MABEL Technology

Android Security

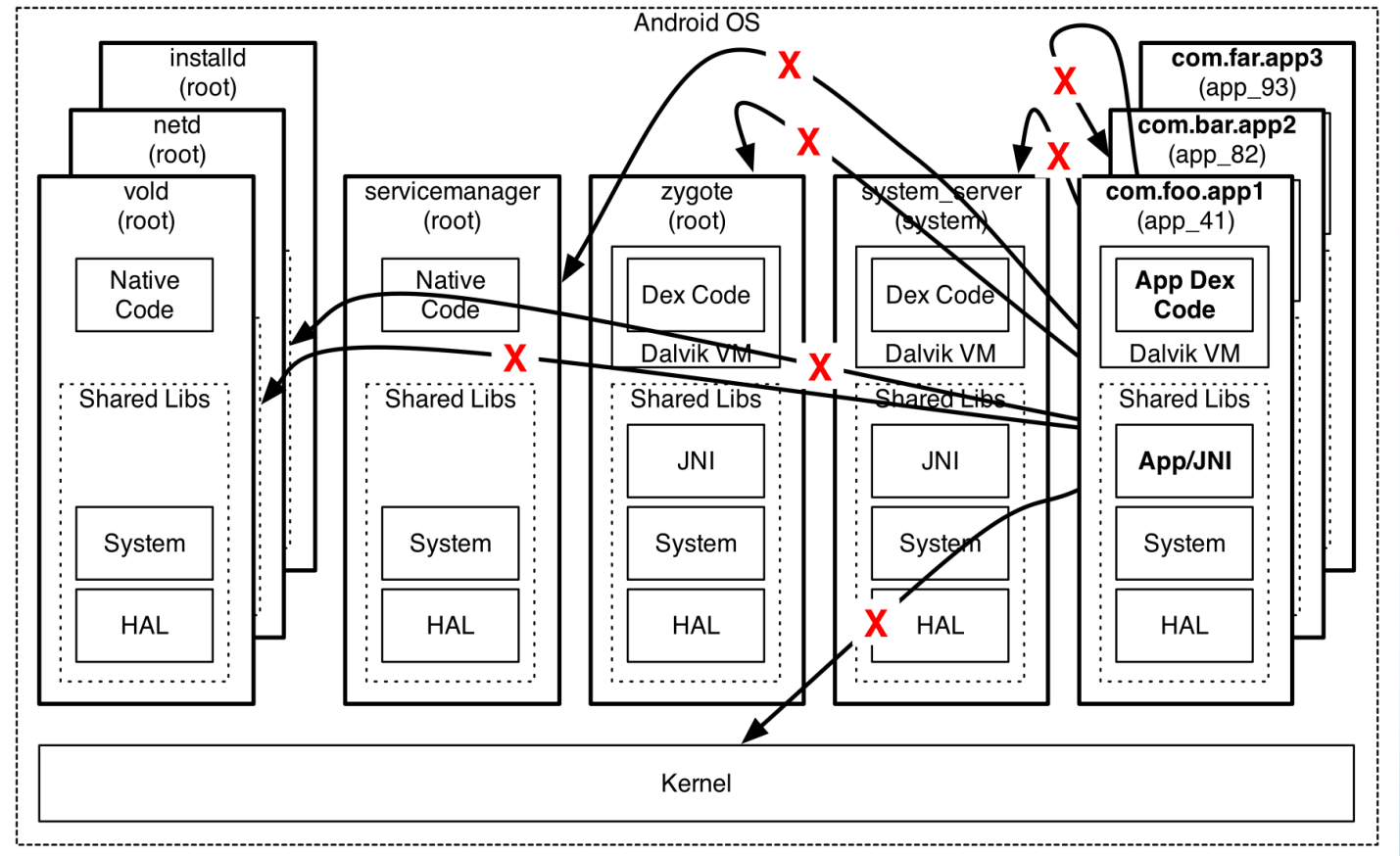
MABEL Technology

Foundations of Android Security

- Application Isolation and Permission-Control
  - Can we control what applica]ons are able to do?
  - Can a misbehaving app affect the rest of the system?
- Application "Provenance"
  - Can we trust the author of an app?
  - Can we trust our apps to be tamper-resistant?
- Data Encryption
  - Is our data safe if our device is hacked/lost/stolen?
- Device Access Control
  - Can we protect our device against unauthorized use?

MABEL Technology

Security Architecture



MABEL Technology

Android Application Isolation

- By default, each app runs in a separate process with a distinct user/group ID (fixed for the lifetime of the app)
  - Possible for multiple apps to share UID and process
  - Based on decades-old, well-understood UNIX security model (processes and file-system permissions)
- Application-framework services also run in a separate process (`system_server`)
- Linux kernel is the sole mechanism of app sandboxing
- Dalvik VM is not a security boundary
  - Coding in Java or C/C++ code – no difference
  - Enables use of JNI (unlike JavaME!)
- Same rules apply to system apps

Default Android Permissions Policy

- No app can do anything to adversely affect – Other apps
  - The system itself
  - The user of the device
- So, by default, apps cannot:
  - Read/write files outside their own directory
  - Install/uninstall/modify other apps
  - Use other apps' private components
  - Access network
  - Access user's data (contacts, SMS, email)
  - Use cost-sensitive APIs (make phone calls, send SMS, NFC)
  - Keep device awake, automatically start on boot, etc.

