Security Enhanced (SE) Android: Bringing Flexible MAC to Android

Stephen Smalley and Robert Craig
Trusted Systems Research
National Security Agency
{sds,rpcraig}@tycho.nsa.gov

Abstract

The Android software stack for mobile devices defines and enforces its own security model for apps through its application-layer permissions model. However, at its foundation, Android relies upon the Linux kernel to protect the system from malicious or flawed apps and to isolate apps from one another. At present, Android leverages Linux discretionary access control (DAC) to enforce these guarantees, despite the known shortcomings of DAC. In this paper, we motivate and describe our work to bring flexible mandatory access control (MAC) to Android by enabling the effective use of Security Enhanced Linux (SELinux) for kernel-level MAC and by developing a set of middleware MAC extensions to the Android permissions model. We then demonstrate the benefits of our security enhancements for Android through a detailed analysis of how they mitigate a number of previously published exploits and vulnerabilities for Android. Finally, we evaluate the overheads imposed by our security enhancements.

1. Introduction

Android is a Linux-based open source software stack for mobile devices [4]. One of Android's design goals was to facilitate a rich and open ecosystem of applications in which critical functionality can be provided or replaced by third party applications or *apps* [40]. *Google Play*, the official app market for Android apps, is also notable for its low barrier to entry and lack of a formal review process for accepting new apps [2]. In such an environment, the need for a strong security model that is capable of addressing the threat of flawed and malicious apps is particularly evident.

Android's existing security model is implemented at two layers: an application-level permissions model (aka *Android permissions*) and a kernel-level sandboxing and isolation mechanism. The application-level permissions model, which is enforced by the Android middleware, controls access to application components, such as the ability to invoke a service provided by another application, and it controls access to system resources, such as the ability to access the camera or the network. The Android permissions model is directly exposed to Android application developers, who must specify the set of required permissions as part of their application's *manifest* file, and to end users, who must approve the set of permissions deemed *dangerous* before installing an application [1].

Underneath the user-visible Android permission model, the Linux kernel provides the foundational mechanism for application isolation and sandboxing. This mechanism normally operates invisibly to the app developers and users, so long as an app does not attempt to violate the restrictions imposed by the kernel. Application isolation and sandboxing in Android is necessarily enforced by the Linux kernel since the *Dalvik* VM is not a security boundary and any app can run native code [3]. At present, Android relies on Linux discretionary access control (DAC) to implement these controls.

In particular, Android employs DAC in two primary ways. First, Android uses DAC to restrict the use of system facilities by apps. In some cases, DAC is used to ensure that certain system resources can only be indirectly accessed by apps through system services which can then mediate access and address sharing concerns. In other cases, DAC is used to directly authorize apps to access system resources, e.g. the ability to create bluetooth or network sockets and the ability to access the filesystem on the SDcard. The socket controls required some custom kernel modifications for Android since Linux DAC does not normally control the use of these sockets.

Second, Android uses DAC to isolate apps from one another in much the same way DAC is employed in conventional Linux systems to isolate users on a multi-user system. Each app is allocated a unique user and group identifier (UID and GID, respectively) when it is installed, and this (UID, GID) pair is assigned to the processes and the private data files associated with the app [1]. This approach is designed to prevent one app from directly accessing the process state or files of another app through the kernel in-

terfaces. Apps signed by the same key may optionally be run in the same (UID, GID) pair if they wish to fully share resources without restriction.

The shortcomings of DAC, particularly with respect to protecting against flawed and malicious applications, are well established [29]. In some respects, by modifying the Linux kernel and by using DAC identities to directly represent applications rather than users, Android has mitigated some of the problems associated with DAC in its security model. Nonetheless, significant weaknesses remain, such as the ability of flawed or malicious apps to leak access to data, the coarse granularity of the DAC permissions, and the inability to confine any system daemons or setuid programs that run with the *root* or *superuser* identity.

Security Enhanced Linux (SELinux) was originally developed as a Mandatory Access Control (MAC) mechanism for Linux to demonstrate how to overcome the shortcomings of DAC [28]. Although SELinux has long been integrated into the mainline Linux kernel and is available in many Linux distributions, neither it nor any other kernel MAC mechanism is presently used by Android. Unlike DAC, SELinux enforces a system-wide security policy over all processes, objects and operations based on security labels that can encode a variety of security-relevant information about each process and object. As a MAC mechanism, SELinux is capable of confining flawed and malicious applications, even ones that run with the *root* or *superuser* identity.

SELinux offers three primary benefits for Android. First, SELinux is capable of confining the privileged Android system daemons in order to protect them from misuse and to limit the damage that can be done via them. In the absence of SELinux, any flaw in any one of these privileged daemons is fatal to the security of the device. Second, SELinux provides a stronger mechanism than DAC for isolating and sandboxing Android apps. SELinux can fully control all possible interactions among Android apps at the kernel layer, and it can control all access to system resources by the apps. Lastly, SELinux provides a centralized policy configuration that can be analyzed for potential information flows and privilege escalation paths.

In order to bring these benefits to Android and motivate the need for similar capabilities in other mobile operating systems, we initiated the Security Enhanced Android or SE Android project. The SE Android project is initially enabling the use of SELinux in Android, although it is not limited in scope to SELinux alone. SE Android also refers to the reference implementation produced by the project. The current SE Android reference implementation provides a worked example of how to enable and apply SELinux at the lower layers of the Android software stack and provides a demonstration of the value provided by SELinux in confining various root exploits and application vulnerabilities.

Our unique contributions in the work described by this paper include:

- Identifying and overcoming several challenges to enabling the effective use of SELinux in Android,
- Integrating SELinux and middleware MAC into Android in a comprehensive and coherent manner,
- Demonstrating concretely how SELinux mitigates real Android exploits and app vulnerabilities, and
- Merging our security enhancements into the mainline Android platform maintained by the Android Open Source Project (AOSP).

The remainder of this paper is as follows. We first provide a description of the challenges to enabling the effective use of SELinux in Android in Section 2. We then identify how those challenges were overcome in the SE Android reference implementation in Section 3. An analysis of how SE Android blocks a variety of previously published exploits and vulnerabilities for Android is captured in Section 4. Our results from measuring the size and performance overheads due to SE Android are reported in Section 5. Related work is discussed in Section 6.

2. Challenges

Prior to our work, the challenges to using SELinux in Android were manifold. These challenges spanned the kernel, userspace, and policy configuration. In prior efforts by others to enable the use of SELinux in Android [44, 35], many of these challenges were either completely overlooked or merely worked around rather than being fully addressed.

2.1. Kernel challenges

The first set of challenges to using SELinux in Android was in the Linux kernel. Even though SELinux is part of the standard Linux kernel, enabling the use of SELinux in Android requires more than merely enabling SELinux in the kernel build configuration. In order to provide per-file protection and to support automatic security context transitions on executables, SELinux requires that the filesystem provide support for security labeling. In Linux, the underlying storage for file security labels is typically provided through the use of extended attributes on files. However, the original preferred filesystem type for Android devices was the *yaffs2* filesystem, which is not part of the mainline Linux kernel and did not originally support extended attributes at all. More recently, *yaffs2* has gained support for extended attributes, but still lacked the necessary support for automatic security labeling of newly created files.

Furthermore, Android introduces a number of new kernel subsystems and drivers to the Linux kernel. These subsystems include new mechanisms for application processes to communicate, such as the *Binder* IPC mechanism and the *Anonymous Shared Memory* (*ashmem*) mechanism, as well as various other Android-specific facilities, such as the Android logging and *wake lock* mechanisms. These Android-unique kernel subsystems have not been previously studied or instrumented for SELinux and thus introduce the possibility of inter-app communication or privileged interfaces that are completely uncontrolled by SELinux.

2.2. Userspace challenges

The second set of challenges was in the Android userspace software stack. In the past, significant work has gone into integrating support for SELinux into conventional GNU/Linux distributions. However, in Android, almost everything above the kernel is different from a typical Linux distribution, from system-level components such as its own unique *init* program, C library (*bionic*) and core daemons up through the *Dalvik* runtime and application frameworks. As a result, none of the prior work to integrate SELinux into Linux userspace could be directly reused for Android.

Android also brings unique challenges for integrating SELinux into the userspace due to its model for starting apps. In Android, a single process, the *zygote*, preloads the *Dalvik* VM and common class files, and then upon request, forks a child process for each app, loading that app's specific classes into the child. This avoids the overhead of the *Dalvik* VM and common class initialization on each app start. However, SELinux normally performs automatic security context transitions upon program execution. Hence, the *zygote* model does not naturally lend itself to running apps in particular SELinux security contexts by default.

Android's rich support for sharing through the Android framework services also poses challenges for enabling the effective use of SELinux in Android. As much of this sharing occurs at the middleware layer and is only visible at the kernel layer as communications between each app and the *system_server* (where the framework service implementation resides) in Android, it is impossible to fully address controlled sharing among apps at the kernel layer.

