also a malicious request accessing various files and resources including privacy sensitive data, with the root privilege.

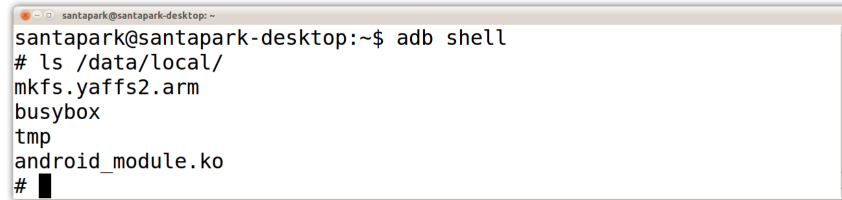

Figure 5: A successful execution of root shell before setting up our *pWhitelist* mechanism

The same test case works differently in the Android system integrated with our proposed scheme, as shown in Figure 6. To initiate the root shell, it needs to open some system libraries such as libc.so with the root privilege. However, our proposed mechanism identifies that the shell is not in pWhitelist, denying the open request, which eventually makes it fail to acquire the root shell.

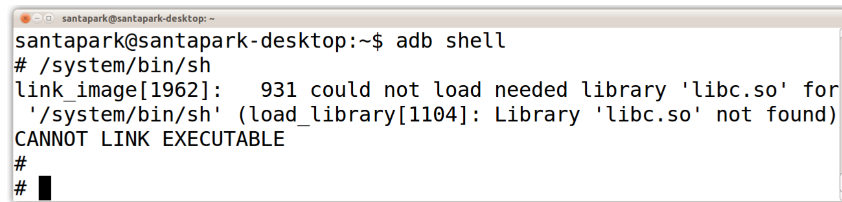

Figure 6: A failure message when trying to execute root shell after setting up our *pWhitelist* mechanism

Contrasting Figure 5 with Figure 6 demonstrates that, even though the Android security model recommends that a shell does not need to be executed with the root privilege, it is not enforced in the traditional Android system, vulnerable to the privilege escalation attacks. However, our proposal restricts the root shell execution, leading to frustrate the privilege escalation attacks since the attackers cannot access any files or resources with the root privilege. In addition, our scheme can prevent the attacks using the reverse shell since they are also filtered out using the *pWhitelist* mechanism.

4.2 Information leakage

Users of an Android system generate a variety of information such as contacts, messages, browsing data and logs. Such information is managed by separated files where each file is owned by the application who creates the file. Android supports the UNIX-like file access control facility so that a file can be manipulated only by the owner application. In addition, an application with the root privilege can access any files. Hence, attackers try to obtaining the root privilege by exploiting various system vulnerabilities and try to access users' personal information by performing the privilege escalation attacks.
Our *pWhitelist* mechanism does not allow an unauthorized application to be executed with the root priv-

ilege. However, the mechanism cannot protect user's personal information from attacks exploiting vulnerabilities of authorized programs. Specifically, attackers can exploit vulnerabilities of the authorized or trusted programs that have the root privilege and can delegate access privileges to the applications using the attacks such as ROP (Return Oriented Programming) attack [12]. Then, since the read and write system call are performed with the root privilege, privacy-sensitive user information can be leaked. Figure 7 shows our experiment that draws information illegally from the contact database file owned by a user contacts application in the traditional Android system. We can extract the file using the root privileged emulator and display the contents with the SQLite Database Browser 2.0b1. It shows that the privilege escalation attacks based on the existing legitimate applications with root privilege are feasible in the current Android systems.

data1	data2	data3
Jhon Jackson	Jhon	Jackson
Carl Iverson	Carl	Iverson
Park HERO	Park	HERO
Golden Spear	Golden	Spear
Welcome Google	Welcome	Google

Figure 7: Screenshot leaking and showing successfully contact DB before setting up our PDP mechanism

The PDP mechanism is devised to prevent such information leakage. It forbids even a root-privileged application from accessing user level resources by monitoring system calls. Figure 8 shows the effect of PDP. Without PDP, as shown in the left screenshot of the figure, user information can be accessed through the root-privileged applications, even though the applications and the data are not associated. On the contrary, with PDP, as shown in the left screenshot of the figure, accessing user information is not permitted.

