Introduction to Android 5 Security

Lukáš Aron and Petr Hanáček

Faculty of Information Technology, Department of Intelligent Systems,
Brno University of Technology
lukas.aron@gmail.com

Abstract. This paper discusses the basic introduction into Android security. It is focused on the last version of Android platform. In the world of numbers it means Android 5.0 with code-name Lollipop. This paper covers description of SELinux and impact on mobile platform, Android Application Sandbox and the whole new virtual machine ART, which was introduced in Android 4.4. The aim of this paper is to introduce the researchers of security principles on Android into new version of thsi operating systems and it's new technologies related to security.

Introduction

Android is a software stack for mobile devices that includes an operating system, middle-ware and key applications. In these days it's the most using operating system on mobile devices on the world. There are a lot of version which are widespread on mobile devices of different manufactures and almost every manufacturer changed the Android user interface. However under user interface it's the same Android operating systems with its core and main functionality. We can say that the most new devices has pre-installed Android version 4.0 and higher. In November 2014 was introduced new version of this mobile platform - Android 5.0 Lollipop and it has some new features and changes according to previous versions. This paper is aimed to these features which has impact to Security. Android 5 brings new view to the security on mobile devices, for example now the device is encrypted from its beginning. There is analyzed the topic of Security-Enhanced Linux which was not mandatory on Android 4. Another new feature is a new virtual machine which replaced the old Dalvik VM. This paper contain basic introduction into this new features and also describe the basic knowledge about functionality of these features.

Security-Enhanced Linux

Security-Enhanced Linux - SELinux [12,13,16] is the primary Mandatory Access Control - MAC [9,10] mechanism built into a number of GNU/Linux distributions. SELinux originally started as the Flux Advanced Security Kernel (FLASK) [11] development by the Utah university Flux team and the US Department of Defence. The development was enhanced by the NSA and released as open source software. There are many views on the usefulness of SELinux on Linux based systems, this section gives a brief view of what SELinux is good at and what it is not (because its not designed to do it). SELinux is not just for military or high security systems where Multi-Level Security (MLS) [3] is required (for functionality such as "no read up" and "no write down"), as using the "type enforcement" functionality applications can be confined (or contained) within domains and limited to the minimum privileges required to do their job, so in a "nutshell":

- If SELinux is enabled, the policy defines what access to resources and operations on them (e.g. read, write) are allowed (i.e. SELinux stops all access unless allowed by policy). This is why SELinux is called a MAC system.
- The policy design, implementation and testing against a defined security policy or requirements is important, otherwise there could be "a false sense of security".
- SELinux can confine an application within its own "domain" and allow it to have the minimum privileges required to do its job. Should the application require access to networks or other applications (or their data), then (as part of the security policy design), this access would need to be granted (so at least it is known what interactions are allowed and what are not - a good security goal).
- Should an application "do something" it is not allowed by policy (intentional or otherwise), then SELinux would stop these actions.
- Should an application "do something" it is allowed by policy, then SELinux may contain any damage that maybe done intentional or otherwise. For example if an application is allowed to delete all of its data files or database entries and the bug, virus or malicious user gains these priviledges then it would be able to do the same, however the good news is that if the policy "confined" the application and data, all your other data should still be there.
- User login sessions can be confined to their own domains. This allows clients they run to be given only the privileges they need (e.g. admin users, sales staff users, HR staff users etc.). This again will confine/limit any damage or leakage of data.
- Some applications (X-Windows for example) are difficult to confine as they are generally designed to have total access to all resources. SELinux can generally overcome these issues by providing sandboxing services.
- SELinux will not stop memory leaks or buffer over-runs (because its not designed to do this), however it may contain the damage that may be done.
- SELinux will not stop all viruses/malware getting into the system (as there are many ways they could be introduced (including by legitimate users), however it should limit the damage or leaks they cause
- SELinux will not stop kernel vulnerabilities, however it may limit their effects.
- Finally, SELinux cannot stop anything allowed by the security policy, so good design is important.

SELinux on Android

This section explains how the previously noted challenges to enabling the effective use of SELinux in Android were overcome in the SE Android reference implementation. Overcoming these challenges required changes to the kernel, changes and new additions to the Android user-space software stack, and the creation of a new policy configuration for Android. Starting with Android 4.3, (SELinux) is used to further define the boundaries of the Android application sandbox.

SELinux operates on the ethos of default denial. Anything that is not explicitly allowed is denied. SELinux can operate in one of two global modes: permissive mode, in which permission denials are logged but not enforced, and enforcing mode, in which denials are both logged and enforced. SELinux also supports a per-domain permissive mode in which specific domains (processes) can be made permissive while placing the rest of the system in global enforcing mode. A domain is simply a label identifying a process or set of processes in the security policy, where all processes labeled with the same domain are treated identically by the security policy. Per-domain permissive mode enables incremental application of SELinux to an ever-increasing portion of the system. Per-domain permissive mode also enables policy development for new services while keeping the rest of the system enforcing.

