An Empirical Evaluation of the Android Security Framework*

Alessandro Armando[1,2], Alessio Merlo[1,3,**], and Luca Verderame[1]

[1] DIBRIS, Università degli Studi di Genova, Italy
name.surname@unige.it
[2] Security & Trust Unit, FBK-irst, Trento, Italy
armando@fbk.eu
[3] Università e-Campus, Italy
alessio.merlo@uniecampus.it

Abstract. The Android OS consists of a Java stack built on top of a native Linux kernel. A number of recently discovered vulnerabilities suggests that some security issues may be hidden in the interplay between the Java stack and the Linux kernel. We have conducted an empirical security evaluation of the interaction among layers. Our experiments indicate that the Android Security Framework (ASF) does not discriminate the caller of invocations targeted to the Linux kernel, thereby allowing Android applications to directly interact with the Linux kernel. We also show that this trait lets malicious applications adversely affect the user's privacy as well as the usability of the device. Finally, we propose an enhancement in the ASF that allows for the detection and prevention of direct kernel invocations from applications.

1 Introduction

Android is the most widely deployed operating system for smartphones and recent estimates [11] indicate that it will continue to remain so in next years. Roughly speaking it consists of a Java stack built on top of a native Linux kernel. Services and functionalities are achieved through the interplay of components living at different layers of the operating system by means of suitable calls.

Security in Android is granted by a set of cross-layers security solutions combining basic Linux security mechanisms (e.g. Discretionary Access Control) with Java native (JVM isolation) and Android-specific (e.g. the Android permission system) mechanisms. These solutions collectively constitute the Android Security Framework (ASF). The ASF supervises the cross-layer interplay among components in order to detect malicious or unwanted interactions and intervene if necessary.

Recently, the security offered by the ASF has been challenged by the discovery of a number of vulnerabilities involving different layers of the Android stack and the corresponding interplay (see, e.g., [1,8,7]). By analyzing interplay-related vulnerabilities, two peculiarities arise:

* This work has been partially founded by EU project FP7-257876 SPaCIoS.
** Corresponding author.

L.J. Janczewski, H.B. Wolfe, and S. Shenoi (Eds.): SEC 2013, IFIP AICT 405, pp. 176–189, 2013.

- the security mechanisms of the Android stack (both Java native and Android-specific) are not completely integrated with those in the Linux kernel, thus potentially allowing for insecure interplay;
- malicious and unprivileged Android applications can force the execution of insecure interplay, thereby by-passing the controls performed by the ASF.

For instance, the Zygote vulnerability reported in [1] allows a malicious application to force the Linux kernel to fork an unbounded number of processes thereby making the device totally unresponsive. In this case, the problem is due to the fact that the ASF is not able to discriminate between a legal interplay (performed by trusted Android services) and an insecure one (executed by applications), thereby permitting the direct invocation of a critical kernel functionality (i.e. the fork operation) by any application. This is basically due to a lack of control on Linux system calls involved in the launch of the new application.

An interesting question is whether such lack of control between the Android stack and the Linux kernel is limited to some type of calls only or else it is a more general issue in the ASF. To ascertain this, we have defined and carried out an empirical assessment on the interplay between the Android stack and the Linux kernel. To this end we have implemented a new kernel module, called *Monitoring Kernel Module*. The Monitoring Kernel Module once installed captures all the invocations targeted to the Linux kernel. We then implemented an Android application (i.e. *KernelCallTester*) that systematically tries to replicate all the calls captured by the Monitoring Kernel Module. This has allowed us to assess to which extent the ASF is able to discriminate between trusted and untrusted invocations of core system functionalities. Our tests—executed involving a number of actual smartphone users—show that very little control is exercised by the ASF and that malicious applications may force and exploit insecure interplays. To show this, we have semi-automatically analyzed all logs produced by the Monitoring Kernel Module. This has led to the discovery of two interplays that adversely affect the user's privacy as well as the usability of the device. We have then implemented and tested two malicious applications (i.e. *WriteTest* and *CacheHooker*)) that execute the malicious interplays. Our experiments on our testbeds showed also in this case that the ASF does not prevent the leakage of private information nor the unavailability of the device. Finally, we designed and implemented an improvement of the ASF (i.e. the *Kernel Call Controller* module) that recognizes and rules the insecure interplays between the Android stack and the Linux kernel we have identified. Again, we tested the effectiveness of the proposed improvement by using our experimental setup (that involves a number of actual users and devices).

