Securing Android-Powered Mobile Devices Using SELinux

Google's Android framework incorporates an operating system and software stack for mobile devices. Using a general-purpose operating system such as Linux in mobile devices has advantages but also security risks. Security-Enhanced Linux (SELinux) can help reduce potential damage from a successful attack.

Asaf Shabtai, Yuval Fledel, and Yuval Elovici
Ben-Gurion University

Google's Android is an open source operating system and associated software framework that targets mobile communication devices (mainly smart phones). Google announced the system in November 2007 and released the first Android-powered phone in October 2008. The company released the platform's full source code immediately after the first device hit the market.

Smart phones, in general, were designed as open, programmable network devices that can provide various PC-like services, such as messaging, email, and Web browsing. As such, they're vulnerable to attacks that can compromise the confidentiality, integrity, and availability of data and services.[1] Attack vectors of malware propagating into smart phones include cellular networks, Bluetooth, the Internet (via Wi-Fi, General Packet Radio Service/Enhanced Data Rates for GSM Evolution, or 3G network access), USB, and other peripherals.[2] Security mechanisms for mobile phones such as antimalware and antispam software, host-based intrusion detection tools, and firewalls are available, although most cellular phone owners don't count smart phones as regular computers and rarely run any of these security solutions.

The Android software stack is based on the Linux kernel, which provides low-level services to the rest of the system. It also provides the basis for security enforcement in Android. Using a Linux-based operating system gives rise to several security issues that aren't handled by common security solutions. One of the best known of these issues is that critical processes and services in Linux run with the highest privileges (that is, as a root user). This exposes the system to *privilege escalation attacks* when an attacker detects vulnerability in one of these services. In such a scenario, the attacker could gain full control of the device. For example, the security community has already found several critical security vulnerabilities in Android, ranging from buffer overflows in early software developer kits to interpreting all local keystrokes as root shell commands on G1 devices. In some cases, a security solution must protect the device's capabilities against even the owner's normal usage. For example, network providers might want to protect devices against modifications of network setting configurations.[3]

Regular Linux-based systems don't include security mechanisms to mitigate these risks. Therefore, we propose the use of the Linux Security Modules (LSM) framework for hardening Android-powered devices. LSM gives administrators low-level, fine-grained access control capabilities to confine applications or processes to a tight environment in which they can perform only specific (that is, legitimate) actions and have limited access to unnecessary resources. Thus, untrusted applications are limited to safe actions[4] and so can't affect the system. If a trusted application is maliciously exploited, access control could contain the damage or even block the attack. The main attack class that we set out to mitigate is privilege escalation, which is usually achieved by exploiting vulnerabilities (such as buffer overflows, inadequate security checks, or bad permission assignment). Limiting damage refers to protecting confidential information and allowing continual service to unaffected subsystems, with particular focus on phone capabilities (dialing, conversing, and messaging). Preventing the attack in the first place is an additional benefit.

We integrated the Security-Enhanced Linux LSM (SELinux; www.nsa.gov/selinux) and evaluated its ability to improve Android's security. Previous work showed that tightening the access control (for example, by using SELinux) could reduce damage from a successful attack and even prevent some attacks entirely.[3,5]

The Android Framework

The Linux kernel provides core system services to the Android software stack. These services include device drivers, networking, and file system, memory, power, and process management. Google patched the kernel with kernel enhancements, such as specific drivers and utilities, to support Android.

The next level up in the software stack contains the Android native libraries. These libraries are written in C/C++ and used by various system components in the upper layers. Next is the Android runtime, consisting of the Dalvik virtual machine (VM) and the core libraries. The Dalvik VM runs the Dalvik executable (`.dex`) files that are designed to be more compact and memory efficient than Java class files. The core libraries are written in Java and provide a substantial subset of the Java 5 standard edition packages as well as some Android-specific libraries, which are needed for accessing the capabilities provided by the hardware, operating system, and native libraries.

The application framework layer, written fully in Java, includes tools provided by Google as well as proprietary extensions or services. The top layer is the application layer, which provides such applications as a phone, Web browser, and email client. Each Android application is packaged in an `.apk` archive for installation. The `.apk` archive is similar to a Java standard `jar` file in that it holds all code and noncode resources (such as images and manifest) for the application. Android applications are written in Java based on the APIs provided by the Android software development kit.