2.3. Policy challenges

The third set of challenges to enabling the use of SELinux in Android was in the policy configuration. For Linux distributions, the SELinux reference policy [43, 47] provides the standard base policy from which the various distribution policies are constructed. This reference policy has been developed over many years based on the feedback and contributions of many SELinux users and developers.

However, the reference policy was developed based on the normal operation of the conventional Linux userspace and the typical ways in which Linux distributions are used. As Android has its own unique userspace software stack, and as its filesystem layout and usage model differs substantially from that of conventional Linux, the reference policy does not provide a good foundation for constructing a SELinux policy for Android. The reference policy is also quite large, and thus is not well suited to the resource constraints of small devices.

Lastly, the reference policy aims to provide comprehensive least privilege for Linux distributions. This requires that the distributions and end users often have to customize the policy for their specific environment and needs. While this is feasible for Linux developers and administrators, it would impose a significant usability challenge for typical Android users and app developers if they had to understand or write SELinux policy.

3. Implementation

This section discusses 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 userspace software stack, and the creation of a new policy configuration for Android.

3.1. Kernel support

Kernel support for SELinux in Android can be divided into two categories. Basic enablement of SELinux and filesystem support for security labels is required in order to use SELinux at all. Instrumenting Android-specific subsystems for SELinux is required in order to provide complete control.

3.1.1. Basic enablement and filesystem support. 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 [49], the audit subsystem, and filesystem support for extended attributes and security labels for each relevant filesystem for the device. Filesystems 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.

As noted earlier, the *yaffs2* filesystem used for NAND storage on Android devices did not originally support extended attributes or security labeling. Filesystem support for extended attributes was recently added to the *yaffs2*

filesystem, thereby providing storage for security labels. To provide full security labeling functionality, we implemented a fix to the *yaffs2* getxattr implementation and we implemented support for automatically setting a security label on new *yaffs2* files when they are created.

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 filesystems 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.

3.1.2. Android-specific subsystems. We began our study of the Android-specific kernel subsystems and drivers with the *Binder* subsystem as it is the central IPC primitive for Android apps. The *Binder* enables transparent invocation of objects whether local (intra-process) or remote (inter-process), and it provides lifecycle management of objects shared across multiple processes. At the kernel level, the core *Binder* functionality is implemented by a binder driver that presents a */dev/binder* interface to applications. This */dev/binder* interface can be opened by all Android apps to perform IPC transactions via specific *ioctl* commands implemented by the driver.

One process, known as the *Binder* context manager, provides name service functionality for the *Binder* and enables the bootstrapping of communications with other services. The Android *servicemanager* program registers itself as the *Binder* context manager via the */dev/binder* interface during system startup and handles initial requests by applications to obtain references to other services.

Once the *servicemanager* is operating, the various Android framework services can register object references with the *servicemanager*. These object references can then be looked up by Android apps from the *servicemanager* and used to initiate IPC to the Android framework services. The *Binder* references are kernel-managed capabilities, similar to open file descriptors. Initially, each app can only initiate IPC to the servicemanager, and it can only initiate IPC on a given *Binder* object if it receives a reference to the object from a process that already has one.

To support SELinux or any other MAC mechanism, it was necessary to define new LSM security hooks and insert calls to these hooks into the binder driver on IPC transactions and on security-relevant control operations. These hooks were then implemented for SELinux to provide SELinux permission checks over inter-app communication and over binder control operations. In particular, permission checks were implemented to control which processes can communicate with one another, to control the propagation of *Binder* references and open file descriptors via *Binder* IPC, and to control which process can serve as the *Binder* context manager.

We next looked at the *Anonymous Shared Memory* or *ashmem* subsystem. As *ashmem* regions are represented by open file descriptors and backed by regular Linux *shmem* objects, the existing support for SELinux appeared sufficient to provide basic control over read and write access to *ashmem* regions by Android apps. However, an area for further exploration of security hooks and SELinux permission checks lies in the *ashmem*-specific *ioctl* commands.

Study and possible instrumentation of the remaining Android-specific subsystems remains as future work. We expect that some of these subsystems will require their own set of security hooks and corresponding SELinux permission checks in order to fully control security-relevant operations and potential means of communications among apps.

3.2. Userspace support

Enabling userspace support for SELinux in Android required changes to a wide variety of software components, ranging from Android's C library implementation, filesystem generation tools, and *init* program up through the Android framework services. It also required porting the core SELinux userspace components to Android. In this section, we examine each aspect of enabling support for SELinux in the Android userspace software stack.

3.2.1. C library and dynamic linker support. The SELinux userspace code makes extensive use of the Linux extended attribute system calls in order to get and set file security labels. Therefore, we had to first extend Android's C library implementation, called *bionic*, with wrappers for these calls. Previously, *bionic* did not provide these wrappers as no other Android userspace code was using these calls. While our motivation for adding these calls was for SELinux, Linux extended attributes are generally useful for other purposes as well, and thus this was a general feature enhancement for Android.

We also had to modify Android's dynamic linker to recognize and use the kernel-supplied *AT_SECURE* auxiliary vector (auxv) flag for determining whether to enable secure mode. The Linux kernel provides this flag to inform userspace whether or not a security transition has occurred. Checking the flag is more reliable than checking the uid/gid against the euid/egid as was done by the Android linker prior to this change, because the flag covers not only the setuid/setgid case but also file capabilities, SELinux, and any other security module state transitions. Using the flag is also a more efficient test since it does not require the additional system calls to get the uid/gid.

3.2.2. SELinux libraries and tools. In order to leverage the SELinux facilities from the Android userspace, we had

Command	Action
seclabel	Set service security context.
restorecon	Restore file security context.
setcon	Set init security context.
setenforce	Set enforcing mode.
setsebool	Set policy boolean.

Table 1. Android init language extensions.

Command	Action
chcon	Change file security context.
getenforce	Get enforcing mode.
getsebool	Get policy boolean values.
id	Display process security context.
load_policy	Load policy file.
ls -Z	Display file security context.
ps -Z	Display process security context.
restorecon	Restore file security context.
runcon	Run program in security context.
setenforce	Set enforcing mode.
setsebool	Set policy boolean values.

Table 2. Android toolbox extensions.

to port the core SELinux userspace libraries and tools to Android. In order to minimize the userspace footprint of SE Android, we created a minimal port of the SELinux API library (called *libselinux*) for Android. As *bionic* lacks various GNU extensions which the mainstream *libselinux* presently leverages, we also had to adapt our port of *libselinux* to eliminate these dependencies.

The other SELinux libraries, namely *libsepol* and *libsemanage*, are not required on the Android device, as they are only required for creating and manipulating SELinux policy, which can be done off-line. However, we did need to adapt *libsepol* and the SELinux policy compiler, called *checkpolicy*, so that they could be built and used not only on Linux but also MacOS X, as many Android developers use the latter platform as a build host OS for Android.

Specific SELinux utilities were ported to Android on an as-needed basis. As Android's *init* program directly interprets the initialization configuration files (the *init.rc* files) rather than running a shell interpreter, support was added for several of the SELinux utilities as *init* built-in commands, as shown in Table 1. Support was also added for several of these utilities as part of the Android *toolbox*, as shown in Table 2, which supports various Linux commands via a single binary for use from a shell or by apps.

3.2.3. Labeling files. Android filesystem images for the device are generated using special purpose tools, namely *mkyaffs2image* and *make_ext4fs*, during the Android build process. These tools have Android-specific support for setting the UID, GID, and mode bits on the files within the images when they are initially generated. However, the tools had no support for setting security labels on files within the generated filesystem image. We extended both tools to support setting of file security labels based on the SELinux *file_contexts* configuration, which specifies the initial assignment of security labels for each file. As a result, the images generated during the Android build process for the system and the userdata partitions have correct security label assignments from the very beginning, and there is no need for a separate relabeling step in the SE Android installation process.

We also extended the Android recovery console and updater programs to ensure that files created from the recovery console, e.g. system updates, are correctly labeled. This was necessary to ensure that the system remains in a securely labeled state even across updates and likewise avoids the need for a separate relabeling step after updates or other changes are applied from recovery. We included a copy of the *file_contexts* configuration file within the recovery image so the updater program can determine the correct file labels, as the recovery image is the only visible mounted file system during the update process.

3.2.4. System initialization. Android has its own unique *init* program for system initialization and its own unique *ueventd* program for managing device nodes. We extended Android's *init* program to load the SELinux policy early during startup prior to executing any commands from the *init.rc* files, and we extended the *ueventd* program to label device nodes in accordance with policy. The security context of the *init* process is set from the *early-init* section of the *init.rc* configuration using the new *setcon* built-in command. Processes and files created after this initial setup can then be labeled automatically in most cases.

Most system services are automatically transitioned into their own security contexts by the kernel when they are executed from the system image by *init*. A few early services, such as *ueventd* and *adbd*, require explicit labeling via a new *seclabel* option in the *init.rc* configuration because their executables live in the root filesystem, which is merely an in-memory filesystem unpacked from the initial *ramfs* image passed to the kernel and thus does not have per-file security labels. This allows each system service to be distinguished in the SELinux policy.