MABEL Technology

Escaping The Sandbox

- Actually, apps can talk to other apps via
  - Intents
  - IPC (a.k.a. Binder)
  - ContentProviders
- Otherwise, to escape our sandbox, we need to use permissions
  - Some permissions are only available to system apps

Built-in Android Permissions

- ACCESS_FINE_LOCATION, ACCESS_NETWORK_STATE,
- ACCESS_WIFI_STATE, ACCOUNT_MANAGER,
- BLUETOOTH, BRICK, CALL_PHONE, CAMERA,
- CHANGE_WIFI_STATE, DELETE_PACKAGES,
- INSTALL_PACKAGES, INTERNET, MANAGE_ACCOUNTS,
- MASTER_CLEAR, READ_CONTACTS, READ_LOGS,
- READ_SMS, RECEIVE_SMS, RECORD_AUDIO,
- SEND_SMS, VIBRATE, WAKE_LOCK, WRITE_CONTACTS,
- WRITE_SETTINGS, WRITE_SMS, ...
  http://developer.android.com/reference/android/Manifest.permission.html

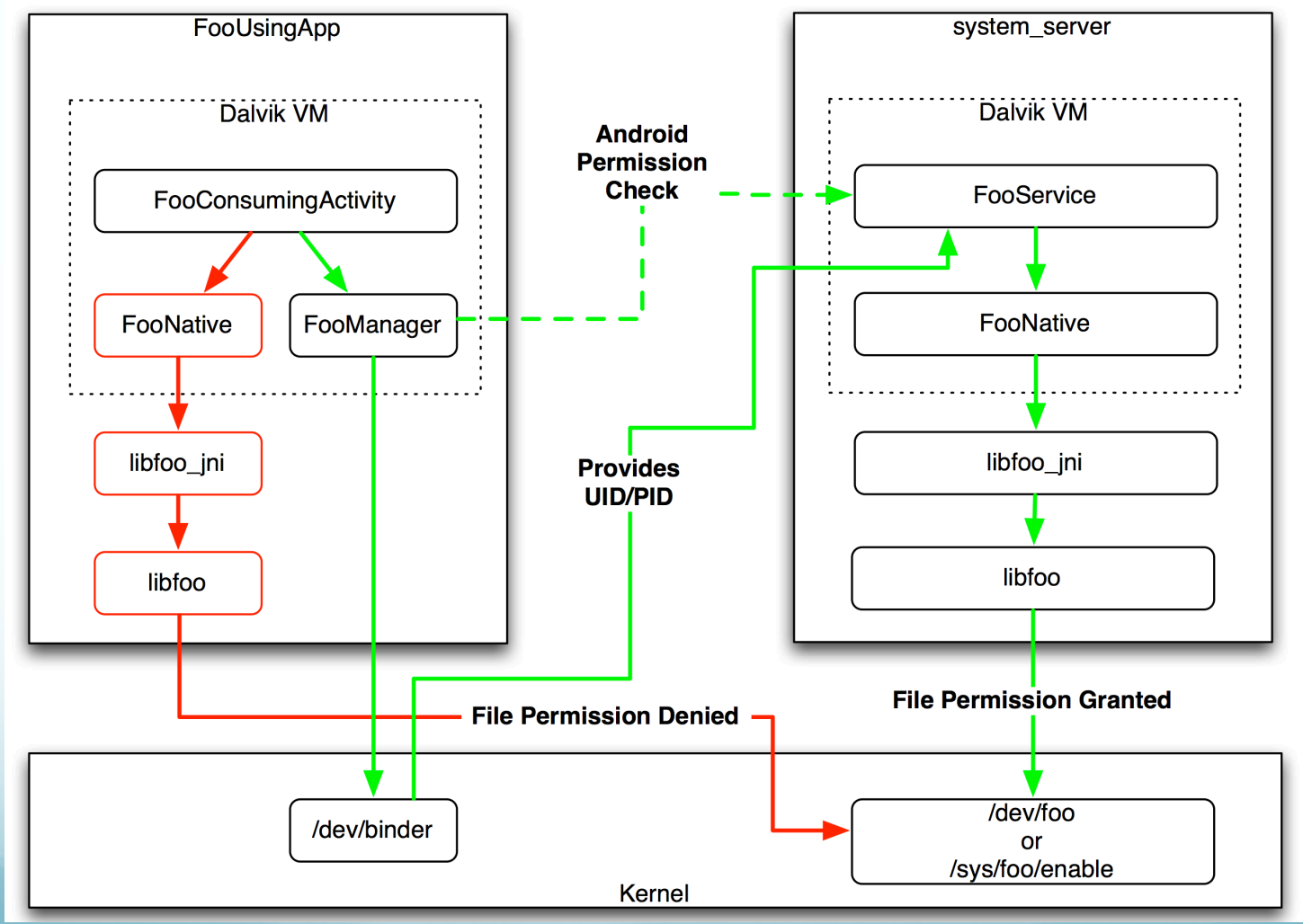
Example

- For example, an app that vibrates your phone any time you get in close vicinity to a friend would need to use at least the following permissions:

```
<manifest package="com.marakana.android.trackapp" …>
 <uses-permission
   android:name="android.permission.ACCESS_FINE_LOCATION"/>
 <uses-permission
   android:name="android.permission.INTERNET" />
 <uses-permission
   android:name="android.permission.VIBRATE" />
 …
</manifest>
```