Every Android application runs in its own Linux process, with its own instance of the Dalvik VM. The application is also assigned its own user ID at installation time. Therefore, in principle, the code of two applications can't run in the same process or harm each other. The Dalvik VM relies on Linux for its underlying functionality (for example, process isolation, threading, and low-level memory management).

Android's critical system processes run with high-privileged users such as root, system, or rild. These processes include the *zygote*, which is a master Dalvik VM process that forks new Dalvik processes whenever a Dalvik-based application needs to start; the *system_server*, which is a Dalvik-based process (started by the zygote) that runs many managed and native services; the first process started by the kernel, named *init*, which initializes the system, starts daemons, cleans up the system during shutdown, invokes the power-off command, and cleans up zombie processes during normal operation; the *mountd*, which automatically mounts mass-storage (such as USB flash drives or SD cards); and the *rild* process, which manages communication with the cellular radio (GSM).

Security in Android

Linux provides several access control mechanisms. The basic element of these mechanisms is users (that is, entities). Users (represented by an integer number or user id) own objects (a process, file, or directory). Users are further assigned to groups.

In the Linux file permissions mechanism, each file is associated with an owner user and group IDs and three tuples of read, write, and execute (rwx) permissions. The kernel enforces the first tuple on the owner, the second on users belonging to the group, and the third on the remaining users. Files in Android (both application and system files) are subject to the Linux permission mechanism. Generally, system files in Android are owned by either the system or root user, and application files are owned by an application-specific user. Separate users for each application and for the system files and proper permissions provide the needed security for file access. Files created by an application won't be accessible to other applications (unless explicitly specified).

The heart of the application-level security in Android is the application-permission mechanism. Developers use this mechanism to enforce restrictions on specific operations that an application can perform. Android offers roughly 100 built-in permissions to control operations such as dialing (`CALL_PHONE`), taking pictures (`CAMERA`), using the Internet (`INTERNET`), and listening to key strokes (`READ_INPUT_STATE`). Any Android application can declare additional permissions. To obtain permission, an application must explicitly request it.

As mentioned previously, unlike a typical Linux-based desktop system, different applications in Android are executed as different users. Such a preventive measure raises the bar on successful exploitation because one application can't affect others in a normal way. However, two or more processes could still run as the same user, and, in particular, several system daemons run as root, system, and radio users.

Android's security mechanisms (both Android-specific and Linux-inherited) are insufficient and too coarse-grained to tackle this security issue. For example, an application granted Android's application-level `INTERNET` permissions can listen on any port, create any type of socket, communicate with all protocols, and more. The file-permission mechanism protects files, but not from a root user.

Several mitigation strategies deal with the "all-powerful root" problem. In Linux capabilities, the superuser (root) privileges are split into several dozens of capabilities. A root-owned process can voluntarily discard capabilities it no longer needs and therefore limit the superuser. A similar method involves voluntarily switching to a nonroot user when root privileges are no longer required.

LSM is an additional solution.[6] It supports various access control models that complement the previously described methods. The system integrator chooses the particular access control model. LSM is implemented through a set of hooks embedded in the Linux kernel in many locations (such as the file system or fork) that require authorization decisions. Security modules can register themselves on these hooks and forbid the execution of certain actions.

Security-Enhanced Linux

SELinux, introduced in 2000, is the best-known LSM implementation. It evolved from the US National Security Agency's Flask research project.[7] SELinux provides several types of enforcement, such as type enforcement, role-based access control, and multilevel security. Authorization decisions are based on a policy, which is loaded from a file. In this research, we focus only on type enforcement because it's the most basic and powerful method that SELinux provided, and it supports most of our required functionalities.

Type enforcement operates by tagging each entity (such as processes) and object (such as files and packets) in the system. These tags are distinct from the regular Linux users, owners, or groups. For example, in Android, we would assign the `init_t` tag for the init process (the process that initializes the system), and the `init_exec_t` tag for the /sbin/init executable file (the file that executes the init process). These tags are then used when an authorization decision needs to be made. Tag assignment and authorization rules are defined in a policy file. A tag on a process is called a *domain*, whereas a tag on an object is called a *type*. Tagging entities and objects lets system integrators define finer-grained access permissions in the policy. For example, two processes running under the same user can be defined with different permissions.

Generally, access is denied unless an *allow* rule is explicitly defined. Such rules consist of the entity's domain, the object type, and the permission required. Permissions are selected from a set of about 200 predefined operations, whereas tags are user-defined.

