Solutions for Adjusting SELinux To Android-Powered Devices

Anh-Duy Vu, Jea-Il Han, Young-Man Kim
Dept of Computer Science, Kookmin University
e-mail : {anhduyvu,jhan,ymkim}@kookmin.ac.kr

안드로이드 응용 단말기를 위한 SELinux 환경설정 방법

안 두이 부, 한재일, 김영만
국민대학교 컴퓨터 공학부

Abstract

Google Android framework consists of an operating system and software platform for mobile devices. Using a general-purpose Linux operating system in mobile device has some advantages but also security risks. Security-Enhanced Linux (SELinux) is a kernel-based protection approach which can help to reduce potential damage from successful attacks. However, there are some challenges to integrate SELinux in Android. In this research, we do a study on how to do the integration and find out four challenges. The first one is that the Android file system (yaff2) does not support security namespace for extended attribute (xattr) which is required by SELinux. The second one is that it's difficult to apply SELinux policy to Dalvik process on which an Android application runs on. The third one is that Android lacks methods, tools and libraries to interact with SELinux. The last one is how to update the SELinux policy automatically when installing or removing an application. In this paper, we propose solutions for the above limitations that make the SELinux more adaptive and suitable for Android framework.

1. Introduction

The idea of utilizing SELinux in Linux-based embedded devices is not novel and was first proposed by Bojorn Vogel et al. in [1]; however some technological issues remain challenging such as

- The boot loader does not provide the capability of passing boot options to the kernel so that SELinux can be switched on and off at the startup.
- Some embedded file systems (jff2, yaff2…) do not officially support extended attribute (xattr) which is used by SELinux to store its own addition data.
- The system does not support methods, tools and libraries to interact with SELinux.

As mention above, the Android framework also gets the last two challenges, another one from its specific design, and the last one from the convince purpose. In details

- The Android file system (yaffs2) supports extended attribute but not security namespace which is specific part of xattr and mainly utilized by SELinux to store its own additional information called security context.
- As a fundamental design choice, only one Dalvik-based process, zygote, is executed in the Android framework. All other Dalvik-based processes are forked from zygote to share memory. SELinux can label an implicit process only upon file execution, which isn't the case for Dalvik-based processes. As such, these Dalvik processes currently aren't labeled.
- The Android framework lacks methods, tools and libraries to interact with SELinux so it is impossible to compile the SELinux

policy on the device.

- It is not practical to force users themselves to study and update the SELinux policy whenever they install or remove an application.

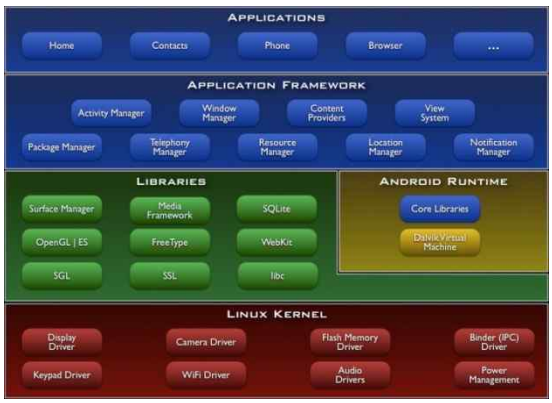
In this paper, our study does not stop at identifying these challenges but also provides ideas and implementations to solve them as discussion below.

The rest of the paper is organized as follows. The next coming Section 2 discusses the background of Android architecture and SELinux. In section 3, we discuss how to integrate SELinux with Android and solve the four mentioned limitations. Finally, conclusions and future works are discussed in section 4.

2. Background

2.1. Android architecture

Figure 1 shows the major components of the Android operating system.



[Fig. 1] Android architecture [2]

Applications are placed at the top of the architecture. Some core applications include an email client, SMS program, calendar, maps, browser, contacts, and others.

The second layer is application framework. By providing an open development platform, Android offers developers the ability to build extremely rich and innovative applications. Developers are free to take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and much, much more.

The library layer provides set of C/C++ libraries used by various components of the Android framework and exposed to developers through the Android application framework. This layer also consists of Android Runtime component which contains Dalvik virtual machine relying on the Linux Kernel for underlying functionality such as threading and low-level memory management. Every Android application runs in its own process, with its own instance of the virtual machine [3].

