Android Security Underpinnings

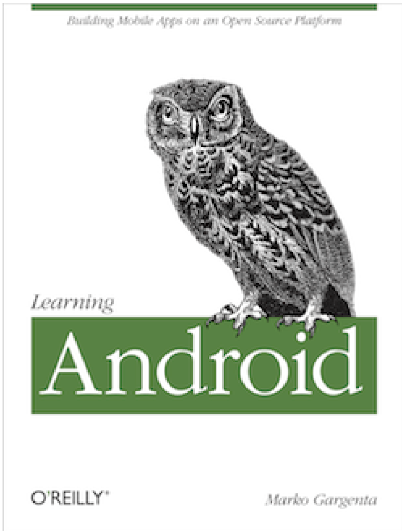
Press ➡ or space to move to the next page. Press 'h' for help.

Agenda

- Android Stack
- Security Essentials
  - Security Architecture
  - Application Signing
  - User IDs
  - File Access
  - Multiuser Support
  - Permissions
- Advanced Security
  - Encryption
  - Rooting
  - Device Admin
  - Malware
  - SE Android
  - Other Security Concerns

Slides and screencast from this class will be posted to: http://mrkn.co/cgcsn

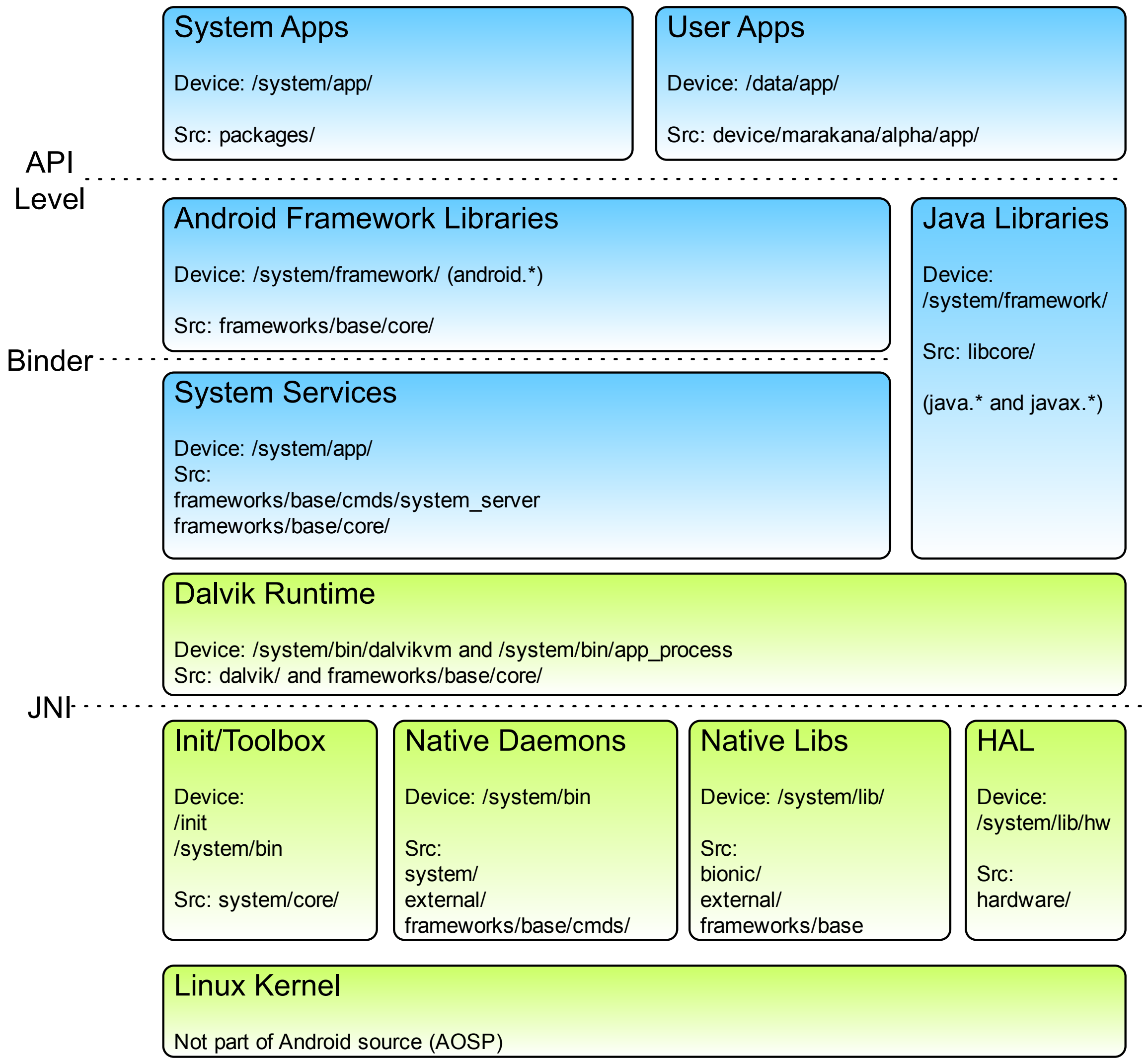
About Marko Gargenta



Entrepreneur, Author, Speaker

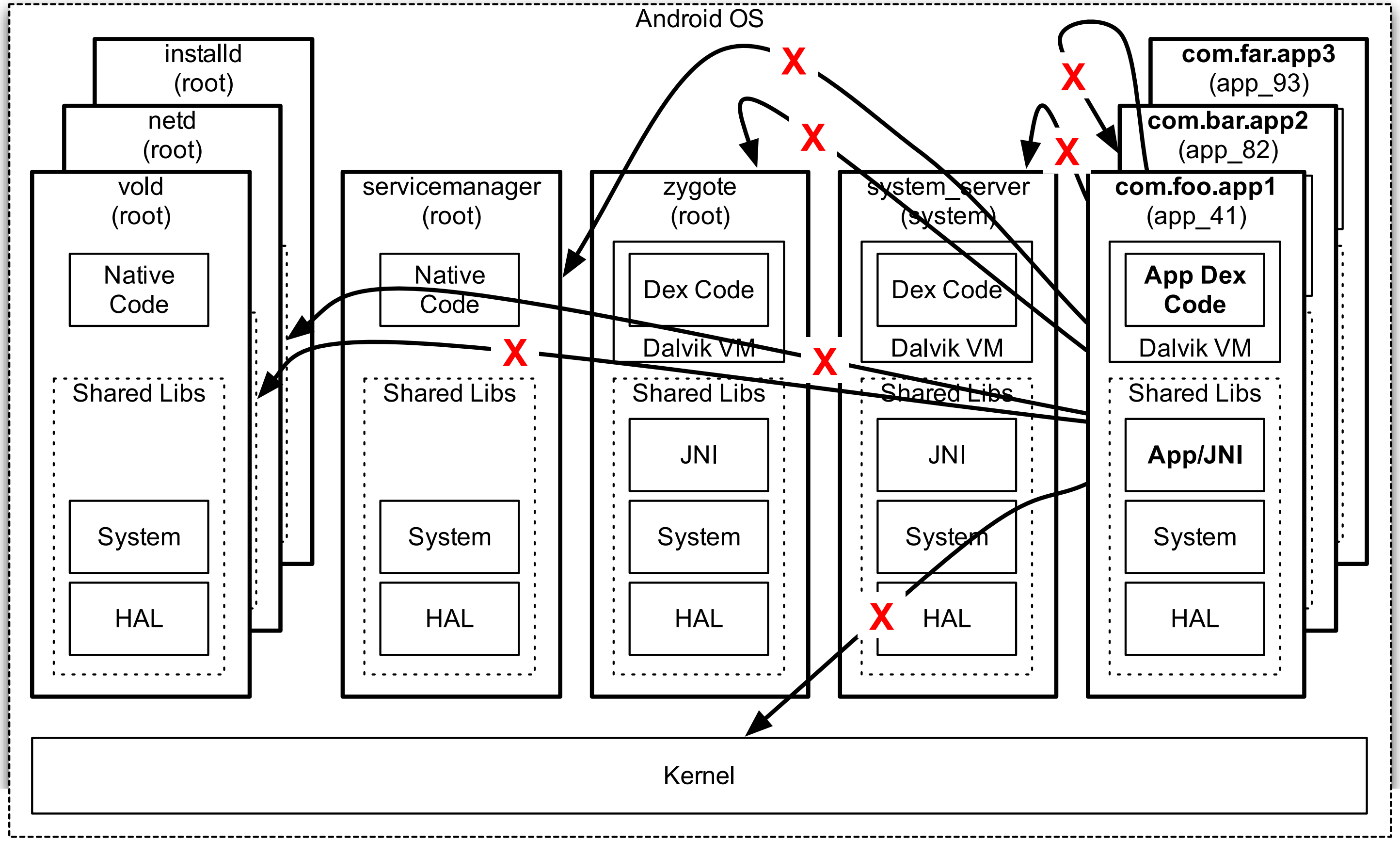
- Developer of Android Bootcamp for Marakana.
- Instructor for 1,000s of developers on Android at Qualcomm, Cisco, Motorola, Intel, DoD and other great orgs.
- Author of Learning Android published by O'Reilly.
- Speaker at AnDevCon (3x), Andriod Builder Summit (3x), OSCON (4x), DroidCon, MobileTechCon, ACM, IEEE, SDC, etc.
- Co-Founder of SFAndroid.org
- Co-Chair of Android Open conference: Android Open

Android Stack



Android Stack Details

Android Security Architecture



- Android is privilege-separated operating system
  - Each app runs with a separate process with a distinct system identity (user/group ID), as do parts of the system
  - OS (Linux) provides app isolation (sandboxing)

- By default, no app can do anything to adversely affect other apps, the system, or the user

    - E.g. reading/writing user data, modifying other apps'/system' files and settings, accessing network, keeping the device awake, etc.

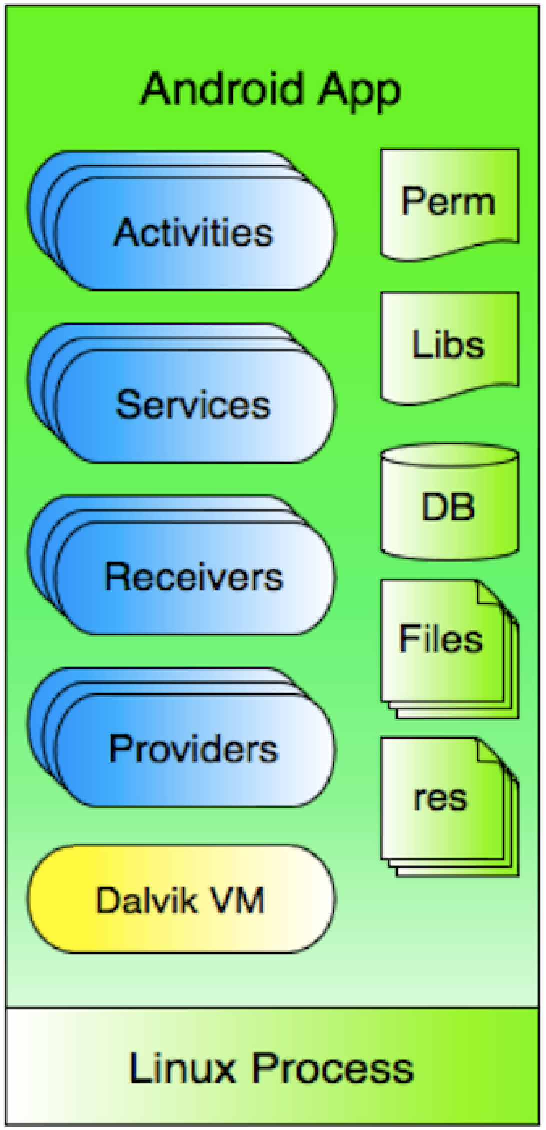
- Android provides fine-grained permission system that restricts what apps can do if they want to play outside the sandbox
    - Apps statically declare permissions they need (use)
    - The Android system prompts the user for consent at app installation-time (and on update if changed)
    - No support for dynamic (run-time) granting of permissions (complicates user experience)
    - The responsibility lies with the user to make educated decisions as to which apps to install based on the list of requested permissions
    - Prone confused deputy attacks where a non-privileged malicious app exploits a vulnerable interface of another privileged (but "confused") application

- Apps can explicitly share resources/data, via ContentProviders, Intents, Binder/IPC, local sockets, or the file system
    - This sharing goes un-checked by Android
    - The entire system prone to collusion attacks where malicious applications can collude to combine their permissions, allowing them to perform actions beyond their individual privileges

- Linux kernel is the sole mechanism of application (i.e. process) sandboxing
    - Dalvik VM is not a security boundary
    - Native code (via NDK) is subject to the same restrictions (i.e. no extra privileges)

- All apps are created equal
    - Sandboxed in the same way
    - Same level of security from each other

Application Sandboxing



Sandboxing

Application Signing

- All apps (`.apk` files) *must be* digitally signed prior to installation on a device with a certificate whose private key is kept confidential by the developer of the application
- Android uses the certificate as a means of:
  - Identifying the author of an application
    - Used to ensure the authenticity of future application updates
  - Establishing trust relationships between applications
    - Applications signed with the same key (i.e. share the certificate) can share signature-level permissions, user-id (file system resources), and runtime process
- The signing process is based on public-key cryptography, as defined by Java's JAR specification
- By default, when we Run As → Android Application our apps, they get automatically signed by Eclipse, using the debug key
  - Typically located at `~/.android/debug.keystore`
  - Automatically gets regenerated if we delete it
  - We can change this key from Eclipse/ADT → Preferences → Android → Build → Default Debug keystore
  - For obvious security reasons, we cannot upload apps signed using the debug key to the Android Market (a.k.a. Google Play store)
- To create a public-private key pair, we can use Java's `keytool` command:

```
$ keytool -genkey -v -keystore marakana.keystore -alias android -keyalg RSA -keysize 2048 -validity 10000 \
  -dname "CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US"
Enter keystore password:
Re-enter new password:
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
    for: CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US
Enter key password for <android>
    (RETURN if same as keystore password):
New certificate (self-signed):
…
```

The certificate can be self-signed - **no trusted 3rd party (i.e. certification authority) is required**.

The certificate's validity period should be **25 years** or longer. In fact, it must be valid until at least October 22, 2033 in order to upload apps signed by this key to Google Play store.

The certificate validity is only verified during the installation/update process (not at startup) - so apps with expired certificates would continue to function normally, but could not be updated.

The private key and the password used to encrypt it **must** be kept confidential! Do not use `–storepass` and/or `-keypass` command-line options when generating or using your key.

If our key is compromised, it could be used to sign and distribute:

- New versions of our apps that replace the existing/authentic installations and introduce malicious behavior
- New malicious apps that attack our existing apps, other apps, or the system itself

There is no official mechanism for revoking compromised keys; though Google does maintain its own private certificate revocation list.