Consider an audio player that should have access to the sound card. The audio player will be assigned an `audio_player_t` tag and the sound card pseudo-file will be assigned a `sound_card_t` tag. The policy will include an allow rule that enables *audio_player_t* to read, write, and get the attributes of any *sound_card_t* file, but not to remove it, change its ownership, or change access permissions to it. We could further control the access such that audio players may access only local sound cards but not Bluetooth earphones, even though they use the same interface.

By default, when creating a new file, the file is assigned the same label as the parent directory. Similarly, when executing a file, the resulting process is labeled the same as the executing process. The *type-transition* feature lets us define a different label for the resulting process. The label is applied automatically, without any additional action by any process in the system. Explicit transitions (that is, a command from userspace that requests a label change and specifies the target label) are also allowed.

SELinux supports two modes of operation: enforcement and permissive. In the former, the system enforces policy decisions, such as denying access or auditing. The permissive mode only logs policy violations, and it doesn't enforce the policy. Typically, a policy is shaped in permissive mode, and, when the system isn't logging any more violations, it can be put in enforcement mode.

Creating a policy is difficult. A *targeted* policy confines only known applications and leaves unknown applications subject to other existing access control mechanisms. A *strict* policy confines all processes and thus requires policy rules to be defined on every process. Critical servers are advised to run with a strict policy (when everything is confined), whereas desktops usually run with a more liberal targeted policy. Given that we can't assume to know all the processes and applications that a user will run on an Android device in advance, we used the targeted policy approach. In this way, we focus on confining only known privileged processes, while leaving unknown applications with their default privileges.

SELinux's tighter access control can confine user and system programs to the minimum amount of privilege they require to do their jobs. First, it can restrict privileged daemons. So, if such a daemon is exploited, the attacker can perform only the operations that the daemon is supposed to perform, thereby limiting his or her playing field. Second, an unprivileged process, such as a browser, can only invoke calls that it's supposed to, thereby protecting the system from potential vulnerabilities in unused calls.

A positive outcome of tightening the control over an application is that during policy editing, a system integrator sometimes detects existing vulnerabilities. An example of such a detected vulnerability is leakage of file descriptors, which allows processes to access files they should not be able to access.[7]

Researchers enabled SELinux on embedded devices and mobile phones that predate Android. Björn Vogel and Bernd Steinke tested SELinux on a Nokia

770 Internet Tablet.[3] They modified the file system to support extended attributes and reduced and updated the policy for the specific requirements. The authors didn't do a performance evaluation, but they refer to a reference stating that systems with SELinux suffer a 7 percent performance loss.

Yuichi Nakamura and Yoshiki Sameshima addressed SELinux's high resource usage.[8] They tackled three major overhead types. They handled the large policy size by creating a policy editor and a simplified policy language that lets system integrators write smaller policies. They dealt with the CPU and memory overhead in the kernel using micro-optimizations of the data structures and code paths. They also reduced memory overhead at the userspace by removing the need for code that's required only by policy creators from the SELinux library. Linux version 2.6.24 includes these kernel changes. Xinwen Zhang and his colleagues implemented an integrity model on LiMo using the SELinux framework in order to achieve the goals of integrity measure and attestation.[9]

Using SELinux in Android

We present several scenarios demonstrating SELinux's usefulness in Android. In the scenarios, we assume an existing vulnerability in Android and show how SELinux would reduce the damage it could cause when exploited, thus extending the length of time available to properly fix the vulnerability.

Protecting Processes

Android contains several components that attackers might exploit to gain additional privileges. Five processes run as a root user, two as a system user, and two more as a radio user. Note that system and radio user ids are hardcoded into the kernel to provide them with additional privileges. Although some trusted components can be audited to ensure that they aren't susceptible to attacks, others are large enough to make formal validation difficult. The cost of ensuring that these components aren't vulnerable can be high, especially in the face of upgrades, such as the addition of new features or optimizations.

Android daemons running as root include init, mountd, debuggerd, zygote, and installd. The installd daemon is in charge of unpacking and handling `apk` files (Android installation packages). It must be run as root because it requires ownership and permissions to be set on the unpacked files. Such files might arrive from an untrusted source, and we can assume that an attacker could create a malformed `apk` file. If an attacker detects vulnerability in installd, the malformed `apk` can exploit a buffer overflow in the unpacking routines, or crash installd during content handling, such as signature verification or manifest parsing. Once a malformed file exploits such a buffer overflow bug in a privileged daemon, it can execute arbitrary code using the daemon's privileges (in this case, root privileges), which might result in bricking the device, causing excessive billing, or theft of sensitive information. In addition, from our experience with the Android-powered G1 device, an attacker that gained root privileges on an Android device can maintain a backdoor that will give him or her full control of the device as well as disable patching updates.