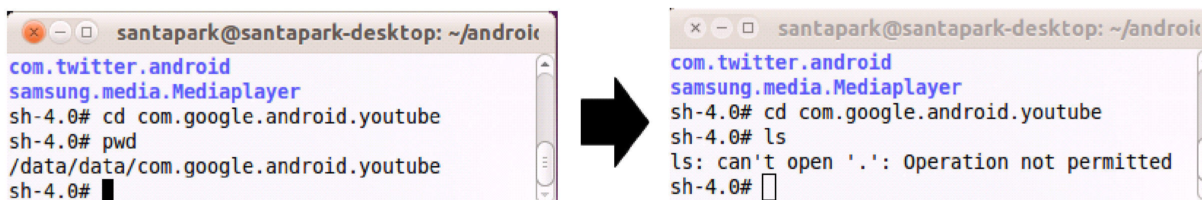

Figure 8: Information accessibility before and after setting up our PDP mechanism

The results presented in Figure 8 reveal that our proposed PDP mechanism controls appropriately the accesses requested from applications to resources using the resource classification discussed in Figure 2. Therefore, even though an attacker disguises his application as a legitimate program, the PDP mechanism can detect and block the attacker, leading to protect users' personal information against illegal accesses or malicious behaviors.

5 Performance evaluation

One issue of our proposed scheme is the performance concern. It may cause non-trivial computational overheads since the *pWhitelist* and *PDP* mechanisms require system to monitor every system calls listed in Table 1. To evaluate the overheads quantitatively, we measure the I/O throughput using the AndroBench 3.1, a storage performance measurement benchmark, with and without our scheme in our

experimental environment. We choose this benchmark since our scheme mainly monitors the file I/O related system calls such as open and link, which affects the storage performance greatly.

5.1 I/O throughput

To measure I/O throughput, we carry out insert, update, and delete transactions on SQLite, 300 times respectively, using the AndroBench 3.1. Table 2 and Figure 9 summarize the results. To enhance the measurement accuracy, we conduct the measurement ten times and use the average values for comparison.

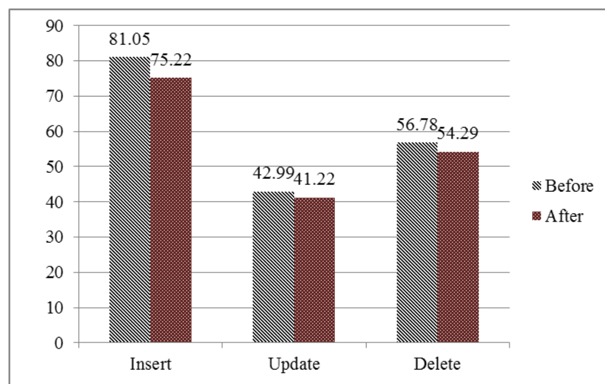

Figure 9: Comparison of I/O performance before and after introducing our mechanisms (Unit: TPS)

The measurement results show that the average TPSs for insert, update and delete transactions in the traditional Android system (without our scheme) are 81.05, 42.99 and 56.78, respectively. When we introduce our scheme, these become 75.22, 41.22 and 52.49, respectively. It means that our scheme incurs 7.19%, 4.12% and 4.39% additional overheads for insert, update, and delete transactions with an average of 5.58%.

Table 2: I/O performance before and after introducing our mechanisms (Unit: TPS [Transactions Per Second])

Count	Without our scheme			With our scheme		
	Insert	Update	Delete	Insert	Update	Delete
1	81.9	44.29	51.11	77.67	42.2	50.7
2	87.51	44.2	56.03	79.8	44.67	54.97
3	84.86	38.61	57.5	73.18	41.3	48.3
4	79.82	37.55	58.37	77.72	43.29	59.37
5	93.77	41.59	61.34	73.89	38.25	62.15
6	63.11	42.42	58.51	76.31	41.64	50.67
7	82.34	42	50.78	69.01	40	53.46
8	82.03	41.6	59.19	72.79	37.02	55.72
9	75.96	36.01	57.01	72.79	41.66	56.45
10	79.17	43	57.95	79.03	42.18	51.15
Average	81.05	42.99	56.78	75.22	41.22	54.29

5.2 Analysis

The results from Table 2 and Figure 9 reveal that, even though our scheme monitors file related system calls, it does not cause considerable overheads. It decreases the I/O throughput 5.58% on average. We carefully argue that, considering the security enhancement achieved by the scheme, the overhead can be acceptable. Also, we find out that the privilege escalation attacks in the Android environment are rather simple and predictable, comparing with the PC environment. We make use of such characteristics to design and implement our scheme with a lightweight manner.
Our proposal is a kind of multi-level protection mechanism. It first protects the illegal root shell acquisition using the *pWhitelist* mechanism, which is one of the popular ways for attacking user information. Then, for the attacks that evade the *pWhitelist* mechanism, it employs the second countermeasure, that is, the *PDP* mechanism, improving security one step further. Hence, when a performance concern becomes serious, we can apply the mechanisms selectively, while balancing the tradeoffs between performance and security.