MABEL Technology

Logical Permission Enforcement



MABEL Technology

Permission Enforcement Example

Only the `system` user (i.e. SS proc) can write to the vibrator driver:

```
$ adb shell ls -l /sys/class/timed_output/vibrator/enable
-rw-r--r-- system   system       4096 2011-09-30 23:23 enable
```

Only apps with `android.permission.VIBRATE` permissions can access `VibratorSevice.vibrate(…)` method:

`frameworks/base/services/java/com/android/server/VibratorService.java`

```
package com.android.server;
…
public class VibratorService extends IVibratorService.Stub {
  …
  public void vibrate(long milliseconds, IBinder token) {
    if (mContext.checkCallingOrSelfPermission(
          android.Manifest.permission.VIBRATE)
            != PackageManager.PERMISSION_GRANTED) {
      throw new SecurityException(
        "Requires VIBRATE permission");
    }
    …
  }
  …
}
```

Kernel Permission Enforcement

Some Android permissions directly map to group IDs, which are then enforced by the kernel/FS:

/system/etc/permissions/platform.xml:

```
<permissions>
  …
  <permission name="android.permission.INTERNET" >
   <group gid="inet" />
  </permission>
  <permission name="android.permission.CAMERA" >
    <group gid="camera" />
  </permission>
  <permission name="android.permission.READ_LOGS" >
    <group gid="log" />
  </permission>
  <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
    <group gid="sdcard_rw" />
  </permission>
  …
</permissions>
```

Interesting example: `android.permission.INTERNET -> inet -> 3003 -> ANDROID_PARANOID_NETWORK` (kernel patch)

Application Provenance

- Can we trust the developer of an applica]on we are about to install? (mostly, no)
- Can we trust that our apps are resistant to tampering once installed? (mostly, yes)
- To get onto Android Market, a developer just needs to register with Google and pay $25 with a valid credit card
  - A mild deterrent against authors of malicious apps
- Apps can also be side-loaded

Application Provenance (Signing)

- All apps (`.apk` files) must be digitally signed prior to installation on a device (and uploading to Android Market)
- The embedded certificate can be self-signed (no CA needed!) and valid for 25+ years
- App signing on Android is used to:
  - Ensure the authenticity of the author on updates
  - Establish trust relationship among apps signed with the same key (share permissions, UID, process)
  - Make app contents tamper-resistant (moot point)
- An app can be signed with multiple keys

MABEL Technology

Application Provenance (Signing)

- Lost/expired key? No way to update the app(s)
- Stolen key? No way to revoke
- How do we trust the author on the first install?
  - Is this the real author, or an imposter? Can I check the cert?
  - Has this app been vetted?
  - Go by the number of installs?
    - Follow the sheep?

Application Provenance (Signing)

- The result?
  - Android.Rootcager
  - Android.Pjapps
  - Android.Bgserv
- All took advantage of weak trust relationship
  - Take an existing (popular) app
  - Inject malicious code (e.g. a trojan)
  - Re-package and re-sign with a new key/cert
  - Upload to market (or distribute via web)
  - Wait for the "sheep" to come (not really our fault)

MABEL Technology

Safeguarding Apps' Data

- Apps' files are private by default
  - Owned by distinct apps' UIDs
- Exceptions
  - Apps can create files that are
    - `MODE_WORLD_READABLE`
    - `MODE_WORLD_WRITABLE`
  - Other apps (signed with the same key) can run with the same UID – thereby gexng access to shared files
  - `/mnt/sdcard` is world-readable and world-writable (with `WRITE_TO_EXTERNAL_STORAGE`)

MABEL Technology

Data Encryption

- ☑ VPN (IPSEC) with 3DES and AES and cert auth.
  - ☑ VPN Client API available as of ICS/4.0
- ☑ 802.11 with WPA/2 and cert auth.
- ☑ OpenSSL
- ☑ JCE (based on BouncyCastle provider)
  - ☑ Apache HTTP Client (supporting SSL)
  - ☑ java.net.HttpsUrlConnection
  - – Using encryption well is non-trivial (e.g. IV)
  - – Does not help if the key is stored on the device
- ☑ Keychain API – apps can install and store user certificates and CAs securely as of ICS/4.0
- ☑ Whole-disk encryption (requires >= 3.0)