Protecting Files

To preserve system integrity, we must prevent unauthorized users from writing to critical executables or altering configuration files. But even read-only access should sometimes be prevented because some files might contain confidential information (such as passwords).

The basic method for restricting access to files in Linux is using the file permissions mechanism. However, the root user has an override capability that results in ignoring the permissions. SELinux can limit a process's access to files even if it runs with the root user. For example, the init process doesn't need to write to disk, and SELinux can ensure that it never will, even though it runs as root.

Applying access control on files in Android is particularly necessary in corporate environments, where devices owned by management personnel contain data that can cause the company severe damage if leaked.[10]

Additionally, Linux handles many system functionalities as pseudo-files. Consequently, methods that protect files can therefore also protect critical system resources and services such as drivers, sockets, character devices, block devices, and directories.

Protecting the System

For most operating systems, once a vulnerability is discovered, the system vendor can provide an update that patches this vulnerability. Android also provides an updating mechanism. However, in practice, once the system is exploited, the updating mechanism might be disabled. For example, Android's unofficial updates disabled the updating mechanism so that official updates won't override them. Moreover, the exploit could prevent a legitimate user from fixing the problem manually. It's therefore crucial that we can prevent the updating mechanism from being disabled even if a vulnerability is found.

The critical component of the updating code resides on a separate recovery partition, which isn't used during the device's normal operation. By denying the ability to replace this trusted code, a user could reboot into the recovery code manually (using a special button combination) and update the system, similarly to rebooting in safe mode on PCs.

We must also prevent the security mechanism itself from being overridden. Again, the resources and actions required are known (although in the LSM case, we assume that the kernel itself isn't vulnerable to arbitrary code-execution attacks). Basically, this means that we want to maintain the kernel's integrity. SELinux can do so by limiting actions, such as the ability to load kernel modules, replace the kernel image on the disk, load a different security policy, or directly access system memory or the disk.

Integrating SELinux in Android

We applied SELinux to Android and tested a policy that was created specifically for Android, with its unique file system layout, init system, and tailored daemons. We encountered several challenges during the integration.

Android Doesn't Support SELinux

Security modules are disabled in the Linux kernel that Android runs; amongst them is SELinux. To solve this problem, we reconfigured and deployed the kernel that supports SELinux. Reconfiguring and deploying the SELinux kernel requires a rooted device (that is, a device running with root privileges).

Android Doesn't Have a Method or Tool for Loading the SELinux Policy

The most straightforward solution to this problem is to cross-compile the standard policy-loading tool or use an embedded policy-loading applet in a tool such as Busybox. However, these solutions don't solve the problems of loading the policy early on boot or of labeling the init process correctly. Because other platforms had SELinux-specific code in init, we added our code to Android's init. The Android init process executes commands from an init.rc text file.

We added three new commands to the interpreter:

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

Figure 1a depicts the modified part of init.rc.

Creating a Custom SELinux Policy for Android

The default SELinux reference policy, which is highly modular, is irrelevant on Android for two main reasons:

- it assumes a Linux standard base- (LSB)-like layout of the file system, and
- it's too big for an embedded system.

To create a policy for Android, we initially removed modules that are irrelevant to it (such as Apache). The resulting trimmed policy file was still too big for an embedded system (approximately 500 Kbytes). In addition, the reference policy assumes an LSB-like system, whereas the Android file-system layout is highly different. So, we opted to construct a policy from scratch, without using the reference policy macro language. We started with an empty policy as generated by the mdp (make dummy policy) script. We started the system in permissive mode and added rules to the policy until all warnings during the device's regular usage were silenced. This policy is based on an Android-like file system layout and confines only processes existing on Android (for example, system_server and zygote).

Android's File System Doesn't Support Extended Attributes

The default file system (yaffs2) doesn't support extended attributes (xattrs). Other works have mentioned this problem and solved it on other file systems for other platforms.[3,7]

We used the `chcon` command to set xattrs on memory file systems that support xattrs (tmpfs, rootfs, and proc). To bypass the lack of xattr support in yaffs2, we used the context directive to label processes explicitly. This workaround specifies the label in the init script and therefore doesn't require labeling the file system.

