The Case for SE Android

Stephen Smalley
sds@tycho.nsa.gov
Trust Mechanisms (R2X)
National Security Agency

Android: What is it?

- Linux-based software stack for mobile devices.
- Very divergent from typical Linux.
  - Almost everything above the kernel is different.
    - Dalvik VM, application frameworks
    - bionic C library, system daemons
    - init, ueventd
  - Even the kernel is different.
    - Unique subsystems/drivers: Binder, Ashmem, ...
    - Hardcoded security checks.

Binder & Ashmem

- Android-specific mechanisms for IPC and shared memory.
- Binder
  - Primary IPC mechanism.
  - Inspired by BeOS/Palm OpenBinder.
- Ashmem
  - Shared memory mechanism.
  - Designed to overcome limitations of existing shared memory mechanisms in Linux (debatable).

Android Security Model

- Application-level permissions model.
  - Controls access to app components.
  - Controls access to system resources.
  - Specified by the app writers and seen by the users.
- Kernel-level sandboxing and isolation.
  - Isolate apps from each other and the system.
  - Prevent bypass of application-level controls.
  - Relies on Linux discretionary access control (DAC).
  - Normally invisible to the users and app writers.

Discretionary Access Control (DAC)

- Typical form of access control in Linux.
- Access to data is entirely at the discretion of the owner/creator of the data.
- Some processes (e.g. uid 0) can override and some objects (e.g. sockets) are unchecked.
- Based on user & group identity.
- Limited granularity, coarse-grained privilege.

Android & DAC

- Restrict use of system facilities by apps.
  - e.g. bluetooth, network, storage access
  - requires kernel modifications, “special” group IDs
- Isolate apps from each other.
  - unique user and group ID per installed app
  - assigned to app processes and files
- Hardcoded, scattered “policy”.

SELinux: What is it?

- Mandatory Access Control (MAC) for Linux.
  - Defines and enforces a system-wide security policy.
  - Over all processes, objects, and operations.
  - Based on security labels.
- Can confine flawed and malicious applications.
  - Even ones that run as “root” / uid 0.
- Can prevent privilege escalation.

How can SELinux help Android?

- Confine privileged daemons.
  - Protect them from misuse.
  - Limit the damage that can be done via them.
- Sandbox and isolate apps.
  - Strongly separate apps from each other and from the system.
  - Prevent privilege escalation by apps.
- Provide centralized, analyzable policy.

What can't SELinux protect against?

- Kernel vulnerabilities, in general.
  - Although it may block exploitation of specific vulnerabilities. We'll see an example later.
  - Other kernel hardening measures (e.g. grsecurity) can be used in combination with SELinux.
- Anything allowed by the security policy.
  - Good policy is important.
  - Application architecture matters.
    - Decomposition, least privilege.

SE Android: Goals

- Improve our understanding of Android security.
- Integrate SELinux into Android in a comprehensive and coherent manner.
- Demonstrate useful security functionality in Android using SELinux.
- Improve the suitability of SELinux for Android.
- Identify other security gaps in Android that need to be addressed.

Enabling SELinux in Android: Challenges

- Kernel
  - No support for per-file security labeling (yaffs2).
  - Unique kernel subsystems lack SELinux support.
- Userspace
  - No existing SELinux support.
  - All apps forked from the same process (zygote).
  - Sharing through framework services.
- Policy
  - Existing policies unsuited to Android.

Enabling SELinux in Android: Kernel

- Implemented per-file security labeling for yaffs2.
  - Using recent support for extended attributes (xattr).
  - Enhanced to label new inodes at creation.
- Analyzed and instrumented Binder for SELinux.
  - Permission checks on IPC operations.
  - Sender security label information.
- To Do:
  - Study and (if needed) instrument other Android-specific kernel subsystems (e.g. ashmem).

Enabling SELinux in Android: SELinux Libraries/Tools

- Ported minimal subset of libselinux to Android.
  - Added xattr syscalls to bionic.
  - Removed glibc-isms from libselinux.
- Other libraries not required on the device.
  - Policy can be built offline.
- Specific tools ported as needed.
  - init built-in commands for use by init.rc
  - toolbox extensions for use from shell

Enabling SELinux in Android: Build Tools

- Filesystem images generated using special purpose tools.
  - mkyaffs2image, make_ext4fs
  - no support for extended attributes / security labels
- Modified tools to label files in images.
  - required understanding on-disk format
  - used to generate labeled /system, /data partitions

Enabling SELinux in Android: init

- init / ueventd
  - load policy, set enforcing mode, set context
  - label sockets, devices, runtime files
- init.rc
  - setcon, restorecon commands
  - seclabel option

Enabling SELinux in Android: Zygote & Installd

- zygote
  - Modified to set SELinux security context for apps.
  - Maps DAC credentials to a security context.
- installd
  - Modified to label app data directories.
- To Do:
  - Generalize assignment of security contexts.
  - Augment existing policy checks with SELinux permission checks.