Data Encryption

- Privacy and integrity of data can be achieved using encryption
- Data being transported off device is usually encrypted via TLS/SSL, which Android supports
  - At the native level - via OpenSSL (`/system/lib/libssl.so`)
  - At the Java level - using Java Cryptography Extension (JCE), which on Android is implicitly provided by BouncyCastle provider
  - For example, for HTTPS with client-side authentication, we could use HttpsURLConnection:

To Encrypt/Decrypt byte array

```
byte[] secret = // some secret
byte[] plainText = // some plain text
Key key = CryptUtil.getKey(secret, true);
Cipher cipher = CryptUtil.getEncryptCipher(key);
byte[] iv = CryptUtil.getIv(cipher);
byte[] encrypted = cipher.doFinal(plainText);
// store iv and encrypted
```

```
byte[] secret = // some secret
byte[] encrypted = // read encrypted buffer from somewhere
byte[] iv = // read iv from somewhere
Key key = CryptUtil.getKey(secret, true);
Cipher cipher = CryptUtil.getDecryptCipher(key, iv);
byte[] plainText = cipher.doFinal(encrypted);
```

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”.

MABEL Technology

Capabilities

installd.c – drop_privileges

```
capdata[CAP_TO_INDEX(CAP_DAC_OVERRIDE)].permitted |= CAP_TO_MASK(CAP_DAC_OVERRIDE);
capdata[CAP_TO_INDEX(CAP_CHOWN)].permitted        |= CAP_TO_MASK(CAP_CHOWN);
capdata[CAP_TO_INDEX(CAP_SETUID)].permitted       |= CAP_TO_MASK(CAP_SETUID);
capdata[CAP_TO_INDEX(CAP_SETGID)].permitted       |= CAP_TO_MASK(CAP_SETGID);
capdata[CAP_TO_INDEX(CAP_FOWNER)].permitted       |= CAP_TO_MASK(CAP_FOWNER);

capdata[0].effective = capdata[0].permitted;
capdata[1].effective = capdata[1].permitted;
capdata[0].inheritable = 0;
capdata[1].inheritable = 0;

if (capset(&capheader, &capdata[0]) < 0) {
    ALOGE("capset failed: %s\n", strerror(errno));
    exit(1);
}
```

See man 7 capabilities

SELinux

- Mandatory Access Control (MAC) for Linux.
  - 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.

SELinux in Android

- Confine privileged daemons.
  - Protect from misuse.
  - Limit the damage that can be done via them.
- Sandbox and isolate apps.
  - Strongly separate apps from one another.
  - 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.
  - 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

Modes

- Disabled – no policy is loaded
- Permissive – logging only
- Enforcing – SE mode

- Use setenforce/getenforce commands

Labels

- User:Role:Type:Sensitivity
- The only part used in android is the type and it is evaluated using TE files
- Domains – subjects (process types)
- Types – objects (files, resources)
- Some file systems support security labels in inode level

Enabling SELinux

- Kernel configuration
  - Security options – NSA SELinux
  - File systems security labels (ext3, ext4, ..)
- Kernel command line
  - androidboot.selinux=permissive (enforcing)
- See /proc/filesystems:
  - selinuxfs

sepolicy

- The policy file:
  - How to label processes and objects with domains and types
  - How domains can interact with each other (IPC, Signals, ...)
  - How domains can access types
- Loaded by init from the root directory
  - /system/core/init/init.cpp
- Policy files and tools:
  - /system/sepolicy
- Generated file:
  - policy.conf
- Libraries
  - /external/selinux (libselinux, libsepol,...)
    - See Android modification in libselinux/src/android/
  - Java API
    - /frameworks/base/core/java/android/os/SELinux.java

Initial_sid_contexts

```
sid kernel u:r:kernel:s0
sid security u:object_r:kernel:s0
sid unlabeled u:object_r:unlabeled:s0
sid fs u:object_r:labeledfs:s0
sid file u:object_r:unlabeled:s0
sid file_labels u:object_r:unlabeled:s0
sid init u:object_r:unlabeled:s0
sid any_socket u:object_r:unlabeled:s0
sid port u:object_r:port:s0
sid netif u:object_r:netif:s0
```

Default label for objects

Example
If you create a file for example in a filesystem with no rules it will be labeled 'unlabeled'

fs_use

- Labels for filesystems
- fs_use_xattr:
  - FS with security labels

```
# Label inodes via getxattr.
fs_use_xattr yaffs2 u:object_r:labeledfs:s0;
fs_use_xattr jffs2 u:object_r:labeledfs:s0;
fs_use_xattr ext2 u:object_r:labeledfs:s0;
fs_use_xattr ext3 u:object_r:labeledfs:s0;
fs_use_xattr ext4 u:object_r:labeledfs:s0;
fs_use_xattr xfs u:object_r:labeledfs:s0;
fs_use_xattr btrfs u:object_r:labeledfs:s0;
fs_use_xattr f2fs u:object_r:labeledfs:s0;
fs_use_xattr squashfs u:object_r:labeledfs:s0;
```

fs_use

```
# Label inodes from task label.
fs_use_task pipefs u:object_r:pipefs:s0;
fs_use_task sockfs u:object_r:sockfs:s0;
```