It's Difficult to Apply SELinux Policy to Dalvik Processes

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

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

Steps in Porting SELinux to Android

To port SELinux to Android, we therefore took the following steps:

1. We compiled the Android kernel with SELinux support.
2. We created and compiled an Android-specific security policy.

```
on init
 loglevel 3
 # SELinux must be initialized very early
 mkdir / selinux
 mount selinuxfs selinuxfs / selinux
 loadpolicy / se-policy
 write /proc/1/attr/current system_u:system_r:init_early_t
 chcon / init system_u:object_r:init_exec_t
 chcon /sbin/abdb system_u:object_r:adbd_exec_t
 chcon /dev/null system_u:object_r:devnull_t
 chcon /dev/ ashmem system_u:object_r:ashmem_t
```
(a)

```
1  allow kernel_t rootfs_t:dir { search };
2  allow init_early_t tmpfs_t:filesystem { mount };
3  allow init_t init_t:process { fork setpgid };
4  allow init_t init_t:unix_stream_socket { create bind };
5  allow init_t tmp_t:sock_file { create setattr unlink };
6  allow init_t adbd_t:process { transition };
7  allow adbd_t devnull_t:chr_file { read write };
8  allow adbd_t ashmem_t:chr_file { read write };
9  type_transition init_t adbd_exec_t:process adbd_t;
```
(b)

```
# ps -Z
     PID CONTEXT                         STAT    COMMAND
1    system_u:system_r:init_t            S       /init
2    system_u:system_r:kernel_t          SW<     [ kthreadd ]
3    system_u:system_r:kernel_t          SW<     [ ksoftirqd / 0 ]
4    system_u:system_r:kernel_t          SW<     [events0/]
...
16   system_u:system_r:kernel_t          SW<     [ rpciod / 0 ]
17   system_u:system_r:unconfined_t      S       /system/bin/sh
18   system_u:system_r:servicemanager    S       /system/bin/servicemanager
19   system_u:system_r:mountd_t          S       /system/bin/ mountd
20   system_u:system_r:debuggerd_t       S       /system/bin/ debuggerd
23   system_u:system_r:unconfined_t      S       zygote /bin/ app_process -Xzygote /s
24   system_u:system_r:mediaserver_t     S       /system/bin/ mediaserver
26   system_u:system_r:dbusd_t           S       /system/bin/ dbus -daemon --system --
27   system_u:system_r:installd_t        S       /system/bin/ installd
30   system_u:system_r:unconfined_t      S       /system/bin/ qemud
33   system_u:system_r:adbd_t            S       / sbin / adbd
51   system_u:system_r:unconfined_t      S       system_server
86   system_u:system_r:unconfined_t      S       com.android.phone
92   system_u:system_r:unconfined_t      S       android.process.acore
107  system_u:system_r:unconfined_t      S       com.android.mms
123  system_u:system_r:unconfined_t      S       com.google.process.gapps
138  system_u:system_r:unconfined_t      S       android.process.media
152  system_u:system_r:unconfined_t      S       com.android.alarmclock
```
(c)

Figure 1. Configuration of SELinux on Android. This figure shows (a) init.rc script modifications, (b) an example of SELinux authorization rules, and (c) an example of SELinux-enabled process listing.

3. We modified the init process code and init.rc script to support additional commands to load the policy at system startup and set initial labels.
4. We built a new disk image containing the updated init and policy files and updated it on the device.

Figure 1b depicts an example of the authorization rules section we used in our evaluation. In line 1, we see that processes with a `kernel_t` label can list files in directories that have the `rootfs_t` label. In line 3, we can see that init is allowed to fork a child of itself. In line 4, the `init_t` process can create and bind a Unix domain socket. In line 7, `adbd` can read and write to the `/dev/null` char file, and in line 8 to the `/dev/ashmem` char file. Finally, line 9 shows the type-transition rule, which specifies that when the `init_t` process executes an `adbd_exec_t` file, the resulting `adbd` process will be labeled as `adbd_t`. Without the type transition, the `adbd` process would retain the `init_t` label and all the privileges it allows.