In the Android 5.0 L release, Android moves to full enforcement of SELinux. This builds upon the permissive release of 4.3 and the partial enforcement of 4.4. In short, Android is shifting from enforcement on a limited set of crucial domains (installd, netd, vold and zygote) to everything. This means manufacturers will have to better understand and scale their SELinux implementations to provide compatible devices. In other words these changes has impact to:

- Everything is in enforcing mode in the Android 5.0 release
- Only `init` process should run in the `init` domain
- Any generic denial (for a block_device, socket_device, default_service, etc.) indicates that device needs a special domain

Using SELinux in Android first requires enabling SELinux and its dependencies in the kernel configuration and rebuilding the kernel. SELinux dependencies in the kernel include the Linux Security Module (LSM) framework, the audit subsystem, and file-system support for extended attributes and security labels for each relevant file-system for the device. File-systems that do not support extended attributes or security labeling such as `vfat` can still be used, but can only be labeled and protected at per-mount granularity rather than per-file granularity. More recent Android devices have begun using the Linux `ext4` filesystem for `eMMC` storage on Android devices, since `eMMC` exposes a traditional block-based interface on which conventional disk-based Linux file-systems can be layered. Ext4 already incorporates all the necessary support for extended attributes and security labeling and is already tested with SELinux as part of conventional Linux distributions.

Enabling user-space support for SELinux in Android required changes to a wide variety of software components, ranging from Android's C library implementation, file-system generation tools, and init program up through the Android framework services. It also required porting the core SELinux user-space components to Android.

Let's clarify some concepts related with SEAndroid:

- Security-Enhanced Linux is an implementation of mandatory access control using Linux Security Modules (LSM) in the Linux kernel, based on the principle of least privilege. It is not a Linux distribution but instead a set of modifications that can be applied to UNIX*-like operating systems, such as Linux and BSD.
- Discretionary Access Control (DAC) [8] is the standard security model for Linux. In this model, access privileges are based on the user identity and object ownership. Such DAC had been used for previous versions of Android until Android 4.3.
- Mandatory Access Control (MAC) limits privileges for subjects (processes) and objects (file, socket, device, etc.).

SELinux does not change any existing security in the Linux environment; instead, SELinux extends the security model to include Mandatory Access Control (e.g., both MAC and DAC are enforced in the SELinux environment). SEAndroid enhances the Android system by adding SELinux support to the kernel and user space to:

- Confine privileged daemons to protect them from misuse and limit the damage that can be done via privileged daemons.
- Sandbox and isolate apps from each other and from the system. Details about Android Application Sandbox folows in next section.
- Prevent privilege escalation by apps.
- Allow application privileges to be controlled at installation and runtime using middleware mandatory access control.
- Provide a centralized, analyzable policy.

Furthermore, in Android 4.4, SEAndroid is enabled in the Enforcing mode, instead of the non-functional disabled mode or the notification-only permissive mode, which means that any invalid operation will be prohibited in the Android run-time environment. In Android 5.0 has been setup full enforcement, as was described in this section.

Android Application Sandbox

Application sandboxing, also called application containerization, is an approach to software development and mobile application management (MAM) that limits the environments in which certain code can execute. The goal of sandboxing is to improve security by isolating an application to prevent outside malware, intruders, system resources or other applications from interacting with the protected app. The term sandboxing comes from the idea of a child's sandbox, in which the sand and toys are kept inside a small container or walled area. Android uses the concept of a sandbox [1,7] to enforce inter-application separation and permissions to allow or deny an application access to the device's resources such as files and directories, the network, the sensors, and APIs in general. For this, Android uses Linux facilities such as process-level security, user and group IDs that are associated with the application, and permissions to enforce what operations an application is allowed to perform. Conceptually, a sandbox can be represented as in Figure 1.

