Daemon-Guard: Towards Preventing Privilege Abuse Attacks in Android Native Daemons

Cong Zheng[†] Heqing Huang[‡]
[†] *Palo Alto Networks, CA, USA*
[‡] *IBM TJ Watson Research Center, NY, USA*
cozheng@paloaltonetworks.com, *hhuang@us.ibm.com*

ABSTRACT

With essential privileges, native daemons provide core system services for apps in the Android system. However, we find that exploiting Android native daemons can still lead to another security issue: the privilege abuse within the confined privilege. So, in this paper, we firstly demonstrate the privilege abuse problem in native daemons through two types of attacks: the data leakage attack and the Denial-of-Service (DoS) attack. To mitigate the privilege abuse issue, we then propose the Daemon-Guard framework, in which we build a dispatcher to fork a new daemon process for handling each service request from apps. The dispatcher can check the ownership of data and determine whether a data access operation is authorized, and check the speed of the service requests from an app by a reference monitor. To restrict a daemon process accessing data in the file system, we deploy Seccomp, a capability system supported by the Linux kernel. At last, we implement the Daemon-Guard framework on the keystore daemon through the static instrumentation. The evaluation of the keystore case shows that Daemon-Guard can successfully prevent these two privilege abuse attacks with an acceptable performance overhead.

1. INTRODUCTION

In recent years, lots of vulnerabilities have been discovered and exploited for privilege escalation in Android system (e.g., root exploits). Most vulnerabilities were discovered in Android native daemons. For example, "GingerBreak" [1] exploit sends a specific Netlink message the "*vold*" daemon to trigger the vulnerability and get the root privilege. Once malicious Android apps make use of root exploits to get the root privilege, they can launch serious attacks without the restriction of Android sandbox mechanism. Due to the serious fragmentation problem of Android ecosystem [12], many old devices are even so under threat of those public vulnerabilities, as well as unpublished or unknown vulnerabilities in customized daemons by OEM vendors.

From Android 4.3 version, SEAndroid was introduced to strengthen the security of Android system [2], especially for mitigating the rooting attack by exploiting native daemons. SEAndroid can enforce mandatory access control (MAC) over all processes. With correct SEAndroid polices, the exploit process is in a restricted domain and can only do limited operations. By nature, SEAndroid can prevent all rooting exploits, which leverage vulnerabilities in Android native daemons to perform privilege escalation. Even though the exploit process can create a *setuid-root* shell and execute it successfully to get the root UID, the assigned SEAndroid policy of the exploited process (or native daemon) still remains the same and thus no superuser capability is allowed in the exploit process [10]. To put it simple, SEAndroid uses the security context to confine the capability of different native daemons and application processes.

In this paper, we claim that, even under such a restrictive defense by SEAndroid, exploiting Android native daemons can still lead to another security issue: *privilege abuse*. As the service processes, native daemons have many sensitive privileges, but the Android system does not restrict the privileges for different context, so that the privileges can be abused by malicious apps. By abusing privileges of Android native daemons, attackers can launch two kinds of attack: the *data leakage attack* and the *denial-of-service (DoS) attack*. The *data leakage attack* is that by exploiting vulnerable native daemons, a malicious app can read or write critical data, which is stored by native daemons for other apps. Some native daemons, such as the *keystore daemon*, can provide a specific service that helps apps manage and store their sensitive data, like key pairs. Once malicious apps exploit daemons, other apps' sensitive data managed by native daemons, would be leaked out, even under the current Android protection mechanism. The *DoS attack* is that malicious apps can send a large number of service request to native daemons to suspend or slow down daemon processes, so that normal apps cannot use the service of native daemons. As native daemons can be used by apps with certain permissions, the service of native daemons can be easily abused without a context-aware reference monitor in the current Android system.

Even the new-added SEAndroid cannot mitigate the privilege abuse. In SEAndroid, each file and process is labeled, and the file operation is restricted by SEAndroid policies. Obviously, the daemon process is allowed to open, create, read and write all files belonging to each app. For example, according to the SEAndroid policy, the *keystore* daemon can operate all key pair files, which are created by the *keystore* daemon itself. If app A exploits the *keystore* daemon, app A definitely can indirectly read the key pair file (key_B) of app B by injecting malicious codes to the *keystore* daemon. At last, SEAndroid does not provide the reference monitor mechanism for preventing DoS attacks in nature.