The Android *init* program handles the creation and binding of local sockets for many of the system services. As these sockets should be labeled to reflect the security attributes of the individual service and not just those of the *init* process, we also extended *init* to set the security context for the sockets and the socket files. This allows connections

to each service socket to be controlled by SELinux policy.

The other new built-in commands can be used from the *init.rc* to specify the initial state for SELinux operation. In particular, the *setenforce* command can be used to put SELinux into enforcing mode and the *setsebool* command can be used to set specific SELinux policy booleans to a desired initial state. Other approaches to managing SELinux enforcing mode and policy booleans are described in Section 3.2.8.

3.2.5. App security labeling. All Android apps are created by the *zygote* process, typically at the request of the *ActivityManagerService* (AMS) running within the *system_server* process. The AMS invokes the *Process.start()* method, which sends a command string over a local socket to the *zygote*. The *zygote* then calls the *Zygote.forkAndSpecialize()* method, which in turn performs a JNI call to the native implementation within the *Dalvik* VM. The *Dalvik* VM then forks a child process and sets the DAC credentials (UID, GID, supplementary groups) for the child process to reflect the credentials of the app.

Even the Android *system_server*, which provides the Android framework services for the apps and runs in the *system* UID with a number of Linux superuser capabilities, is created by the *zygote*. This is done using a variant interface, *forkSystemServer*, which is also implemented by the *Dalvik* runtime and internally uses the same function, *forkAndSpecializeCommon*, within the *Dalvik* VM.

To enable the *system_server* and the Android apps to be labeled differently from the *zygote* process, we had to extend the *Dalvik* VM to also set the SELinux security context for the child process. To support this functionality, we inserted a hook within the *Dalvik* VM to call a new interface in the *libselinux* library at the right point during setup of the new child process. The setting of the SELinux security context for the child must occur after the setting of the DAC credentials as the new SELinux security context may not be allowed to change DAC credentials at all, and it must occur before any other threads or objects are created by the child to ensure proper labeling of all associated threads and objects.

In order to support a greater range of inputs to select a SELinux security context for app processes, we extended the relevant interfaces and their callers to take an additional *seinfo* string argument that can be used to pass higher-level information from the AMS about the particular app being started. These changes spanned the *Dalvik* VM, the *zygote*, and the AMS in order to convey the additional argument all the way through the call chain for spawning an app process. We also extended the relevant interfaces and their callers to provide the name of the application package as one of the inputs.

This mechanism allows SELinux to distinguish the *system_server* process from all apps, even apps running in the *system* UID. It also allows SELinux to distinguish apps from one another based on their platform UID, package name, or other higher-level information provided by the AMS of the *system_server*. The *seinfo* string for the app is determined from the middleware MAC policy configuration, described in Section 3.3.2.

In addition to setting the security label of each app process, we also needed to set the security label of each app data directory. The creation of the app data directories is performed by the *installd* daemon in Android, which receives commands over a local socket from the *PackageManagerService* running in the *system_server*. As with the *Dalvik* VM, we inserted hooks into *installd* to call a new interface in the *libselinux* library when the app data directories are being created to set the directory security context correctly.

The new functions in *libselinux* that implement these hooks for both *Dalvik* VM and *installd* are driven by a single shared configuration file, the *seapp_contexts* configuration. This configuration was introduced specifically for SE Android but is similar to other SELinux policy configuration files. The configuration allows specification of how to label app processes and data directories based on the available inputs.

3.2.6. Userspace policy enforcement. SELinux provides support to allow seamless extension of the Mandatory Access Control model to application layer objects and operations. The *libselinux* library provides interfaces for use by applications to obtain security contexts for their own objects and to apply SELinux permission checks on operations performed on these objects. These userspace components are typically referred to as *userspace object managers* in the terminology of the Flask architecture on which SELinux is based [46]. This approach has been previously applied to various Linux applications such as the D-Bus message bus [42], the X Window System [48], the PostgreSQL DBMS [20], and the GConf [8] configuration system.

In Android, there are number of applications that implement their own permission checking logic, typically based on the DAC UID of the requesting process. Two such components include the *init* program, which provides a global property name/value space for all Android processes, and the *zygote*, which provides the interface for spawning new apps.

The *init* program was extended to apply security labeling for the system properties and to enforce mandatory access controls over attempts to set their values. Fine-grained control over reading the system properties is not presently possible as they are stored in a single shared mapping that is mapped read-only by all processes in Android. By con-

trolling the ability to set the system properties, SELinux can prevent a compromise of one of the privileged system services from being leveraged to set any arbitrary property. SELinux can also support finer-grained distinctions over the ability to set properties than the existing DAC controls.

The *zygote* was likewise extended to enforce mandatory access controls over the use of its socket interface for spawning new apps. While the kernel can directly control what processes can connect to the *zygote* via this socket interface, finer-grained distinctions over the use of privileged commands issued over the socket require permission checking by the *zygote* itself. In order to enable the use of the SELinux APIs from the *zygote* Java code and from the Android framework services written in Java, we created a set of SELinux JNI bindings for a subset of the *libselinux* interfaces. The *ZygoteConnection* class was then extended to use these APIs to obtain the security context of the client process and apply SELinux permission checks for any privileged operation, such as specifying the UID and GID for the new app being spawned. As with the property MAC controls, these controls enable SELinux to prevent a compromise of one of the privileged system services from being leveraged to spawn apps with arbitrary credentials or resource limits.

3.2.7. Middleware policy enforcement. Although the SELinux userspace object manager approach worked well for the *init* property service and the *zygote*, it proved far more problematic at the Android middleware layer. First, apps communicate with the Android middleware via *Binder* IPC rather than socket IPC. Thus, the lack of support for obtaining the sender security context for binder IPC was an initial obstacle. We overcame this problem by implementing support for passing the sender security context on binder transactions.

However, we then discovered that binder transactions often involve multi-stage call chains that require saving and restoring caller identity for permission checking purposes. For example, content providers are accessed indirectly via the AMS, which saves the caller identity in thread-local storage before invoking the content provider, so that it can look up the original caller identity when the content provider later queries the AMS for a permission check. Providing similar support for saving and restoring the sender security context would have required an invasive and potentially costly set of changes.

Further, Android permission checks are often invoked by application components using the public *checkPermission* API, which only supports passing the sender PID and UID. This would have required that we extend the public *checkPermission* API in order to fully support permission checking based on SELinux security contexts, thereby creating compatibility problems for existing apps and impacting Android app developers. We were further concerned about the potential implications of using the SELinux policy to capture middleware MAC semantics on our goals for keeping the SELinux policy small, simple, and relatively static.

As a result of these considerations, we chose to introduce a separate middleware MAC (MMAC) layer for Android. The MMAC layer should only interact with the kernel MAC layer with respect to determining the *seinfo* string used for app security labeling, as described in Section 3.2.5. Otherwise, the two layers should largely function independently, with the kernel MAC layer enforcing a small set of fixed security goals based on the assigned security contexts. As this design decision removed the need to pass sender security context information on *Binder* IPC, we reverted the corresponding code changes from our reference implementation.

We have developed several MMAC mechanisms for Android to explore the design and implementation space. These MMAC mechanisms provide different forms of mandatory restrictions over the Android permissions model. One of these mechanisms, known as *install-time MAC*, has been integrated as part of the core SE Android implementation as it provides the basis for determining *seinfo* strings for apps and is the most mature mechanism.

Our install-time MAC mechanism applies install-time checks of the permissions requested by an app, or implicitly granted by the system, against a MAC policy configuration. This mechanism allows a policy-driven approach to authorizing app installation that preserves the all-or-nothing contract that Android presently offers to apps. Install-time MAC differs from Android's existing permission model in that it ensures that policy has final determination when installing apps rather than the user. This approach enables enforcement of organizational limits on the maximum permissions for apps. The policy is expressed in a new *mac_permissions.xml* configuration file described in Section 3.3.2, and is enforced by extensions to the Android *PackageManagerService* (PMS).

Integration of install-time MAC into the Android PMS ensures that the policy checks are unbypassable and always applied when apps are installed and when they are loaded during system startup. As a result, restrictions can be applied even to pre-installed system apps via this mechanism, in which case the app is completely disabled from running if it is not authorized by the policy. For conventional third party app installs, installation of the app is aborted if the policy denies one of the requested permissions.

3.2.8. Runtime policy management. SE Android policy is integrated into the Android build process and included in the ramdisk image placed within the boot partition so that it can be loaded by *init* very early in boot, before starting any other processes. This approach ensures that all subse-

quent processes and files are labeled properly when created and that policy is enforced as early as possible. However, this approach does not directly allow runtime changes to the policy without updating the entire boot image.

In order to support basic SELinux management functionality, we first developed SELinux JNI bindings to support setting of the SELinux enforcing mode and policy booleans by the *system_server* or *system* UID apps, which must be signed by the platform certificate. We developed a *SEAndroidManager* app to permit user management of these settings. This mechanism however does not support modifying policy beyond setting policy booleans.