Structure of the Paper. In Sect. 2 we briefly introduce the architecture and the interplay of Android, while in Sect. 3 we discuss peculiarities and limitations of the ASF. In Sect. 4 we describe the setup and the implementation of the *Monitoring Kernel Module* and the *KernelCallTester* application. In Sect. 5 we analyze the testing phase and the experimental results. In Sect. 6 we describe the development and the testing of two malicious applications (i.e. *WriteTest* and *CacheHooker*) able to exploit the lack of control in the ASF. Then, in Sect. 7

we propose an improvement in the ASF able to solve the problem. In Sect. 8 we discuss the related work and we conclude in Sect. 9 with some final remarks and future directions.

2 Android in a Nutshell

The Android Architecture consists of 5 layers. The Linux kernel lives in the bottom layer (henceforth the *Linux kernel*). The remaining four layers are Android-specific and we therefore collectively call them *the Android stack*:

Application Layer (A***).*** Applications are at the top of the stack and comprise both user and system applications that have been installed and execute on the device. Each application is made of a set of components each performing a different role in the logic of the application (see [2] for further details).

Application Framework Layer (AF***).*** The Application Framework provides the main services of the platform that are exposed to applications as a set of APIs. This layer provides the System Server, that is a process containing Android core components [1].

Android Runtime Layer (AR***).*** This layer consists of the Dalvik Virtual Machine (Dalvik VM, for short), i.e. the Android runtime core component that executes application files built in the Dalvik Executable format (`.dex`).

Libraries Layer (L***).*** The Libraries layer contains a set of C/C++ libraries that support the direct invocation of basic kernel functionalities. They are widely used by Application Framework services to interact with the Linux kernel and to access data stored on the device. Examples of libraries are the *Bionic libc*, a custom implementation of `libc` for Android, and *SQLite*, a self-contained and transactional database engine.

Kernel Layer (K***).*** Android relies on the Linux kernel version 2.6 for core system functionalities. These functionalities include *i)* the access to physical resource (i.e. device peripherals, memory, file system) and *ii)* the Inter-Process Communication (IPC). Device peripherals (e.g. GPS antenna, Bluetooth/Wireless/3G modules, camera, accelerometer) are accessed through Linux drivers installed as kernel modules. Triggering peripheral drivers, as well as accessing file system and memory are achieved by means of *system calls* (e.g. `open`, `read` and `write` for files management). IPC may be carried out through the use of the Binder driver or by reading from/writing on native Unix Domain Sockets. Binder driver is activated through *binder* calls (i.e. `ioctl`) , while sockets are accessed through *socket calls* (e.g. `connect`, `bind`, `sendmsg`).

2.1 Notes on the Interplay in Android

Operations in Android are carried out through interactions among layers. Such interactions constitute the interplay of Android and are implemented through

[1] `http://events.linuxfoundation.org/slides/2011/abs/abs2011_yaghmour_internals.pdf` for a comprehensive list of such service components.

6 kinds of calls (namely, *function*, *dynamic load*, *jni*, *system*, *binder*, and *socket* calls) involving distinct subsets of layers and libraries (see [2] for details on calls). As previously discussed, *system*, *binder*, and *socket* calls allow to trigger directly the Linux kernel functionalities. Hereafter, we refer to these kinds of calls as *kernel calls*. Android provides OS functionalities to applications by means of combinations of calls. For instance, the launch of a new application in Android is normally provided by the following interplay:

1. a requesting application (i.e. the home screen of the device) executes a *binder call* to the Activity Manager Service (AMS) at the Application Framework layer to start the launching process.
2. The AMS checks the permissions of the requesting application and, in case they are sufficient, executes a set of *socket calls* (i.e. `connect`, `sendmsg`, `listen`) to the Zygote socket at K layer for writing down a command aimed at requesting the launch of a new application. The command contains information related to the application to launch.
3. The controlling process of the Zygote socket (i.e. the Zygote process) parses the command and invokes a *JNI call* to load a proper library function at L layer for accessing kernel functionalities.
4. The invoked function directly executes a `fork` *system call* at K layer, building a new Linux process that will host the launching application. If something goes wrong, the command provided by the AMS to the Zygote socket forces the kernel to destroy the created process, otherwise a new Dalvik VM with the code of the launching application is bound to the process and the execution starts.