Android's ADT Export Wizard for Eclipse (Project → Android Tools → Export Signed Application Package…) handles key creation automatically

- To view the contents of a keystore file (for a given alias), we can also use `keytool`:

```
$ keytool -list -v -keystore marakana.keystore -alias android
Enter keystore password:
Alias name: android
Creation date: Nov 11, 2012
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US
Issuer: CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US
Serial number: 3229e4c2
Valid from: Sun Nov 11 20:26:13 PST 2012 until: Thu Mar 29 21:26:13 PDT 2040
…
```

- Before we can sign our own apps with our custom key-pair, we should re/build our APKs without any debug keys: Eclipse/ADT → Project → Android Tools → Export Unsigned Application Package…
- Now we can sign our apps using Java's `jarsigner` command:

```
$ jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore marakana.keystore HelloAndroid.apk android
Enter Passphrase for keystore:
   adding: META-INF/ANDROID.SF
   adding: META-INF/ANDROID.RSA
  signing: res/layout/scan.xml
  signing: AndroidManifest.xml
  signing: resources.arsc
  signing: res/drawable-hdpi/ic_launcher.png
  signing: res/drawable-ldpi/ic_launcher.png
  signing: res/drawable-mdpi/ic_launcher.png
  signing: classes.dex
```

Prior to JDK 7, we did not need to specify `-sigalg` and `-digestalg` command-line switches. We can obviously omit asking for `-verbose` output.

Android's ADT Export Wizard for Eclipse (Project → Android Tools → Export Signed Application Package…) also handles signing automatically

About Signed APK (JAR) Files

When we sign an APK (JAR-based) archive, every file entry, except for signature related files, will be signed individually. The signature related files are:

- `META-INF/MANIFEST.MF`
- `META-INF/*.SF`
- `META-INF/*.DSA`
- `META-INF/*.RSA`
- `META-INF/SIG-*`

A signed APK file is exactly the same as the original APK file, except that:

- Its `META-INF/MANIFEST.MF` is updated: an entry is added for each signed file, along with a secure (e.g. SHA1) digest of that file
- A new signature file is added (`META-INF/ANDROID.SF` with `-alias android`), containing a secure digest of the manifest file (and its entries)

- A new signature block file is added (`META-INF/ANDROID.RSA` with `-alias android`), containing the digital signature for the JAR file that was generated with the private key, as well as the certificate, which itself contains the public key

We can examine the contents of the signature block file with the `keytool` command:

```
$ jar -xvf HelloAndroid.apk META-INF/ANDROID.RSA
$ keytool -printcert -file META-INF/ANDROID.RSA
Owner: CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US
Issuer: CN=Android Application Signer, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US
Serial number: 3229e4c2
…
```

More more info, see Understanding Signing and Verification

- Upon signing, we should `zipalign` our APK, so that it can be more efficiently memory-mapped by Android at runtime (yielding lower memory utilization):

```
$ zipalign -f 4 HelloAndroid.apk HelloAndroid-aligned.apk
$ mv HelloAndroid-aligned.apk HelloAndroid.apk
```

- Finally, we can verify that our steps worked correctly:

```
$ jarsigner -verify HelloAndroid.apk
jar verified.
```

If someone tampered with our APK, `jarsigner` would fail to `-verify` it.

- Android also allows a single app to be signed using multiple keys (each with a different alias):

```
$ keytool -genkey -v -keystore marakana2.keystore -alias android2 -keyalg RSA -keysize 2048 -validity 10000 \
   -dname "CN=Android Application Signer2, OU=Android, O=Marakana Inc., L=San Francisco, ST=California, C=US"
…
$ jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore marakana2.keystore multi/HelloAndroid.apk android2
Enter Passphrase for keystore:
   adding: META-INF/ANDROID2.SF
   adding: META-INF/ANDROID2.RSA
   …
```

Multi-signing apps is not common in Andorid. When used, it enables multiple different vendors to establish a trust to a "common" intermediary app that is signed by keys from different vendors.

Make sure that they keys are stored (and used) under different aliases.

- For more info see http://developer.android.com/guide/publishing/app-signing.html

Platform Keys

- AOSP comes with four platform keys in `build/target/product/security/`
  - `platform` - used to sign core parts of the system (configured with `LOCAL_CERTIFICATE := platform`): framework core, BackupRestoreConfirmation, DefaultContainerService, SettingsProvider, SharedStorageBackup, SystemUI, VpnDialogs, Bluetooth, CertInstaller, KeyChain, Nfc, PackageInstaller, Phone, Provision, Settings, Stk, TelephonyProvider, etc.
  - `shared` - used to sign home/contacts part of AOSP (configured with `LOCAL_CERTIFICATE := shared`): Contacts, Launcher2, QuickSearchBox, LatinIME, PinyinIME, ApplicationsProvider, ContactsProvider, UserDictionaryProvider, and built-in live wall papers
  - `media` - used to sign media/download framework parts of AOSP (configured with `LOCAL_CERTIFICATE := media`): Gallery, DownloadProvider, DrmProvider, MediaProvider
  - `testkey` - used to sign everything else - i.e. this is the default key for packages that don't specify one of the keys above

- The keys provided by default cannot be used to ship an actual product (enforced by CTS)
- For example, to generate our own keys, we can use the supplied `development/tools/make_key` command for each `platform`, `media`, `shared`, and `testkey`:
  1. `$ rm build/target/product/security/platform.p*`
  2. `$ SIGNER="/C=US/ST=California/L=San Francisco/O=Marakana Inc./OU=Android/CN=Android Platform Signer/emailAddress=android@marakana.com"`
  3. `$ echo | development/tools/make_key build/target/product/security/platform "$SIGNER"`

4. Repeat for `media`, `shared`, and `testkey`

- To sign our own app using the platform key:
  - If we are building it as part of the ROM, we can just add `LOCAL_CERTIFICATE := platform` to our app's `Android.mk` file
  - To sign our app outside the build process, assuming we have access to platform sources and keys, we could use the `signapk.jar` tool:

```
$ java -jar /path/to/aosp/out/host/linux-x86/framework/signapk.jar \
    /path/to/aosp/build/target/product/security/platform.x509.pem \
    /path/to/aosp/build/target/product/security/platform.pk8 \
    MyApp.apk MySignedApp.apk
```

User IDs

- Each app (package) is assigned an arbitrary, but distinct, OS user ID at installation time
    - Typically something like `app_XX` (e.g. `app_79`), where the actual UID is offset from `10000`
    - Does not change during app's life on a device
    - For example, the Browser app is assigned UID `10001` (though it could be different), which translates to `app_1`:

    ```
    $ adb -e shell cat /data/system/packages.list |grep com.android.browser
    com.android.browser 10001 0 /data/data/com.android.browser
    ```

- Each app process runs under its own UID:

  ```
  $ adb -e shell ps |grep com.android.browser
  app_1     682   37    192592 53144 ffffffff 40011384 S com.android.browser
  ```

- All app resources are owned by its UID

```
adb -e shell ls -l /data/data/com.android.browser
drwxrwx--x app_1    app_1             2012-02-06 12:47 app_appcache
drwxrwx--x app_1    app_1             2012-02-06 12:47 app_databases
drwxrwx--x app_1    app_1             2012-02-06 12:47 app_geolocation
drwxrwx--x app_1    app_1             2012-02-06 12:47 app_icons
drwxrwx--x app_1    app_1             2012-02-06 12:47 cache
drwxrwx--x app_1    app_1             2012-02-06 12:48 databases
drwxr-xr-x system   system            2012-01-17 15:42 lib
drwxrwx--x app_1    app_1             2012-02-06 12:47 shared_prefs
```

- Apps that are signed with the same certificate can share data, user ID, as well as run in a single process
    - They just need to specify the same `sharedUserId` and `process`

AndroidManifest.xml

```
<manifest package="com.marakana.android.myapp"
          android:sharedUserId="marakana.uid.myapp"
          android:sharedUserLabel="@string/myapp_uid" … >
  <application android:process="myapp" … >
    …
  </application>
</manifest>
```

From the security standpoint, packages with the same `sharedUserId` are treated as being parts of the same application, with the same UID and file permissions.

File Access

- Android has a number of different partitions:

```
$ adb shell mount
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
none /acct cgroup rw,relatime,cpuacct 0 0
tmpfs /mnt/secure tmpfs rw,relatime,mode=700 0 0
tmpfs /mnt/asec tmpfs rw,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,relatime,mode=755,gid=1000 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
/dev/block/platform/dw_mmc.0/by-name/system /system ext4 ro,relatime,data=ordered 0 0
/dev/block/platform/dw_mmc.0/by-name/cache /cache ext4 rw,nosuid,nodev,noatime,nomblk_io_submit,errors=panic,data=ordered 0 0
/dev/block/platform/dw_mmc.0/by-name/userdata /data ext4 rw,nosuid,nodev,noatime,nomblk_io_submit,errors=panic,data=ordered 0 0
/dev/block/platform/dw_mmc.0/by-name/efs /factory ext4 ro,nosuid,nodev,noatime,data=ordered 0 0
adb /dev/usb-ffs/adb functionfs rw,relatime 0 0
/sys/kernel/debug /sys/kernel/debug debugfs rw,relatime 0 0
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,default_permissions,allow_other 0 0
/dev/block/dm-0 /mnt/asec/com.zeptolab.ctr.paid-1 ext4 ro,dirsync,nosuid,nodev,noatime 0 0
…
```

- The notable ones include:
  - `/` - the read-only root file system, containing the `init` process plus some bootstrapping configuration files
  - `/system` - the read-only home of the Android OS, containing the system libraries (including HAL and application framework),

executables (daemons), fonts, media, and system apps

  - `/data` - the read-write file system containing user apps as well as application data (e.g. settings, preferences, etc) and system state information
  - `/cache` - the read-write file system containing transient user state (e.g. browser cache)

- Apps could be stored in several locations:
  - `/system/app/<App-Name>.apk` (for system apps)
    - The optimized dex code for this app would be stored as `/system/app/<App-Name>.odex`
    - When booted into safe-mode (by holding pre-configured keys pressed on power-on), Android will only make system apps available to the user
  - `/data/app/<App-Name>.apk` (for pre-loaded user apps)
    - The optimized dex code for this app would be stored as `/data/dalvik-cache/<App-Name>.odex`
  - `/data/app/<app-package-name>-1.apk` (for downloaded user apps)
    - The optimized dex code for this app would be stored as `/data/dalvik-cache/data@app@<app-package-name>-1.apk@classes.dex`
  - `/mnt/secure/asec/<app-package-name>-1.asec` (for apps moved to SD Card)
    - The optimized dex code for this app would be stored as `/data/dalvik-cache/mnt@asec@<app-package-name>-1@pkg.apk@classes.dex`

- To find out where a particular application is stored, we can ask the package manager:

```
$ adb shell pm path com.android.browser
package:/system/app/Browser.apk
```

- Files created by apps are owned by apps' distinct user/group ID and are not world-accessible
    - Stored under `/data/data/<app-package-name>/`

- Native libraries (generated by NDK) are copied to `/data/data/<app-package-name>/lib/lib<library-name>.so`
- Exceptions
    - `/mnt/sdcard` is FAT32, so free-for-all (though it requires `android.permission.WRITE_TO_EXTERNAL_STORAGE` permission)
    - Apps can create files, preferences, database with `MODE_WORLD_READABLE` and/or `MODE_WORLD_WRITABLE`, which affect world access, but not file ownership

Multi-User Support

- Android 4.2 adds support for multiple (human) users of tablet devices
  - The default setting only allows a single user:

    frameworks/base/core/res/res/values/config.xml:

    ```
    <resources …>
      …
      <integer name="config_multiuserMaximumUsers">1</integer>
      …
    </resources>
    ```

  - For tablets, this is often overridden to support up to 8 users:

    device/asus/grouper/overlay/frameworks/base/core/res/res/values/config.xml:

    ```
    <resources …>
      …
      <integer name="config_multiuserMaximumUsers">8</integer>
      …
    </resources>
    ```

- Users are created via Settings → Users → Add User
- Each user is assigned a unique user ID: 0, 10, 11, 12, … and profile info

  /data/system/users/userlist.xml:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<users nextSerialNumber="12" version="1">
<user id="0" />
<user id="10" />
<user id="11" />
</users>
```

/data/system/users/0.xml:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<user id="0" serialNumber="0" flags="19" created="0" lastLoggedIn="0" icon="/data/system/users/0/photo.png">
<name>Testuser</name>
</user>
```

/data/system/users/0/:

```
$ adb shell ls /data/system/users/0/
accounts.db
accounts.db-journal
appwidgets.xml
package-restrictions.xml
photo.png
wallpaper_info.xml
```

- When multiple users are enabled, apps get new effective UIDs, which are in the form of: `100000 x <user-id> + <app-user-id>`

  Another way to think of this, we just "concatenate" the two user IDs: `<user-id><app-user-id>`.
  The user-id offset of `100000` comes from `AID_USER` defined by `system/core/include/private/android_filesystem_config.h`.

  - `u0_a3` → `10003`

    - `u0_a4` → `10004`
    - ...
    - `u10_a3` → `1010003`
    - `u10_a4` → `1010004`
    - ...
    - `u11_a3` → `1110003`
    - `u11_a4` → `1110004`
    - ...

- Additionally, each user gets his/her own copy of all apps' data (including settings):

`/data/user/`:

```
$ adb shell ls -l /data/user/
lrwxrwxrwx root     root              2012-12-16 07:59 0 -> /data/data/
drwxrwx--x system   system            2012-12-17 10:53 10
drwxrwx--x system   system            2012-12-18 06:58 11
```

`/data/user/0/`:

```
$ adb shell ls -l /data/user/0/
drwxr-x--x u0_a39   u0_a39            2012-12-16 08:00 com.android.apps.tag
drwxr-x--x u0_a1    u0_a1             2012-12-16 07:59 com.android.backupconfirm
drwxr-x--x bluetooth bluetooth          2012-12-16 08:00 com.android.bluetooth
drwxr-x--x u0_a3    u0_a3             2012-12-16 07:59 com.android.browser
drwxr-x--x u0_a4    u0_a4             2012-12-16 07:59 com.android.calculator2
drwxr-x--x u0_a5    u0_a5             2012-12-16 08:00 com.android.calendar
drwxr-x--x u0_a7    u0_a7             2012-12-16 07:59 com.android.certinstaller
drwxr-x--x u0_a0    u0_a0             2012-12-16 08:01 com.android.contacts
…
drwxr-x--x system   system            2012-12-16 08:00 com.android.settings
…
```

/data/user/11/:

```
$ adb shell ls -l /data/user/11/
drwxr-x--x u11_system u11_system          2012-12-18 06:58 android
drwxr-x--x u11_a39  u11_a39           2012-12-18 06:58 com.android.apps.tag
drwxr-x--x u11_a1   u11_a1            2012-12-18 06:58 com.android.backupconfirm
drwxr-x--x u11_bluetooth u11_bluetooth          2012-12-18 06:58 com.android.bluetooth
drwxr-x--x u11_a3   u11_a3            2012-12-18 06:58 com.android.browser
drwxr-x--x u11_a4   u11_a4            2012-12-18 06:58 com.android.calculator2
drwxr-x--x u11_a5   u11_a5            2012-12-18 06:59 com.android.calendar
drwxr-x--x u11_a7   u11_a7            2012-12-18 06:58 com.android.certinstaller
drwxr-x--x u11_a0   u11_a0            2012-12-18 07:00 com.android.contacts
…
drwxr-x--x u11_system u11_system          2012-12-18 06:59 com.android.settings
…
```

- Each app runs in a separate process for each user (under a per-app, per-user effective user ID):