Figure 1c depicts an output of `ps -Z` under an SELinux-enabled Android. The `-Z` argument instructs ps to show the security context of processes running on the system. In this listing, kernel threads are labeled as `kernel_t`, and critical processes are correctly identified and labeled.

Benchmarking

We ran several benchmarks to check the performance overhead when using SELinux on Android and to understand the possible effect on the user. The benchmarking focused on metrics such as I/O bandwidth, latency, CPU consumption, and memory footprint. We collected all benchmarking measurements on a G1 device, which is based on a Qualcomm MSM7201A processor, 192 Mbytes of RAM, internal flash storage of 256 Mbytes, and a microSD expansion slot. On the software side, we performed the benchmarks on the RC30 version of T-Mobile G1 USA with minimal modifications: we enabled `adb` support and added a root shell and a custom kernel. The RC30 kernel is based on Linux 2.6.25, with platform (msm) patches and, of course, Android patches.

We compiled both the SELinux-enabled and non-SELinux kernels. The non-SELinux-enabled kernel had the exact same configuration as the official RC30 kernel, and the SELinux-enabled kernel had the minimum additional configuration options needed to support SELinux.

For the evaluation we used the bonnie++ (www.coker.com.au/bonnie++) and lmbench (www.bitmover.com/lmbench) benchmarking applications. We executed each benchmark 33 times on the SELinux-enabled kernel and 33 times on the non-SELinux kernel. We maintained all other conditions unchanged and used the same device for both evaluations.

Bonnie++ is a file system and disk performance benchmarking application for POSIX-compatible systems. We performed the tests on a microSD card with the vfat file system. This test sought to measure the degree of slowdown in the I/O paths caused by the additional SELinux overhead.

We used the microSD card because the internal storage is small enough to fit in RAM and so could be served entirely by memory cache, rendering an I/O benchmark useless. The file size used in the benchmark was 192 Mbytes, which ensured that the I/O operations reached the physical storage.

The lmbench suite is a set of microbenchmarks for low-level Linux functionalities. Chris Wright and his colleagues first used lmbench to benchmark LSM implementations during the initial introduction of the LSM framework.[6] Nakamura and Sameshima also used lmbench to compare the overhead of SELinux in mobile devices.[8] They used Linux 2.6.22, and their optimizations were merged into Linux 2.6.24, which means that our benchmark includes their optimizations. lmbench contains a lengthy list of metrics, and we used relevant ones in our evaluation.

Table 1 lists the benchmarking results. We performed statistical significance testing using nonparametric ANOVA (Mann-Whitney test), which doesn't assume normal distribution of the measurements. An asterisk indicates significant changes ($p < 0.0013$). Comparing many parameters in separate tests raises the well-known multiple comparisons problem—that is, the probability of making at least one type I error in any of the comparisons is much larger than the alpha (α) value used in each separate test. It's common practice to use the Bonferroni correction to address this issue—that is, dividing alpha by the number of comparisons made.

Most of the bonnie++ results proved insignificant. Three metrics—sequential input bandwidth (char), sequential input bandwidth (block), and input latency—showed significant yet moderate slowdowns. Overall, we can't conclude that SELinux impacts I/O performance significantly, or that it impacts CPU usage due to I/O operations.

Most of the lmbench metrics proved significant. We observed notable slowdowns in operations requiring additional permission checks (such as open, close, read, write, and stat) on both files and sockets. The remaining actions showed moderate slowdowns.

We encountered two cases of notable speedup with SELinux: the protection fault (lmbench) and `seq output Chr` (bonnie++). Our review of the benchmark's source code and the kernel code couldn't pinpoint the speedup's cause. We suspect that it might be caused by compiler heuristics, CPU cache layout, or the write-combine optimization in the case of the `seq output Chr` metric.

We measured memory usage at system startup, taking special care to ensure that the system had completed its initialization and was waiting for user input. Table 1 compares memory usage (free RAM) with and without SELinux support. The results show that SELinux doesn't significantly impact memory consumption.

Enabling SELinux doesn't affect the disk costs on G1 for several reasons. The device has five partitions: recovery, boot, system, cache, and data. All our additions increased either the kernel or the ramdisk that resides on the boot partition. The boot partition is statically allocated and can't be used for storing files, and, therefore, there is no decrease in free space on the file system. In total, an update file for only kernel and ramdisk weighed 1,379,032 bytes compressed for the non-SELinux version and 1,477,188 bytes for the SELinux version (a 7.11 percent increase over the non-SELinux version). The SELinux binary policy file, containing two users, three roles, 47 types, 70 classes, and 533 rules, was 17,400 bytes in size.