To prevent the privilege abuse, we propose the **Daemon-Guard** framework for strengthening Android native daemons. Our Daemon-Guard framework is based on the capability system supported by Linux kernel, namely *Seccomp* [3], which could complement the existing MAC on Android. In the design, we build a *dispatcher*, which receives each service request to the native daemon and forks a new daemon process for handling the request. Before the execution of the new process, *Seccomp* is enabled to prohibit the "open" system call. So, the new *keystore* process can only read and write already-opened file descriptors, which are created by *dispatcher*. The *dispatcher* process can guarantee that an opened file always belongs to the requesting app by maintaining a mapping of UID to filename. In this way, a malicious app, which triggers vulnerabilities in native daemons, can only access files which are created by itself. Thus, the *data leakage attack* can be mitigated. On the other hand, the *dispatcher*, working as a reference monitor, can record the count of service request from each app and refuse to serve malicious apps, which send a lot of requests in a short period (e.g., 100 times per minute) for *DoS attack*.

We implement a prototype of the *Daemon-Guard* framework on the *keystore* daemon by instrumenting the Android system middleware (e.g., the Binder IPC mechanism) and enabling *Seccomp* in the Linux Kernel of Android. In the evaluation, the performance of instrumented the *keystore* daemon is acceptable with 463 microseconds overhead on average through the test of mini-benchmark.

To summarize, this paper makes the following contributions:

- We discover the privilege abuse problem of Android native daemons and demonstrate the problem with two types of attacks: the *data leakage attack* and the *DoS attack*. We also discuss the current Android protection mechanism including the SEAndroid mechanism cannot mitigate this security issue.
- To the best of our knowledge, it is the first time to provide a defense framework (i.e., the *Daemon-Guard*) to the privilege abusing problem by deploying the capability system *Seccomp* for Android.
- We also implemented the *Daemon-Guard* framework on the *keystore* daemon by instrumenting the daemon program.
- The evaluation result shows that our system can successfully prevent the proposed two attacks with an acceptable performance overhead.

2. BACKGROUND

2.1 SEAndroid

SEAndroid applies SELinux to Android system from Android 4.3. SEAndroid enforces mandatory access control (MAC) on all processes. The privileged processes are confined by the SEAndroid policy. The goal of deploying SEAndroid is confining privileged processes, such as native daemons, most of which are in the root user or system user group. The defense principle is that the exploit process is always in a unprivileged domain, in which a process cannot access any root privileged types, such as files under the "/system/bin". SEAndroid policy rules are compiled to a single kernel policy file as a part of Android ROM in ramdisk image. So, the kernel policy file can be loaded by *init* at a very early stage of booting, even before the mounting of system partition. Therefore, the SEAndroid policy cannot be dynamically adjusted in a specific security context.

2.2 Android Native Daemon Security

In Android, all native daemons are implemented in C/C++ code as background services in system-level. In the booting phase, native daemons are booted after the Zygote process. Each daemon process will be created and assigned to a specific SEAndroid subject domain(s). The subject domain will have a set of capabilities associated with a predefined object domains (e.g., file, socket and etc.). For instance, the keystore daemon process will be added into the *keystore* subject domain based on the *init_daemon_domain(keystore)* policy. Then, the capability of the *keystore* subject domain is predefined in the SEAndroid policy type files (here the keystore.te). To our best knowledge, currently SEAndroid is the main security enhancement that Google has deployed for the Android system.

2.3 Seccomp

Secure computing (*Seccomp*) is a capability system enforced by Linux kernel since Linux 2.6.12. After *Seccomp* is enabled for a specific process, the process enters the *Seccomp* mode, where the process is only allowed to execute "exit()", "sigreturn()", "read()" and "write()" four system calls. If other system calls are executed, a "SIGKILL" signal is throwed. *Seccomp* is fit for the scenario, where an application needs to execute untrusted codes, which are received from others or injected by exploits.