Enabling SELinux in Android: Policy

- Confined domains for system daemons.
  - Only kernel and init are unconfined.
- Parallel existing Android DAC model for apps.
  - Use domains to represent system permissions.
  - Use categories to isolate apps.
- Benefits:
  - Small, fixed policy.
  - No policy writing for app writers.
  - Normally invisible to users.

Enabling SELinux in Android: Current State

- Basic working prototype
  - on the Android emulator
  - on the Nexus S
- Kernel, userspace, and policy support
- Capable of enforcing (some) security goals.
- Still a long way from a complete solution.
  - But let's see how well it does...

Case Study: vold

- vold - Android volume daemon
  - Runs as root.
  - Manages mounting of disk volumes.
  - Receives netlink messages from the kernel.
- CVE-2011-1823
  - Does not verify that message came from kernel.
  - Uses signed integer from message as array index without checking for < 0.
- Demonstrated by the Gingerbreak exploit.

GingerBreak: Overview

- Collect information needed for exploitation.
  - Identify the vold process.
  - Identify addresses and values of interest.
- Send carefully crafted netlink message to vold.
  - Trigger execution of exploit binary.
  - Create a setuid-root shell.
- Execute setuid-root shell.
- Got root!

GingerBreak: Collecting Information

- Identify the vold process.
  - /proc/net/netlink to find netlink socket users.
  - /proc/pid/cmdline to find vold PID.
- Identify addresses and values of interest.
  - /system/bin/vold to obtain GOT address range.
  - /system/lib/libc.so to find “system” address.
  - /etc/vold.fstab to find valid device name
  - logcat to obtain fault address in vold.

GingerBreak: Would SELinux help?

- Let's walk through it again with our SELinux-enabled Android.
- Using the initial example policy we developed.
  - Before we read about this vulnerability and exploit.
  - Just based on normal Android operation and policy development.

GingerBreak vs SELinux #1

- Identify the vold process.
  - /proc/net/netlink allowed by policy
  - /proc/pid/cmdline of other domains denied by policy
- Existing exploit would fail here.
- Let's assume exploit writer recodes it based on prior knowledge of target or some other means.

GingerBreak vs SELinux #2

- Identify addresses and values of interest.
  - /system/bin/vold denied by policy.
  - /system/lib/libc.so allowed by policy.
  - /etc/vold.fstab allowed by policy
  - /dev/log/main denied by policy.
- Existing exploit would fail here.
- Let's assume that exploit writer recodes exploit based on prior knowledge of target.

GingerBreak vs SELinux #3

- Send netlink message to vold process.
  - netlink socket create denied by policy
- Existing exploit would fail here.
- No way around this one - vulnerability can't be reached.
- Let's give the exploit writer a fighting chance and allow this permission.

GingerBreak vs SELinux #4

- Trigger execution of exploit code by vold.
  - execute of non-system binary denied by policy
- Existing exploit would fail here.
- Let's assume exploit writer recodes exploit to directly inject code or use ROP to avoid executing a separate binary.

GingerBreak vs SELinux #5

- Create a setuid-root shell.
  - remount of /data denied by policy
  - chown/chmod of file denied by policy
- Existing exploit would fail here.
- Let's give the exploit writer a fighting chance and allow these permissions.

GingerBreak vs SELinux #6

- Execute setuid-root shell.
  - SELinux security context doesn't change.
  - Still limited to same set of permissions.
  - No superuser capabilities allowed.
- Exploit “succeeded”, but didn't gain anything.

GingerBreak vs SELinux: Conclusion

- SELinux would have stopped the exploit six different ways.
- SELinux would have forced the exploit writer to tailor the exploit to the target.
- SELinux made the underlying vulnerability completely unreachable.
  - And all vulnerabilities of the same type.
  - Other vulnerabilities of the same type have been found, e.g. ueventd.

Case Study: ueventd

- ueventd - Android udev equivalent
  - Runs as root
  - Manages /dev directory
  - Receives netlink messages from the kernel
- Same vulnerability as CVE-2009-1185 for udev.
  - Does not verify message came from kernel.
- Demonstrated by the Exploid exploit.

Exploid vs SELinux

- Similar to GingerBreak scenario.
- Exploit would be completely blocked in at least two ways by SELinux:
  - creation/use of netlink socket by exploit
  - write to /proc/sys/kernel/hotplug by ueventd
- Vulnerability can't be reached.
- Exploit code can't be invoked with privilege.