Interplay in Android is poorly documented and not standardized. As a matter of fact, the interplay of only a few operations is discussed in the official literature ([4]) and the interplay related to the launch of a new application is not documented. The above description (borrowed from [1]) has been inferred by systematically analyzing Android source code. Moreover, the lack of documentation and standardization implies that the same functionalities could be potentially carried out through different interplay, some of which may lead to security flaws. To prevent this, the Android Security Framework discriminates whether an interplay is secure or not, according to the permissions of applications and the basic Android security policy. We introduce in the following the basis of the ASF and then we reason about its limitations related to the analysis of the interplay.

3 The Android Security Framework

The Android Security Framework (ASF) provides a cross-layer security solution (i.e. sandboxing) built by combining native per-layer security mechanisms. Each layer in the Android stack (except the Libraries layer) comes with its own security mechanisms:

- **Application layer (Android Permissions)**. Each application comes with a file named `AndroidManifest.xml` that contains the permissions that the application may require during execution. During installation the user is asked to grant all the permissions specified in the manifest.
- **Application Framework (Permission Enforcement)**. Services at this layer enforce the permissions specified in the manifest and granted by the user during installation.
- **Runtime (VM Isolation)**. Each application is executed in a separate Dalvik VM machine. This ensures isolation among applications.
- **Linux (Access Control)**. As in any Linux kernel, resources are mapped into files (e.g. sockets, drivers). The Linux Discretionary Access Control (DAC) model associates each file with an owner and a group. Then, DAC model allows the owner to assign an access control list (i.e. read, write, and/or execute) on each file to the owner itself (UID), the owner's group (GID) and other users.

Sandboxing is a cross-layer solution adopted in Android to provide strong isolation among applications. In detail, Android achieves sandboxing of the applications by binding each Android application to a separate user at K layer, thereby combining the native separation due to the execution of applications on different Dalvik VMs with the isolation provided by native Linux access control.

Once an application is installed on the device (i.e. the user accepts all required permissions in the `AndroidManifest.xml` file) a new user at the Linux layer is created and the corresponding user id (UID) is bound to the installed application. As stated in the previous section, once the application is launched, a new process, with such UID, and a novel Dalvik VM are created in order to execute the application.

This solution forces any non privileged UID to have at most one process running (i.e. the one containing the running application). Rarely, more than one active process for the same UID can be allowed if explicitly requested in the `AndroidManifest.xml`. However, the maximum number of active processes is upper-bounded by the number of components composing the application.

At runtime, sandboxing and other per-layer security mechanisms are expected to avoid illegal interplay. For instance, if an application tries to invoke a `kill` system call on the process hosting another application, the sandboxing is violated (i.e. a Linux user tries to kill a process belonging to another user) and the system call is blocked.

3.1 Security Considerations on the Android Security Framework

Some recent vulnerabilities indicate that the integration of the Android-specific security mechanisms with those provided by Linux may suffer from unknown security flaws.

For instance, as shown in [1] in the launching flow presented in Sect. 2.1 an application can invoke *socket calls* directly and send an ad-hoc command through the Zygote socket, thereby by-passing the Activity Manager Service. The ASF

identifies such interplay as legal, since it is not able to notice that an application is invoking calls targeted to the Linux kernel instead of a trusted service in the Application Framework layer. More in detail, each running application, as well as any trusted service, is hosted in an unprivileged process at Linux layer due to *sandboxing*; this makes hard for the Linux kernel to discriminate whether a socket or system call comes from an application rather than a trusted server; this impersonation should be noticed by security mechanisms in the Android stack. However, in the specific case no mechanism recognizes the unexpected caller, thus allowing the malicious interplay. Furthermore, such an interplay can be repeated an arbitrary number of times by the malicious application until the device becomes totally unresponsive [1].