Seccomp with Berkeley Packet Filter (*Seccomp-BPF*) is an extension of *Seccomp* since Linux kernel 3.10 [3]. The filters of *Seccomp-BPF* can allow or deny any system calls, and can even be more fine-grained by filtering system call arguments, but numeric values only.

To apply *Seccomp-BPF* to applications, it must instrument the original program by adding some codes to define filters, install filters and enable the *Seccomp* mode. For security reasons, one should always enable the *Seccomp* before executing untrusted codes. In our design, we use the *Seccomp-BPF* as the base protection mechanism to fulfill our fine-grained security requirements (i.e., *Daemon-Guard* framework) of Android.

3. PRIVILEGE ABUSE ATTACK IN NATIVE DAEMONS

In this section, we will explain how the Android native daemon could be attacked (i.e., the privilege abuse attack) even under the protection of SEAndroid.

Some Android native daemons, as service processes, serve Android apps by exposing the socket interface or the API interface. Most socket interfaces of daemons are protected well by Linux-based privileges currently, so that normal apps cannot easily access these sockets. However, normal apps can invoke APIs to indirectly communicate with daemon processes by declaring corresponding Android permissions, or even without permissions for some specific daemons. So, normal apps can somehow use the services of native daemons. This gives a chance for apps to abuse the privilege of daemons, since the current Android system does not restrict apps using the privilege of native daemons. Once apps can use the privilege of native daemons, apps also use the full privilege set without any limits.

3.1 Data Leakage Attack

To launch the data leakage attack, there are three conditions for Android native daemons:

- *Condition 1*: The vulnerability in native daemons can be exploited for the arbitrary code execution.
- *Condition 2*: Apps can directly or indirectly send the attack payload to native daemons for triggering vulnerabilities.
- *Condition 3*: The vulnerable native daemons can store sensitive data in the file system for apps.

In Android system, the *keystore* daemon allows an app to generate and save key pairs by APIs (*Condition 2*), and key pairs can only be accessed by the app itself. The service provided by *keystore* includes the key pair generation, signature generation and validation, etc. The secret key pairs are stored by *keystore* daemon in a key directory (*Condition 3*). A stack buffer overflow vulnerability (CVE-2014-3100) was found in *keystore* daemon before Android 4.4 version [6]. This vulnerability could be exploited for the arbitrary code execution after bypassing DEP [5], ASLR [8] and stack cookie [9] (*Condition 1*).

The data leakage attack scenario is displayed by Figure 1. App 2 (uid:10054) is benign, and it sends a request to keystore daemon for generating a key pair (alias: aa). The key pair is stored in the path "/data/misc/keystore/user_0/10054_USRCERT_aa" and "/data/misc/keystore/user_0/10054_USRPKEY_aa". The permission of these two key files is *"-rw——-*

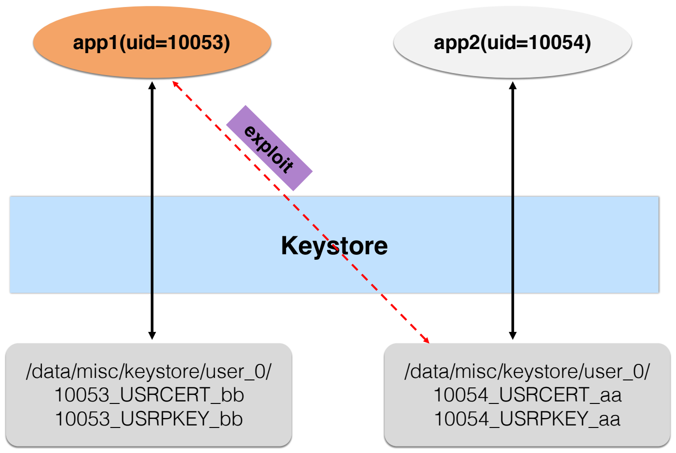

Figure 1: The Attack on the *keystore daemon*

keystore keystore". Then, an attacker can develop a malicious app (app 1, uid:10053), which sends a malformed request message to *keystore* to trigger that vulnerability and execute malicious payloads. Then, the malicious app can read the "10054_USRCERT_aa" key file. In fact, app 1 is not allowed to read "10054_USRCERT_aa" key file, which is enforced by the default data isolation mechanism in *keystore* daemon. But, this data isolation will be easily bypassed once attackers exploit the vulnerability in *keystore* daemon and execute arbitrary codes.