In order to support runtime policy management, we added support for reloading any of the policy files from the standard Android */data/system* directory that is already used for various runtime system configuration data. This directory is only writable by the *system* UID and thus these configuration files can only be updated by the Android *system_server* or *system* UID apps. Once the updated policy files have been written to */data/system*, a policy reload is initiated by setting a new Android system property, the *selinux.reload_policy* property, which also can only be set by *system* UID processes. The ability to write these files and set this property can be further restricted using SE Android policy.

Each time the property is set, the *init* process reloads the kernel policy as well as any other policy configuration files it uses, e.g. the *file_contexts* and *property_contexts* configurations. The *init* process then executes any triggers defined for the *selinux.reload_policy* property in the *init.rc* configuration. We added a trigger to the *init.rc* configuration to restart the *ueventd* and *installd* daemons so that they also reload the policy configuration files relevant to their operation. We confirmed that restarting of these daemons at runtime does not cause any problems for Android. An alternative approach would have been to notify the daemons of the policy reload via their existing socket interfaces and allow them to reload policy without restarting.

With this support in place, we created device admin APIs to allow management of the SELinux settings and provisioning of alternate policy configurations via a device admin app. We also created a sample device admin app, *SEAndroidAdmin*, to demonstrate these APIs. The code from this sample app could be leveraged by MDM vendors as a starting point for integrating support for SE Android management.

3.3. Policy configuration

Policy configuration for SE Android can be divided into two categories. The kernel layer MAC mechanism is governed by the SELinux policy configuration, which includes both the kernel policy and various configuration files leveraged by userspace to look up SELinux security contexts. The middleware MAC mechanisms are governed by their own configurations. This section describes each configuration for SE Android.

3.3.1. SELinux policy configuration. Our goal for SE Android was to apply SELinux to enforce a small set of platform security goals in a manner that would avoid any user-visible or app developer-visible changes. Recognizing that Android was unlikely to discard its existing DAC model altogether, we focused on using SELinux to reinforce the DAC model and to address the gaps left by the DAC model. Primarily, we wanted a policy that would confine the privileged daemons in Android, ensure that the Android middleware components could not be bypassed, and strongly isolate apps from each other and from the system.

As explained in Section 2.3, the SELinux reference policy was not suitable as a starting point for Android SELinux policy configuration. Instead, we created a small policy from scratch tailored to Android's userspace software stack and to our security goals. The Type Enforcement (TE) portion of the policy was configured to define confined domains for the system daemons and apps. The Multi-Level Security (MLS) portion of the policy was configured to isolate app processes and files from each other based on MLS categories. This approach yielded a small, fixed policy at the kernel layer with no requirement for policy writing by Android app developers. As with the existing DAC model, the kernel layer MAC is normally invisible to users and to apps, only manifesting if an app violates one of the security goals.

	SE Android	**Fedora**
Size	71K	4828K
Domains	39	702
Types	182	3197
Allows	1251	96010
Transitions	65	14963
Unconfined	3	61

Table 3. Policy size and complexity.

The SE Android policy is significantly smaller and simpler than the SELinux policies used in conventional Linux distributions. Table 3 shows some statistics for the SE Android policy compared to the Fedora SELinux policy. The SE Android policy is notably smaller in terms of the binary policy size, the number of domains (subjects), the number of types (objects), the number of allow rules, and the number of type transitions. The SE Android policy also differs in that it defines very few unconfined domains (i.e. domains that are allowed all permissions); in particular, only the domains for kernel threads, the *init* process, and the *su* program (which is only included in debug builds) are uncon-

fined. No app domains are unconfined in SE Android. The source files for the SE Android policy can be found within the *external/sepolicy* directory of the Android Open Source Project (AOSP) master branch.

Two new configuration files were added to the Android SELinux policy configuration for use by applications. The *property_contexts* configuration is used to specify the security context of Android system properties. This configuration is used by the *init* property service as described in Section 3.2.6. The *seapp_contexts* configuration is used to label app processes and app package directories. This configuration is used by the *Dalvik* VM and by *installd* as described in Section 3.2.5.

3.3.2. Middleware policy. The *mac_permissions.xml* configuration for the install-time MAC mechanism is written in XML format and follows the conventions of other Android system configuration files. Recognizing that managing access control policies for potentially hundreds of apps on a per-app basis was infeasible, we provided a scalable policy that does not require a new policy for each app. In order to express app-permission authorizations without having to specify per-app rules, we devised a method to specify X.509 certificates as part of our policy. Android already requires each installed app to be signed with such a certificate. We leveraged this existing attribute of Android apps by identifying groups of apps in our configuration by their certificate.

The *mac_permissions.xml* configuration uses the AOSP signing keys to organize apps into equivalence classes and to allow or deny, based on whitelist/blacklist logic, an appropriate permission set. Each entry in the configuration can also contain a *seinfo* tag to specify the seinfo string used for app security labeling. Individual apps can be specified when appropriate by package name in addition to specifying their certificate. As enumerating all possible third party app certificates is infeasible, we provide a *default* tag that is used to match any apps that are not otherwise specified by the configuration.

4. Analysis

This section surveys a set of previously published exploits and vulnerabilities for Android and describes the results of analysis and testing performed to assess the effectiveness of SE Android in addressing the threats of flawed and malicious apps. It then provides a general discussion of the threats that can and cannot be mitigated by SE Android. The analysis and testing was performed using the initial SE Android policy configuration developed before reading about any of these specific exploits or vulnerabilities in Android. The policy configuration was developed based on normal Android operation and SELinux policy development practices.

4.1. Root exploits

The first class of exploits and vulnerabilities that was evaluated were Android *root exploits*. These exploits escalate privilege from an unprivileged app or user shell to gain full *root* access to the device, enabling the exploit to then perform arbitrary actions on the device. Often these root exploits are developed by the Android *modding* community for the purpose of enabling them to modify their own devices for customization and optimization. However, these exploits can also be leveraged by malware to gain complete control of a user's device.

4.1.1. GingerBreak and Exploid. The Android volume daemon or *vold* is a system service that runs as *root* on Android and manages the mounting of the SDcard (and in Android 4.0 and later, the mounting of the encrypted storage). In order to support this functionality, *vold* listens for messages on a *netlink* socket in order receive notifications from the kernel.

CVE-2011-1823 identifies a vulnerability in *vold*'s handling of the netlink messages [33]. First, *vold* did not verify that the netlink messages originated from the kernel, and it was possible for unprivileged applications to generate messages on these sockets. Second, *vold* used a signed integer from the message as an array index, checking only for an upper bound but not whether the integer was non-negative.

The *GingerBreak* exploit demonstrated how to exploit this vulnerability in order to gain root access from a user shell [23]. The exploit has also been packaged as an Android app and has shown up in the wild in Android malware in the *GingerMaster* malware [19].

The *GingerBreak* exploit code is able to dynamically survey the device in order to find all of the information it requires to mount a successful attack on *vold*. It obtains the PID of the *vold* process from the *proc* filesystem, and it obtains information required to craft the malicious payload by reading several files from the system partition. It also makes use of access to the Android logging facility (*logcat*) in order to dynamically observe the effect of its attacks on *vold* and refine its attack incrementally until successful.

Once *GingerBreak* has found the *vold* process and crafted the malicious payload, it sends the payload to *vold* via a netlink socket. The payload triggers execution of the exploit binary by *vold* and the exploit code is then running with full root privileges. The exploit code then proceeds to create a setuid-root shell, and the original *GingerBreak* process executes this setuid-root shell to give the user a root shell.

We performed an analysis and testing of the impact of SE Android on *GingerBreak*. During the information collection stage of *GingerBreak*, the policy denied the exploit's attempt to read the proc information for the *vold* process,

and it denied the exploit's attempt to read the *vold* binary to discover the target address range. This caused the exploit to fail immediately. However, we assumed for the sake of further analysis that the exploit writer could have rewritten the exploit based on prior knowledge of the target, and allowed the corresponding permissions for the sake of further testing.

Next, the policy denied *GingerBreak*'s attempt to create the netlink socket, as there is no legitimate need for user shells or apps to create this type of socket. Thus, the exploit could not even reach the vulnerable code in *vold*. This provided unbypassable protection for the vulnerability; however, we allowed the necessary permissions and continued our analysis.

GingerBreak was then able to send the malicious payload to *vold*. However, this merely triggered an attempt to execute the exploit binary from *vold*, which was also denied by policy. There was no legitimate case where *vold* executed non-system binaries, so the attempt to execute a binary from the data partition was not allowed by the policy we had developed. The exploit again failed, but we assumed for the sake of further analysis that the exploit writer could have rewritten the exploit to avoid executing a separate binary from the filesystem and simply allowed this permission.

Once running as root, the exploit code then attempts to create a setuid-root shell. SE Android denied the attempt to set the ownership and mode of the exploit file (or any file writable by the exploit). We allowed these permissions for the sake of further testing.