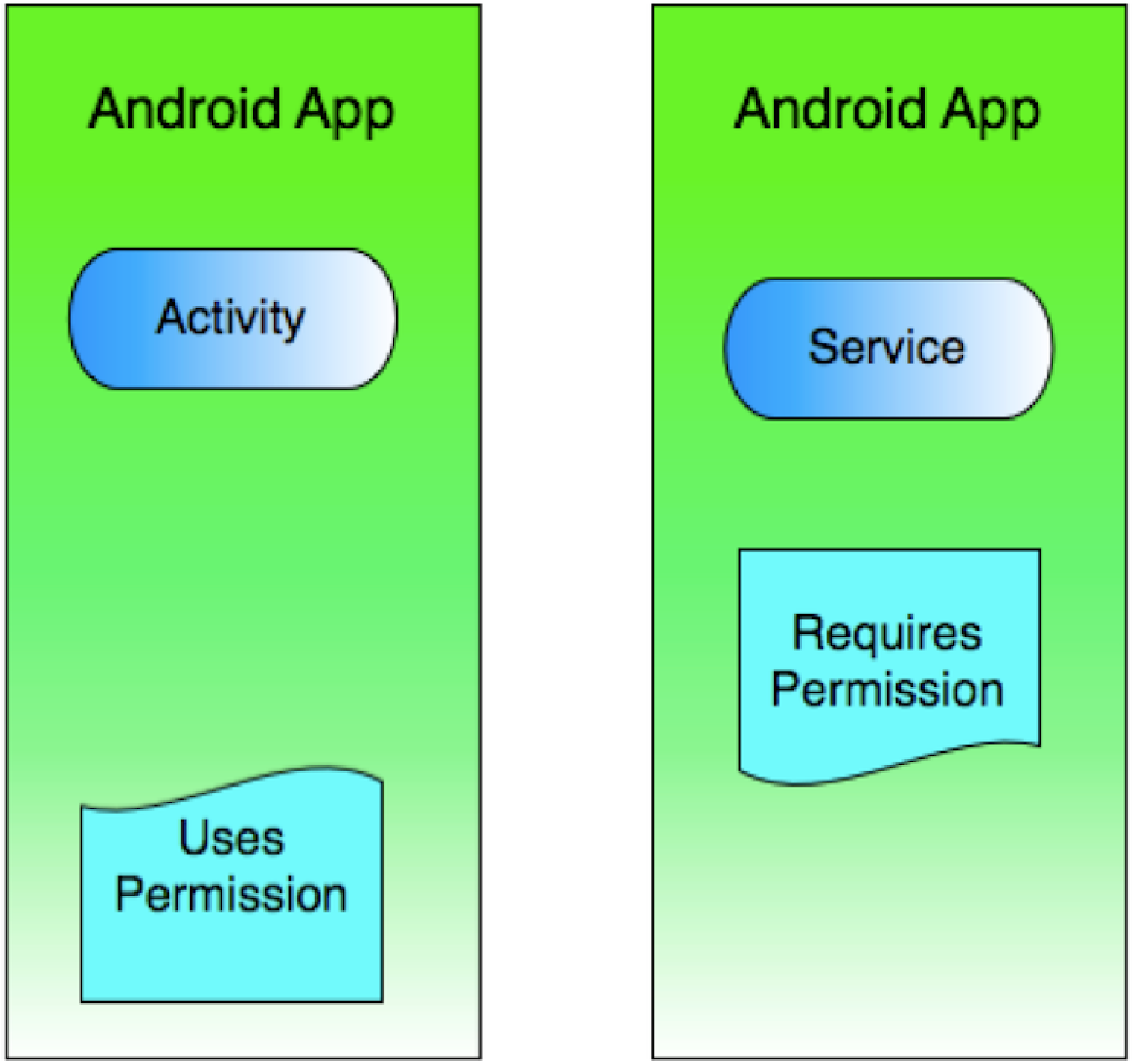
```
$ adb shell ps
u0_a38    466   129   495220 49988 ffffffff 40172ee4 S com.android.systemui
radio     585   129   489620 30712 ffffffff 40172ee4 S com.android.phone
nfc       596   129   489292 26824 ffffffff 40172ee4 S com.android.nfc
u0_a19    603   129   487792 40388 ffffffff 40172ee4 S com.android.launcher
…
u11_system 13883 129   517012 63408 ffffffff 40172ee4 S com.android.settings
u11_a19   13921 129   501672 60092 ffffffff 40172ee4 S com.android.launcher
u11_a18   13940 129   498656 52084 ffffffff 40172ee4 S com.android.inputmethod.latin
…
u11_a38   14051 129   477004 39404 ffffffff 40172ee4 S com.android.systemui
u11_a9    14068 129   470616 25560 ffffffff 40172ee4 S com.android.deskclock
…
u11_radio 14187 129   469228 24456 ffffffff 40172ee4 S com.android.phone
…
u0_a25    14361 129   466972 22672 ffffffff 40172ee4 S com.android.musicfx
u11_a25   14375 129   466972 22676 ffffffff 40172ee4 S com.android.musicfx
```

- To save on space apps are still installed only once under `/data/app/`
    - Apps’s optimized Dalvik code (i.e. `/data/dalvik-cache/`) is still shared
    - Apps' native lib directories (i.e. `/data/user/<user-id>/<package>/lib/`) are remapped to `/data/app-lib/<package>/` with symbolic links

```
$ adb shell ls -l /data/user/0/com.marakana.android.auldlangsyne/
drwxrwx--x u0_a45   u0_a45            2012-12-17 10:53 cache
lrwxrwxrwx install  install           2012-12-18 06:01 lib -> /data/app-lib/com.marakana.android.auldlangsyne-2
```

Permissions

Permissions



Permissions

Using Permissions

- By default, apps cannot do much outside their sandbox
- Attempts to access restricted resources without holding the appropriate permission
  - Fail with `SecurityException` (explicit failures)
    - For example, dialing a number (i.e. starting an activity with action `android.intent.action.CALL` for some data URI/phone-number) is restricted

```
$ adb shell run-as com.example.helloworld am start -a android.intent.action.CALL -d tel:4155551234
Starting: Intent { act=android.intent.action.CALL dat=tel:xxxxxxxxxx }
java.lang.SecurityException: Permission Denial: starting Intent { act=android.intent.action.CALL dat=tel:xxxxxxxxxx
flg=0x10000000 cmp=com.android.phone/.OutgoingCallBroadcaster } from null (pid=1533, uid=10045) requires
android.permission.CALL_PHONE
    at android.os.Parcel.readException(Parcel.java:1327)
    at android.os.Parcel.readException(Parcel.java:1281)
    at android.app.ActivityManagerProxy.startActivity(ActivityManagerNative.java:1631)
    at com.android.commands.am.Am.runStart(Am.java:433)
    at com.android.commands.am.Am.run(Am.java:107)
    at com.android.commands.am.Am.main(Am.java:80)
    at com.android.internal.os.RuntimeInit.finishInit(Native Method)
    at com.android.internal.os.RuntimeInit.main(RuntimeInit.java:238)
    at dalvik.system.NativeStart.main(Native Method)
```

    - For example, trying to read a private system directory is restricted:

```
$ adb shell run-as com.example.helloworld ls /data/misc/keystore
opendir failed, Permission denied
```

- Ignored but logged by the system (implicit failures) - e.g. `BroadcastReceiver` listening on a protected intent

- To access restricted features of the system or other apps, apps developers are are required to *use* permissions via explicit `<uses-permission>`-s in `AndroidManifest.xml`
- Subject to one-time user-approval, permissions are granted at install time with no support for run-time per-use user approval
- For example, an app that wishes to track user by GPS and report location via network/SMS would require three permissions: access (fine) location, access the internet, and send SMS
- Its `AndroidManifest.xml` would look like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" 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.SEND_SMS" />
  …
</manifest>
```

- For the list of built-in permissions:
  - See http://developer.android.com/reference/android/Manifest.permission.html
  - Or run:

```
$ adb shell pm list permissions
All Permissions:

permission:android.permission.CLEAR_APP_USER_DATA
permission:android.permission.SHUTDOWN
permission:android.permission.BIND_INPUT_METHOD
permission:android.permission.ACCESS_DRM
permission:android.permission.DOWNLOAD_CACHE_NON_PURGEABLE
permission:android.permission.INTERNAL_SYSTEM_WINDOW
```

```
permission:android.permission.SEND_DOWNLOAD_COMPLETED_INTENTS
permission:android.permission.MOVE_PACKAGE
permission:android.permission.ACCESS_CHECKIN_PROPERTIES
permission:android.permission.CRYPT_KEEPER
permission:android.permission.READ_INPUT_STATE
permission:android.permission.DEVICE_POWER
permission:android.permission.DELETE_PACKAGES
permission:android.permission.ACCESS_CACHE_FILESYSTEM
permission:android.permission.REBOOT
permission:android.permission.STATUS_BAR
permission:android.permission.ACCESS_DOWNLOAD_MANAGER_ADVANCED
permission:android.permission.ACCESS_ALL_DOWNLOADS
permission:android.permission.STOP_APP_SWITCHES
permission:android.permission.BIND_VPN_SERVICE
permission:android.permission.CONTROL_LOCATION_UPDATES
permission:android.permission.ACCESS_DOWNLOAD_MANAGER
permission:android.permission.MANAGE_APP_TOKENS
permission:android.permission.BIND_PACKAGE_VERIFIER
permission:android.permission.DELETE_CACHE_FILES
permission:android.permission.BATTERY_STATS
permission:android.permission.COPY_PROTECTED_DATA
permission:com.android.email.permission.ACCESS_PROVIDER
permission:android.permission.INSTALL_DRM
permission:android.permission.MASTER_CLEAR
permission:android.permission.SET_ACTIVITY_WATCHER
permission:android.permission.BRICK
permission:android.permission.MODIFY_NETWORK_ACCOUNTING
permission:android.permission.READ_NETWORK_USAGE_HISTORY
permission:android.permission.BACKUP
permission:android.permission.SET_TIME
permission:android.permission.STATUS_BAR_SERVICE
permission:android.permission.INSTALL_PACKAGES
permission:android.permission.PERFORM_CDMA_PROVISIONING
permission:android.permission.INJECT_EVENTS
```

```
permission:android.permission.SET_POINTER_SPEED
permission:com.android.browser.permission.PRELOAD
permission:android.permission.WRITE_SECURE_SETTINGS
permission:android.permission.INSTALL_LOCATION_PROVIDER
permission:android.permission.CONFIRM_FULL_BACKUP
permission:android.permission.PACKAGE_USAGE_STATS
permission:android.permission.ACCESS_SURFACE_FLINGER
permission:android.permission.CALL_PRIVILEGED
permission:android.permission.PACKAGE_VERIFICATION_AGENT
permission:android.permission.CHANGE_COMPONENT_ENABLED_STATE
permission:android.intent.category.MASTER_CLEAR.permission.C2D_MESSAGE
permission:android.permission.WRITE_GSERVICES
permission:android.permission.MANAGE_NETWORK_POLICY
permission:android.permission.ALLOW_ANY_CODEC_FOR_PLAYBACK
permission:android.permission.BIND_TEXT_SERVICE
permission:android.permission.READ_FRAME_BUFFER
permission:android.permission.FORCE_BACK
permission:android.permission.UPDATE_DEVICE_STATS
permission:android.permission.BIND_WALLPAPER
permission:android.permission.BIND_REMOTEVIEWS
permission:android.permission.SET_ORIENTATION
permission:android.permission.FACTORY_TEST
permission:android.permission.BIND_DEVICE_ADMIN
```

Add `-f` for the full description of permissions - i.e. `adb shell pm list permissions -f`

- Or take a look at `frameworks/base/core/res/AndroidManifest.xml` in AOSP source tree

Top Ten *Bad* Permissions (on Google Play)

Using the following permissions will significantly lower the likelihood for an Android app/game to be featured in Google Play (from Google I/O 2012):

1. `android.permission.SEND_SMS` and `android.permission.RECEIVE_SMS`
2. `android.permission.SYSTEM_ALERT_WINDOW`
3. `com.android.browser.permission.READ_HISTORY_BOOKMARKS` and `com.android.browser.permission.WRITE_HISTORY_BOOKMARKS`
4. `android.permission.READ_CONTACTS`, `android.permission.WRITE_CONTACTS`, `android.permission.READ_CALENDAR`, `android.permission.WRITE_CALENDAR`
5. `android.permission.CALL_PHONE`
6. `android.permission.READ_LOGS`
7. `android.permission.ACCESS_FINE_LOCATION`
8. `android.permission.GET_TASKS`
9. `android.permission.RECEIVE_BOOT_COMPLETED`

10. android.permission.CHANGE_WIFI_STATE

Avoid Using Permissions (When You Can)

- Instead of requesting `android.permission.CAMERA` permission (and directly using the Camera APIs), start an intent (for result) for `android.provider.MediaStore.ACTION_IMAGE_CAPTURE`; for example:

```
...
public class MyActivity extends Activity {
  private static final int CAPTURE_IMAGE_REQ = 1;
  ...
  public void onClick(View view) {
    Intent intent = new Intent(
      android.provider.MediaStore.ACTION_IMAGE_CAPTURE,
      CAPTURE_IMAGE_REQ);
    intent.putExtra(
      android.provider.MediaStore.EXTRA_OUTPUT,
      "/sdcard/myphoto.jpeg");
    super.startActivityForResult(intent);
  }

  protected void onActivityResult (int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
      case CAPTURE_IMAGE_REQ:
        if (resultCode == RESULT_OK) {
          // we have the image!
        }
        break;
    }
  }
}
```

- Instead of requesting `android.permission.SEND_SMS` permission (and using the SMS APIs), start the SMS composer activity to send the SMS and pre-fill the data:

```
...
public class MyActivity extends Activity {
  ...
  public void onClick(View view) {
    Uri smsNumber = Uri.parse("sms:14155551234");
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(smsNumber);
    intent.putExtra(Intent.EXTRA_TEXT, "Hello");
    super.startActivity(intent);
  }
}
```

- Instead of requesting `android.permission.READ_CONTACTS` permission (and accessing contacts provider) use the contacts' chooser:

```
...
public class MyActivity extends Activity {
  private static final int GET_CONTACT_REQ = 1;
  ...
  public void onClick(View view) {
    Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
    intent.setType(Phone.CONTENT_ITEM_TYPE);
    super.startActivityForResult(intent, GET_CONTACT_REQ);
  }

  public void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == GET_CONTACT_REQ && resultCode == RESULT_OK && data != null) {
      Uri uri = data.getData();
      if (uri != null) {
        try {
          // best to do on another thread
          Cursor c = getContentResolver().query(uri,
            new String[] {Contacts.DISPLAY_NAME, Phone.NUMBER}, null, null, null);
          if (c.moveToFirst()) {
            String name = c.getString(0);
            String phone = c.getString(1);
            // we have the contact info!
          }
        } catch (RuntimeException e) {
          // handle
        }
      }
    }
  }
}
```

- Instead of requesting `android.permission.READ_PHONE_STATE` permission (and using

`TelephonyManager.getDeviceId()`), use `android.provider.Settings.Secure.ANDROID_ID` or create/store your own `UUID.randomUUID()`:

```
String deviceId = UUID.randomUUID().toString();
// store deviceId in the application preferences
```

For more choices on how to get a unique device ID, see http://android-developers.blogspot.com/2011/03/identifying-app-installations.html

Permission Enforcement

There are a number of trigger points for security/permission checks:

- Kernel
- Native Layer Daemons
- Applications Framework Services
- Application Components

Kernel / File-system Permission Enforcement

- Access to system resources (files/drivers/unix-sockets and net/BT-sockets) is restricted via a combination of
  - File-system permissions
  - Paranoid network security (Android-specific kernel-patches)

- File/driver/unix-socket ownership/permissions are set in:
  - `/init` process
  - `/init.rc` file(s)
  - `/ueventd.rc` file(s)
  - system ROM (via `system/core/include/private/android_filesystem_config.h` in AOSP)

- Application's logical permissions (i.e. ones defined via `<uses-permission>` in `AndroidManifest.xml`) are mapped to system groups via:

`frameworks/base/data/etc/platform.xml` (Android source code):

```
<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>
```

Run `adb shell cat /system/etc/permissions/platform.xml` to see all the mappings

- Permissions assigned to individual applications are stored in `/data/system/packages.xml`

UID-based Permission Enforcement

- Some system processes (daemons) explicitly check for the UID of the calling process (via IPC)
- For example, `servicemanager` explicitly limits registration of new services to processes running as root, system, radio, media, nfc, and drm (with some additional restrictions)

frameworks/base/cmds/servicemanager/service_manager.c:

```
…
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
#ifdef LVMX
    { AID_MEDIA, "com.lifevibes.mx.ipc" },
#endif
    { AID_MEDIA, "media.audio_flinger" },
    { AID_MEDIA, "media.player" },
    { AID_MEDIA, "media.camera" },
    { AID_MEDIA, "media.audio_policy" },
    { AID_DRM,   "drm.drmManager" },
    { AID_NFC,   "nfc" },
    { AID_RADIO, "radio.phone" },
    { AID_RADIO, "radio.sms" },
    { AID_RADIO, "radio.phonesubinfo" },
    { AID_RADIO, "radio.simphonebook" },
/* TODO: remove after phone services are updated: */
    { AID_RADIO, "phone" },
    { AID_RADIO, "sip" },
    { AID_RADIO, "isms" },
    { AID_RADIO, "iphonesubinfo" },
    { AID_RADIO, "simphonebook" },
```

```
};
…
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM))
        return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;

    return 0;
}
…
```

Paranoid Network Security

- Restricts access to some networking features depending on the group of the calling process
- Enabled via `CONFIG_ANDROID_PARANOID_NETWORK` kernel build option, which defines process group IDs that have special network access
- For example, for the Browser application to browse the web it needs to use the `INTERNET` permission:

`packages/apps/Browser/AndroidManifest.xml`:

```
<manifest … package="com.android.browser">
  …
  <uses-permission android:name="android.permission.INTERNET" />
  …
</manifest>
```

- When the the Browser app is launched, its `INTERNET` permission is mapped onto the `inet` group

`frameworks/base/data/etc/platform.xml` (or `/system/etc/permissions/platform.xml` on the system image):

```
<permissions>
  …
  <permission name="android.permission.INTERNET" >
    <group gid="inet" />
  </permission>
  …
</permissions>
```

- The `inet` group is mapped onto group id `3003`:

`system/core/include/private/android_filesystem_config.h` (AOSP source-tree)

```
…
#define AID_INET          3003  /* can create AF_INET and AF_INET6 sockets */
…
static const struct android_id_info android_ids[] = {
  …
  { "inet",             AID_INET, },
  …
};
…
```

- When the Browser app runs and tries to open a socket, the kernel will check that its process is a member of group 3003:

`include/linux/android_aid.h` *(kernel source-tree):*

```
…
#define AID_INET         3003
…
```

`net/ipv4/af_inet.c` *(kernel source-tree):*

```c
…
#ifdef CONFIG_ANDROID_PARANOID_NETWORK
#include <linux/android_aid.h>

static inline int current_has_network(void)
{
  return in_egroup_p(AID_INET) || capable(CAP_NET_RAW);
}
#else
static inline int current_has_network(void)
{
  return 1;
}
#endif
…
static int inet_create(struct net *net, struct socket *sock, int protocol)
{
  …
  if (!current_has_network())
    return -EACCES;
  …
}
…
```

- In addition to protecting IPv4 sockets, `CONFIG_ANDROID_PARANOID_NETWORK` option is also used to control access to:
  - `net/ipv6/af_inet6.c`
  - `net/bluetooth/af_bluetooth.c`
  - `net/netfilter/xt_qtaguid.c`

Other Kernel Changes

- Timed output / Timed GPIO
  - Generic GPIO allows user space to access and manipulate GPIO registers
  - Timed GPIO allows changing a GPIO pin and having it restored automatically after a specified timeout
  - Implementation at `drivers/staging/android/timed_output.c` and `drivers/staging/android/timed_gpio.c`
  - Used by the vibrator by default

- Linux Scheduler
  - Not a custom scheduler, just Android-specific configuration in `init.rc`

```
write /proc/sys/kernel/panic_on_oops 1
write /proc/sys/kernel/hung_task_timeout_secs 0
write /proc/cpu/alignment 4
write /proc/sys/kernel/sched_latency_ns 10000000
write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
write /proc/sys/kernel/sched_compat_yield 1
write /proc/sys/kernel/sched_child_runs_first 0
```

- Switch events - userspace support for monitoring GPIO used by `vold` to detect USB
- USB gadget driver for ADB (`drivers/usb/gadget/android.c`)
- yaffs2 flash filesystem, though this is switching to ext4
- RAM console
  - Kernel's `printk` goes to a RAM buffer

    - A kernel panic can be viewed in the next kernel invocation via `/proc/last_kmsg`

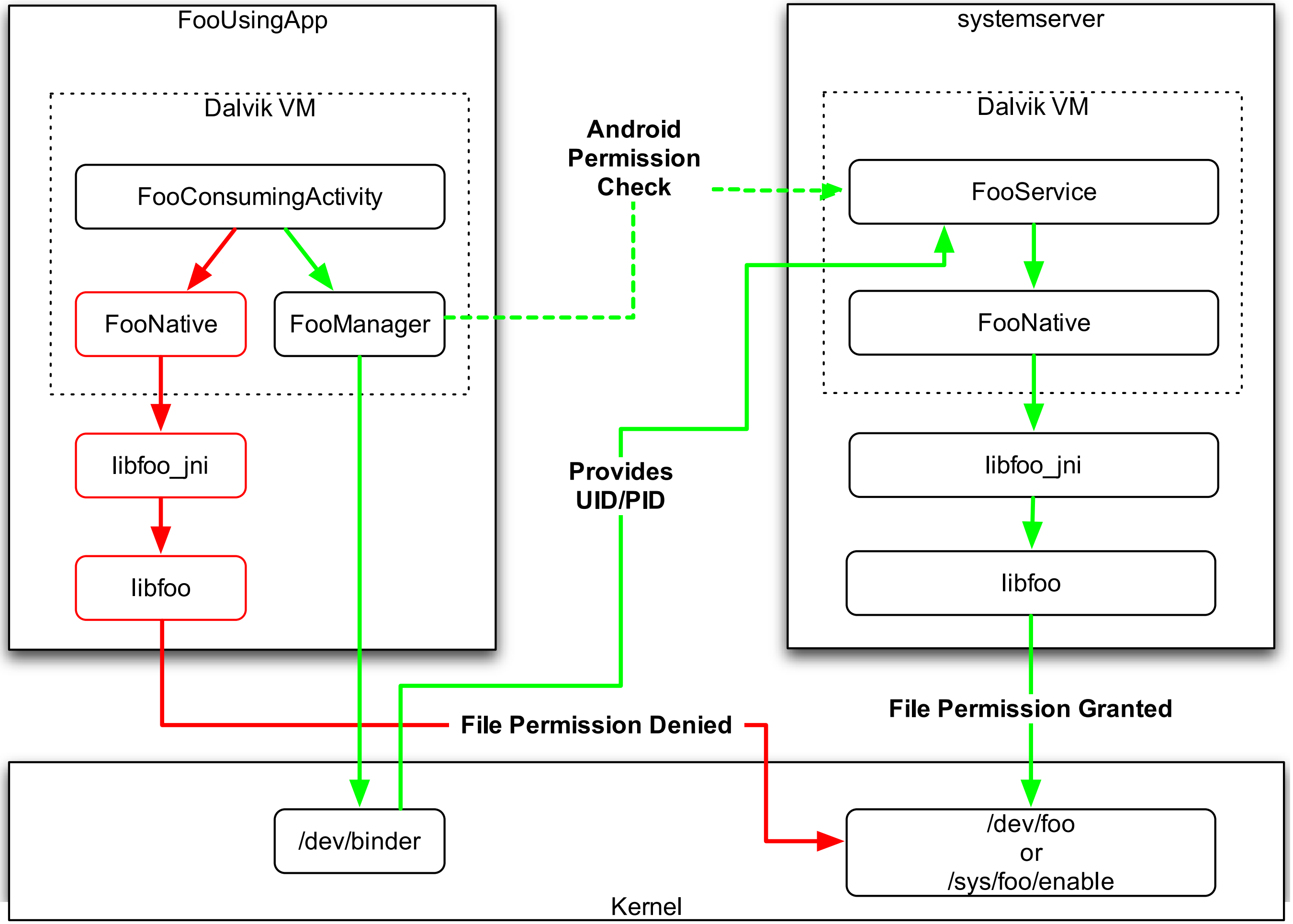
- Support in FAT filesystem for `FVAT_IOCTL_GET_VOLUME_ID`

Static Permission Enforcement

- Automatically managed by `ActivityManagerService`
- In `AndroidManifest.xml` an application restricts access to its components via `android:permission` attribute
    - On `<activity>`, controls who can `Context.startActivity()` and `startActivityForResult()`
    - On `<service>`, controls who can `Context.startService()`, `stopService()`, and `bindService()`
    - On `<provider>`, controls who can access it via a `ContentResolver`
        - `android:readPermission` specifically controls who can `ContentResolver.query()`;
        - `android:writePermission` specifically controls who can `ContentResolver.insert()`, `ContentResolver.update()`, and `ContentResolver.delete()`
    - On `<receiver>`, controls who can send broadcasts to the receiver
        - Receivers can also be registered programmatically, so the sender's permission requirement can be specified via `Context.registerReceiver(BroadcastReceiver receiver, IntentFilter filter, String broadcastPermission, Handler scheduler)` method
        - Senders can also programmatically require that the receivers hold the appropriate permission via `Context.sendBroadcast(Intent intent, String receiverPermission)`
        - Some broadcasts are declared as `<protected-broadcast android:name="…" />` in `frameworks/base/core/res/AndroidManifest.xml` (AOSP source tree) - these can only be sent by the system
- For example:

```xml
<manifest …>
  …
  <application …>
    …
    <activity android:name=".GetPasswordActivity"
      android:permission="com.marakana.android.permission.GET_PASSWORD_FROM_USER" … >
      …
    </activity>
    <service android:name=".UserAuthenticatorService"
      android:permission="com.marakana.android.permission.AUTHENTICATE_USER" … >
      …
    </service>
    <provider android:name=".EnterpriseDataProvider"
      android:readPermission="com.marakana.android.permission.READ_ENTERPRISE_DATA"
      android:writePermission="com.marakana.android.permission.WRITE_ENTERPRISE_DATA" … >
      …
    </provider>
    <receiver android:name=".UserAuthStatusReceiver"
      android:permission="com.marakana.android.permission.SEND_USER_AUTH_STATUS">
      …
    </receiver>
  </application>
</manifest>
```

Dynamic Permission Enforcement



- Android provides an API to *programmatically* check whether a particular process (PID) running under a particular user (UID) is granted a specific permission:
  `android.content.Context.checkPermission(String permission, int pid, int uid)`
  - The `permission` parameter specifies the name of the `<permission … >` defined in a `AndroidManifest.xml` file
    If the permission is defined in the same application (which is the most usual case), then we can statically access its name via the

auto-generated `Manifest` class. For example: `Manifest.permission.MY_PERMISSION`

  - The `pid` parameter specifies the process ID being checked against
  - The `uid` parameter specifies the user ID being checked against (`0` will pass every permission check)
  - This method returns android.content.pm.PackageManager.PERMISSION_GRANTED if the calling process has been granted the `permission`; android.content.pm.PackageManager.PERMISSION_DENIED otherwise

- In the context of IPC (which is the most typical use-case), we can verify the *calling process* using: `android.content.Context.checkCallingPermission(String permission)`
  - PID comes from `android.os.Binder.getCallingPid()`
  - UID comes from `android.os.Binder.getCallingUid()`

- In the event that we want to automatically allow our own process to execute permission-protected code, we can use: `android.content.Context.checkCallingOrSelfPermission(String permission)`
  - Otherwise, we would have to *use* our own permission

- Instead of handling `PERMISSION_DENIED` *return value*, we could have `SecurityException` *thrown* if the other (e.g. calling) process does not have the requested permission by using:
  - `android.content.Context.enforcePermission(String permission, int pid, int uid, String message)`
  - `android.content.Context.enforceCallingPermission(String permission, String message)`
  - `android.content.Context.enforceCallingOrSelfPermission(String permission, String message)`

- This is how many of the application framework services enforce their permissions

`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");
    }
    …
  }
  …
}
```

frameworks/base/services/java/com/android/server/LocationManagerService.java:

```
package com.android.server;
…
public class LocationManagerService extends ILocationManager.Stub implements Runnable {
  …
  private static final String ACCESS_FINE_LOCATION =
      android.Manifest.permission.ACCESS_FINE_LOCATION;
  private static final String ACCESS_COARSE_LOCATION =
      android.Manifest.permission.ACCESS_COARSE_LOCATION;
  …
  private void checkPermissionsSafe(String provider) {
    if ((LocationManager.GPS_PROVIDER.equals(provider)
             || LocationManager.PASSIVE_PROVIDER.equals(provider))
        && (mContext.checkCallingOrSelfPermission(ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
        throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION permission");
    }
```

```
    if (LocationManager.NETWORK_PROVIDER.equals(provider)
        && (mContext.checkCallingOrSelfPermission(ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)
        && (mContext.checkCallingOrSelfPermission(ACCESS_COARSE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
        throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission");
    }
  }
  …
  private Location _getLastKnownLocationLocked(String provider) {
    checkPermissionsSafe(provider);
    …
  }
  …
  public Location getLastKnownLocation(String provider) {
    …
    _getLastKnownLocationLocked(provider);
    …
  }
}
```

- Not very common in application code
  - Can be used in bound services to differentiate access to specific methods (e.g. "administrative" operations)

- Use to avoid confused deputy exploits - check whether the calling app has the same permission
  - For example, the `mediaserver` uses this idea to enforce `INTERNET` and `WAKE_LOCK` permissions

Custom Permissions

Before we can enforce our own permissions, we have to declare them using one or more `<permission>` in