3.2 DoS Attack

Launching a DoS attack on native daemons only requires *Condition 2* in §3.1. For *keystore* daemon, apps can communicate with it by APIs. For example, apps invoke "KeyPairGenerator().genKeyPair()" API to generate a key pair. However, as Android system does not restrict service requests for *keystore* daemon, the DoS attack is feasible. Attackers can launch the DoS attack to slow down performance and consume resources. A malicious app can frequently invoke some time-costly APIs. For instance, when the "verify()" API that checks whether a given signature can be verified by the public key or a certificate of the signer is abused to be called frequently, the requests from benign apps will delayed for a long time. Meanwhile, lots of CPU resources are occupied by the malicious app. The DoS attack would also consume a large number of file system resources. If the "genKeyPair()" function is invoked by malicious apps very frequently, the used size of the file system will increase very quickly.

3.3 SEAndroid Bypass

SEAndroid cannot defend these two attacks by nature. The fundamental limitations of the SEAndroid mechanism is that: 1) the security context is predefined statically for subject domains (e.g., application and native daemons are assigned to the subject domain) and object domains (e.g., files or folders). 2) What's worse, there is no way further restriction on the assigned capability to the subject domains, which makes it a very coarse-grained granularity confinement. Due to these two limitations of SEAndroid, it cannot dynamically adjust policies to restrict file operations on different files and restrict the speed of service request according to different requesting apps. Hence, once the native daemon is exploited, all the granted capability [1] to the daemon could be leveraged directly, which can lead to privilege abuse attacks, including the data leakage attack, the DoS attack, etc.

4. DAEMON-GUARD DESIGN

To prevent the privilege abuse attack on Android native daemons, we propose the *Daemon-Guard* framework. Generally, the framework includes two main components: *dispatcher* module and *Seccomp* enforcing module. The *dispatcher* receives and dispatches each service request and forks a child daemon process for handling the request. *Seccomp* enforcing module is designed to prohibit specific system calls based on the security context. The purpose of forking several isolated processes for one native daemon, and restricting the capabilities of each forked process is to further confine the process with a more restricted privilege. Hence, the data leakage attack can be prevented. Also, a reference monitor in the *dispatcher* module records the speed of service request (e.g., API calls), so that the DoS attack can be directly detected and prevented.

In the following, we will firstly introduce the *dispatcher* and then describe how to apply *Seccomp* to native daemons. We use the *keystore daemon* as an example, and the *Daemon-Guard* framework can be applied to other *daemons* similarly.

4.1 Dispatcher

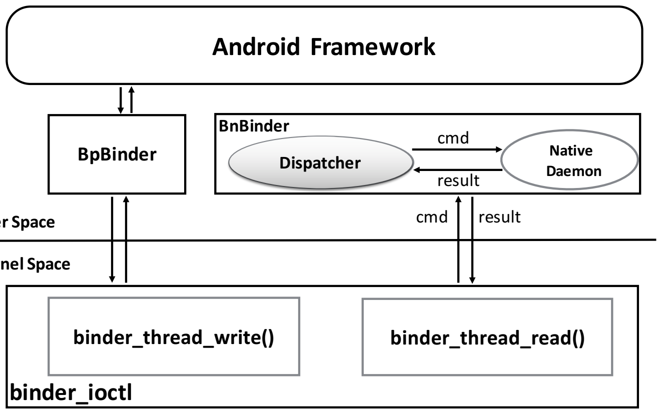

Figure 2: Android Binder of native daemons

Native daemon uses Android Binder as its IPC to communicate with Android framework, showed in Figure 2. The Binder includes a service BnBinder and a client BpBinder. BnBinder communicates with BpBinder through a *binder_-ioctl* module in the system kernel. As BnBinder is a service process for handling each command request, we can split BnBinder process to a *dispatcher* process and a *daemon business* process. For the IPC, the *dispatcher* can use a socket to communicate with the new daemon process.