Other vulnerabilities suggest that security breaches can be hidden in the communications among applications. For instance, single messages exchanged among applications (by means of *binder calls*) can be individually compliant with the applications permissions and hence permitted by the ASF; however, some maliciously crafted interplay can lead to undetected privilege escalation [8] or attacks on legal applications [7].

We argue that at the above problems are due to a lack of control on:

- the identification of the caller for direct kernel invocations (i.e. *binder*, *socket* and *system calls*) that may allow malicious applications to operate undetected on the Kernel instead of legal Android services;
- the monitoring of cross-layer interplay, that may allow malicious interactions if single calls do no violate any system policy or the sandboxing;
- the identification of repeated interplay, that can make the OS weak against Denial-of-Service attacks.

In the following, we substantiate our claims by means of an empirical evaluation carried out on the Android platform.

4 Assessing the ASF on Kernel Calls

As discussed, the Zygote vulnerability is basically due to a lack of control on the identity of the components invoking a *socket call* targeted to the Kernel layer that is normally expected to be executed by trusted services in the Application Framework layer. However, the same problem may affect other calls normally invoked by the same trusted services.

Unfortunately, due to a very limited documentation on this topic, it is impossible to rely exclusively on current technical and research literature to retrieve reliable information on the set of kernel calls invoked by trusted services. To this aim, static analysis techniques could help (they are widely adopted to retrieve models of Android applications from the Dalvik code). However, due to the complexity and the size of the Android source code, static analysis can be complex and cumbersome. Hence, we opted for an empirical approach with the two-fold aim of i) relating trusted services with the kernel calls they invoke, and ii) verifying whether the ASF is able to recognize that an execution of a kernel call, as well as it is invoked by a legal service, is invoked by a malicious application instead.

To this aim, we set up the experiment into two steps: we implemented 1) a *Monitoring Kernel Module* (MKM) able to intercept kernel calls invoked by the whole *Android stack* and 2) a tester application (*KernelCallTester*) that is able to replicate the calls intercepted by the MKM. Then, we set up the MKM to intercept kernel calls executed by trusted services in the AF layer and the *KernelCallTester* application to reproduce each kernel call as soon as it has been intercepted (see Fig.1), in order to assess whether the ASF recognizes such attempt as malicious.

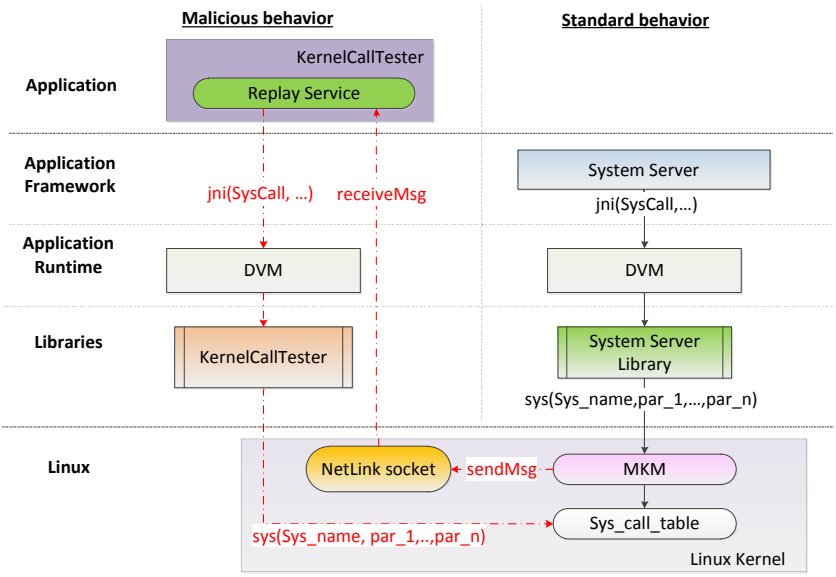

Fig. 1. Interaction between the MKM and the `KernelCallTester`

The MKM is a kernel module which customizes the way in which kernel calls are invoked. Upon installation, the MKM retrieves the kernel call prototypes from `systemcalls.h` and the kernel calls numbers from `unistd.h`. Then, it modifies each entry in the `sys_call_table` structure, which contains the kernel calls routines; in particular, the MKM substitutes each routine in the table with a customized one. Each customized routine gets the calling thread name and process pid (using the Linux macro `current`) as well as the optional parameters passed to the call and, then, it executes the normal routine. At runtime, the MKM creates a *netlink socket* in order to store the intercepted kernel calls and the corresponding parameters. Each time the MKM intercepts a kernel call, the custom routine writes a message on the netlink socket.