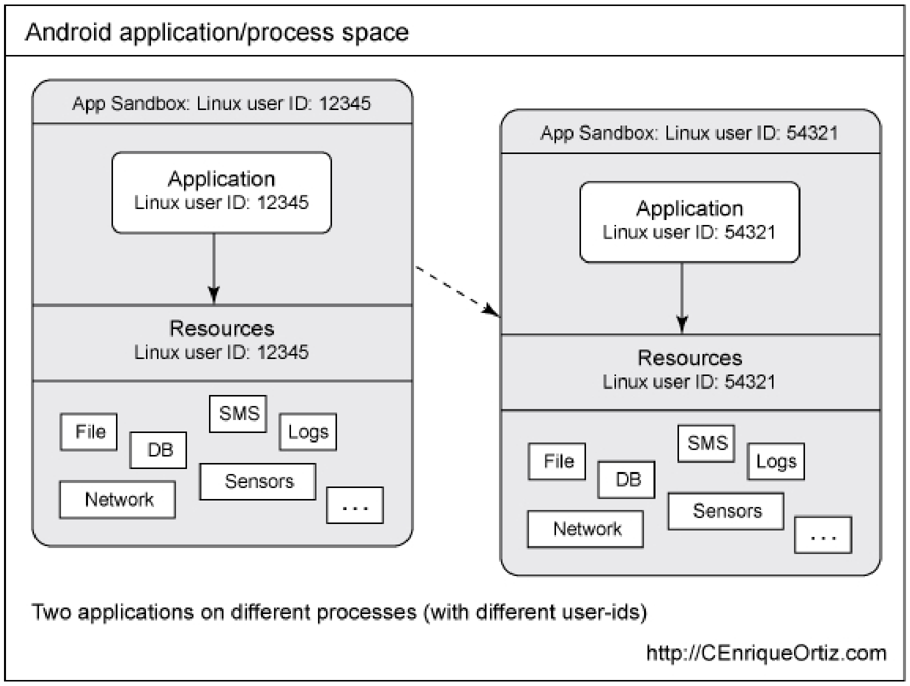

Fig. 1. Two Android applications, each on its own basic sandbox or process

Sandboxes are often located within kernel space since access to critical parts of the OS can be realized. The kernel is a very essential part of a system because it acts as bridge between hardware and software. One approach of sandbox systems is to monitor system and library calls including their arguments. This is often done trough system call redirecting, also known as system call hijacking. System calls, short system calls, are function invocations made from user space into the kernel in order to request some services or resources from the operating system. Figure 2 shows an example for the read() system call on a Linux based system. Android uses a modified Linux basis to host a Java-based middleware running the user applications. This implies that calls should not be monitored on Java level since other calls being made by native Linux application might get lost.

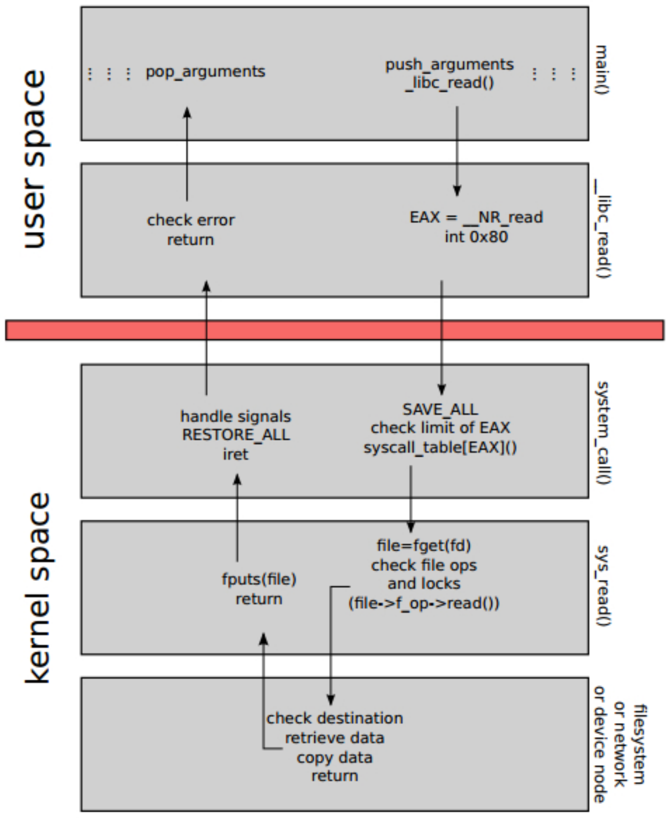

Fig. 2. Steps involved in performing a `read()` system call from user space. Each arrow in the figure represents a jump in the instruction flow

Virtual Machine

Android is a mobile operating system mainly used for mobile devices, tablets or TVs. Android is based on Linux kernel as was discussed earlier, supplemented with middleware and libraries written in C/C++, yet Android application (app) itself is written in Java and run with the Android framework and Java-compatible libraries. Android can enjoy the full benefit of Java such as platform independence, added security, and ease of app development. Android had its own virtual machine (VM) to execute Java application, called Dalvik VM [2,4,5]. The Dalvik VM differs from conventional Java VM [15]. First, the Dalvik VM has its own register-based bytecode rather than the Java VM's stack-based bytecode. This can in theory lead to more efficient interpretation due to fewer bytecode instructions fetched for interpretation, reducing the fetch overhead which is crucial to the interpreter performance. Secondly, the Dalvik VM employs trace-based just-in-time (JIT) compilation [14], while high-performance Java VMs use method-based JIT compilation. The motivation for the trace-based JIT compilation is that it would reduce the memory overhead of the mobile devices and compilation time, without affecting the performance benefit of JIT compilation since only hot paths within a method will be compiled.