In its final act, assuming all of the previous denials were allowed, *GingerBreak* then executed the setuid-root shell. This did provide the user with a uid 0 shell, but the SELinux security context remained the same, and the shell was still limited to the same set of permissions with no superuser privileges allowed by SELinux.

In summary, SE Android would have blocked the exploit at many points in its execution and would have forced the exploit writer to tailor the exploit to the target rather than being able to survey the target device at runtime for all of the necessary information. SE Android also would have made the underlying vulnerability in *vold* completely unreachable from an app or an unprivileged user shell.

A similar vulnerability was discovered in Android's *ueventd* daemon. The vulnerability was the same flaw reported as CVE-2009-1185 [30] for the Linux *udev* implementation, simply replicated in Android's *ueventd* implementation. The *Exploid* exploit demonstrated how to exploit this vulnerability in order to gain root access from a user shell [22]. In our analysis and testing of *Exploid* on SE Android, we found that *Exploid* would have been completely blocked in at least two ways by SE Android. SE Android would have blocked not only these two specific vulnerabilities, but all vulnerabilities that fall into the same class, i.e. vulnerabilities in netlink socket message handling in privileged daemons.

4.1.2. Zimperlich and RageAgainstTheCage. The Android *zygote* is a system service that runs as root and is responsible for spawning all Android apps. The *zygote* receives requests to spawn apps over a local socket. The *zygote* forks a child process for each app and the child process uses setuid() to switch to the unprivileged UID for the app before executing any app code. The particular code implementing this logic lives in the *Dalvik* VM.

However, the *Dalvik* VM did not check for failure on the setuid() call, as this call normally does not fail for root processes, and therefore did not abort the process on a failed setuid(). The *Zimperlich* [24] exploit demonstrated how to exploit this vulnerability in order to gain root access from an Android app. It achieves this access by inducing a failure in the setuid() call through a subtle interaction with Linux resource limits.

First, the exploit code forks itself repeatedly in order to reach the maximum number of processes allowed per uid (*RLIMIT_NPROC*) for the app UID. It then issues a request to the *zygote* over the local socket to spawn one of its components in a new process. When the *zygote* forks a child process and attempts to set the app UID, the setuid() call fails due to reaching the resource limit. As the *Dalvik* VM did not abort in this situation, execution proceeds and the malicious app's code is run in the same UID as the *zygote*, i.e. as a root process. The *Zimperlich* exploit then proceeds to re-mount the system partition read-write and creates a setuid-root shell in the system partition for later use in obtaining root access at any time.

In our analysis and testing of *Zimperlich* on SE Android, we found that although the malicious app succeeds in running with the root UID, the SELinux security context is still correctly set by the *Dalvik* VM based on the app's credentials, as this operation is not subject to a resource limit. Consequently, the app runs in an unprivileged security context with no superuser capabilities and is unable to re-mount the system partition or perform other privileged actions.

A similar vulnerability was discovered in the Android debug bridge daemon or *adbd*. The *RageAgainstTheCage* [25] exploit demonstrated how to exploit this vulnerability. When tested on SE Android, the user shell created by *adbd* stills runs with the root UID but transitions to an unprivileged security context automatically based on SELinux policy. As a result, the user shell is not allowed any superuser capabilities and remains confined.

The subtle interaction of setuid() and *RLIMIT_NPROC* in Linux has been the source of similar bugs in various root daemons and setuid-root programs in conventional Linux distributions and has led to some recent changes in the

Linux kernel [12]. As a result, recent Linux kernels defer the resource limit failure until a subsequent call to execve(), such that the setuid() always succeeds.

4.1.3. KillingInTheNameOf and psneuter. The *Anonymous Shared Memory* or *ashmem* subsystem is an Android-specific kernel subsystem used to provide a shared memory abstraction for inter-app data sharing. It was also originally used to implement the global system property space managed by the Android *init* process. A read-only mapping of this global property space is mapped into every process on the system and used to read the property values at any time. CVE-2011-1149 identifies multiple vulnerabilities in the *ashmem* implementation with respect to the memory protection settings of the system property space [31]. These vulnerabilities are demonstrated by the *KillingInTheNameOf* and *psneuter* exploits [21].

The *KillingInTheNameOf* exploit invokes the mprotect() system call to add write access to its mapping of the system property space and is then free to modify any system property at will via direct memory write. The exploit makes use of this ability to modify the *ro.secure* property value so that it can obtain a root shell via *adbd* upon its next restart. When tested on SE Android, the *KillingInTheNameOf* exploit is denied the attempt to add write access to the mapping because the policy does not allow write access to the memory mapping owned by the *init* process.

The *psneuter* exploit makes use of a different vulnerability in the *ashmem* implementation. It uses an *ashmem*-specific *ioctl* command (*ASHMEM_SET_PROT_MASK*) to set a protection mask on the system property space to zero. Upon the next re-start of *adbd* (or any other process), the mask is applied on its attempt to map the system property space and thus the mapping fails. This leads to the *ro.secure* system property being treated as zero since it cannot be read, which again provides a root shell via *adb*. SE Android does not prevent this exploit from modifying the protection mask, although the shell still transitions to an unprivileged security context and is therefore confined by SELinux.

The *psneuter* exploit suggests that SELinux instrumentation of *ashmem ioctl* commands may be worth exploring to provide better control of the security-relevant operations. However, modern versions of Android have worked around these issues by changing the *ashmem* implementation and by switching the implementation of the *init* property space from using *ashmem* to using a conventional Linux *tmpfs* file.

4.1.4. Mempodroid. The Linux *proc* pseudo file system provides an interface for accessing various global system state and for accessing per-process information. The */proc/pid/mem* file provides a kernel interface for accessing the memory of the process with the specified PID. In the past, due to security concerns, this interface has been restricted to read-only access to the memory of another process, even for processes with the same DAC credentials. In Linux 2.6.39, support for write access to */proc/pid/mem* files was enabled because it was believed that the prior security concerns had been adequately addressed.

CVE-2012-0056 [34] identifies a vulnerability in the Linux kernel permission checking for */proc/pid/mem* that can be used to induce a setuid-root program into writing its own memory. This vulnerability was demonstrated for conventional Linux distributions via the *mempodipper* exploit [11] and for Android by the *mempodroid* exploit [16]. *mempodroid* makes use of the Android *run-as* program as the target setuid-root program. It invokes the setuid-root program with an open file descriptor to */proc/pid/mem* as its standard error (stderr) stream, and passes the shellcode as an argument string. The *run-as* program proceeds to overwrite its own executable with the supplied shellcode, which sets the UID/GID to 0 and executes a shell or command string with full privileges.

On SE Android, we first tested the exploit without defining any specific policy for the *run-as* program. In this situation, the exploit will succeed in overwriting the memory of the *run-as* process due to the kernel vulnerability and can therefore execute the exploit payload with the root UID. However, the security context of the *run-as* process remains the same as its caller, and thus no privilege escalation occurs. The exploit remains confined by SELinux and cannot exercise any superuser privileges or any other permissions not allowed to the original caller.

However, under these restrictions, the *run-as* program cannot perform its legitimate function for Android, i.e. enabling app developers to debug their own apps on production devices. To support this functionality, we defined policy for the *run-as* program and we modified the *run-as* program to switch to the correct app security context before running the specified command or shell. With these changes, the exploit fails to overwrite the memory of the *run-as* process due to the SELinux file checks on the */proc/pid/mem* access. This protection blocks the exploit completely.

Even if we allowed this access in the policy, the exploit would be limited to the permissions allowed to the *run-as* program by the policy rather than having arbitrary root access. The SE Android policy for the *run-as* program only grants it the specific superuser capabilities required for its function, namely *CAP_DAC_READ_SEARCH*, *CAP_SETUID*, and *CAP_SETGID*. The exploit is unable to exercise privileged operations requiring any of the other superuser capabilities defined by Linux, such as performing raw I/O to devices, loading kernel modules, or remounting partitions. Further, the policy does not allow *run-as* to execute any programs without first changing to an app security context, and thus the exploit cannot execute a command or shell without first shedding even these capabilities.

4.2. Application vulnerabilities

The second class of vulnerabilities that was evaluated were Android app vulnerabilities. These are vulnerabilities in legitimate Android apps that allow data to be leaked or modified without being authorized by the user in any way. These vulnerabilities can then be leveraged by malware in order to gain access to sensitive user data or to modify security-relevant settings.

4.2.1. Skype. The *Skype* app for Android provides VOIP functionality via the *Skype* service. CVE-2011-1717 identifies a vulnerability in the *Skype* app in which the app stores sensitive user data without encryption in files within its data directory that are world readable and writable [32]. The information included the user's account balance, date of birth, home address, contacts, chat logs, etc. As a result, any other app on the device could potentially read the data, and if allowed *INTERNET* permission, could leak the data remotely. Other apps on the device could also maliciously tamper with the files.