Pseudo file systems,
process creates a pipe(2) or socket(2)

fs_use

```
# Label inodes from combination of task label and fs label.
# Define type_transition rules if you want per-domain types.
fs_use_trans devpts u:object_r:devpts:s0;
fs_use_trans tmpfs u:object_r:tmpfs:s0;
fs_use_trans devtmpfs u:object_r:device:s0;
fs_use_trans shm u:object_r:shm:s0;
fs_use_trans mqueue u:object_r:mqueue:s0;
```

For pseudo file systems
Combined with type_transition rules in TE files for example (drmserver.te):

type_transition drmserver apk_data_file:sock_file drmserver_socket;

This rule state that when a process with type drmserver create an object of sock_file type in a filesystem labeled apk_data_file , the resoult object should be labeled drmserver_socket

MABEL Technology

genfs_contexts

```
# Label inodes with the fs label.
genfscon rootfs / u:object_r:rootfs:s0
# proc labeling can be further refined (longest matching prefix).
genfscon proc / u:object_r:proc:s0
genfscon proc /config.gz u:object_r:config_gz:s0
genfscon proc /interrupts u:object_r:proc_interrupts:s0
genfscon proc /iomem u:object_r:proc_iomem:s0
genfscon proc /meminfo u:object_r:proc_meminfo:s0
genfscon proc /misc u:object_r:proc_misc:s0
genfscon proc /modules u:object_r:proc_modules:s0
genfscon proc /net u:object_r:proc_net:s0
```

Rules for labeling files in FS without xattr (security labels)

Test:
adb shell
mount –o remount,rw /
touch testfile
ls –Z

-rw-rw-rw- root root u:object_r:rootfs:s0 testfile

file_contexts

- Security labels for files in:
  - /
  - /system
  - /dev
  - /data
  - /vendor
  - ...

Mounting options

- You can also mount file system with a context:

mount -ocontext=u:object_r:system_file:s0 -t ramfs ramfs /ramfs

mount -ocontext=u:object_r:vfat1:s0 /dev/block1 /mnt/vfat1

Attributes

```
# All types used for devices.
# On change, update CHECK_FC_ASSERT_ATTRS
# in tools/checkfc.c
attribute dev_type;

# All types used for processes.
attribute domain;

# All types used for filesystems.
# On change, update CHECK_FC_ASSERT_ATTRS
# definition in tools/checkfc.c.
attribute fs_type;

# All types used for context= mounts.
attribute contextmount_type;

# All types used for files that can exist on a labeled fs.
# Do not use for pseudo file types.
# On change, update CHECK_FC_ASSERT_ATTRS
# definition in tools/checkfc.c.
attribute file_type;

# All types used for domain entry points.
attribute exec_type;
```

Used by the code in /system/sepolicy/tools

mac_permissions.xml

- Mapping between application certificate and seinfo string
- Add LOCAL_CERTIFICATE := platform to Android.mk file to sign the app

Domain

- /system/sepolicy/**public**/platform_app.te

type platform_app, **domain**;

- /system/sepolicy/**private**/platform_app.te
  - All the rules
- seapp_contexts
  - Map application UID and (optionaly) seinfo to a domain for app and type for objects
  - user=_app seinfo=platform domain=platform_app type=app_data_file levelFrom=user

Services

- Declare service type in service.te

type user_service, app_api_service, ephemeral_app_api_service,

system_server_service, service_manager_type

- service_contexts:
- user u:object_r:user_service:s0

property_contexts

- Property security contexts
- Labels for **init** property service permission checks
- property.te
  - Properties types and rules