Generally, one of Java's advantages as a mobile software platform is a platform independence, achieved by using the bytecode that can be executed by the interpreter on any platform without porting. Since this software-based execution is much slower than hardware-based execution, just-in-time compilation for translating bytecode into machine code at runtime has been used in the Java VM. The Dalvik VM is not an exception and it also employs JIT compilation. However, there are some differences in the JIT compilation techniques used in the two VMs. Dalvik VM has been used until Android 4.4 was introduced. In this version of operating system the user can choose between two types of VM - Dalvik VM or ART. In Android 5.0 Lollipop is only one version and it is the new VM called ART or Android RunTime.

Android runtime (ART) is the managed runtime used by applications and some system services on Android. ART and its predecessor Dalvik were originally created specifically for the Android project. ART as the runtime executes the Dalvik Executable format and Dex bytecode specification. ART and Dalvik are compatible runtimes running Dex bytecode, so apps developed for Dalvik should work when running with ART. However, some techniques that work on Dalvik do not work on ART. ART introduces ahead-of-time (AOT) compilation [6], which can improve app performance. ART also has tighter install-time verification than Dalvik. At install time, ART compiles apps using the on-device `dex2oat` tool. This utility accepts DEX files as input and generates a compiled app executable for the target device. The utility should be able to compile all valid DEX files without difficulty. However, some post-processing tools produce invalid files that may be tolerated by Dalvik but cannot be compiled by ART. Garbage collection (GC) can impair an app's performance, resulting in choppy display, poor UI responsiveness, and other problems. ART improves garbage collection in several ways:

- One GC pause instead of two
- Parallelized processing during the remaining GC pause
- Collector with lower pause time for the special case of cleaning up recently-allocated, short-lived objects
- Improved garbage collection ergonomics, making concurrent garbage collections more timely, which makes GC_FOR_ALLOC events extremely rare in typical use cases

ART currently does not use compacting GC, but this feature is under development in the Android Open Source Project (AOSP). In the meantime, don't perform operations that are incompatible with compacting GC, such as storing pointers to object fields. Information about ART comes from official site of Android project.

There are some basic information, that comes from comparison of virtual machines. The new ART VM has better performance on executing application, because the whole application now runs from compiled code. On the opposite the application takes more memory on flash disc. This is related to bytecode (or compress bytecode), where one instruction means more instruction in binary code. In these day when the cost of memory is low we do not have a problem with space on mobile devices. The performance is no the most important pointer in development.

Conclusion

Improvement of the whole security model in Android version 5.0 has impact to keep users safe and try to protect them before hackers, attackers and also from processes or malware applications. This paper explains new security features of new Android operating system and how these features work. Combination of these changes make a big jump to forward in comparing to older versions. The new Android 5.0 Lollipop offers mandatory access control, encrypted disc, encrypted memory, new virtual machine which improves performance and still keep application separate through Sandbox. There are some questions about backward compatibility, but it is not related to this article.

Acknowledgement. This article was created within the research plan of the Security-Oriented Research in Information (MSM0021630528). This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the project Advanced Secured, Reliable and Adaptive IT (FIT-S-11-1).

References

1. Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In Malicious and unwanted software (MALWARE), 2010 5th international conference on, pp. 55–62, IEEE (2010)
2. Bornstein, D.: Dalvik vm internals. In Google I/O developer conference, vol. 23, pp. 17–30 (2008)
3. Chen, J.F., Wang, J.-S.: Application level security system and method, February 11 1997. US Patent 5,602,918 (1997)
4. Fresko, N., Lam, M., Wong, H.: Platform-independent selective ahead-of-time compilation, May 1 2007. US Patent 7,213,240 (2007)
5. Li, N.: Discretionary access control. Encyclopedia of Cryptography and Security, pp. 353–356 (2011)
6. Lindqvist, H.: Mandatory access control. Master's Thesis in Computing Science, Umea University, Department of Computing Science, SE-901, 87 (2006)

7. Smalley, S.: Flask: Flux advanced security kernel (1997)
8. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., Nakatani, T.: Overview of the ibm java just-in-time compiler. IBM systems Journal, 39(1):175–193 (2000)
9. Venners, B.: Inside the Java virtual machine. McGraw-Hill, Inc. (1996)
10. Vermeulen, S.: SELinux Cookbook. Packt Publishing Ltd. (2014)