6 Conclusions

The paper has proposed a new Android security scheme to prevent privilege escalation attacks as well as to protect users' personal information against illegal accesses by the programs with root privileges. The proposed scheme is an extension of RGBDroid, our previous work, which detects and responds to privilege escalation attacks. RGBDroid is effective for maintaining system integrity too. However, it cannot protect personal information of smartphone users against illegal accesses or unauthorized collection by malicious programs with root privileges while our scheme can do. The scheme has made use of two mechanisms: *pWhitelist* and *PDP* (Private Data Protection). *pWhitelist*, a list of trusted programs with root privileges, prevents attackers or malicious applications from being executed with root-level privileges on the Android platform. However, it can be bypassed by the attacks exploiting vulnerabilities of trusted programs with root privileges. To address the problem, *PDP* enforces the policy that privacy sensitive information can be accessed only by its owner program. This disallows even authorized programs with root privileges to access the personal data owned by applications. Therefore, PDP can protect user's personal data from the programs that have root privileges but are compromised by malware or attackers. Experimental results have shown that our scheme is effective to prevent personal information and causes little overhead.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0007372) and by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development

References

[1] Android. Android open source project, android security overview. `http://source.android.com/tech/security/index.html`.

[2] T. Bradley. *DroidDream becomes Android market nightmare*. March 2011.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report 04, Technische Universität Darmstadt, 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proc. of the 19th Annual Network & Distributed System Security Symposium (NDSS'12), San Diego, CA, USA*, February 2012.

[5] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. of the 20th USENIX Security Symposium (USENIX Security'11), San Francisco, California, USA*, pages 51–58, August 2011.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the 9th USENIX conference on Operating systems design and implementation (OSDI'10), Vancouver, British Columbia, Canada*, pages 1–6. USENIX Association, October 2010.

[7] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of the 16th ACM conference on Computer and communications security (CCS'09), Chicago, Illinois, USA*, pages 235–245. ACM, November 2009.

[8] B. Hatch. Linux Intrusion Detection System (LIDS). `http://www.lids.org`, November 2001.

[9] N. Husted, H. Saïdi, and A. Gehani. Smartphone security limitations: conflicting traditions. In *Proc. of the 2011 Workshop on Governance of Technology, Information, and Policies (GTIP'11), New York, USA*, pages 5–12. ACM, December 2011.

[10] X. Jiang. Gingermaster: First android malware utilizing a root exploit on android 2.3 (gingerbread). `http://www.cs.ncsu.edu/`, August 2011.

[11] X. Jiang. Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets. NC State University, June 2011. `http://www.csc.ncsu.edu/faculty/jjang/DroidKungFu.html`.

[12] L. Le and T. Nguyen. Payload already inside: Data re-use for ROP exploits. Technical report, Black Hat USA, 2010.

[13] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10), Beijing, China*, pages 328–332. ACM, April 2010.

[14] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proc. of the 2009 Annual Computer Security Applications Conference (ACSAC'09), Honolulu, Hawaii, USA*, pages 340–349. IEEE, December 2009.

[15] 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. In *Proc. of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats (LEET'12), Berkeley, California, USA*, 2012.

[16] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 33rd IEEE Symposium on the Security and Privacy (SP'12), San Francisco, California, USA*, pages 95–109. IEEE, May 2012.

Yeongung Park received the B.S. degree in Computer Science and the M.E. degree in Computer Engineering from Dankook University, Korea, in 2011 and 2012 respectively. During 2009̃2012, he was a member of Secure Software & System Lab. He is now a researcher of The attached institute of ETRI. His research interests include computer security, internet web security, smartphone security, and software security testing.

Chanhee Lee is a B.S student in Department of Computer Engineering at the Dankook University, Korea. His research interests include computer security, smartphone security, and software protection.

Jonghwa Kim received the B.S., the M.S. degree in computer science from the Dankook University, Korea. He is currently a Ph.D. student in the Department of Software Science at the Dankook University. His research interests include Linux kernel, SW architecture for Flash memory and SSD, and virtualizations.

Seong-je Cho received the B.E., the M.E. and the Ph.D. in Computer Engineering from Seoul National University, Korea, in 1989, 1991 and 1996 respectively. He was a visiting scholar at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. He is a Professor in Department of Software Science, Dankook University, Korea, from 1997. His current research interests include computer security, operating systems, software protection, real-time scheduling, and embedded software.

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993 and the MS and Ph.D. degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. Previously, he was a senior engineer at Ubiquix Company, Korea. He held a visiting faculty position at the Department of EECS, University of California, Santa Cruz from 2005 to 2006. He is now an associate professor in Department of Software Science, Dankook University, Korea. His research interests include Linux kernel optimization, SW architecture for Flash memory and SSD, OS for Manycore systems, I/O virtualizations, and big data processing.