Enabling SELinux in Android: init

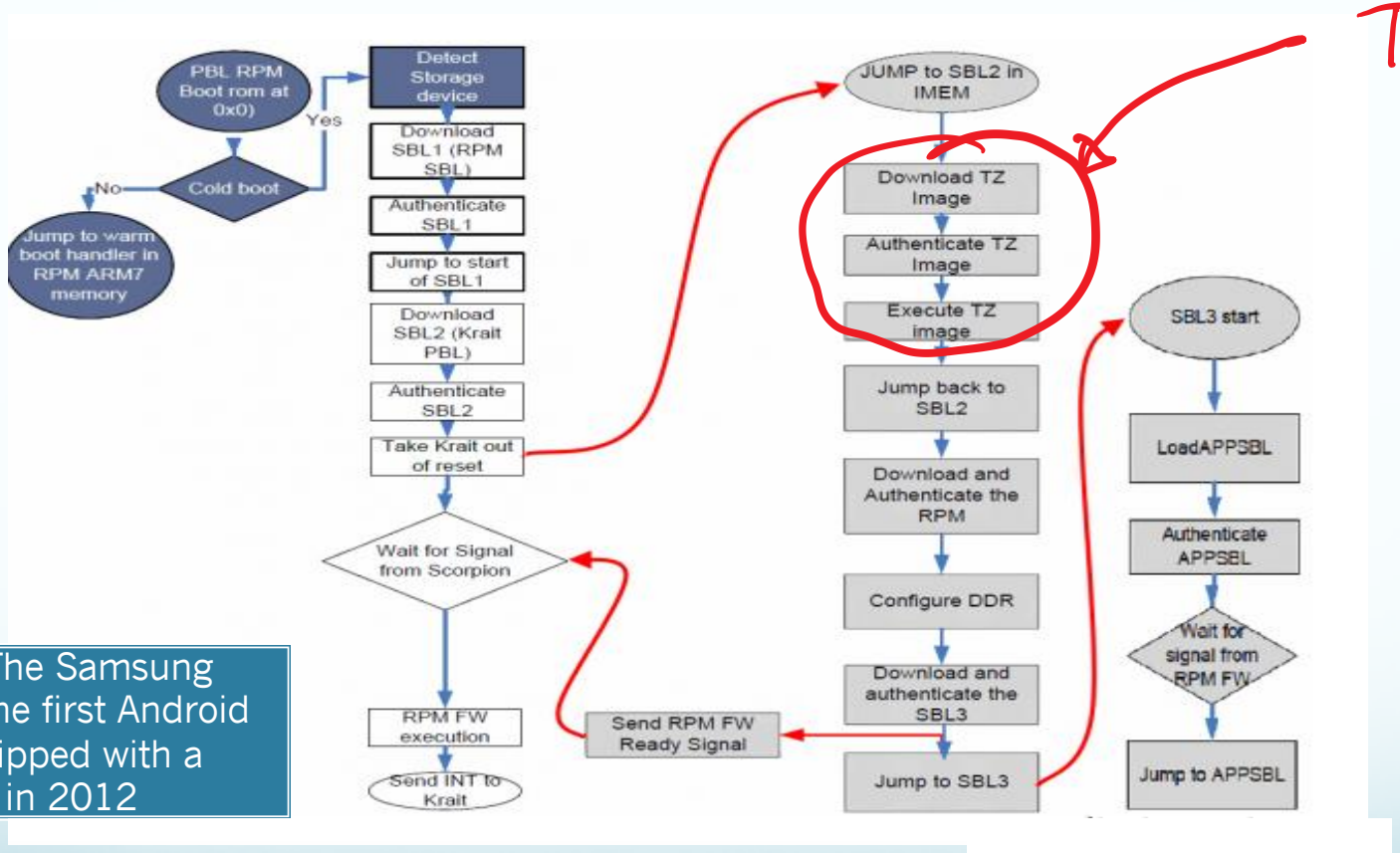
- init / ueventd
  - load policy, set enforcing mode, set context
  - label sockets, devices, runtime files
- init.rc
  - setcon, restorecon commands
  - seclabel option

MABEL Technology

Trusted Execution Environment

- The TEE is a separate secure execution environment
  - Shares the CPU with the main device but can time-slice (has its own MMU)
- **Available on 99% of 2017 Android devices** as part of the ARM SoC (and now mandatory 7+)
- Use is to run code securely outside of the normal (AOSP) environment

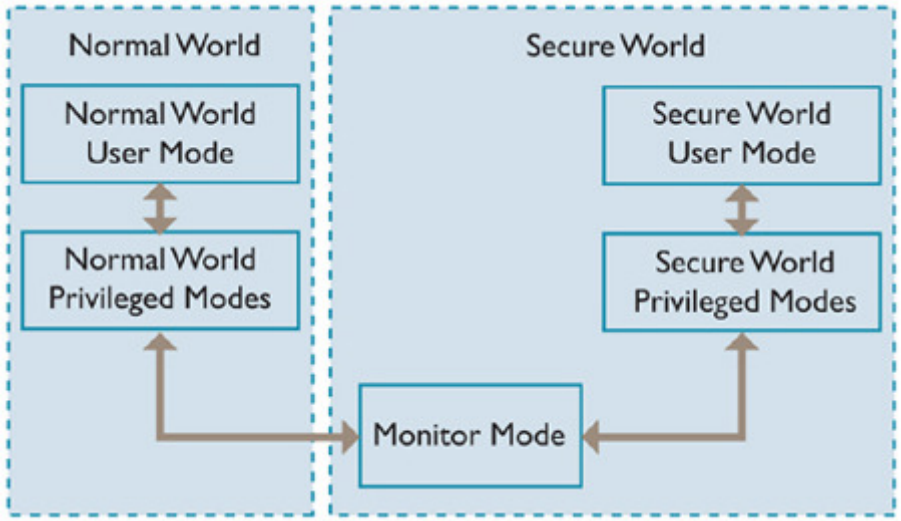
How does the TEE fit into our (SIII) boot sequence?