KernelCallTester has been designed to replicate kernel calls invocations intercepted by the MKM and stored on the netlink socket. For each kernel call invocation, *KernelCallTester* tries to replicate it a random number of times. *KernelCallTester* is composed by a a broadcast receiver, an Android service (called `Replay Service`), running on the device in background, and a C++

pre-compiled library. The broadcast receiver is in charge to launch the `Replay Service` once the device completes the boot. The `Replay Service` connects to the netlink socket created by the MKM and parses data contained on it. For each invocation stored on the socket, the `Replay Service`, by means of a *jni* call to the *KernelCallTester* library, starts replicating the corresponding kernel call with the same parameters as the original one. Depending on the kind of kernel calls, the `Replay Service` may also execute other ad-hoc calls. For instance, if a `read` call is invoked on a file, the `Replay Service` tries to execute an `open` on the same file before invoking the `read`. After replicating the call, `Replay Service` removes the entry from the socket. The `Replay Service` keeps track in proper log files of the success/failure of each replication attempt. Besides, error messages for failed invocations are stored.

4.1 Deploying and Configuring MKM and *KernelCallTester*.

We deployed the MKM and the *KernelCallTester* into two Android builds, namely v. 2.3.3 (API 10) and v. 4.0.3 (API 15), as the most representative distributions currently available on commercial devices[2], respectively for entry-level and top-notch smartphones.

Since the ability to load modules is natively disabled in the Linux kernel deployed in Android, we enabled such feature by recompiling the kernel for a generic ARM architecture. Such modification does not alter any kernel functionality, thus the behavior of the recompiled kernel is equivalent to the original one. Moreover, we have developed a custom `rc.module` script, executed as a service in the `init.rc`, which installs the MKM automatically at startup.

We configured the MKM to intercept kernel calls executed by trusted services and to keep track of a subset of the most representative kernel calls[3], including core system calls (e.g. for I/O and process management: `open`, `close`, `read`, `write`, `lseek`, `mkdir`, `rmdir`, `exit_group`, `exit`, `getpid`, `gettid`, `kill`, `lstat64`, `prctl`, `setuid`, `setgid`, `waitid`, `shutdown`, `gettuid`, `geteuid`, `getgid`, `mount`, `umount`), socket calls (`bind`, `connect`, `sendmsg`, `sendto`, `socket`, `recvfrom`, `recvmsg`, `listen`) and binder calls (which rely on the `ioctl` system call).

This selection covers a wide range of Android security relevant operations, like file management, Internet connection, IPC communication and launch of new applications. The motivation of reducing the subset of monitored kernel calls is to limit the overhead generated by both the MKM and *KernelCallTester* on the testing devices.

5 Testing and Experimental Results

We installed the two customized Android builds presented in Sect.4.1 into a set of ten smartphones. In particular, we deployed the Android build v.2.3.3 to five smartphones (i.e. HTC Desire HD, LG Optimus One p550, LG Optimus 3D,

[2] `http://developer.android.com/about/dashboards/index.html`
[3] `https://github.com/android/platform_bionic/blob/master/libc/SYSCALLS.TXT`

LG Optimus L3, Galaxy Next GT-S5570) and the Android build v.4.0.3 to other five smartphones (i.e. Galaxy Nexus, HTC Sensation XL, Motorola Droid RAZR MAXX, Galaxy Tab 7.1, HTC Vivid). Then, we delivered the smartphones to a heterogeneous set of users (i.e. university students, teenagers, professors and clerks) for normal use for two weeks. Users have been left free to install and use every kind of application.

During the testing period the execution of applications had forced services in the AF layer to invoke kernel calls to provide functionalities to applications. By analyzing the MKM logs from different smarpthones, which reported more than 100.000 kernel call invocations, we were able to relate services in the AF layer with the kernel calls they invoke. Results are presented in Table 1.