The last layer is the Linux kernel. This layer is responsible for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

2.2. Android security mechanisms

2.2.1. Discretionary Access Control (DAC) mechanism [4]

The DAC mechanism is inherited from Linux, which controls access to files by process ownership. Each running process is assigned a UserID, while each file access rules are specified. Each file is assigned access rules for three set of subjects: user, group, and everyone. Each subject set may have permissions to read, write, and execute a file.

2.2.2. Permission mechanism [5]

Android's permissions mechanism for applications enforces restrictions on specific operations that an application can perform. Android has roughly 100 built-in permissions that control operations ranging from dialing the phone (CALL_PHONE), taking pictures (CAMERA), using the Internet (INTERNET), listening to key strokes (READ_ INPUT_STATE), and even disabling the phone perma-nently (BRICK). Any Android application can declare additional permissions. To obtain a permission, an ap-plication must explicitly request it.

2.2.3. Component ecapsulation mechanism [5]

Android's permissions mechanism for applications enforces restrictions on specific operations that an application can perform. Android has roughly 100 built-in permissions that control operations ranging from dialing the phone (CALL_PHONE), taking pictures (CAMERA), using the Internet (INTERNET), listening to key strokes (READ_ INPUT_STATE), and even disabling the phone perma-nently (BRICK). Any Android application can declare additional permissions. To obtain a permission, an ap-plication must explicitly request it.

2.2.4. Application signing mechanism [5]

Each application in Android is packaged in an

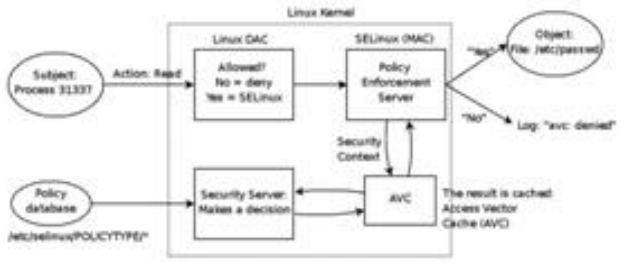
apk archive for installation. The Android framework requires that all installed applications be digitally signed (code and noncode resources). The signed apk is valid as long as its certificate is valid and the enclosed public key successfully verifies the signature. Signing applications in Android verifies that two or more applications are from the same au-thor ("same-origin" verification). The sharedUserId mechanism and the permission mechanism use this method to verify signature and signature-or-system protection-level permissions.

2.3. Security-Enhanced Linux (SELinux)

Security-Enhanced Linux (SELinux)[6] is an enhancement for the standard Linux kernel, which implements fine-grained Access Control based on the FLASK[7][8] concept to provide Mandatory Access Control (MAC).

In contrast to the classical DAC mechanism of Linux with owner, group, and anyone, SELinux attaches a special collection of security attributes called security context to each subject (such as process) and object (such as file and packets) within the system. The security context is used when an authorization decision needs to be made. The security information and authorization rules are defined in a policy file.

Figure 2 depicts the decision making in kernel supported by SELinux. When a subject (process) wants to access an object (passwd file), it must first be allowed by DAC. Then the decision is sent to SELinux. In SELinux the Policy Enforcement Server does a lookup in the Access Vector Cache (AVC) where earlier subject and objects permission are cached. If the decision is not found in the AVC, the request is forwarded to the Security Server which looks up the SELinux security context of the file and consults the policy predefined by System Security Administrator. Permission is then either denied or granted. The result is cached in the AVC.



[Fig. 2] Decision making in Linux Kernel supported by SELinux

3. Integrating SELinux in Android framework

We integrated SELinux in Android framework on both Nexus 1 and Nexus S phone, and test a policy that is specific for Android due to its unique file system, initialization process, and application startup. We encountered some challenges which are mentioned above

3.1. The Android file system lacks security namespace of extended attribute

The default file system (yaffs2) doesn't support security namespace of ex-tended attributes (xattrs). Other research has men-tioned this problem and solved it on other file systems for other platforms [1], [9].