This vulnerability provides a classic example of the difference between DAC and MAC. Under DAC, file permissions are left to the discretion of each application and thus are subject to intentional or accidental misconfiguration. Under MAC, the policy is defined by the policy writer and enforced on all applications running on the system. The SE Android policy was configured to ensure that no app can read or write files created by another app by assigning each app and its files a unique MLS category. Thus, data isolation of app data files is not dependent on the correctness of the apps. As a result, on SE Android, although the *Skype* data files are still created with weak DAC file permissions, SE Android prevents any malicious app from reading or writing the files.

An obvious concern with this approach is that it does not allow for intentional app data sharing via files. However, in general, Android's own model gives preference to app data sharing via Binder IPC rather than direct file access. Thus, SE Android offers a way to enforce Android's own preferred system structure. Second, if two apps should have fully shared access to data files, they can declare a shared user id (as long as they are signed with the same certificate), and in this case, SE Android also will label them with the same category, enabling such sharing. Third, the assignment of MLS categories is configurable as part of the *seapp_contexts* configuration, so if direct sharing by file is required among apps with different UIDs, SE Android can be configured to place particular sets of apps within the same category even if they do not share the same UID and thus allow various sharing relationships.

4.2.2. Lookout Mobile Security. The *Lookout Mobile Security* app for Android provides security, backup, lost device tracking, and management functionality for Android devices [27]. A vulnerability in this app was discovered where the app created configuration and database files via native calls without setting the umask for the process, leading to the files being world-readable and world-writable [26]. As a result, any app running on the device could disable or reconfigure the *Lookout* app or could cause the app to execute arbitrary code.

As in the *Skype* example, this example highlights the difference between DAC and MAC permissions, where a subtle flaw in the application (in this case, the combination of using native calls for file creation combined with the failure to set the umask) can subvert the DAC protections altogether. The SE Android policy would have prevented any other app from reading or writing the private data files of the *Lookout* app regardless of such application flaws.

4.2.3. Opera Mobile. The *Opera Mobile* app for Android is a version of the Opera web browser built for the Android platform [41]. A vulnerability in this app was discovered where the app created its cache files world-readable and world-writable [17]. As a result, any app on the device could both read and write to the browser's cache, potentially leaking sensitive user information and altering data or code (e.g. JavaScript).

As with the prior vulnerabilities, this vulnerability stems from the dependency of DAC on application correctness. The SE Android policy would have ensured that no other app on the device could read or write the cache files of the browser, thereby preventing exploitation of this vulnerability.

4.3. General analysis

In the preceding sections, we described our analysis and testing of the impact of SE Android on specific Android exploits and vulnerabilities. That analysis and testing demonstrated concretely that SE Android can mitigate real root exploits and app vulnerabilities for Android. In this section, we provide a more general discussion of what threats SE Android can and cannot mitigate.

SE Android's kernel layer MAC provides an effective means of preventing privilege escalation by apps and of preventing unauthorized data sharing by apps via the kernel level interfaces. It also provides a foundation for ensuring that higher level security functionality is unbypassable and protected against tampering by apps. For example, SE Android can rigorously ensure that hardware devices can only be accessed by the authorized system services and not directly by apps, so that the system services can then enforce the higher level Android permissions model. SE Android also provides a way of protecting the integrity of apps and their data. These same protections are provided by SELinux for conventional Linux systems.

Similarly, the install-time MAC mechanism of SE Android ensures that apps can only be installed if their requested permissions are authorized by the middleware policy. This mechanism can help protect users and organizations from installing untrustworthy apps with dangerous sets of permissions, and can even be used to disable preinstalled apps with dangerous permissions. However, this mechanism cannot address privilege escalation attacks or unauthorized data sharing at the middleware layer by apps that have been allowed to be installed. Addressing such runtime threats will require further MMAC mechanisms beyond install-time MAC. We are presently exploring such runtime MMAC mechanisms.

There are a number of threats that SE Android cannot directly address. First, SE Android cannot mitigate anything allowed by the policy. As such, developing good policy is crucial to the effectiveness of SE Android, and the ability to have a good policy while still having a functional system is dependent on the software system architecture. Fortunately, Android already makes good use of process isolation in its existing architecture, thereby enabling the effective application of SE Android.

Second, as a kernel level mechanism, SE Android cannot in general mitigate kernel vulnerabilities. In some cases, as shown in the *KillingInTheNameOf* and *mempodroid* case studies, SE Android prevents a kernel vulnerability from being exploitable by making the vulnerable code unreachable by untrusted applications or by rendering the impact of the vulnerability inconsequential. However, this is not true of all or even most kernel vulnerabilities. Thus, other mechanisms for protecting and measuring kernel integrity are desirable in combination with SE Android.

Lastly, SE Android cannot address threats from other platform components, particularly ones that may have direct access to system resources such as memory and storage. For example, SE Android cannot protect against actions by a compromised baseband processor or network card. Such threats must be addressed through other, hardware-facilitated mechanisms.

In spite of these limitations, we have shown that Android security would benefit from some form of MAC in general, and that SE Android in particular would mitigate many of the exploits and vulnerabilities facing Android today. Mitigating the threat posed by flawed and malicious apps is an important piece of an overall security architecture for mobile devices.

5. Overhead

This section describes our results from measuring the size and performance overheads introduced by SE Android compared to a pristine build of the corresponding AOSP version. The AOSP images were built from the *android-4.2_r1* tag for the Galaxy Nexus (*maguro*) device, using the prebuilt kernel supplied by AOSP for that device. These images were measured to provide the baseline for each set of results. The SE Android images were built from the *seandroid-4.2* branch of the SE Android source code repository for the same device, using a kernel built from the same kernel source tree but with the SE Android modifications and with SELinux enabled. Both the AOSP and SE Android images included the same set of additional apps used for benchmarking. The results for SE Android images can be compared against the AOSP results to determine the overhead introduced by SE Android.

	AOSP	**SE Android**	**Increase**
boot.img	4400K	4552K	+152K
system.img	194072K	194208K	+136K
recovery.img	4900K	5068K	+168K

Table 4. Image sizes for full_maguro-userdebug (4.2).

5.1. Size

Given the limited resources of mobile devices, a goal of SE Android was to keep the number and size of changes to a minimum. Table 4 shows the absolute sizes of the boot, system, and recovery images for the AOSP and SE Android builds and it shows the relative size increase of the SE Android images. The data reveals relatively small size increases for the three images. The userdata image was unchanged in size and is therefore not shown.

The increase in boot image size is primarily due to the increase in kernel size for SE Android (+100K). The SE Android kernel enables filesystem support for extended attributes and security labels as well as the kernel audit subsystem, the Linux Security Module (LSM) framework, and the SELinux security module. The remaining increase in size for the boot image comes from the SE Android policy files and extensions to the *init* program.

The system image increased in size largely from three new components introduced by SE Android: the *libselinux* library (+44K), the *SEAndroidManager* app (+40K) and the *mac_permissions.xml* file (+24K). The Android *toolbox* program and the *libandroid_runtime* library also increased slightly in size (+4K each) due to the SE Android extensions. Since the system image contains the core Android OS, SE Android's small relative increase in size (+.07%) speaks to its small footprint. Further, the *SEAndroidManager* app is not required for SE Android operation and thus could be omitted from the final image for the device if desired.

	AOSP		SE Android	
	Mean	**SD**	**Mean**	**SD**
total score	4172.68	148.83	4165.31	188.28
memory	507.05	51.81	514.27	65.42
integer	838.89	57.61	842.95	65.83
float	672.25	61.48	673.68	72.21
score2d	279.85	36.22	273.23	45.52
score3d	1230.67	0.86	1230.46	1.02
sdread	191.110	0.662	191.010	0.748
sdwrite	115.45	5.61	115.15	4.74
database	337.40	22.85	324.55	19.86

Table 5. AnTuTu comparative benchmarking (n = 200) full_maguro-userdebug (4.2).

The increase in size for the recovery image is similar to the boot image. Like the boot image, the recovery image includes the kernel and a minimal root filesystem, which in the SE Android case includes the SE Android kernel extensions and the policy files. The recovery image further includes the recovery console and updater programs that were extended for SE Android to ensure proper security labeling of files upon updates.

5.2. Performance

In order to be acceptable in mainline Android, SE Android must not impose significant runtime performance overhead. To measure the runtime performance overhead of SE Android, we ran two well known benchmark apps found in the Google Play Store: AnTuTu [5] by AnTuTu Labs and Benchmark by Softweg [45]. Each benchmark was executed on both the AOSP and SE Android builds on the same device. We ran a large number of trials to sharpen the distribution. Under both benchmarks we ensured that the same number of apps/services were loaded and running at any time.