The *dispatcher* is responsible for the following tasks. 1) It handles the service requests and does the ownership check. It maintains a sensitive data ownership mapping for each request UID. The UID of request app can be obtained by *"IPCThreadState::self()->getCallingUid()"* API. For instance, if the request command is *"create"*, it inserts an item into the mapping, and for the *"read"*, *"write"* and *"delete"* command, it always checks the mapping to determine the ownership. 2) It helps forking daemon processes and enabling the *Seccomp* mode. The *dispatcher* forks a new daemon process for each request. Before the new process launches, it enables the *Seccomp* mode in the new process. 3) It helps the native daemon process open a file and send the file descriptor back after passing the ownership check for the file. 4) It receives results from the forked new process. After getting the results, the *dispatcher* can send them back to BpBinder, which is transparent for BpBinder. 5) It keeps a reference monitor for each request UID. When the speed of request from a same UID is greater than a predefined threshold, the *dispatcher* will drop following requests. So, the *DoS attack* can be defended by the reference monitor.

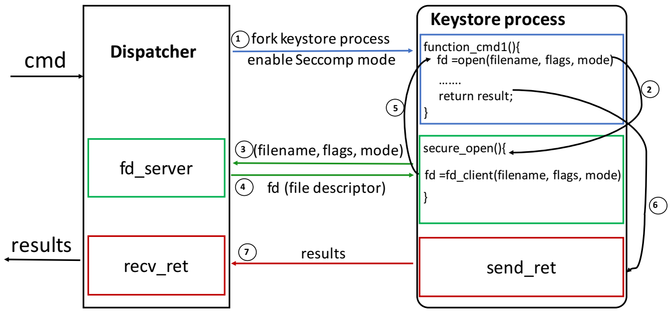

Figure 3: The workflow of *Dispatcher*

By giving an implementation example of *keystore* daemon, we describe the workflow of *dispatcher* in Figure 3.

Step 1: When a new command request arrives at the *dispatcher*, it will fork a new keystore process, which will go to the original workflow of the keystore daemon to parse and handle this command.

Step 2: "open" system call in the keystore process is hooked. Once an "open" system call is invoked, it will be redirected to a "secure_open" function, which is a customized secure open system function in our hooking library.

Step 3: The "secure_open" function will send the parameters of "open" system call, including filename, flags and mode, to the **fd_server** in the *dispatcher*. **fd_server** is a socket server for waiting the requests from the new keystore process to open a file. Before opening a file, the **fd_server** checks the ownership of the file.

Step 4: After passing the ownership check for the requested open file, it can finish "open" and send the file descriptor back

[1]The relevant SEAndroid policy in the "keystore.te" for the keystore native daemon: "allow keystore keystore_data_file:dir create_-dir_perms", "allow keystore keystore_data_file:notdevfile_class_set create_file_perms".

to the child process via the socket.

Step 5: The "secure_open" function returns the file descriptor to the "open" system call.

Step 6: The return results are hijacked by our static instrumentation to send to the **send_ret** socket client.

Step 7: The **send_ret** socket client then sends the return results to the **recv_ret** socket server, which eventually returns the results back to BpBinder.

4.2 Seccomp

Seccomp is a default capability system provided by the Linux kernel. Once *Seccomp* is enabled, all system calls in specified process will be restricted. To use *Seccomp* in the forked native process, we statically modify the source codes of native daemons. At the very beginning of the native daemon process, we insert some codes to enable *Seccomp* mode by the static instrumentation.

When using the *Seccomp-BPF* filter, we adopt a white list strategy that only system calls in the white list are allowed to be executed. Definitely, in addition to put all necessary system calls in the white list, we will not put extra system calls, such as "open" system call, into the white list. Once the *Seccomp* mode is enabled in the new daemon process, the white list *Seccomp-BPF* filter is loaded into the kernel and it can never be modified later in runtime.