Table 1. Evaluation results.

Measure	Without SELinux	With SELinux	Slowdown (%)
bonnie++			
Seq output, Chr (KBps)	52.3 ± 1.26	62.76 ± 1.89	–16.66*
Seq output, Chr (% CPU)	96.82 ± 0.92	96.33 ± 0.89	–0.50
Seq output, block (KBps)	3,674.64 ± 15.76	3,682.06 ± 25.41	–0.20
Seq output, block (% CPU)	13.42 ± 0.5	13.79 ± 0.48	+2.71
Seq output, rewrite (KBps)	2,623.7 ± 5.7	2,624.27 ± 7.64	–0.02
Seq output, rewrite (% CPU)	11.7 ± 0.47	11.94 ± 0.24	+2.07
Seq input, Chr (KBps)	205.45 ± 16.44	157.85 ± 12.97	+30.16*
Seq input, Chr (% CPU)	99 ± 0	98.97 ± 0.17	–0.03
Seq input, block (KBps)	8,914.91 ± 10.22	8,895.91 ± 10.54	+0.21*
Seq input, block (% CPU)	20.3 ± 0.47	20.61 ± 0.5	+1.49
Random, seeks (KBps)	104.75 ± 2.93	103.48 ± 2.29	+1.22
Random, seeks (% CPU)	41.82 ± 2.32	43.88 ± 2.52	+4.93
Seq output, Chr, latency (ms)	340.48 ± 110.83	326.27 ± 109.23	–4.17
Seq output, block, latency (ms)	1,368.21 ± 70.93	1,332.24 ± 108.47	–2.63
Seq output, rewrite, latency (ms)	2,175.7 ± 107.89	2,132.18 ± 234.56	–2.00
Seq input, Chr, latency (ms)	47.66 ± 4.16	60.29 ± 4.64	+26.51*
Seq input, block, latency (ms)	34.72 ± 14.59	33.99 ± 14.17	–2.11
Random, seeks, latency (ms)	225.45 ± 37.23	234.88 ± 36.81	+4.18
Lmbench			
Simple syscall (μs)	4.83 ± 0.05	4.78 ± 0.05	–0.86
Simple read (μs)	7.84 ± 0.12	9.64 ± 0.14	+22.96*
Simple write (μs)	7.25 ± 0.83	10.02 ± 1.72	+38.13*
Simple stat (μs)	97.39 ± 2.03	184.95 ± 5.21	+89.91*
Simple fstat (μs)	12.81 ± 0.18	27.9 ± 1.81	+117.75*
Simple open/close (μs)	139.19 ± 5.36	261.5 ± 6.75	+87.87*
Signal handler installation (μs)	6.64 ± 0.05	6.49 ± 0.02	–2.22
Signal handler overhead (μs)	39.29 ± 0.51	44.25 ± 1.82	+12.63*
Protection fault (μs)	6.77 ± 1.12	1.81 ± 1.31	–73.19*
Pagefault (μs)	146.09 ± 1.13	149.27 ± 3.05	+2.17*
Pipe bandwidth (MBps)	39.85 ± 1.01	38.61 ± 0.3	+3.23*
Pipe latency (μs)	312.91 ± 8.38	454.06 ± 13.66	+45.11*
AF_UNIX sock stream bandwidth (MBps)	43.95 ± 1.09	42.7 ± 0.5	+2.93*
AF_UNIX sock stream latency (μs)	408.68 ± 14.03	677.62 ± 14.25	+65.81*
Context switches, size = 16k (2)	724.01 ± 8.77	826.33 ± 10.83	+14.13*
Context switches, size = 16k (4)	1,057.49 ± 15.33	1,147.86 ± 28.63	+8.55*
Context switches, size = 16k (8)	1,107.91 ± 13.48	1,190.78 ± 23.16	+7.48*
Context switches, size = 16k (16)	1,120.88 ± 19.38	1,200.32 ± 20.3	+7.09*
Context switches, size = 16k (24)	1,106.83 ± 17.98	1,188.16 ± 22.52	+7.35*
Context switches, size = 16k (32)	1,099.9 ± 25.15	1,178.78 ± 21.03	+7.17*
Context switches, size = 16k (64)	1,077.3 ± 31.04	1,157.98 ± 35.53	+7.49*
Context switches, size = 16k (96)	1,054.5 ± 33.86	1,126.61 ± 49.04	+6.84*
Disk and memory usage			
Free RAM	53,177 ± 828 Kbytes	51,945 ± 929 Kbytes	–2.32
Init file size	98,260 bytes	98,296 bytes	+0.04
Init.rc	8,630 bytes	9,469 bytes	+9.72
Kernel size	1,379,032 bytes	1,477,188 bytes	+7.11
Binary policy	N/A	17,400 bytes	N/A