Factoid: The Samsung SIII was the first Android device shipped with a TEE back in 2012

MABEL Technology

Architecture



AOSP (NWd)

TEE/TrustZone/SWd

- Android devices have two Worlds
- Known as Normal World (NWd) and Secure World (SWd)
- They are completely isolated from one another
- Monitor acts as a “gatekeeper”
- Usually we use an API call to communicate from NWd→SWd

TrustZone Uses

- Assist Secure Boot/Verified Boot (hold keys/flags in QFuses)
- SoC Crypto Accelerator – hidden crypto work
- R/T Kernel Integrity checks (Samsung KNOX TIMA)
- Carrier locks on the baseband
- Programs – known in TrustZone as “Trustlets”
- Private key/data storage
- DRM
- Basically anything you want hidden from the NWd

MABEL Technology

ARM Processor Modes

- "Normal" CPU has User and Supervisor Mode (Rings 0/3)
- ARM actually has 4 modes (Exception Levels – i.e. changed during enter/return from an exception)

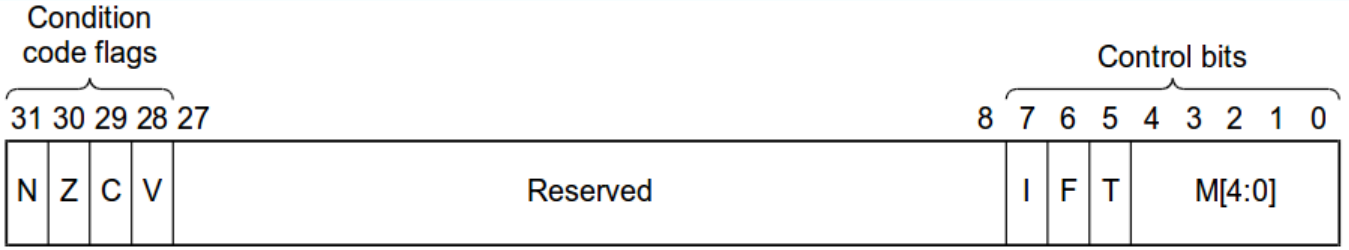
  - **EL0** Normal User (Android User/Secure User)
  - **EL1** Kernel (Android Kernel/Secure User)
  - *EL2* *Hypervisor (not used)*
  - **EL3** TZ (Default Power-up/Reset state)

  These states indicate the ARM CPU privilege mode.

MABEL Technology

ARM Modes

- **Current Process Status Register = “Classic Mode” Reg.**
- 32-bit register – amongst other things indicates current mode

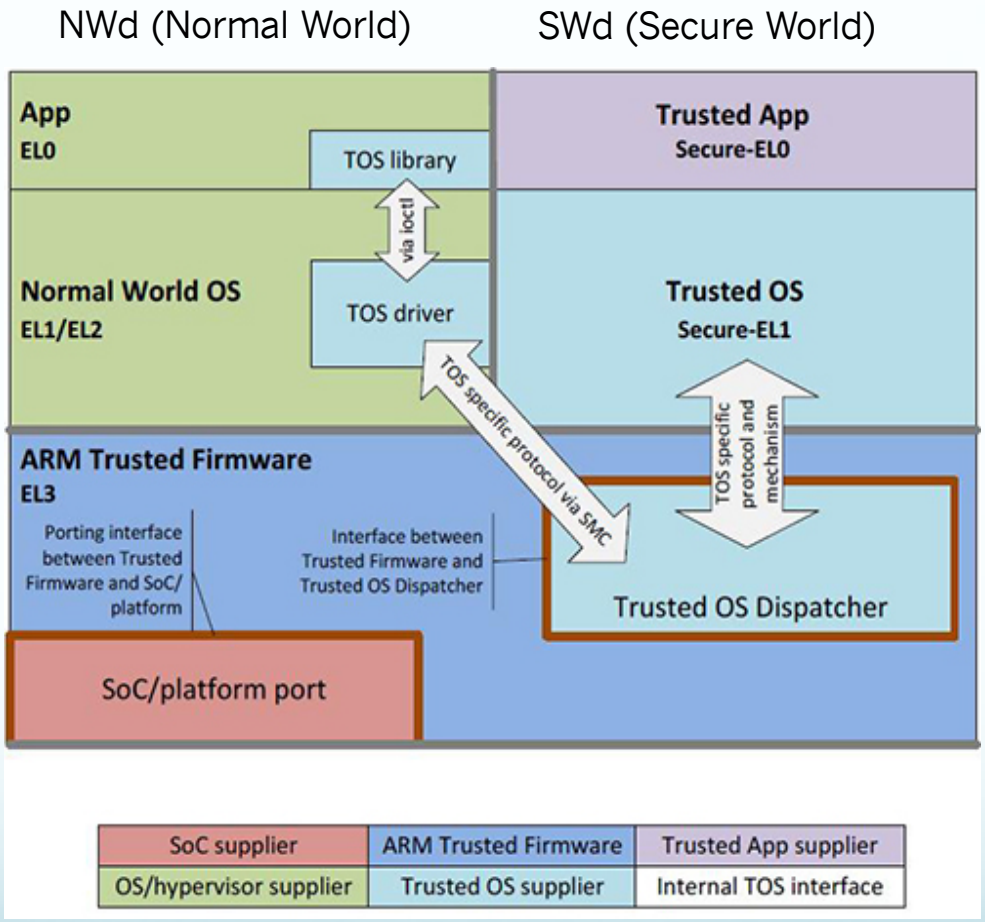


- Determines User/Supervisor mode
- user:b10000 svc:b10011 (need to be SVC for TEE operations)

- **Secure Configuration Register (SCR)**
- 32-bit register use **Non-Secure bit (NS)**
- NS bit determines Secure/Normal World
- To use must be in (NWd) Kernel mode!
- Handled by libraries when we are calling from higher level abstractions – really only interesting at ARM level

Figure 3.29. Secure Configuration Register format

[0]	NS bit	Defines the world for the processor: 0 = Secure, reset value 1 = Non-secure.