In the *keystore* daemon, by enabling *Seccomp* with a white list filter, the forked new *keystore* process cannot call any *libc* "open" functions or "open" system call. Instead, the "open" system call can only be called in *dispatcher*. By that, any injected attack payloads in the *keystore* process cannot open any files by itself, even if the *keystore* daemon has corresponding privileges and does not violate the SEAndroid policy. Therefore, the privilege abuse attack can be prevented even when the *keystore* daemon is compromised.

5. IMPLEMENTATION

In the static instrumentation design, we use a light weight instrumentation, so that this *Daemon-Guard* design can be easily extended to other daemons which are also vulnerable under the privilege abuse attack. We choose to instrument the *keystore* Binder so that each daemon service request can be captured by the dispatcher. The dispatcher will fork a new *keystore* process and enable the *Seccomp-BPF* mode within it. To hook the "open" system call in the new *keystore* process, we use the *LD_PRELOAD* to check all "open" functions in *glibc*.

We choose the Linux 3.18.0 kernel for using *Seccomp-BPF*. In the *Seccomp-BPF*, the white list is applied the filter. For building the white list, we use *objdump* tool to dump all system calls in the "/system/bin/keystore" file.

6. EVALUATION

In this section, we present the evaluation of *Daemon-Guard*. The experiment is set on Android x86 emulator with Android 4.4 and Linux kernel 3.18.0 on the machine with 3.20 GHz and 64 GB RAM.

To test the performance overhead, we built a mini-benchmark, which tests the performance overhead of each request command. The mini-benchmark includes an evaluation app and a log analyzer. The evaluation app invokes the *keystore* APIs to send different commands to the *keystore* daemon. The log analyzer will analyze the timing log information output both before the command service and after finishing it. After comparing the performance of each command between the original *keystore* and the *Daemon-Guard* instrumented version, the average overhead for all commands, in total 23, is about 463 microseconds. Some common commands' evaluation results are listed in Table 1.

From the evaluation results, even though the performance for each command is about 2 to 4 times of its original performance, but it is still acceptable since users could not feel any operation delay. The reason is that the request of each command for the *keystore* daemon is not very frequent. By testing top 500 apps in Google Play, which use the service of the keystore, the total number of used commands is always below 50. Therefore, the total overhead by the average overhead for an app is about 23 milliseconds. In the worst case that *GENERATE* command is invoked for 50 times, the total overhead is about 270 milliseconds, which is close to the user fast response time 200 milliseconds.

To verify whether *Daemon-Guard* can prevent both the data leakage attack and the DoS attack, we perform a security analysis in the following.

Analysis of the Data Leakage Attack: According to the attack scenario in §3.1, we developd a malicious app (app 1) and a benign app (app 2) respectively. There are two attack payloads: 1) One attack payload modifies the file name from "10053_USRPKEY_bb" to "10054_USRPKEY_aa" in memory. 2) The other payload directly invokes the "open" system call to open "10054_USRPKEY_aa" file. Obviously, both attack payloads can read the "10054_USRPKEY_aa" key file in the original *keystore* daemon. However, when we replay this data leakage attack on the instrumented *keystore* daemon with *Daemon-Guard* framework, both attack payloads cannot succeed. The first attack payload fails as the dispatcher does not open the "10054_USRPKEY_aa" file after the ownership checking. The second payload also does not work since the injected direct "open" system calls in the payload is intercepted by *Seccomp* filter and then the new *keystore* process is terminated. The *Seccomp* sends a SIGSYS signal to a crash reporter, which logs information about the syscall and finally terminate the process.

Analysis of the DoS Attack: to verify whether *Daemon-Guard* can prevent the DoS attack, we developd a malicious app, which invokes "verify()" and "genKeyPair()" functions at 10,000 times per minute speed. With this, we can monitor that the usage of CPU and file system resources increases very quickly. However, after testing this malicious app in the *Daemon-Guard* enforced Android system by setting a threshold 100 times per minute, we found that the *Daemon-Guard* can effectively mitigate the DoS attack.