*Indicates significant changes ($\alpha = 0.0013$)

Implementing SELinux in Android hardens the Android system and enforces low-level access control on critical Linux processes that run under privileged users. By choosing this route, we can better protect the system from scenarios in which an attacker exploits a vulnerability in a high-privileged process.[11] This path is a low-cost, high-gain solution. Our future work will focus on additional evaluation of SELinux and other LSMs such as Smack and AppArmor. ☐

References

1. C. Dagon, T. Martin, and T. Starner, "Mobile Phones as Computing Devices: The Viruses Are Coming," *IEEE Pervasive Computing*, vol. 3, no. 4, 2004, pp. 11–15.
2. J. Cheng et al., "SmartSiren: Virus Detection and Alert for Smartphones," *Proc. 5th Int'l Conf. Mobile Systems, Applications and Services* (MobiSys 07), ACM Press, 2007, pp. 258–271.
3. B. Vogel and B. Steinke, "Using SELinux Security Enforcement in Linux-Based Embedded Devices," *Proc. 1st Int'l Conf. Mobile Wireless Middleware, Operating Systems, and Applications* (MobilWare 08), Inst. for Computer Sciences, Social-Informatics and Telecomm. Eng., 2008, article 15.
4. *Open Mobile Terminal Platform (OMTP)*, Application Security Framework, 2008.
5. D. Marti, "A Seatbelt for Server Software: SELinux Blocks Real-World Exploits," *LinuxWorld.com*, 20 Feb. 2008; www.linuxworld.com/news/2008/022408-selinux.html.
6. C. Wright et al., "Linux Security Module Framework," *Proc. Linux Symp.*, Linux Symp., 2002, pp. 604–610.
7. J. Morris, "Have You Driven an SELinux Lately? An Update on the Security Enhanced Linux Project," *Proc. Linux Symp.*, Linux Symp., 2008, pp. 101–113.
8. Y. Nakamura and Y. Sameshima, "SELinux for Consumer Electronics Devices," *Proc. Linux Symp.*, Linux Symp., 2008, pp. 125–133; www.linuxsymposium.org/archives/OLS/Reprints-2008/nakamura-reprint.pdf.
9. X. Zhang, O. Aciicmez, and J.P. Seifert, "Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms," *Proc. 1st Int'l Conf. Security and Privacy in Mobile Information and Comm. Systems* (MobiSec 09), Springer, 2009, pp. 71–82.
10. C. Hanson, "SELinux and MLS: Putting the Pieces Together," *Proc. SELinux Symp.*, SELinux Symp., 2006; http://selinux-symposium.org/2006/papers/03-SELinux-and-MLS.pdf.
11. A. Shabtai et al., "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy*, vol. 8, no. 2, 2010, pp. 35–44.

***Asaf Shabtai**, CISSP, is a PhD student at Ben-Gurion University of the Negev (BGU), Israel, and an R&D manager at Deutsche Telekom Labs. His main areas of interests are computer and network security, data mining, temporal data mining, and temporal reasoning. Shabtai has an MSc in information systems engineering from BGU. Contact him at shabtaia@bgu.ac.il.*

***Yuval Fledel** is an MSc student in the information systems engineering department at Ben-Gurion University of the Negev (BGU), Israel, and a senior system architect and security expert at Deutsche Telekom Labs. His primary areas of interest are computer and network security, operating systems security, general hacking, and reverse engineering. Fledel has a BSc in software engineering from BGU. Contact him at fledely@bgu.ac.il.*

***Yuval Elovici** is the director of Deutsche Telecom Laboratories at Ben-Gurion University of the Negev (BGU), Israel, and is a member of the information systems engineering department. His main areas of interest are computer and network security, information retrieval, and data mining. Elovici has a PhD in information systems from Tel-Aviv University, Israel. Contact him at elovici@bgu.ac.il.*