We provide this namespace by modifying the code of yaffs2 hosted in kernel code of Nexus 1 and Nexus S. The detail of how to implement xattr and security namespace can be found at [10].

3.2. It's difficult to apply SELinux policy to Dalvik process[11]

As a fundamental design choice, only one Dalvik-based process, zygote, is executed in the Android framework. All other Dalvik-based processes are forked from zygote to share memory. The single process execu-tion of Dalvik-based processes limits SELinux's ap-plicability to Android. SELinux can label an implicit process only upon file execution, which isn't the case for Dalvik-based processes. As such, these processes currently aren't labeled.

SELinux can allow processes to explicitly change their labels, for example, by a userspace command that requests a label change and specifies the target label. This capability is supported by the type_change poli-cy directive, which is in contrast to the implicit type-transition feature mentioned earlier. We could make zygote SELinux-aware by altering the zygote code to explicitly label its children using this capability.

3.3. Android lacks methods, tools, and library to interact with SELinux

We solved this problem straightforwardly by cross-compiling libsepol, libselinux, and busybox to Android. However, this solution does not solve the problems of loading the policy early on boot or of la-beling the init process correctly. Because other platforms had SELinux-specific code in init, we also added our code to Android's init. The Android init process executes commands from an init.rc text file.

We added three new commands to the interpreter

- loadpolicy loads a policy file into the kernel,
- chcon changes a file's label, and
- context sets a label for daemons started by init. This let us adapt init.rc to our needs and label most of the early system.

3.4. How to automatically update SELinux policy whenever installing or removing an application

The idea to solve this challenge is to alter the installd code which is responsible for installing and removing an application. At the very end of the code, we add another module that parses the AndroidManifest.xml file to get the permission requests of that application to create a new SELinux policy module, compile, and load the module into SELinux kernel in case of installing a new application or just remove a module out off SELinux kernel in case of removing an application.

4. Conclusions

Integrating SELinux in Android hardens the Android framework and reduces potential damage from successful attacks. In this paper we identify four challenges while doing this integration: (1) The Android file system lacks security namespace of extended attribute; (2) It is difficult to apply SELinux policy to Dalvik process; (3) Android lacks methods, tools, and library to interact with SELinux; (4) How to automatically update SELinux policy whenever installing or removing an application.

Currently, we successfully solve 2 of them - the first and third - and can run SELinux on Nexus One and Nexus S phone. Our future work will focus on implementing the remaining ideas.

References

[1] B. Vogel and B. Steinke. Using SELinux security enforcement in Linux-based embedded devices. 1st Int'l Conf. Mobile Wireless Middleware, Operating Systems, and Applications (MobilWare 08), Inst. for Computer Sciences, Social-Informatics and Telecomm. Eng., 2008, article 15.

[2] Android reference: what is android? Available at http://developer.android.com/guide/basics/what-is-android.ht

[3] KhanS, Banuri H. Analysis of Dalvik virtual machine and class path library. Available at: http://imsciences.edu.pk/serg/wp-content/upload/2009/07/Analysis-of-Dalvik-vm..pdf.Nov 2009

[4] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In 13th Information Security Conference (ISC), 2010.

[5] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, "Google Android: A Comprehensive Security Assessment," IEEE Security & Privacy, vol. 8,no. 2, 2010, pp. 35 - 44

[6] Security-enhanced Linux website. http://www.nsa.gov/selinux

[7] S. Smalley, FLASK: Flux Advanced Security Kernel, http://www.cs.utah.edu/projects/flux/fluke/html/flask.html, Jan. 1997

[8] R. spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security archetecture: System support for the diverse security policy. The 8th conference on USENIX Security Symposium, 1999, pp 123-139.

[9] J. Morris, "Have You Driven an SELinux Lately? An Update on the Security Enhanced Linux Project," Proc. Linux Symp., Linux Symp., 2008, pp. 101 - 113.

[10] Linux extended attribute and ACLS website, http://acl.bestbits.at/.

[11] A. Shabtai, Y. Fledel, Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. Security & Privacy, Volume 8, Issue 3, IEEE, 2010, pp. 35-44.