5.2.1. AnTuTu performance test. The results for 200 runs of the AnTuTu benchmark on both AOSP and SE Android are shown in Table 5. The memory, integer, and float tests should be unaffected by SE Android as they do not involve any system calls. The score3d and score2d tests are measurements of graphical performance and should likewise be unaffected by SE Android. Both tests measure the frames per second (fps) for various images and graphics. The sdwrite and sdread tests perform writes and reads of the SDcard storage, measuring the data transfer rate. The database I/O test exercises the Android SQLite database functionality. For these tests, we expect some small overhead from SE Android due to the need to create and fetch the extended attributes for file security labeling and due to the additional permission checking performed by SE Android. For most of the tests the SE Android result shows negligible overhead and is within one standard deviation of the AOSP result.

	AOSP		SE Android	
	Mean	**SD**	**Mean**	**SD**
Total memory	588.88	68.61	591.71	67.28
Copy memory	535.11	62.35	537.68	61.13
Total CPU	3167.07	149.51	3113.31	138.51
MFLOPS DP	17.61	1.09	17.46	0.98
MFLOPS SP	41.85	5.06	41.22	5.20
MWIPS DP	200.86	8.83	197.16	10.10
MWIPS SP	289.18	19.04	283.73	14.93
VAX MIPS DP	139.03	6.19	136.73	6.62
VAX MIPS SP	191.31	16.08	187.69	15.12
Graphics Scores				
Total score	19.50	0.37	19.62	0.38
Opacity	6.32	0.16	6.37	0.18
Transparent	5.58	0.13	5.62	0.11
Filesystem Scores				
Total Score	236.99	20.88	234.77	20.05
Create files	0.38	0.02	0.44	0.03
Delete files	0.23	0.11	0.25	0.12
Read file	382.54	38.72	375.25	37.58
Write file	100.20	7.90	96.88	7.57
SDcard Scores				
Create files	1.45	0.15	1.62	0.17
Delete files	0.46	0.06	0.49	0.06
Read file	64.73	5.33	63.46	5.21
Write file	33.78	2.54	33.65	2.89

Table 6. Softweg comparative benchmarking (n = 200) full_maguro-userdebug (4.2).

5.2.2. Softweg performance test. The results for 200 runs of the Benchmark by Softweg benchmark on both AOSP and SE Android are shown in Table 6. The memory and CPU scores should be unaffected by SE Android as they do not involve any system calls. The graphics scores should likewise be unaffected by SE Android, measuring the MPixels per sec for transparent and opaque image overlays.

For most of the tests, the SE Android result shows negligible overhead and is within one standard deviation of the AOSP result. As with AnTuTu, for the filesystem and SDcard tests, we expect some small overhead from SE Android due to the need to create and fetch the extended attributes for file security labeling and due to the additional permission checking performed by SE Android. The filesystem write and read tests measured the speed (M/sec) of writing and reading 1M. The create and delete tests were a measure of the time (seconds) it took to create or delete 1000 empty files. These create and delete tests can be viewed as a worst case overhead for SE Android since the overhead of extended attribute creation and removal is not amortized over any real usage of the file.

6. Related work

There has been an extensive body of research into extensions to the Android access control model [14, 39, 13, 36, 38, 7, 18, 15, 10, 6]. Most of these extensions, including Kirin [14], SAINT [39], TaintDroid [13], Porscha [38], AppFence [18], IPC Inspection [15], and QUIRE [10] only attempt to address access control at the Android middleware layer and provide no solution for the underlying weaknesses of the Linux DAC mechanism. As any access control model implemented by the Android middleware is fundamentally dependent on the kernel layer controls to ensure that the access control enforcement is unbypassable, this leaves these solutions still vulnerable to the root exploits and application vulnerabilities that were described in Section 4. Of these solutions, Kirin and SAINT are the closest to our install-time MAC mechanism, and demonstrated the value of such controls for Android. Our work provides a foundation for supporting such install-time policies in Android.

There has also been prior research into integrating and applying SELinux in embedded systems [9] and in Android [35, 44]. While these efforts have shown how to perform basic enablement of SELinux, they have failed to address or even identify many of the challenges present in enabling the effective use of SELinux in Android, as noted in Section 2. Our work is the first to our knowledge to fully address integration of SELinux support into the Android kernel and userspace and to create a suitable SELinux policy for Android, as described in Section 3. Our work is also the first to our knowledge to concretely demonstrate the benefits of SELinux for Android through the analysis and testing described in Section 4.

TrustDroid [7] and XManDroid [6] bear the greatest similarity to our work, both in their goals and approach. Those systems provided MAC at both the middleware layer and at the kernel layer, although the kernel layer solution relied upon TOMOYO Linux [37] rather than SELinux. SELinux provides a more natural and complete way of enforcing data separation goals based on security labels, unlike the pathname-based security model of TOMOYO. Our work fills the gap in mainline Android for kernel layer MAC and provides a sound base on which these systems and others like them can build. Our work would in turn benefit from incorporating some of the ideas for runtime MMAC that have been explored in TrustDroid and XManDroid.

7. Summary

This paper explains the need for mandatory access control (MAC) in Android, identifies the challenges to enabling the effective use of SELinux in Android and presents how we overcame these challenges in our SE Android reference implementation. The benefits of SE Android are concretely demonstrated through a series of case studies of the impact of SE Android on public Android root exploits and application vulnerabilities. The size and performance overhead imposed by our implementation is evaluated and shown to be negligible.

Availability

The Security Enhanced Android software is available from `http://selinuxproject.org/page/SEAndroid`. Many of the changes have already been merged onto the Android Open Source Project (AOSP) master branch, and work is ongoing to bring the remaining changes to mainline Android.

Acknowledgments

We thank the rest of the SE Android development team for their contributions, particularly Joman Chu for his work on device admin support and James Carter for his work on SEAndroidManager. We thank William Roberts, Haiqing Jiang, Joshua Brindle, Bryan Hinton, Tim Radzykewycz, and the rest of the external SE Android community for their feedback, code review, and contributions of code and policy to SE Android. We thank Kenny Root, Jean-Baptiste Queru, Ying Wang, Matthew Finifter, and the rest of the Google Android team for shepherding the SE Android changes through the AOSP submission process, reviewing the code, integrating it into the AOSP build, and helping to improve it. We thank the anonymous reviewers of the paper for their comments.

References

[1] Android Open Source Project. Android Security Overview. `http://source.android.com/tech/security/`.

[2] Android Open Source Project. Publishing on Google Play. `http://developer.android.com/guide/publishing/publishing.html`.

[3] Android Open Source Project. Security and permissions. `http://developer.android.com/guide/topics/security/security.html`.

[4] Android Open Source Project. What is Android? `http://developer.android.com/guide/basics/what-is-android.html`.

[5] AnTuTu Labs. AnTuTu Benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark`.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium*, February 2012.

[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*, October 2011.

[8] J. Carter. Using GConf as an Example of How to Create an Userspace Object Manager. In *3rd Annual SELinux Symposium*, pages 25–32, March 2007.

[9] R. Coker. Porting NSA Security Enhanced Linux to Handheld Devices. In *2003 Linux Symposium*, July 2003.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium*, August 2011.

[11] J. Donenfeld. Linux Local Privilege Escalation via SUID /proc/pid/mem Write. `http://blog.zx2c4.com/749`.

[12] J. Edge. RLIMIT_NPROC and setuid(). *Linux Weekly News*, July 2011. `http://lwn.net/Articles/451985`.

[13] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[14] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *20th USENIX Security Symposium*, August 2011.

[16] J. Freeman. Mempodroid. `https://github.com/saurik/mempodroid`.

[17] R. Hay. Opera Mobile Cache Poisoning XAS. `http://blog.watchfire.com/wfblog/2011/09/opera-mobile-cache-poisoning-xas.html`.

[18] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droid You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *18th ACM Conference on Computer and Communications Security (CCS'11)*, October 2011.

[19] X. Jiang. GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread). `http://www.csc.ncsu.edu/faculty/jiang/GingerMaster`.

[20] K. Kohei. Security Enhanced PostgreSQL. `http://code.google.com/p/sepgsql`.

[21] S. Krahmer. C-skills: adb trickery #2. `http://c-skills.blogspot.com/2011/01/adb-trickery-again.html`.

[22] S. Krahmer. C-skills: android trickery. `http://c-skills.blogspot.com/2010/07/android-trickery.html`.

[23] S. Krahmer. C-skills: yummy yummy, GingerBreak! `http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html`.

[24] S. Krahmer. C-skills: Zimperlich sources. `http://c-skills.blogspot.com/2011/02/zimperlich-sources.html`.

[25] S. Krahmer. RageAgainstTheCage. `http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz`.

[26] Lookout. LOOK-11-001. `http://blog.lookout.com/look-11-001/`.

[27] Lookout. Lookout Mobile Security: Android Security for Mobile. `http://www.lookout.com`.

[28] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[29] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *21st National Information Systems Security Conference*, pages 303–314, October 1998.

[30] MITRE. CVE-2009-1185. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1185`.

[31] MITRE. CVE-2011-1149. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1149`.

[32] MITRE. CVE-2011-1717. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1717`.

[33] MITRE. CVE-2011-1823. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823`.