Table 1: Performances

Command	Time(Keystore)	Time(Daemon-Guard)	Overhead
DEL_KEY	4239	4636	397
DEL	213	425	212
GENERATE	8340	13734	5394
GET_PUBKEY	145	579	434
GET	252	671	419
SIGN	4390	8988	4598
INSERT	112	435	323
EXIST	98	280	182
VERIFY	7513	8428	915

* The unit of performance time is microsecond.

7. DISCUSSIONS

In *Daemon-Guard* framework, the identification relies on the security of Android Binder and our built *dispatcher*. One assumption of this paper is that Binder cannot be compromised by attackers, as attackers can launch much more harmful attacks after exploiting Binder. If Binder is compromised, the attacker can send a fake UID to *dispatcher*, and the ownership checking would be bypassed. In short, the identification of the design is guaranteed by the secure *dispatcher*.

In the current design of our *Daemon-Guard*, one needs to manually instrument code and integrate *Seccomp* into daemons, so that the security-enhanced daemons have different security properties to prevent different attacks. Since the current design of each daemon could be different, defining a security context of each daemon is case by case. Hence, this instrumentation part is not fully automated yet, and we leave the automated instrumentation as our future work.

8. RELATED WORK

Android attacks: Attack targets of Android system mainly include: *Linux kernel* and *native daemons*. For attacking Linux kernel, TowerRoot abuses an vulnerability in the futex system call to root devices. For native daemons, previous public attacks are only used for the rooting attack, such as "RAtC" [11] getting the root privilege by forking bombs to prevent *adbd* from dropping its root privilege. But, this paper introduces the privilege abuse problem in Android native daemons, including the data leakage attack and the DoS attack, which have not been revealed before. *Privilege Abuse* attack [4] abuses the Unix socket with incorrect configured permission in the Zygote process to fork tons of Zygote processes to launch a DoS attack on Android. However, we are focusing on the unexplored problem, namely the privilege abuse attack on Android native daemons. Not only have we provided detailed analysis on the consequences, but also we propose and implemented the defense solutions as well.

Capability system: Mbox [7] is a lightweight sandbox for preventing applications from modifying the host file system by adding a transparent layer of the file system. For interposing on system calls, Mbox also uses *Seccomp-BPF*, with its another option that the filter returns a ptrace event for further proceeding. Different with Mbox to interpose system calls, *Daemon-Guard* uses *Seccomp-BPF* to prohibit the "open" system call.

9. CONCLUSION

In this paper, we first explore the privilege abuse problem in Android native daemons and propose two proof-of-concept attacks of this kind. We show that the privilege abuse attack can evade all the existing preventions deployed by Android system, including the newly added SEAndroid mechanism. We then propose a defense framework, namely the *Daemon-Guard* framework, which leverages the static instrumentation and dynamic hooking based on Seccomp. Furthermore, we implemented and instantiated our framework in one of the critical native daemons (i.e., keystore daemon). The evaluation result shows our framework can successfully prevent the proposed two privilege abuse attacks with an acceptable performance overhead. The result demonstrates that our *Daemon-Guard* framework can be easily extended to other native daemons to prevent against privilege abuse attacks.

References

[1] Gingerbreak. URL http://www.cvedetails.com/cve/CVE-2011-1823.

[2] Seandroid. URL https://source.android.com/security/selinux/.

[3] Seccomp-bpf. URL http://lwn.net/Articles/475043/.

[4] A. Alessandro, M. Alessio, M. Mauro, and V. Luca. Would you mind forking this process? a denial of service attack on android. In *27th International Information Security and Privacy Conference*, 2012.

[5] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. In *Privilege Escalation Attacks on Android*, pages 346–360, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] R. Hay and A. Dayan. Android keystore stack buffer overflow, 2014.

[7] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 139–144, Berkeley, CA, USA, 2013.

[8] K. Lu, S. Nurnberger, M. Backes, and W. Lee. How to make aslr win the clone wars: Runtime re-randomization. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[9] H. Marco-Gisbert. Preventing brute force attacks against stack canary protection on networking servers. In *Proceedings of 2013 12th IEEE International Symposium on Network Computing and Applications (NCA)*, 2013.

[10] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Proceedings of the 2013 Network and Distributed System Security Symposium*, 2013.

[11] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[12] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.