`AndroidManifest.xml`

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.marakana.android.myapp" >
  <permission
    android:name="com.example.app.DO_X"
    android:label="@string/do_x_label"
    android:description="@string/do_x_desc"
    android:permissionGroup="android.permission-group.PERSONAL_INFO"
    android:protectionLevel="dangerous" />
  …
</manifest>
```

- `name` - arbitrary but unique (name-spaced) identifier of this permission
- `protectionLevel` - specifies if/how will Android inform the user when another app uses this permission
    - `normal` (0) - A low-risk permission that is automatically granted (by default), though users have an option to review these before installing (users often just ignore)
    - `dangerous` (1) - A higher-risk permission that requires explicit user approval at install time (preferred)
    - `signature` (2) - A permission that will be granted automatically if the requesting app shares the signature of the declaring app (preferred for application "suites")
    - `signatureOrSystem` (3) - A permission that will be granted only to packages in `/system/app/` (i.e. burned to ROM) or that are signed with the same certificates.
        - This is useful in special-cases where different vendors supply apps to be built into system image, and those apps need to work together

- `label` - a short description of the functionality protected by the permission (ignored when `protectionLevel=signature*`)
- `description` - a longer description (warning) of what can go wrong if the permission is abused (ignored when `protectionLevel=signature*`)
- `android:icon` - an drawable resource describing this permission
- `permissionGroup` - helps organize (group) permissions by some pre-determined categories (e.g. `….COST_MONEY`, `….PERSONAL_INFO`, `….SYSTEM_TOOLS`, etc.)
    - See http://d.android.com/reference/android/Manifest.permission_group.html for the existing Android categories
    - For example

```
$ adb shell pm list permission-groups
permission group:android.permission-group.DEVELOPMENT_TOOLS
permission group:android.permission-group.PERSONAL_INFO
permission group:android.permission-group.COST_MONEY
permission group:android.permission-group.LOCATION
permission group:android.permission-group.MESSAGES
permission group:android.permission-group.NETWORK
permission group:android.permission-group.ACCOUNTS
permission group:android.permission-group.STORAGE
permission group:android.permission-group.PHONE_CALLS
permission group:android.permission-group.HARDWARE_CONTROLS
permission group:android.permission-group.SYSTEM_TOOLS
```

note We can list permissions currently defined an an Android device/emulator: `$ adb shell pm list permissions -s`

Adding Custom Permissions Dynamically

- Android allows permissions to be defined programmatically via `PermissionManager.addPermission(PermissionInfo)` API
- The new permission can only be added relative to a permission tree setup via `<permission-tree/>` in the `AndroidManifest.xml` file
- Permissions added through this API are persisted (across device reboots)
- Existing permissions (that already exist) are updated
- For example:

AndroidManifest.xml:

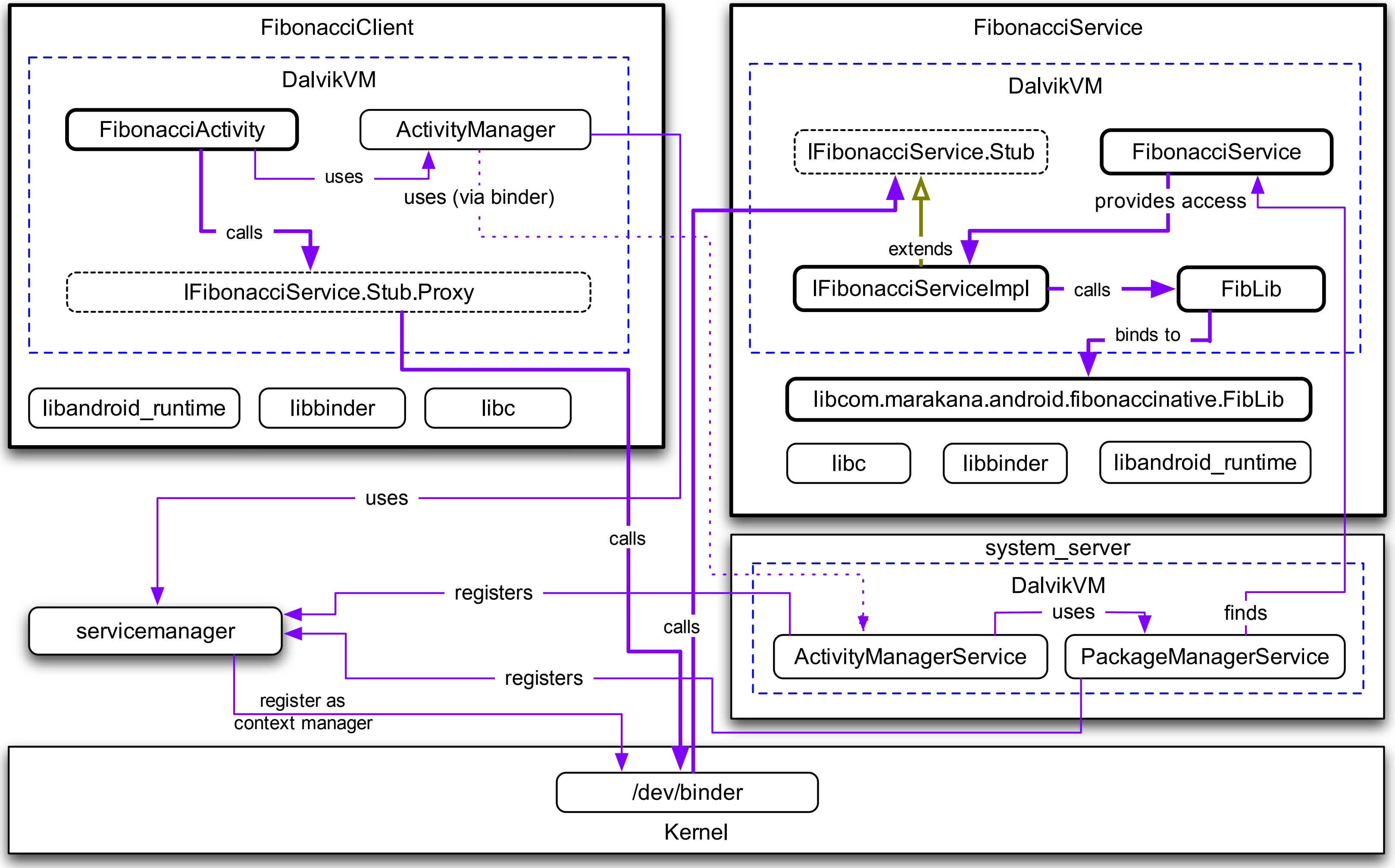
```
<manifest …>
    …
    <permission-tree android:name="com.marakana.android.foo" />
    …
</manifest>
```

src/com/marakana/android/foo/FooActivity.java:

```
…
public class FooActivity extends Activity … {
    public void onCreate() {
        super.onCreate();
        …
        PermissionInfo permission = new PermissionInfo();
        permission.name = "com.marakana.android.foo.DO_X_WITH_FOO";
        permission.protectionLevel = PermissionInfo.PROTECTION_SIGNATURE;
        super.getPackageManager().addPermission(permission);
        …
    }
}
```

Permissions by Example

- In this example, we are given two applications, `FibonacciClient` and `FibonacciService`



- These two apps communicate via Binder/IPC
- Common files for these applications reside in a library project called `FibonacciCommon`

- The finished code for these applications (as Eclipse projects) is available:
  - As a ZIP archive: https://github.com/marakana/FibonacciBinderDemo/zipball/secured
  - By Git: `git clone https://github.com/marakana/FibonacciBinderDemo.git -b secured`

Static Permission Enforcement

Here, we want to restrict access to the `com.marakana.android.fibonacciservice.FibonacciService` to applications (i.e. clients) that hold *USE_FIBONACCI_SERVICE* custom permission

1. We start by by creating a custom permission group (making sure that we name-space it):

   FibonacciService/res/values/strings.xml:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <resources>
       …
       <string name="fibonacci_permissions_group_label">Fibonacci Permissions</string>
       …
   </resources>
   ```

   FibonacciService/AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <permission-group
        android:name="com.marakana.android.fibonacciservice.FIBONACCI_PERMISSIONS"
        android:label="@string/fibonacci_permissions_group_label" />
    …
</manifest>
```

This permission group is optional - as we could instead use one of the already provided groups

2. Next, we create a custom permission (again, making sure that we name-space it), while taking advantage of our newly-created permission group:

FibonacciService/res/values/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    …
    <string name="use_fibonacci_service_permission_label">use fibonacci service</string>
    <string name="use_fibonacci_service_permission_description">
      applications with this permissions get fibonacci results for free
    </string>
    …
</resources>
```

FibonacciService/AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <permission-group …/>
    <permission
        android:name="com.marakana.android.fibonacciservice.USE_FIBONACCI_SERVICE"
        android:description="@string/use_fibonacci_service_permission_description"
        android:label="@string/use_fibonacci_service_permission_label"
        android:permissionGroup="com.marakana.android.fibonacciservice.FIBONACCI_PERMISSIONS"
        android:protectionLevel="dangerous" />
    …
</manifest>
```

3. Now we can statically require the permission on our `FibonacciService` service:

FibonacciService/AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <permission-group …/>
    <permission …/>
    <application …>
        <service
            android:name=".FibonacciService"
            android:permission="com.marakana.android.fibonacciservice.USE_FIBONACCI_SERVICE" >
            …
        </service>
    </application>
    …
</manifest>
```

4. If we now re-run the `FibonacciService` and re-run the `FibonacciClient`, we will notice that the client will fail to launch and `adb logcat` will show something like:

```
…
W/ActivityManager(   85): Permission Denial: Accessing service
ComponentInfo{com.marakana.android.fibonacciservice/com.marakana.android.fibonacciservice.FibonacciService} from pid=540, uid=10043
requires com.marakana.android.fibonacciservice.USE_FIBONACCI_SERVICE
D/AndroidRuntime(  540): Shutting down VM
W/dalvikvm(  540): threadid=1: thread exiting with uncaught exception (group=0x409c01f8)
E/AndroidRuntime(  540): FATAL EXCEPTION: main
E/AndroidRuntime(  540): java.lang.RuntimeException: Unable to resume activity
{com.marakana.android.fibonacciclient/com.marakana.android.fibonacciclient.FibonacciActivity}: java.lang.SecurityException: Not
allowed to bind to service Intent { act=com.marakana.android.fibonaccicommon.IFibonacciService }
E/AndroidRuntime(  540):    at android.app.ActivityThread.performResumeActivity(ActivityThread.java:2444)
…
E/AndroidRuntime(  540):    at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime(  540): Caused by: java.lang.SecurityException: Not allowed to bind to service Intent {
act=com.marakana.android.fibonaccicommon.IFibonacciService }
E/AndroidRuntime(  540):    at android.app.ContextImpl.bindService(ContextImpl.java:1135)
E/AndroidRuntime(  540):    at android.content.ContextWrapper.bindService(ContextWrapper.java:370)
E/AndroidRuntime(  540):    at com.marakana.android.fibonacciclient.FibonacciActivity.onResume(FibonacciActivity.java:65)
…
W/ActivityManager(   85):   Force finishing activity com.marakana.android.fibonacciclient/.FibonacciActivity
…
```

5. Finally, we can give `FibonacciClient` a fighting chance by allowing it to *use* the `USE_FIBONACCI_SERVICE` permission:

FibonacciClient/AndroidManifest.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <uses-permission android:name="com.marakana.android.fibonacciservice.USE_FIBONACCI_SERVICE"/>
    …
</manifest>
```

6. We can now observe that our client is again able to use the service

7. In the Emulator, if we go to *Home → Menu → Manage apps → Fibonacci Client → PERMISSIONS*, we should see the *Fibonacci Permissions* group and under it, *use fibonacci service* permission

Dynamic Permission Enforcement

Here, we want to restrict access to the `com.marakana.android.fibonacciservice.IFibonacciServiceImpl`'s recursive operations (`fibJR(long n)` and `fibNR(long n)`) for `n` > `10` to applications (i.e. clients) that hold *USE_SLOW_FIBONACCI_SERVICE* custom permission

1. Like before, we start off by creating a custom permission:

FibonacciService/res/values/strings.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    …
    <string name="use_slow_fibonacci_service_permission_label">
        use slow fibonacci service operations
   </string>
    <string name="use_slow_fibonacci_service_permission_description">
        applications with this permissions can melt the CPU and drain the battery
        by using slow fibonacci operations
    </string>
    …
</resources>
```

FibonacciService/AndroidManifest.xml:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <permission-group …/>
    <permission …/>
    <permission
        android:name="com.marakana.android.fibonacciservice.USE_SLOW_FIBONACCI_SERVICE"
        android:description="@string/use_slow_fibonacci_service_permission_description"
        android:label="@string/use_slow_fibonacci_service_permission_label"
        android:permissionGroup="com.marakana.android.fibonacciservice.FIBONACCI_PERMISSIONS"
        android:protectionLevel="dangerous" />
    …
</manifest>
```

2. Next, we update our `IFibonacciServiceImpl` to enforce this permission dynamically - via a `android.content.Context` that get expect to get through the constructor:

FibonacciService/src/com/marakana/android/fibonacciservice/IFibonacciServiceImpl.java:

```
package com.marakana.android.fibonacciservice;

import android.content.Context;
…
public class IFibonacciServiceImpl extends IFibonacciService.Stub {
    …
    private final Context context;

    public IFibonacciServiceImpl(Context context) {
        this.context = context;
    }

    private long checkN(long n) {
        if (n > 10) {
            this.context.enforceCallingOrSelfPermission(
                    Manifest.permission.USE_SLOW_FIBONACCI_SERVICE, "Go away!");
        }
        return n;
    }
    …
    public long fibJR(long n) {
        …
        return FibLib.fibJR(this.checkN(n));
    }
    …
    public long fibNR(long n) {
        …
        return FibLib.fibNR(this.checkN(n));
    }
    …
}
```

3. We have to update `FibonacciService` to invoke the new `IFibonacciServiceImpl`'s constructor:

FibonacciService/src/com/marakana/android/fibonacciservice/FibonacciService.java:

```
…
public class FibonacciService extends Service {
    …
    @Override
    public void onCreate() {
        …
        this.service = new IFibonacciServiceImpl(super.getApplicationContext());
        …
    }
    …
}
```

4. If we now re-run the `FibonacciService` and re-run the `FibonacciClient` for a recursive operation with `n` > `10`, we will notice that the client will fail and `adb logcat` will show something like:

```
…
D/IFibonacciServiceImpl(  617): fib(15, RECURSIVE_NATIVE)
D/IFibonacciServiceImpl(  617): fibNR(15)
W/dalvikvm(  604): threadid=11: thread exiting with uncaught exception (group=0x409c01f8)
E/AndroidRuntime(  604): FATAL EXCEPTION: AsyncTask #1
E/AndroidRuntime(  604): java.lang.RuntimeException: An error occured while executing doInBackground()
…
E/AndroidRuntime(  604):     at java.lang.Thread.run(Thread.java:856)
E/AndroidRuntime(  604): Caused by: java.lang.SecurityException: Go away!: Neither user 10043 nor current process has
com.marakana.android.fibonacciservice.USE_SLOW_FIBONACCI_SERVICE.
…
```

5. Finally, we can allow `FibonacciClient` to melt our CPU and drain our battery by allowing it to *use* the `USE_SLOW_FIBONACCI_SERVICE` permission:

FibonacciClient/AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest …>
    …
    <uses-permission android:name="com.marakana.android.fibonacciservice.USE_SLOW_FIBONACCI_SERVICE"/>
    …
</manifest>
```

6. We can now observe that our client is again able to use recursive fibonacci operations even for `n > 10`

7. In the Emulator, if we go to *Home* → *Menu* → *Manage apps* → *Fibonacci Client* → *PERMISSIONS* → *Fibonacci Permissions*, we should see both *use fibonacci service* and *use slow fibonacci service operations* permissions