ARM Conceptual
EL0 → EL1 → EL3 → Secure-EL1 → Secure EL-0

Trustlets

- Dominant by far in the Android/Qualcomm TEE universe (Google have “trusty OS” nowhere to be seen)
- Software environment is known as **TrustZone**
- TrustZone runs on the TEE
- TrustZone can run programs known as “**Trustlets**” (.tlbin) – analogous to AOSP executables
- Example: www.trustonic.com

Loading Trustlets

- Usually located at system/vendor/firmware
- Loaded by secondary boot loader (secured)

MABEL Technology

Build Trustlets

- We can't use Java – there are no bindings (unlikely in near future)
- The **supported languages are C** (and C++)
- This can be via an autonomous executable on the machine or via the NDK (we simply use JNI to glue the C)
- We call the TrustZone via **qseecom** (Qualcomm Secure Execution Environment) SCM device driver
- Trustonic have an SDK (time limited and bound by your PGP key) available – NDA required as IP issues – and export licence (to RU)!!!
- >=3.10 Kernel have built-in TZ support

MABEL Technology

Start a trustlet

QSEECom_start_app call from the libQSEECom.so library

```
 * @param[in/out] handle The device handle
 * @param[in] fname The directory and filename to load.
 * @param[in] sb_size Size of the shared buffer memory  for sending requests.
 * @return Zero on success, negative on failure. errno will be set on
 *  error.
 */
int QSEECom_start_app(struct QSEECom_handle **clnt_handle, const char *path,
                      const char *fname, uint32_t sb_size);
```

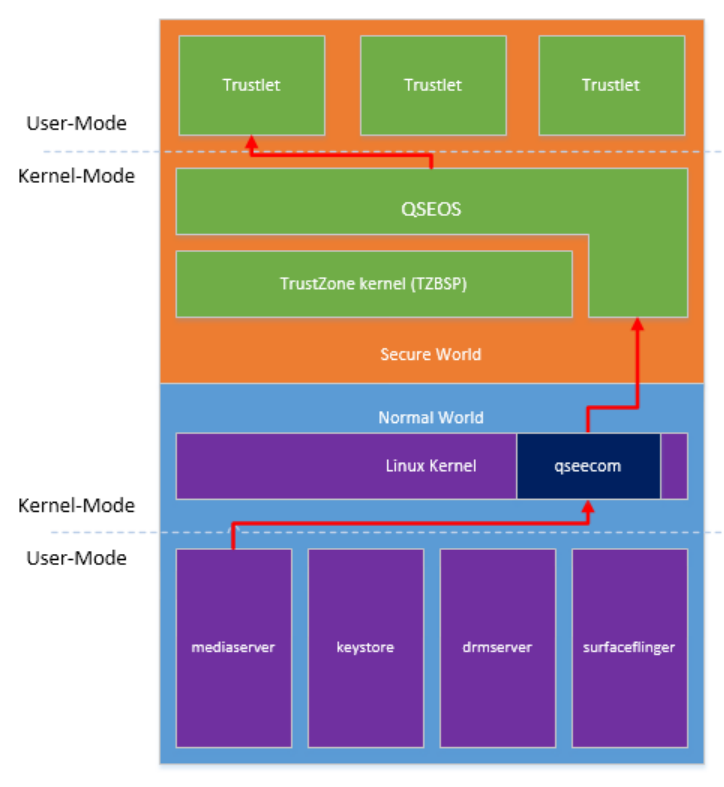
Client handle (unsigned char buffer*), pointer to path and filename (of the Trustlet in Android file space), buffer size to allocate for Trustlet I/O

Returns 0 success, negative error (err no).

* Handle to the qseecom device for kernel clients

MABEL Technology

AOSP and Trustlets



Secure World Overview

- Very few Android Processes have access to the qseecom driver (used to access the Secure World) – and they are secured by a GID
  - Mediaserver (indexes device media)
  - keystore (handles crypto keys) → keymaster trustlet
  - drmserver (manages drm) → widevine trustlet
  - surfaceflinger (handles buffer for screen writes)

MABEL Technology

Keymaster

- Allows storage of crypto material securely through a hardware backed device (the TEE)
  - Keystore provided digital signing and verification operations, plus generation and import of asymmetric signing key pairs.
- *No sensitive user space operations* – all must be done on the TEE
- Access via an OEM-provided, dynamically-loadable library used by the Keystore service (using framework services and Keystore daemon) accessible from Java