Table 1. Kernel calls invoked by services in the AF layer

AF service	**Kernel calls**
Alarm Manager	`getpid`, `ioctl`, `open`
Activity Manager	`close`, `getpid`, `gettid`, `ioctl`, `lseek`, `mkdir`, `open`, `prctl`, `read`, `write`
Audio Service	-
BatteryStats	`close`, `exit`, `gettid`, `open`
GpsLocationProvider	`getpid`, `ioctl`
Location Manager Service	`getpid`, `ioctl`, `lseek`, `open`, `read`
Package Manager	`close`, `getpid`, `gettid`, `ioctl`, `lstat64`, `open`, `sendmsg`, `write`
Power Manager Service	`getpid`, `ioctl`, `open`, `read`, `write`
ServerThread	`close`, `connect`, `getpid`, `gettid`, `ioctl`, `lseek`, `lstat64`, `open`, `prctl`, `read`, `recvmsg`, `sendmsg`, `sendto`, `socket`, `write`
ThrottleService	`close`, `exit_group`, `getpid`, `gettid`, `ioctl`, `open`, `prctl`, `read`, `sendmsg`, `write`
VoldConnector	`getpid`, `gettid`, `ioctl`, `open`, `recvmsg`, `write`
Window Manager	`close`, `getpid`, `gettid`, `ioctl`, `open`, `read`, `write`

Furthermore, 28 out of 33 (85%) of the kernel call types intercepted by the MKM have been successfully replicated by the *KernelCallTester* both on Android v. 2.3.3 and v. 4.0.3. Only 5 calls (15%) failed due to Linux permissions errors or wrong parameters. More in detail:

System Calls. System calls reproduced by *KernelCallTester* can be divided into file management and process management system calls.

- **File Management System Calls.** The MKM intercepts system calls related to files management as well as the parameters used (e.g. the data written and absolute path of the file opened). Since files are accessed in Linux by means of *file descriptors*, whenever a file system call occurs, *KernelCallTester* tries to reopen the targeted files, then reproduces the corresponding operation (`write`, `read` or `lseek`). Moreover, *KernelCallTester* handles `mkdir` and `rmdir` system calls. In all tests executions,

we noticed that possible failures are only due to Linux permission errors (e.g. *KernelCallTester* is not in the owner group of a certain file).
- **Process Management System Calls.** *KernelCallTester* is also able to reproduce successfully process management system calls like `gettid`, `getpid` or `exit_group`. Only the `kill` call cannot be reproduced because an unprivileged user is authorized to kill only process she owns.

Socket Calls. Since sockets are mapped on files, *KernelCallTester* firstly tries to `connect` to the socket before reproducing socket calls like `recvmsg` or `sendmsg`. Failure is again only due to insufficient Linux access permissions (e.g. sockets owned by *root* with permissions set to *660*) or unaccepted parameters (e.g. `bind` fails because the targeted socket is already bound).

Binder Calls. Binder calls rely on `ioctl` system call which allows sending simple commands and data to a file descriptor. Regarding IPC, the sender process performs a `ioctl` call to the Binder, specifying the addressee and including the data of the message. The Binder, which handles the passing mechanism, reads incoming messages and routes them to the appropriate destinations. *KernelCallTester* is not able to reproduce exactly the same call since one of the arguments is the pointer to the data sent which is located within addresser's memory space, causing a *Bad Address* failure.

Above all, our tests show that **every kernel call invoked by trusted services in the** AF **layer can be reproduced by any unprivileged application.** More specifically, our experiments show that the ASF never intervenes whether a kernel call invocation is attempted by an application in stead of a trusted service, revealing that Android does not perform any discrimination according to the caller of a kernel call. Moreover, no intervention is performed on repeated (and suspicious) invocation of the same kernel call; in particular, each attempts to replicate (also 10K times in a few seconds) the same kernel call has never been recognized as suspicious by the ASF.

Thus, the possibility to properly forge and directly execute kernel calls from the A layer may allow any application to take advantage of the kernel functionalities for malicious purposes. To this aim, in the next section we show two interplays that a malicious application can execute to reduce the performance of the device and to violate the privacy of the user.

6 Kernel Calls and Malicious Interplay

A central design point of the Android security architecture is that applications cannot adversely impact other applications and the operating system, or the user [4]. Such statement represents the final goal of the ASF, according to the official Android documentation. Starting from this statement, we identified two security goals that should be obliviously granted:
- an application cannot exhaust system resources;
- an application cannot access data of another application.