Lab: Custom Permissions

- For this lab, you are given two applications, `LogClient` and `LogService`
    - These two apps communicate via Binder/IPC, where the common files reside in a library project called `LogCommon`:

      LogCommon/src/com/marakana/android/logcommon/ILogService.aidl:

```
package com.marakana.android.logcommon;
import com.marakana.android.logcommon.LogMessage;

interface ILogService {
    void log(in LogMessage logMessage);
}
```

LogCommon/src/com/marakana/android/logcommon/LogMessage.java:

```java
package com.marakana.android.logcommon;

import android.os.Parcel;
import android.os.Parcelable;

public class LogMessage implements Parcelable {
    private final int priority;
    private final String tag;
    private final String msg;

    public LogMessage(int priority, String tag, String msg) {
        this.priority = priority;
        this.tag = tag;
        this.msg = msg;
    }

    public int getPriority() {
        return priority;
    }

    public String getTag() {
        return tag;
    }

    public String getMsg() {
        return msg;
    }
    …
}
```

- To get started:

1. Get `LogCommon`, `LogService`, and `LogClient` Eclipse projects
   - As a ZIP archive: https://github.com/marakana/LogBinderDemo/zipball/master
   - By Git: `git clone https://github.com/marakana/LogBinderDemo.git`
2. Unzip or `git clone` into your workspace directory
3. Import `LogCommon`, `LogClient`, and `LogService` (as existing) projects into Eclipse

   If you get errors on import, try to clean all projects (menubar → *Project* → *Clean…* → *Clean all projects* → *OK*), and/or close and reopen all projects, and/or restart Eclipse

- In the provided Log client-service applications:
  1. Restrict access to the `com.marakana.android.logservice.LogService` to applications that hold *USE_LOG_SERVICE* custom permission
     1. Create a custom permission group (make sure to name-space it)
     2. Create a custom permission (make sure to name-space it)
     3. Then require the permission on the service
     4. Test that a client without the required permission cannot bind to the service
        1. Look for an exception stack trace in `adb logcat` when you launch the client

5. Have the client use the required permission

6. Test again - the client should now be able to bind to the service as before

2. Restrict access to "long" log messages on `com.marakana.android.logservice.ILogService` to applications that hold *USE_LONG_LOG_SERVICE* permission (e.g. where "long" == `logMessage.getTag().length() > 10 || logMessage.getMsg().length() > 80`)

   1. Create another custom permission (make sure to name-space it)

   2. Dynamically enforce your permission

      For you to do this, you'll need access to a `android.content.Context` object inside the provided `com.marakana.android.logservice.ILogServiceImpl`. Conveniently enough, `com.marakana.android.logservice.LogService` extends `android.app.Service`, which in turn implements `android.content.Context` and has access to *application context* via `getApplicationContext()` method.

   3. Test that a client without the required permission cannot log "long" messages

   4. Have the client use the required permission

   5. Test again - the client should now be able to use the service as before

- Solution

- As a ZIP archive: https://github.com/marakana/LogBinderDemo/zipball/secured
- By Git: `git clone https://github.com/marakana/LogBinderDemo.git -b secured`

`ContentProvider` URI Permissions

- Simple `ContentProvider` read/write permissions on are not always flexible enough
  - E.g. an image viewer app wants to get access to an email attachment for viewing – it would be an overkill to give it read-permission over all of email (assuming that email is exposed via a content provider)
  - E.g. a contact picker (selector) activity wants to select a contact

- Use per-URI permissions instead
  - When starting another Activity, caller sets `Intent.FLAG_GRANT_READ_URI_PERMISSION` or `Intent.FLAG_GRANT_WRITE_URI_PERMISSION`
  - This grants the receiving activity permission to access the specific URI regardless of whether it holds a permission over the ContentProvider managing the data behind the URI

- ContentProviders need to add explicit support for URI permissions via `android:grantUriPermission`

Public vs. Private Components

- We can avoid using permissions to protect our application components, simply by keeping them *private*
- By default, all components are `android:exported="false"`
- Once we define an `<intent-filter>`, this default flips to `android:exported="true"`
- We can revert it back, by *explicitly* setting `android:exported="false"`
- Non-exported components cannot be *used* by external applications
- For example:

```
<manifest …>
  …
    <application …>
      …
      <activity android:name=".GetPasswordActivity" android:exported="false" … >
        <intent-filter>
          …
        </intent-filter>
      </activity>
    </application>
</manifest>
```

Most exported components *should* be protected with permissions.

Intent Broadcast Permissions

- Broadcast senders can specify which permission the `BroadcastReceiver`-s must hold to access the intent
- If they don't, the broadcast intent is leaked to all applications on the system
- Always specify read permission via `Context.sendBroadcast(Intent i, String receiverPermission)` unless we use an explicit destination

Pending Intents

- PendingIntent allows delayed triggering of an actual intent (to start an activity, send a broadcast or start a service) in another application
  - E.g. Notification
  - E.g. Alarms

- When the application given a pending intent triggers the actual intent, it does so with **the same permissions and the identity** as the application that created the pending intent
- The application given the pending intent can fill-in unspecified values of the actual intent (subject to the rules of Intent.fillIn()), which may influence destination and/or integrity of the actual intent's data
- Best to only use pending intents as triggers for our own private components - i.e. explicitly specify our own component's class in the actual Intent - so that it can go there and nowhere else

Encryption

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:

```
KeyStore keyStore = ...;
KeyManagerFactory kmf = KeyManagerFactory.getInstance("X509");
kmf.init(keyStore);
SSLContext context = SSLContext.getInstance("TLS");
context.init(kmf.getKeyManagers(), null, null);
URL url = new URL("https://www.example.com/");
HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
...
```

- Android also comes with `android.net.SSLCertificateSocketFactory` (a more-specialized version of `javax.net.ssl.SSLSocketFactory`) with extra features:
  - Timeout specification for SSL handshake operations
  - SSL certificate and hostname verification checks
    - Hostname verification requires that the socket is created with a hostname, as opposed to an IP address
    - Can be disabled with `adb shell setprop socket.relaxsslcheck yes` on development devices (i.e. requires root)

- Optional, persistent, and file-based SSL session caching with `SSLSessionCache`
    - Saves time, power, and bandwidth by skipping the long SSL handshake when re-establishing a connection to the same server

- For example:

```
int timeout = 10000;
String host = "www.fortify.net";
int port = 443;
SSLSessionCache cache = new SSLSessionCache((Context) this);
Socket socket = SSLCertificateSocketFactory.getDefault(timeout, cache).createSocket(host, port);
OutputStream out = socket.getOutputStream();
InputStream in = socket.getInputStream();
String request = "GET /sslcheck.html HTTP/1.1\r\nHost: 68.178.217.222\r\n\r\n";
out.write(request.getBytes());
byte[] response = new byte[1024];
int nRead = in.read(response);
```

- The key store of trusted root certs (CAs) is at
    - `/system/etc/security/cacerts.bks` and can only be changed by rebuilding the ROM (< ICS)
    - `/system/etc/security/cacerts/*.0` and certificates can be individually disabled via the system settings (>= ICS)

- On-device data encryption is usually also done via JCE
    - Use a basic infrastructure like CryptUtil:

```
package com.marakana.android.securenote;

import java.io.EOFException;
import java.io.IOException;
```

```java
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidParameterSpecException;
import java.util.Arrays;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class CryptUtil {

    public static final int IV_LENGTH = 16;

    private static final String ENCRYPTION_ALGORITHM = "AES/CBC/PKCS5Padding";

    private static final String KEY_ALGORITHM = "AES";

    private static final int KEY_SIZE = 256;

    public static Key getKey(byte[] secret) throws NoSuchAlgorithmException {
        return getKey(secret, false);
    }

    public static Key getKey(byte[] secret, boolean wipeSecret) throws NoSuchAlgorithmException {
        // generate an encryption/decryption key from random data seeded with
        // our secret (i.e. password)
        SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
        secureRandom.setSeed(secret);
```

```
        KeyGenerator keyGenerator = KeyGenerator.getInstance(KEY_ALGORITHM);
        keyGenerator.init(KEY_SIZE, secureRandom);
        Key key = new SecretKeySpec(keyGenerator.generateKey().getEncoded(), KEY_ALGORITHM);
        if (wipeSecret) {
            Arrays.fill(secret, (byte)0);
        }
        return key;
    }

    public static Cipher getEncryptCipher(Key key) throws NoSuchAlgorithmException,
            NoSuchPaddingException, InvalidKeyException {
        Cipher cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, key);
        return cipher;
    }

    public static Cipher getDecryptCipher(Key key, byte[] iv) throws NoSuchAlgorithmException,
            NoSuchPaddingException, InvalidKeyException, InvalidAlgorithmParameterException {
        Cipher cipher = Cipher.getInstance(ENCRYPTION_ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(iv));
        return cipher;
    }

    public static byte[] getIv(Cipher cipher) throws InvalidParameterSpecException {
        return cipher.getParameters().getParameterSpec(IvParameterSpec.class).getIV();
    }

    public static byte[] getIv(InputStream in) throws IOException {
        byte[] iv = new byte[IV_LENGTH];
        for (int i = 0; i < iv.length;) {
            int nRead = in.read(iv, i, iv.length - i);
            if (nRead == -1) {
                throw new EOFException("Unexpected EOF");
            } else {
```

```
                i += nRead;
            }
        }
        return iv;
    }
}
```

- To encrypt to a byte array, we could do:

```
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
```

- To decrypt from a byte array, we could do:

```
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);
```

- To encrypt to a file/stream, we could do:

```java
byte[] secret = // some secret
byte[] plainText = // some plain text
Key key = CryptUtil.getKey(secret, true);
OutputStream out = new FileOutputStream("some-file"); // or = socket.getOutputStream()
try {
    Cipher cipher = CryptUtil.getEncryptCipher(key);
    byte[] iv = CryptUtil.getIv(cipher);
    out.write(iv);
    out = new CipherOutputStream(out, cipher);
    out.write(plainText);
    out.flush();
} finally {
    out.close();
}
```

- To decrypt a file/stream, we could do:

```java
InputStream in =  new FileInputStream("some-file"); // or = socket.getInputStream();
try {
    byte[] iv = CryptUtil.getIv(in);
    Cipher cipher = CryptUtil.getDecryptCipher(key, iv);
    in = new CipherInputStream(in, cipher);
    in.read(plainText); // do properly
} finally {
    in.close();
}
```

In May 2011, Google was caught with their pants down. They passed authentication tokens from their Android client applications to their backend services, including contacts, calendars, and photos (picassa) over a plain-text (i.e. unencrypted) channel. This enabled potential

attackers to get access and modify private content of users whose auth tokens were captured.
See http://money.cnn.com/2011/05/18/technology/android_security/?section=money_latest

Whole Disk Encryption

- Android 3.0 (Honeycomb) release introduced a new feature, Settings → Location & Security → Encryption → Encrypt tablet, which enables transparent encryption of the `/data` partition using Linux kernel's dm-crypt functionality (http://www.saout.de/misc/dm-crypt/)
- dm-crypt, which presents itself as a block-device, wraps another block device such as eMMC and similar flash devices (but not raw flash chips, so no yaffs2) and offers on-the-fly encryption/decryption of the underlying data
  - Note that at the moment encryption is done "in software", rather than by an optimized set of hardware instructors on the SoC
- To avoid issues with GPL, Android does not use dm-crypt's `cryptsetup` command and `libdevmapper` shared library, but rather moves that functionality to the volume daemon (`vold`), which directly `ioctl`-calls on the dm-crypt's kernel device
- Additionally, Android's `init` process had to be extended to support password-entry at boot
  - With encryption enabled, `init` gets the password and then restarts into the normal boot process with `/data` properly initialized as a real filesystem
  - See http://source.android.com/tech/encryption/android_crypto_implementation.html for details
- The rest of the Android OS as well as all the apps are unaware of any encryption of the underlying `/data` partition
- At present, the Android uses 128 AES with CBC and ESSIV:SHA256 for `/data`, and 128 bit AES for the master key (stored in the last 16KB of the encrypted partition)
- On an unencrypted tablet:
  - The `/data` partition is mounted as a memory block device:

```
$ adb shell mount |grep /data
/dev/block/mmcblk0p8 /data ext4 rw,nosuid,nodev,noatime,barrier=1,data=ordered 0 0
```

- Writing a 1GB file takes on average 103.919 seconds (10,333,296 bytes/sec)

  ```
  cd /data/local/tmp/
  time dd if=/dev/zero of=out bs=4096 count=262144
  262144+0 records in
  262144+0 records out
  1073741824 bytes transferred in 104.006 secs (10,323,845 bytes/sec)
      1m44.02s real     0m0.45s user     0m8.42s system
  rm out
  time dd if=/dev/zero of=out bs=4096 count=262144
  …
  rm out
  time dd if=/dev/zero of=out bs=4096 count=262144
  …
  ```

- Reading a 1GB file takes on average 45.99 seconds (23,348,197 byes/sec)

  ```
  time dd if=out of=/dev/null bs=4096 count=262144
  262144+0 records in
  262144+0 records out
  1073741824 bytes transferred in 45.692 secs (23499558 bytes/sec)
      0m45.70s real     0m0.54s user     0m8.84s system
  time dd if=out of=/dev/null bs=4096 count=262144
  …
  time dd if=out of=/dev/null bs=4096 count=262144
  …
  rm out
  ```

- After encryption
  - The `/data` partition is mounted as a dm-crypt device-mapper target, which wraps the original memory block device:

```
$ adb shell mount |grep /data
/dev/block/dm-0 /data ext4 rw,nosuid,nodev,noatime,barrier=1,data=ordered 0 0
```

  - Writing a 1GB file takes on average 107.288 seconds (10,014,686 bytes/sec), a mare 3.2% degradation in performance, mostly because writing to NAND is slow
  - Reading a 1GB file takes on average of 70.616 seconds (15,209,002 byes/sec), a significant **54% degradation** in performance
  - The same benchmark on a OMAP4460-powered Galaxy Nexus running Android 4.2.1 showed the write latency going up by 5.8%, whereas the read latency went up by 98.85%!
    - Note, that at some point dm-crypt could be optimized to take advantage of any special encryption facilities offered in the hardware (on the SoC)

- Android's whole-disk encryption is still vulnerable to various attacks:
  - "Evil maid attack"
    - Only the `/data` is encrypted, so we don't know if we can trust the bootloader and `/system` not to contain any "keyloggers"
    - Everything along the boot path would have to be encrypted, which it is not at the moment
  - Cold-boot attack (http://citp.princeton.edu/memory/)
    - Since dm-crypt stores the encryption keys in RAM, it would theoretically be possible to reboot the device with something like `msramdmp` (McGrew Security RAM Dumper) or `ram2usb` to dump the contents of RAM (containing the encryption key) to a USB drive
      - The device has to be running in order for this to work
      - Even if the host device itself does not support booting from an alternative device (or USB), the RAM could be cooled (so that it retains its state), and then transferred to a device that would support alternative boot methods
      - This is a problem for almost all disk-encryption "solutions" in popular OSs, like Windows, Mac OS X, and Linux

- To protect against this, we would need encryption key to be stored somewhere other than RAM, like the CPU (debug) registers, which may be hardware-dependent (e.g. AES-NI on new Intel chips works well), requires changes to the Linux kernel (see `TRESOR` patch), and is not supported by dm-crypt/Android at the moment

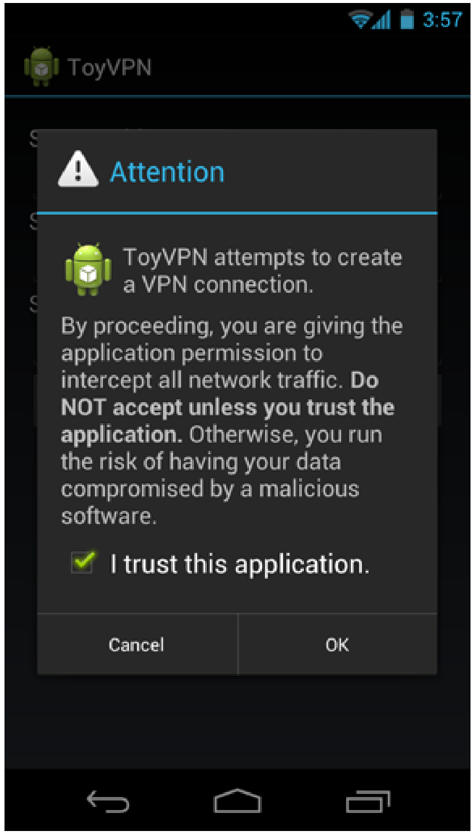
Breaks during 3.0 to 3.1 OS upgrade. A 3.0-encrypted device had to be master-reset (i.e. all data on `/data` had to be wiped) on upgrading to 3.1.

