Pervasive and Mobile Computing 30 (2016) 113–131

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Android vs. SEAndroid: An empirical assessment[☆]

Alessio Merlo[a,*], Gabriele Costa[a], Luca Verderame[a], Alessandro Armando[a,b]

[a] DIBRIS, Università degli Studi di Genova, Italy

[b] Security & Trust Unit, FBK-irst, Trento, Italy

A R T I C L E   I N F O

Article history:
Received 6 August 2015
Received in revised form 18 November 2015
Accepted 7 January 2016
Available online 21 January 2016

Keywords:
Android OS
Android security
Android Security Framework
SEAndroid
SELinux MAC

A B S T R A C T

Android has a layered architecture that allows applications to leverage services provided by the underlying Linux kernel. However, Android does not prevent applications from directly triggering the kernel functionalities through system call invocations. As recently shown in the literature, this feature can be abused by malicious applications and thus lead to undesirable effects. The adoption of SEAndroid in the latest Android distributions may mitigate the problem. Yet, the effectiveness of SEAndroid to counter these threats is still to be ascertained. In this paper we present an empirical evaluation of the effectiveness of SEAndroid in detecting malicious interplays targeted to the underlying Linux kernel. This is done by extensively profiling the behavior of honest and malicious applications both in standard Android and SEAndroid-enabled distributions. Our analysis indicates that SEAndroid does not prevent direct, possibly malicious, interactions between applications and the Linux kernel, thus showing how it can be circumvented by suitably-crafted system calls. Therefore, we propose a runtime monitoring enforcement module (called *Kernel Call Controller*) which is compatible both with Android and SEAndroid and is able to enforce security policies on kernel call invocations. We experimentally assess both the efficacy and the performance of KCC on actual devices.

1. Introduction

Android consists of a Java stack built on top of a native Linux kernel. Services and functionalities are achieved through the interplay of components residing at different layers of the operating system. The Android Security Framework (ASF) consists of a number of cross-layer security solutions combining basic Linux security mechanisms (e.g. Discretionary Access Control), the app isolation offered by the Java Virtual Machine execution environment and Android-specific mechanisms (e.g. the Android permission system).

The security offered by the ASF has been recently challenged by the discovery of a number of vulnerabilities involving all layers of the Android stack (see, e.g., [1–3]). The analysis of these interplay-related vulnerabilities indicates that

- the security mechanisms of the Android stack (both Java native and Android-specific) are not completely integrated with those in the Linux kernel, thereby allowing insecure interplay among layers;
- malicious and unprivileged Android applications can directly interact with the underlying Linux kernel, thereby by-passing the controls performed by the ASF.

☆ This work has been partially funded by the Italian PRIN project *Security Horizons* (no. 2010XSEMLC).

* Corresponding author.
E-mail addresses: alessio.merlo@unige.it (A. Merlo), gabriele.costa@unige.it (G. Costa), luca.verderame@unige.it (L. Verderame), armando@fbk.eu (A. Armando).

http://dx.doi.org/10.1016/j.pmcj.2016.01.006
1574-1192/© 2016 Elsevier B.V. All rights reserved.

In [1] and [4] we reported a vulnerability in cross-layer interaction that allowed to mount a fork bomb attack on all Android distributions up the release of a patch for 4.0.3 version, in [5] we have carried out an empirical evaluation aimed at determining to which extent such a lack of control in the ASF may allow applications to maliciously trigger the Linux kernel functionality by means of properly-forged invocations.

The introduction of the *Security Enhancements for Android* project (SEAndroid) [6] in the Android Open Source Project (AOSP) since version 4.4.3 calls for a reconsideration of the results reported in [5]. In fact, according to the official documentation, the SEAndroid project aims at enabling the use of SELinux in Android *"in order to limit the damage that can be done by flawed or malicious applications and in order to enforce separation guarantees between applications"* [7] and it should therefore mitigate the aforementioned problems. It must also be observed that SEAndroid suffers from a number of limitations. For instance, the development of a *"good policy"* and the trustworthiness of the ASF are *"crucial to the effectiveness of SE Android"* [7].

To illustrate, let us consider the Zygote vulnerability reported in [1]. Such a vulnerability 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 (carried out 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 applications. Although SEAndroid is able to address the Zygote vulnerability (by limiting the usage of socket devices by applications), it *"cannot in general mitigate kernel vulnerabilities"* [7].

This paper extends the methodology proposed in [5] in a number of ways:

1. An experimental setup for SEAndroid is discussed. The experimental setup is aimed at (i) systematically capturing invocations to the Linux kernel from different layers in the Android stack, and (ii) replicating the invocations through a properly-crafted application.
2. A new empirical assessment, involving both recent Android and SEAndroid builds, has been carried out and a comparison between the detection capabilities of Android and SEAndroid is discussed.
3. An interplay that allows a malicious application to circumvent SEAndroid is shown. This witnesses the limited control exercised by SEAndroid on the interactions between applications and the Linux kernel.
4. A policy enforcement module, called *Kernel Call Controller* (KCC for short), that provides both Android and SEAndroid with the possibility to recognize and limit the direct interaction between the Android stack and the Linux kernel is presented.
5. An empirically-inferred KCC policy for filtering kernel call invocations according to (i) the identity of the caller and (ii) the number of repeated invocations, is discussed.
6. KCC performance and reliability is evaluated and discussed using the sample policy.

Structure of the paper. Section 2 provides a general introduction to the cross-layer architecture of Android. Section 3 discusses the peculiarities and limitations of the ASF whereas Section 4 introduces SEAndroid and discusses its security features. Section 5 describes the implementation and setup of our assessment environment for both Android and SEAndroid. Section 6 is devoted to describing the experimental setup and results. Section 7 discusses the effectiveness of the security countermeasures provided by the ASF and SEAndroid against two malicious applications. Section 8 introduces the *Kernel Call Controller* module, the language used for defining enforcement policies and its empirical assessment using a empirically-inferred policy. Section 9 discusses some related work while Section 10 concludes the paper 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 executed on the device. Android applications are built as a set of independent execution modules called *components*. The reader can refer to [4] for further details regarding application components and their interactions.

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 the Android core components[1] for managing the device and for interacting with the underlying Linux drivers.

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). The Dalvik VM is specifically optimized for efficient concurrent execution of virtual machines in a resource constrained environment.

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