[4] `https://developer.android.com/guide/topics/security/security.html`

Then, we analyzed all logs produced by the MKM during the testing phase in order to find potential vulnerabilities that, if exploited, allow to violate the previous security goals. The analysis has been carried out with both manual and automatic inspection of whole set of logs. The analysis activity allowed to infer that:

- file system operations with proper parameters and sufficient permissions are unbounded;
- files located in `/data/data/com.android.browser/cache/webviewCache`, containing the cache (e.g. images, javascript codes, web pages accessed by the user) of installed browsers, can be read by any application.

Thus, we implemented two malicious applications, namely *WriteTest* (requiring no privileges upon installation) and *CacheHooker* (requiring only the INTERNET permission) that exploit such characteristics to perform attacks by means of kernel calls.

WriteTest repeatedly execute an `open`, a `write` and a `close` system call at different periods of time. Each interplay create a dummy file of 4 MB in the internal memory of the phone.

CacheHooker cyclically invokes a `read` on the cache file, followed by a `write` on its data folder. Moreover it connects (using `socket` and `connect` calls) and sends (relying on `sendmsg` call) every copied file to a remote server. The connecting operation exploits the `android.permission.INTERNET` permission.

We installed both applications to the testing smartphones, so that they execute as services in background after a random period of time after the boot completion. Users reported that other applications had started to crash or to terminate at launch, and that Android provides back pop-ups on the exhaustion of the phone memory. Uninstalling other applications did not solve the problem. Only very few users was able to identify the *WriteTest* application and solve the problem by flashing the device. Furthermore, during the testing period *CacheHooker* has successfully and silently grabbed users browsing data delivering them to the remote server.

7 Enhancing the ASF

We argue that malicious interplay related to kernel calls may be limited by providing the ASF with the possibility to identify the caller of a kernel call. To this aim, we propose an improvement to the ASF requiring the modification of the *Bionic Libc* and the development of a kernel call evaluation module, called *Kernel Call Controller* (KCC). KCC, placed at the Library layer, exposes a set of options for evaluating a kernel call, which can be set by the *Activity Manager* at the AF layer:

Application Kernel Call Restriction. It allows to deny/permit kernel call invocations performed by the Application layer (i.e. caller with PID greater than 10000).

Kernel Call Frequency Restriction. It allows to upper-bound the number of calls of a given type (also with different parameters) that can be invoked by a single caller in an amount of time (i.e. 1, 10 or 60 seconds).

Bionic Libc is a derivation of the standard C implementation for a mobile environment. In Android, this library is included in each Linux process and it performs the actual invocation of the kernel calls. Each call is implemented by a different tiny assembler source fragment (called *syscall stub*), which is responsible for executing the corresponding kernel code. Syscall stubs are generated automatically by a Python script (i.e. *gensyscalls.py*).

We extended the Bionic Libc by modifying the syscall stubs, i.e. changing the script that generates them. In particular, before executing the kernel code of the call, the modified stub invokes the main routine of KCC , which is able to i) identify the PID of the caller, ii) calculate the frequency of each invocation, and then iii) assess whether the behavior is compliant with the KCC settings, allowing/denying the execution of the call accordingly. Furthermore, information collected by the KCC are stored in the */data* folder as a text file for offline analysis.

The idea of hooking the system call procedures has been also adopted by SEAndroid[5]. However, the adoption of SEAndroid has a considerable impact on the system architecture. In fact, whereas our patch involves only a modification of the Bionic Libc, SEAndroid requires a deep customization of some parts of the Android Framework and the recompilation of the Linux Kernel.

We deployed KCC as well as *WriteTest* and *CacheHooker* in our two Android builds and then installed them into the ten smartphones. Then, we repeated the same test described in Sect. 5, providing users with devices. We configured KCC disabling the invocations of kernel calls from the A layer and limiting the number of allowed calls to 10 per second. KCC was able to block unexpected direct kernel calls invocations from *WriteTest* and *CacheHooker* applications, thus fully preventing the memory exhaustion and the web cache privacy leak (i.e. no data has been sent to the remote server).