While Honeycomb's whole-disk encryption based on dm-crypt is clearly a step in the right direction, it is far from being a NIST FIPS 140-2-certified solution, which requires two-factor encryption and is mandated for most of DOD applications.

Apple's iOS 4 256-bit hardware encryption was cracked in May 2011 by ElcomSoft through a "simple" brute-force attack in as little as 30 mins using CPU and GPUs of modern host machines. See http://www.geek.com/articles/chips/apples-ios-4-hardware-encryption-has-been-cracked-20110525/ for more info.
Also, according to Nguyen from Symantec, iOS encryption key is stored on the device, but itself is not encrypted by the user's master key. This means that if a potential attackers successfully jailbreak the device, they would be able to access the data without knowing the passcode.

VPN

- Pre ICS/4.0, Android supported L2TP, L2TP/IPSec PSK, L2TP/IPSec RSA, and PPTP VPNs
- ICS/4.0 adds support for pure IPSec VPNs - for better compatibility with commonly deployed VPN endpoints/routers
- Also new in ICS/4.0 is the new VPN API that enables SSL VPN clients to be deployed as apps
    - Via `android.net.VpnService.Builder`, an app can configure addresses, routing rules, DNS/search domains, and even MTUs
    - Via `android.net.VpnService`, the app can then create a virtual network interface tunnel, which is exposed to it as a file descriptor
        - Outgoing packets are *read* from the FD and sent to the VPN tunnel
        - Incoming packets are received from the VPN tunnel and *written* to the FD
        - Each FD read/write operation retrieves/injects one IP packet at a time (so the packets start with IP headers)
    - To address security/exclusivity issues:
        - User has to approve the creation of a new VPN connection
        - There can be only one VPN connection running at the same time (any existing one is deactivated)
        - A system-managed notification is shown during the lifetime of a VPN connection and provides information on the current connection as well as a way to disconnect
        - The network is restored automatically when the file descriptor is closed (in the event that the app process is terminated)
- Example app provided by *ToyVpn* sample application:


3:57
ToyVPN
Attention
ToyVPN attempts to create a VPN connection.
By proceeding, you are giving the application permission to intercept all network traffic. Do NOT accept unless you trust the application. Otherwise, you run the risk of having your data compromised by a malicious software.
I trust this application.
Cancel
OK

Keystore and Keychain API

- Since Donut/1.6, Android has had a system-wide `keystore` for storage of VPN and (later) WiFi authentication keys
  - See Working with secure certificates
  - Pre ICS/4.0, applications were not able to access it, so those needing to authenticate via client certificates ended up maintained their own key store (hard to manage across apps)

- In ICS/4.0, Android adds support for `android.security.KeyChain`, which provides apps with
  - Access to private keys in the system key store and their corresponding certificate chains (subject to one-time user approval)
  - Ability to initiate installation of credentials from X.509 certificates and PKCS#12 key stores (e.g. for installation of organization CA certs)

- Also, in ICS/4.0, Android uses the screen lock password to protect the system credential storage, which works well with device administration APIs and prevents the user from disabling the password as long as they use the secured credentials
- See Unifying Key Store Access in ICS

Application Encryption in Android 4.1

- Jelly Bean introduced an anti-piracy feature such that paid apps in Google Play are encrypted with a device-specific key before they are delivered and stored on the device
- See a 3rd party analysis of this feature

Android Rooting

- Why root?
  - Use applications that require root (e.g. backup, tethering, overclocking, etc.)
  - Customize the existing ROM (e.g. remove bloatware, themes, etc.)
  - Install (e.g. upgrade) new custom ROMs
  - Understand how it works
  - Malware rootkits!

- To obtain "root" on an Android device usually involves a number of steps:
  1. Exploit a vulnerability of the system to give us root once (see below)
  2. Remount the `/system` partition read-write:

     ```
     $ mount -o remount,rw -t ext4 /dev/block/mmcblk0p1 /system
     ```

  3. Create a setuid version of `/system/bin/sh`

     ```
     $ cat /system/bin/sh > /system/bin/su
     $ chmod 06755 /system/bin/su
     ```

  4. Remount the `/system` partition read-only

```
$ mount -o remount,ro -t ext4 /dev/block/mmcblk0p1 /system
```

5. Use `/system/bin/su` to become root at any time afterwards

Controlling access to `/system/bin/su` with *Superuser*

- In the real world, most people would not leave `su` wide open to anyone (or any application) - for obvious security implications
- Instead they use something like *Superuser* (http://code.google.com/p/superuser/)
  - Superuser is a combination of a custom `/system/bin/su` binary, as well as a `Superuser.apk` (application).
  - When a 3rd-party application wants super-user access, it runs `/system/bin/su` (possibly with parameters of what it wishes to execute)
    1. The `su` binary first checks whether the calling process is white-listed
        1. First, it checks in the SQLite database owned by the Superuser application: `/data/data/com.koushikdutta.superuser/databases/superuser.sqlite`
            - Its `whitelist` table stores three values: the calling process UID, process name (usually just the package name), and a flag (-1=not allowed, 1=temporary allowed, 10000=always allowed)
        2. If the calling process is already not-approved `su` exits
        3. If the calling process is not already approved,
            1. `su` launches a simple activity dialog of the Superuser application by passing the calling process' UID and PID as extra intent parameters:

```
…
char sysCmd[1024];
sprintf(sysCmd, "am start -a android.intent.action.MAIN -n
com.koushikdutta.superuser/com.koushikdutta.superuser.SuperuserRequestActivity --ei uid %d --ei pid %d >
/dev/null", g_puid, ppid);
if (system(sysCmd))
    …
```

4. Now `com.koushikdutta.superuser` application starts and `SuperuserRequestActivity` prompts the user to make their selection
    1. The user's selection is saved into the database and `SuperuserRequestActivity` terminates
    2. `su` now checks the database again, and if the process is not approved, it exits
5. `su` then `setuid(uid)` -s and `setgid(gid)` -s the calling process
6. `su` finally executes `/system/bin/sh` with the same parameters that the original `su` was invoked with

2. The 3rd party application can now use `sh` to pass any commands that it wishes to run as `root`

Now, let take a look at some of the past Android exploits (Getting root the first time):

UDEV exploit (CVE-2009-1185)

- On a standard Linux OS, `udev` enables dynamic management of devices - specifically ones that can be hot-plugged while the system is running (like USB)
  - When a new device is detected, Linux kernel passes a message (containing executable code) to the `udev` daemon, which runs as root and acts on this event
  - Prior to version 1.4.1, `udev` did not verify that the message came from the kernel, which made it possible for a rouge application to fake a device and have `udev` execute arbitrary code
  - Newer kernels sent authenticated messages, but `udev` needs to be smart to verify them

- On Android, `udev` functionality is rolled int Android's `init` process (actually `ueventd`), which still runs as root
  - This "exploid2" (a.k.a "Exploid") roughly works as follows:
    1. The user `copies` to the device (e.g. `adb push exploid2 /data/local/tmp/exploid2`)
    2. The user then runs the "exploid2" process (e.g. adb shell `/data/local/tmp/exploid2`)
    3. On the first run, the "exploid2" copies itself to `/sqlite_stmt_journals/exploid2`
    4. The "exploid2" then sends a `NETLINK_KOBJECT_UEVENT` message (via a local unix socket) to `init` (i.e., `udev` code within `init`) to tell it to run a copy of itself next time a device is plugged in (basically it presents itself as `FIRMWARE` update for this device)

5. The user then "hot-plugs" a device by clicking Settings → Wireless → Airplane, WiFi, etc. or plugs in a USB device (if USB host port is available)

6. The "exploid2" runs again, this time as `root` (as part of `init`) and it then

    1. Remounts the `/system` in read-write mode

    2. Copies itself to `/system/bin/rootshell` and sets its permission as `04711` (i.e. setuid-bit enabled)

7. If the user now wants root, the user simply runs `/system/bin/rootshell`, which then

    1. Switches to `root` via a simple `setuid(0); setgid(0);`

    2. Executes `/system/bin/sh` (now as root) with the parameters passed to `/system/bin/rootshell`

ADB setuid exhaustion attack (CVE-2010-EASY)

- Android Debug Bridge Daemon (`adbd`) starts as root, but calls `setuid(shell)` when forking itself to execute remote requests (i.e. to run `/system/bin/sh`)
- In this case, a program called `rageagainstthecage` exploits a known condition where a call to `setuid()` fails once we reach `RLIMIT_NPROC`, preventing `adbd` from dropping its privileges, and since it does not check for this failure, it remains running as root
- The program "fork-bombs" `adbd` by creating client requests to it (which causes it to fork) until the system reaches the maximum number of processes (RLIMIT_NPROC) - typically around 2-5K
- At this point, `rageagainstthecage` tries to fill the last slot with its connection to `adbd` while it is still running as `root` before `adbd` has a chance to `setuid()` itself
- The problem is that `adbd` does not check whether its call to `setuid()` succeeded, which leaves `rageagainstthecage` running with `root` access

Zimperlich attack against Zygote

- Similar to `adbd`, `zygote` (Android app spawner, which also runs as root) did not check for setuid() failures, so it too was prone to this sort of attack
- Fork self repeatedly to reach the process limit (RLIMIT_NPROC)
- The call to `setuid()` fails
- The app that was spawned runs as root

Ashmem memory protection attack (CVE-2011-1149)

- "Android before 2.3 does not properly restrict access to the system property space, which allows local applications to bypass the application sandbox and gain privileges, as demonstrated by psneuter and KillingInTheNameOf, related to the use of Android shared memory (ashmem) and ASHMEM_SET_PROT_MASK."
- `KillingInTheNameOf` exploit:
  1. Changes protections of shared (ashmem) memory space where system properties are stored to allow writing
  2. Sets `ro.secure` to `0`
  3. User restarts `adbd`
  4. User get root via `adb shell`
- `psneuter` exploit:
  1. Disables access to shared (ashmem) memory space where system properties are stored (sets protection mask to 0)
  2. User restarts `adbd`, but since `adbd`cannot read `ro.secure` it assumes `ro.secure=0`
  3. User get root via `adb shell`

Buffer Overrun on `vold` exploit (CVE-2011-1823)

- "The vold volume manager daemon on Android 3.0 and 2.x before 2.3.4 trusts messages that are received from a PF_NETLINK socket, which allows local users to execute arbitrary code and gain root privileges via a negative index that bypasses a maximum-only signed integer check in the DirectVolume::handlePartitionAdded method, which triggers memory corruption, as demonstrated by Gingerbreak"
- On Android, `vold` (volume daemon running as root) is used for operations such as SD-Card mounting/unmounting, as well as encryption of `/data` partition
- Here, an application called Gingerbreak (a.k.a. *Softbreak*) is first uploaded to the device (e.g. to `/data/local/tmp/softbreak`)
- When executed (e.g. via `adb shell`) it tries to exploit an out of bounds array access in `vold` and thus inject code to be executed by `root`
- Because `vold` is configurable by the OEMs (and its memory state changes), this attack is not guaranteed to work every time - in fact, it often causes `vold` to segfault
- A malicious application on the device can exploit the same vulnerability to gain root access
- While this exploit initially targeted Gingerbread, it also works on Froyo and Honeycomb (⇐ 3.1) releases

Linux Local Privilege Escalation via SUID `/proc/pid/mem` Write (CVE-2012-0056)

- In Linux-based systems, `/proc/pid/mem` provides an interface for reading and writing directly to a process's virtual memory
  - Seeking on `/proc/pid/mem` is the same as seeking on the memory of that process

- In v2.6.39, the kernel was simplified to do permission checks on `/proc/pid/mem` at the time it is opened, instead of tracking the process that's using it
  - If we hold the file descriptor to `/proc/pid/mem` open over an `execve()` system call, we'll continue to read from the virtual memory of the original *old* process
  - This vulnerability can be exploited (CVE-2012-0056) by local users to gain root privileges by modifying process memory (assuming no ASLR) of a setuid-enabled executable that writes something deterministic to a file descriptor (like `su`), as demonstrated by Mempodipper
  - Fixed by Linus on Tue, 17 Jan 2012

- Most Android devices running Android 4.0.0 - 4.0.3 (like Galaxy Nexus) are based on kernels >= 2.6.39 (e.g. 3.0.8) and are vulnerable (via setuid-enabled `run-as`) so can be easily rooted
  - Fortunately, `run-as` does an early check to verify that it is running as the `root` or `shell` user so this particular exploit requires ADB-based shell access
  - Since `run-as` is statically linked, the mempodroid exploit needs to be given the exact offsets to `exit()` and `setresuid()` (which happen to be `0xd7f4` and `0xad4b` on Galaxy Nexus)

```
$ adb push mempodroid /data/local/tmp/.
$ adb shell chmod 755 /data/local/tmp/mempodroid
$ adb shell
shell@android:/ # id
uid=2000(shell) gid=2000(shell)
groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),1015(sdcard_rw),3001(net_bt_admin),3002(net_bt),3003(inet)
127|shell@android:/ /data/local/tmp/mempodroid 0xd7f4 0xad4b sh
shell@android:/ # id
uid=0(root) gid=0(root)
groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),1015(sdcard_rw),3001(net_bt_admin),3002(net_bt),3003(inet)
```

- Patched in Android 4.0.4

Incorrect Memory Driver Permissions on Samsung Devices

- On most Samsung devices, running Exynos 42xx family of SoC (GS2/3, GN1/2, etc.), the physical memory manager driver (`/dev/exynos-mem`) handle is set as readable and writable by all:
  - In the kernel source:

    linux/drivers/char/mem.c:

    ```
    #ifdef CONFIG_EXYNOS_MEM
        [14] = {"exynos-mem", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH
                | S_IWOTH, &exynos_mem_fops},
    #endif
    ```

  - In `ueventd.smdk4x12.rc`

- This driver is used by Camera and Gralloc HALs as well as other proprietary code
- It appears to provide complete access to all physical memory, which with simple code injection can be used to elevate access to root-level
- For more info on this bug as well as the controversy on how it was disclosed, see http://forum.xda-developers.com/showthread.php?p=35469999
- For the full list of devices affected, see http://www.zdnet.com/kernel-vulnerability-places-samsung-devices-at-risk-7000008862/

WebKit exploit

- Initially discovered on iOS, which also uses WebKit
- Based on buffer-overruns
- Enables the browser (or any app using a WebKit via Android’s `WebView` or directly via `libwebcore`) to execute arbitrary code
- A proof of concept creates a remote shell
- But, not a root exploit, because the application is sandboxed
    - Still, access to bookmarks, SSL sessions, stored passwords, etc.