Case Study: adbd

- adbd - Android debug bridge daemon
  - Runs as root
  - Provides debug interface
  - Switches to shell UID and executes shell.
- Does not check/handle setuid() failure.
  - Can lead to a shell running as root.
- Demonstrated by RageAgainstTheCage.

RageAgainstTheCage: Overview

- Look up adbd process in /proc.
- Fork self repeatedly to reach RLIMIT_NPROC for shell identity.
- Re-start adbd.
- adbd setuid() call fails.
- shell runs as root.

RageAgainstTheCage vs SELinux

- Look up and restart of adbd.
  - read /proc/pid/cmdline denied by policy
  - signal adbd denied by policy
- adbd setuid() would still fail.
- Security context changes upon exec of shell.
- Shell runs in unprivileged security context.
  - No superuser capabilities.
  - No privilege escalation achieved.

Case Study: zygote

- zygote - Android app spawner
  - Runs as root.
  - Receives requests to spawn apps over a socket.
  - Uses setuid() to switch to app UID.
- Does not check/handle setuid() failure.
  - Can lead to app running as root.
- Demonstrated by Zimperlich exploit.

Zimperlich: Overview

- Fork self repeatedly to reach RLIMIT_NPROC for app UID.
- Spawn app component via zygote.
- Zygote setuid() call fails.
- App runs with root UID.
  - Re-mounts /system read-write.
  - Creates setuid-root shell in /system.

Zimperlich vs SELinux

- Similar to RageAgainstTheCage scenario.
- zygote setuid() would still fail.
- Security context changes upon setcon().
  - Not affected by RLIMIT_NPROC.
- App runs in unprivileged security context.
  - No superuser capabilities.
  - No privilege escalation.

Case Study: ashmem

- ashmem - anonymous shared memory
  - Android-specific kernel subsystem
  - Used by init to implement shared mapping for system property space.
- CVE-2011-1149
  - Does not restrict changes to memory protections.
  - Actually two separate vulnerabilities in ashmem.
- Demonstrated by KillingInTheNameOf and psneuter exploits.

KillingInTheNameOf: Overview

- Change protections of system property space to allow writing.
- Modify ro.secure property value.
- Re-start adbd.
- Root shell via adb.

KillingInTheNameOf vs SELinux

- Changing memory protections of system property space.
  - performed via mprotect, already controlled by SELinux.
  - denied write to tmpfs by policy
- Exploit blocked.
  - Before it can do any harm.

psneuter: Overview

- Set protection mask to 0 (no access) on property space.
- Re-start adbd.
- adbd cannot read property space.
- Defaults to non-secure operation.
- Root shell via adb.

psneuter vs SELinux

- Set protection mask to 0 on property space.
  - ashmem-specific ioctl, not specifically controlled (yet) by SELinux
  - therefore allowed
- Re-start adbd.
  - read of /proc/pid/cmdline denied by policy.
  - signal to adbd denied by policy.
- Exploit blocked, but protection mask modified.
  - Points to need to instrument ashmem for SELinux.

Case Study: Skype for Android

- Skype app for Android.
- CVE-2011-1717
  - Stores sensitive user data without encryption with world readable permissions.
    - account balance, DOB, home address, contacts, chat logs, ...
- Any other app on the phone could read the user data.

SELinux vs Skype vulnerability

- Classic example of DAC vs. MAC.
  - DAC: Permissions are left to the discretion of each application.
  - MAC: Permissions are defined by the administrator and enforced for all applications.
- All apps denied read to files created by other apps.
  - Each app and its files have a unique SELinux category set.
  - App has no control over the categories on its files.

Was the Skype vulnerability an isolated incident?

- Lookout Mobile Security
- Symantec Norton Mobile Security
- Wells Fargo Mobile app
- Bank of America app
- USAA banking app

Application Layer Security

- So far we're only dealing with the kernel level access controls.
- To fully control the apps, we need SELinux integration with the application layer access controls.
- Requires further study of the existing Android security model.
- Requires SELinux instrumentation of the application frameworks.

SELinux & App Layer Security

- SELinux provides interfaces for application layer access control enforcement.
  - Extends security model to higher level objects and operations.
  - Provides same benefits of centralized, analyzable policy for system.
  - Provides infrastructure for caching, revocation, etc.
- Already leveraged by a number of applications, including Xorg, D-BUS, Postgres.

Conclusion

- Android security would benefit from SELinux.
  - In general, Android needs MAC.
  - In practice, SELinux would have stopped a number of published exploits for Android.
- There is still a lot of work to do to bring full SELinux enablement to Android.
- Get Involved!

Questions?

- Email: sds@tycho.nsa.gov