[34] MITRE. CVE-2012-0056. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0056`.

[35] Y. Nakamura and Y. Sameshima. SELinux for Consumer Electronics Devices. In *2008 Linux Symposium*, pages 125–134, July 2008.

[36] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, April 2010.

[37] NTT DATA Corporation. TOMOYO Linux Home Page. `http://tomoyo.sourceforge.jp/`.

[38] M. Ongtang, K. Butler, and P. McDaniel. Porscha: policy oriented secure content handling in Android. In *26th Annual Computer Security Applications Conference (ACSAC'10)*, December 2010.

[39] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *25th Annual Computer Security Applications Conference (ACSAC'09)*, December 2009.

[40] Open Handset Alliance. Android Overview. `http://www.openhandsetalliance.com/android_overview.html`.

[41] Opera Software ASA. Opera Mini & Opera Mobile browsers. `http://www.opera.com/mobile`.

[42] J. Palmieri. Get on D-BUS. *Red Hat Magazine*, 3, Jan 2005. `http://www.redhat.com/magazine/003jan05/features/dbus/`.

[43] C. PeBenito, F. Mayer, and K. MacMillan. Reference Policy for Security Enhanced Linux. In *2nd Annual SELinux Symposium*, pages 25–30, March 2006.

[44] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security and Privacy Magazine*, 8(3):36–44, May-June 2010.

[45] Softweg. Benchmark. https://play.google.com/store/apps/details?id=softweg.hw.performance.

[46] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *8th USENIX Security Symposium*, pages 123–139, Aug. 1999.

[47] Tresys Technology. SELinux Reference Policy. http://oss.tresys.com/projects/refpolicy.

[48] E. Walsh. Application of the Flask Architecture to the X Window System Server. In *3rd Annual SELinux Symposium*, pages 33–40, March 2007.

[49] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *11th USENIX Security Symposium*, August 2002.

Appendices

A Sample SELinux policy

This appendix shows the contents of a sample policy source file from the SE Android policy. These contents were taken from *external/sepolicy/bluetoothd.te* in the Android source tree. This file defines a domain for the Android *bluetoothd* daemon. The policy is written using a combination of macros, such as the *init_daemon_domain* macro, and policy language statements, such as *allow* and *type_transition* rules.

```
type bluetoothd, domain;
type bluetoothd_exec, exec_type, file_type;

init_daemon_domain(bluetoothd)
allow bluetoothd self:capability { setuid \
      net_raw net_bind_service net_admin };
allow bluetoothd self:socket *;
allow bluetoothd bluetoothd_data_file:dir \
      create_dir_perms;
allow bluetoothd bluetoothd_data_file:file \
      create_file_perms;
unix_socket_connect(bluetoothd, dbus, dbusd)
```

B Sample seapp_contexts

This appendix shows an excerpt from the *external/sepolicy/seapp_contexts* configuration used to determine how to assign security contexts to app processes created by the *zygote* and to app data directories created by *installd*. Each line specifies a set of input selectors, such as the *isSystemServer* boolean, the *user* name, the *seinfo* string, and the package *name*, and a set of resulting output values, such as the *domain* name, the *type* name, the *levelFromUid* boolean, and the *level* string. A set of precedence rules are applied for determining which entry to use, with more specific entries taking precedence.

```
isSystemServer=true domain=system
user=system domain=system_app \
     type=system_data_file
user=bluetooth domain=bluetooth \
     type=bluetooth_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=_app domain=untrusted_app \
    type=app_data_file levelFromUid=true
user=_app seinfo=platform domain=platform_app \
    type=platform_app_data_file
user=_app seinfo=release domain=release_app \
    type=platform_app_data_file
user=_app seinfo=release \
    name=com.android.browser \
    domain=browser_app \
    type=platform_app_data_file
```

C Sample property_contexts

This appendix shows an excerpt from the *external/sepolicy/property_contexts* configuration used to determine the security context to use in permission checks on setting Android properties. The longest matching prefix is used. The wildcard (*) character can be specified to match any property names that do not match any specified prefix. The sample configuration matches the ownerships assigned to property prefixes by an existing table in the *init* property service code.

```
net.rmnet0        u:object_r:radio_prop:s0
net.gprs          u:object_r:radio_prop:s0
net.ppp           u:object_r:radio_prop:s0
net.qmi           u:object_r:radio_prop:s0
net.lte           u:object_r:radio_prop:s0
net.cdma          u:object_r:radio_prop:s0
gsm.              u:object_r:radio_prop:s0
persist.radio     u:object_r:radio_prop:s0
ril.              u:object_r:rild_prop:s0
net.              u:object_r:system_prop:s0
dev.              u:object_r:system_prop:s0
runtime.          u:object_r:system_prop:s0
hw.               u:object_r:system_prop:s0
sys.              u:object_r:system_prop:s0
service.          u:object_r:system_prop:s0
wlan.             u:object_r:system_prop:s0
dhcp.             u:object_r:system_prop:s0
debug.            u:object_r:shell_prop:s0
log.              u:object_r:shell_prop:s0
service.adb.root  u:object_r:shell_prop:s0
service.adb.tcp.port u:object_r:shell_prop:s0
persist.sys.      u:object_r:system_prop:s0
persist.service.  u:object_r:system_prop:s0
persist.security. u:object_r:system_prop:s0
selinux.          u:object_r:system_prop:s0
vold.             u:object_r:vold_prop:s0
crypto.           u:object_r:vold_prop:s0
ctl.dumpstate  u:object_r:ctl_dumpstate_prop:s0
ctl.ril-daemon u:object_r:ctl_rildaemon_prop:s0
ctl.           u:object_r:ctl_default_prop:s0
*              u:object_r:default_prop:s0
```

D. Middleware MAC policy

This appendix shows an excerpt from the *external/sepolicy/mac_permissions.xml* configuration used for the install-time MAC mechanism and to assign *seinfo* tags to apps. The signature values and permission names have been abbreviated for readability. The signatures are hex-encoded X.509 certificates.

Values for the *seinfo* string and Android permissions can be specified for all packages with a given signature via a *signer* stanza, or may be refined on a per-package basis for specific packages via a *package* stanza. Allowed Android permissions can be specified via a whitelist (*allow-permission*) or via a blacklist (*deny-permission*) but not both. A given signature or package can be allowed all permissions it requests without any constraints by specifying *allow-all*. The *default* tag is used for any app that does not match any other stanza. A *setool* program can be used to help generate policy stanzas from a set of Android packages.

```
<!-- Platform dev key with AOSP -->
<signer signature="...1b357" >
  <allow-all />
  <seinfo value="platform" />
</signer>

<!-- release dev key in AOSP -->
<signer signature="...e684d" >
  <seinfo value="release" />
  <deny-permission name="BRICK" />
  <deny-permission name="READ_LOGS" />
  <deny-permission name="READ_HISTORY_BOOKMARKS" />
  <deny-permission name="WRITE_HISTORY_BOOKMARKS" />
  <package name="com.android.browser" >
    <allow-permission name="ACCESS_COARSE_LOCATION"/>
    <allow-permission name="ACCESS_DOWNLOAD_MANAGER"/>
    <allow-permission name="ACCESS_FINE_LOCATION"/>
    <allow-permission name="ACCESS_NETWORK_STATE"/>
    <allow-permission name="ACCESS_WIFI_STATE"/>
    <allow-permission name="GET_ACCOUNTS"/>
    <allow-permission name="INTERNET" />
    <allow-permission name="MANAGE_ACCOUNTS" />
    <allow-permission name="NFC" />
    <allow-permission name="READ_CONTACTS" />
    <allow-permission name="READ_EXTERNAL_STORAGE" />
    <allow-permission name="READ_PROFILE" />
    <allow-permission name="READ_SYNC_SETTINGS" />
    <allow-permission name="SEND_DOWNLOAD_COMPLETED_INTENTS" />
    <allow-permission name="SET_WALLPAPER" />
    <allow-permission name="USE_CREDENTIALS"/>
    <allow-permission name="WAKE_LOCK"/>
    <allow-permission name="WRITE_EXTERNAL_STORAGE" />
    <allow-permission name="WRITE_SETTINGS" />
    <allow-permission name="WRITE_SYNC_SETTINGS" />
    <allow-permission name="READ_HISTORY_BOOKMARKS"/>
    <allow-permission name="WRITE_HISTORY_BOOKMARKS"/>
    <allow-permission name="INSTALL_SHORTCUT"/>
    <seinfo value="release" />
  </package>
</signer>

<!-- All other keys -->
<default>
  <seinfo value="default" />
  <deny-permission name="ACCESS_COARSE_LOCATION" />
  <deny-permission name="ACCESS_FINE_LOCATION" />
  <deny-permission name="AUTHENTICATE_ACCOUNTS" />
  <deny-permission name="CALL_PHONE" />
  <deny-permission name="CAMERA" />
  <deny-permission name="READ_LOGS" />
  <deny-permission name="WRITE_EXTERNAL_STORAGE" />
</default>
```