During the testing phase, no user reported any visible performance issue or unexpected behavior. Such results, although limited to a small subset of all available Android applications, indicate that applications may not need to perform direct kernel call invocations to work properly. However, further analysis should be carried out on a more comprehensive set of applications.

8 Related Work

Literature on Android security has considerably spread in recent years, including general surveys, like [15] and [9], vulnerabilities, tools and formal methods to enhance both the Android architecture and the corresponding security model. Regarding vulnerabilities, recent works show that the Android platform may suffer from DoS attacks [1], covert channels [14], web attacks [12] and privilege

[5] `http://selinuxproject.org/page/SEAndroid`

escalation attacks [8]. The same works underline that such vulnerabilities affect each Android build.

Several authors underline the limitation of the current Android security model, thus proposing methodologies and tools to extend the native Android security solutions. For instance, in [13] authors propose an extension to the basic Android permission systems and corresponding new policies, while in [16] authors suggest new privacy-related security policies for addressing security problems related to users' personal data. Other tools are devoted to malware detection (e.g. XManDroid [5] and Crowdroid [6]) and application certification (e.g. Scandroid [10] and Comdroid [7]).

However, none of such solutions takes into consideration interplay between the Android stack and the Linux kernel, focusing on interplay related to the Android stack only.

Monitoring of system calls in Android is discussed by Blasing et al. [3]. They propose an Android Application Sandbox (AASandbox) which is able to perform both static and dynamic analysis in a fully isolated environment. AASandbox relies on a loadable kernel module which monitors system calls, although such approach is particularly different from the one proposed in this paper for a two-fold reason. First, the kernel module is able to log only the return value of each system call, without logging the parameters. Furthermore, authors do not provide any security assessment regarding Android system calls, neither suggest any solution that could mitigate possible malicious interplay.

To the best of our knowledge, our work is the first attempt to empirically investigate correlations and security issues related to the interplay among the Android layers and the Linux kernel.

9 Conclusions

In this paper, we empirically demonstrated that Android allows applications to directly invoke Kernel functionalities. We built an ad-hoc kernel module (i.e. the *Monitoring Kernel Module*) and an application (i.e. *KernelCallTester*) to capture and replicate all kernel calls invoked by trusted services in AF layer. We demonstrated that this trait may lead to undermine the security of the system as well as the privacy of the user; we proved it by means of two malicious applications (i.e. *WriteTest* and *CacheHooker*) implemented after analyzing the logs of the MKM. However, other potential vulnerabilities may be hidden in the logs produced by the MKM. Nevertheless, the discovery of vulnerabilities in Android is far from being automated and exhaustive. In order to achieve such results in the near future, we argue that two problems must be tackled. First, it is currently difficult to assess whether a specific behavior (e.g. connecting to a system socket) violates the Android Security policy, since the same policy is very informally defined. For this, we argue that proper languages for formally stating the expected security properties and behavior of an Android build should be investigated. Then, new techniques for automatically/semi-automatically search for vulnerabilities in Android builds should be identified.

References

1. Armando, A., Merlo, A., Migliardi, M., Verderame, L.: Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 13–24. Springer, Heidelberg (2012)
2. Armando, A., Merlo, A., Migliardi, M., Verderame, L.: Breaking and fixing the android launching flow. Elsevier Computer and Security (2013)
3. Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S., Albayrak, S.: An android application sandbox system for suspicious software detection. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 55–62 (2010)
4. Brady, P.: Anatomy and physiology of an android. Google I/O (2008)
5. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt (April 2011)
6. Burguera, I., Zurutuza, U., Nadjm-Therani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011 (2011)
7. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys 2011, pp. 239–252. ACM, New York (2011)
8. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
9. Enck, W., Ongtang, M., McDaniel, P.: Understanding Android Security. Security & Privacy 7(1), 50–57 (2009)
10. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Technical report (2009)
11. Gartner Group. Press Release (August 2012), `http://www.gartner.com/it/page.jsp?id=2120015`
12. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on webview in the android system. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 343–352. ACM, New York (2011)
13. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, pp. 328–332. ACM, New York (2010)
14. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS) (February 2011)
15. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google android: A comprehensive security assessment. IEEE Security Privacy 8(2), 35–44 (2010)
16. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)