- Patched in 2.2

To Root or Not To Root?

- Dangers on already rooted devices
- A malicious app can gain root access and inject a loadable kernel module into the kernel
- Very hard to detect
- Can open network-channels to leak information from the device

Malware Rootkits

- If we can root our own phone, so can malicious applications (e.g. trojans)
- But they have to be installed first
  - Easy, when they are repackaged versions of legit apps, so they look and feel "official"
  - Users are often confused into installing applications from Market that are not authentic, since they have no easy way to verify
  - Google has ability to both pull apps from Market as well as remotely uninstall them from users' devices (via C2DM) but this process is reactive - not proactive
- OEMs/Carriers are often too slow to patch the devices out in the field - so users remain vulnerable to these root exploits

Device Administration

Objectives of Device Administration Module

Device Administration API provide a framework for creating apps that can enforce certain policies on a device. These policies were designed with the enterprise requirements in mind. In this module you will learn what Device Admin API can and cannot do, as well as how to create an application that becomes an administrator on user’s device. Topics covered include:

- Overview of Device Administration API
- Supported policies
- Developing a Device Administration app
- Managing and enforcing policies

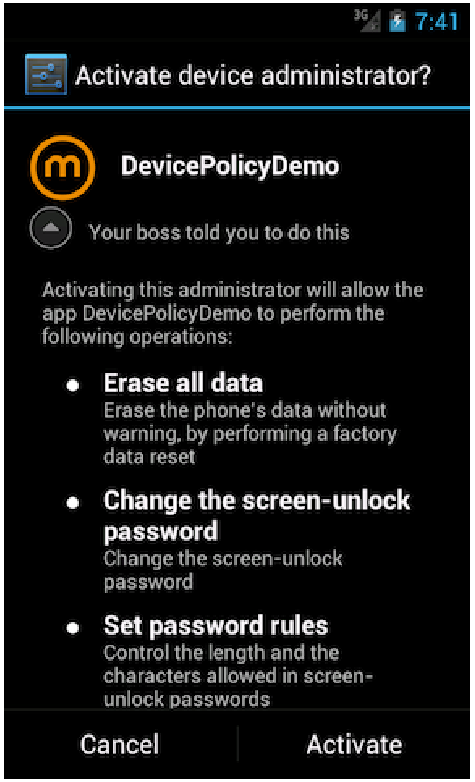
Device Administration Overview

The Android Device Administration API, introduced in Android 2.2, allows you to create security-aware applications that are useful in enterprise settings, such as:

- Email clients
- Security applications that do remote wipe
- Device management services and applications

You use the Device Administration API to write device admin applications that users install on their devices. The device admin application enforces desired *security policies*. Here’s how it works:

- A system administrator writes a device admin application that enforces remote/local device security policies.
- The application is installed on a user’s device.
- The system prompts the user to enable the device admin application.



- Once the users enable the device admin application, they are subject to its policies.

When enabled, in addition to enforcing security policies, the admin application can:

- Prompt the user to set a new password
- Lock the device immediately
- Perform a factory reset on the device, wiping the user data (if it has permission)

If a device contains multiple enabled admin applications, the strictest policy is enforced.

If users do not enable the device admin app, it remains on the device, but in an inactive state.

- Users will not be subject to its policies, but the application may disable some or all of its functionality.

If a user fails to comply with the policies (for example, if a user sets a password that violates the guidelines), it is up to the application to decide how to handle this.

- For example, the application may prompt the user to set a new password or disable some or all of its functionality.

To uninstall an existing device admin application, users need to first deactivate the application as a device administrator.

- Upon deactivation, the application may disable some or all of its functionality, delete its data, and/or perform a factory reset (if it has permission).

Security Policies

An admin application may enforce security policies regarding the device's screen lock PIN/password, including:

- The maximum inactivity time to trigger the screen lock
- The minimum number of PIN/password characters
- The maximum number of failed password attempts
- The minimum number of uppercase letters, lowercase letters, digits, and/or special password characters (Android 3.0)
- The password expiration period (Android 3.0)
- A password history restriction, preventing users from reusing the last *n* unique passwords (Android 3.0)

Additionally, a security policy can require device storage encryption as of Android 3.0 and disabling of camera as for Android 4.0.

The Device Administration Classes

The Device Administration API includes the following classes:

`DeviceAdminReceiver`

Base class for implementing a device administration component. This class provides a convenience for interpreting the raw intent actions that are sent by the system. Your Device Administration application must include a `DeviceAdminReceiver` subclass.

`DevicePolicyManager`

A class for managing policies enforced on a device. Most clients of this class must have published a `DeviceAdminReceiver` that the user has currently enabled. The `DevicePolicyManager` manages policies for one or more `DeviceAdminReceiver` instances.

`DeviceAdminInfo`

This class is used to specify metadata for a device administrator component.

Creating the Manifest

The manifest of your admin application must register your `DeviceAdminReceiver` as a `<receiver>`.

The `<receiver>` should set `android:permission="android.permission.BIND_DEVICE_ADMIN"` to ensure that only the system is allowed to interact with the broadcast receiver.

The `<receiver>` must have an `<intent-filter>` child element including one or more of the following `<action>`s, as defined in the `DeviceAdminReceiver` class:

`ACTION_DEVICE_ADMIN_ENABLED`
(Required) This is the primary action that a device administrator must implement to be allowed to manage a device. This is sent to a device administrator when the user enables it for administration.

`ACTION_DEVICE_ADMIN_DISABLE_REQUESTED`
Action sent to a device administrator when the user has requested to disable it, but before this has actually been done.

`ACTION_DEVICE_ADMIN_DISABLED`
Action sent to a device administrator when the user has disabled it.

`ACTION_PASSWORD_CHANGED`
Action sent to a device administrator when the user has changed the password of their device.

`ACTION_PASSWORD_EXPIRING`

Action periodically sent to a device administrator when the device password is expiring.

ACTION_PASSWORD_FAILED

Action sent to a device administrator when the user has failed at attempted to enter the password.

ACTION_PASSWORD_SUCCEEDED

Action sent to a device administrator when the user has successfully entered their password, after failing one or more times.

```
<receiver android:name="MyDeviceAdminReceiver"
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
        <action android:name="android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED" />
        <action android:name="android.app.action.ACTION_DEVICE_ADMIN_DISABLED" />
    </intent-filter>
    <!-- ... -->
</receiver>
```

Creating the Manifest (cont.)

Your `<receiver>` element must also include a `<meta-data>` child element specifying an XML resource declaring the policies used by your admin application.

- The `android:name` attribute must be `android.app.device_admin`.
- The `android:resource` must reference an XML resource in your application.
- For example:

```
<meta-data android:name="android.app.device_admin"
           android:resource="@xml/device_admin_sample" />
```

An example XML resource requesting all policies would be:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies>
        <limit-password />
        <watch-login />
        <reset-password />
        <force-lock />
        <wipe-data />
        <expire-password />
        <encrypted-storage />
        <disable-camera />
    </uses-policies>
</device-admin>
```

Your application needs to list only those policies it actually uses.

The `DeviceAdminReceiver` Class

The `DeviceAdminReceiver` class defines a set of methods that you can override to handle the device administration events broadcast by the system:

`void onEnabled(Context context, Intent intent)`
Called after the administrator is first enabled, as a result of receiving `ACTION_DEVICE_ADMIN_ENABLED`. At this point you can use `DevicePolicyManager` to set your desired policies.

`CharSequence onDisableRequested(Context context, Intent intent)`
Called when the user has asked to disable the administrator, as a result of receiving `ACTION_DEVICE_ADMIN_DISABLE_REQUESTED`. You may return a warning message to display to the user before being disabled, or `null` for no message.

`void onDisabled(Context context, Intent intent)`
Called prior to the administrator being disabled, as a result of receiving `ACTION_DEVICE_ADMIN_DISABLED`. Upon return, you can no longer use the protected parts of the `DevicePolicyManager` API.

`void onPasswordChanged(Context context, Intent intent)`
Called after the user has changed their password, as a result of receiving `ACTION_PASSWORD_CHANGED`.

`void onPasswordExpiring(Context context, Intent intent)`

Called periodically when the password is about to expire or has expired, as a result of receiving `ACTION_PASSWORD_EXPIRING`. (API 11)

`void onPasswordFailed(Context context, Intent intent)`

Called after the user has failed at entering their current password, as a result of receiving `ACTION_PASSWORD_FAILED`.

`void onPasswordSucceeded(Context context, Intent intent)`

Called after the user has succeeded at entering their current password, as a result of receiving `ACTION_PASSWORD_SUCCEEDED`.

Testing Whether the Admin Application is Enabled

You can query the `DevicePolicyManager` to test if your admin application is enabled:

```
DevicePolicyManager devicePolicyManager
    = (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName deviceAdminComponentName
    = new ComponentName(this, MyDeviceAdminReceiver.class);
boolean isActive = devicePolicyManager.isAdminActive(deviceAdminComponentName);
```

You could then enable or disable features of your application depending on whether it is an active device administrator.

Enabling the Application

Your application must explicitly request the user to enable it for device administration. To do so:

1. Create an implicit Intent with the `DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN` action:

```
Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
```

2. Add an extra identifying your `DeviceAdminReceiver` component:

```
ComponentName deviceAdminComponentName
    = new ComponentName(this, MyDeviceAdminReceiver.class);
intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, deviceAdminComponentName);
```

3. Optionally, provide an explanation as to why the user should activate the admin application:

```
intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION, "Your boss told you to do this");
```

4. Use the Intent with `startActivityForResult()` to display the activation dialog:

```
startActivityForResult(intent, ACTIVATION_REQUEST);
```

5. You can test for successful activation in your Activity's `onActivityResult()` method:

```java
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case ACTIVATION_REQUEST:
            if (resultCode == Activity.RESULT_OK) {
                Log.i("DeviceAdminSample", "Administration enabled!");
            } else {
                Log.i("DeviceAdminSample", "Administration enable FAILED!");
            }
            return;
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

Setting Password Quality Policies

`DevicePolicyManager` includes APIs for setting and enforcing the device screen lock password policy.

- The `setPasswordQuality()` lets you set basic password requirements for your admin application, using these constants:

  `PASSWORD_QUALITY_ALPHABETIC`
  The user must enter a password containing at least alphabetic (or other symbol) characters.

  `PASSWORD_QUALITY_ALPHANUMERIC`
  The user must enter a password containing at least both numeric and alphabetic (or other symbol) characters.

  `PASSWORD_QUALITY_NUMERIC`
  The user must enter a password containing at least numeric characters.

  `PASSWORD_QUALITY_SOMETHING`
  The policy requires some kind of password, but doesn't care what it is.

  `PASSWORD_QUALITY_UNSPECIFIED`
  The policy has no requirements for the password.

  `PASSWORD_QUALITY_COMPLEX`
  (API 11) The user must have entered a password containing at least a letter, a numerical digit and a special symbol.

- Once you have set the password quality, you may also specify a minimum length (except with `PASSWORD_QUALITY_SOMETHING` and

PASSWORD_QUALITY_UNSPECIFIED) using the setPasswordMinimumLength() method.

- For example:

```
devicePolicyManager.setPasswordQuality(deviceAdminComponentName, PASSWORD_QUALITY_ALPHANUMERIC);
devicePolicyManager.setPasswordMinimumLength(deviceAdminComponentName, 6);
```

Your application’s policy metadata resource must request the <limit-password /> policy to control password quality; otherwise these methods throw a security exception.

Setting Password Quality Policies, API 11

Beginning with Android 3.0, the `DevicePolicyManager` class includes methods that give you greater control over the contents of the password. Here are the methods for fine-tuning a password's contents:

- `setPasswordMinimumLetters()`
- `setPasswordMinimumLowerCase()`
- `setPasswordMinimumUpperCase()`
- `setPasswordMinimumNonLetter()`
- `setPasswordMinimumNumeric()`
- `setPasswordMinimumSymbols()`

You can also set the password expiration timeout, and prevent users from reusing the last *n* unique passwords:

- `setPasswordExpirationTimeout()`
- `setPasswordHistoryLength()`

Additionally, Android 3.0 introduced support for a policy requiring the user to encrypt the device, which you can set with:

- `setStorageEncryption()`

Your application's policy metadata resource must request the `<limit-password />` policy to control password quality; otherwise these methods throw a security exception.

Similarly, it must request the `<expire-password />` and `<encrypted-storage />` policies to control those features without throwing a security exception.

Setting the Device Password

You can test if the current device password meets the quality requirements by calling `DevicePolicyManager.isActivePasswordSufficient()`, which returns a boolean result.

If necessary, you can start an activity prompting the user to set a password as follows:

```
Intent intent = new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
startActivity(intent);
```

Your application can also perform a password reset on the device using `DevicePolicyManager.resetPassword()`. This can be useful if your application is designed to support remote administration, with a new password being provided from a central administration system.

Your application's policy metadata resource must request the `<reset-password />` policy to reset the password; otherwise `resetPassword()` throws a security exception.

Locking and Wiping the Device

Your application can lock the device programmatically using `DevicePolicyManager.lockNow()`.

You can wipe the user data of the device, performing a factory reset, using `DevicePolicyManager.wipeData()`.

Additionally, you can set the maximum number of allowed failed password attempts before the device is wiped automatically by calling `DevicePolicyManager.setMaximumFailedPasswordsForWipe()`

Your application's policy metadata resource must request the `<wipe-data />` policy to wipe the data either explicitly or set the maximum failed passwords for wipe; otherwise a security exception is thrown. `setMaximumFailedPasswordsForWipe()` also requires the `<watch-login />` policy.

The `lockNow()` method requires your application to request the `<force-lock />` policy to avoid throwing a security exception.

Device Administration Demo

In this example app, you will see how to write an application that requests to device administration privileges, and once it gets them, allows user to lock or reset the device.

We are going to look at the following files:

- Android Manifest File
- XML Resource File
- Device Admin Receiver Component
- Activity

The source code for this project is available at https://marakana.com/static/courseware/android/DevicePolicyDemo.zip

Android Manifest File

This is where we register our device administration receiver component. It appears as another receiver declaration.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.marakana.android.devicepolicydemo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".DevicePolicyDemoActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- This is where we register our receiver -->
        <receiver
            android:name=".DemoDeviceAdminReceiver"
            android:permission="android.permission.BIND_DEVICE_ADMIN" >
            <intent-filter>
```

```
                <!-- This action is required -->
                <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
            </intent-filter>

            <!-- This is required this receiver to become device admin component. -->
            <meta-data
                android:name="android.app.device_admin"
                android:resource="@xml/device_admin_sample" />
        </receiver>
    </application>

</manifest>
```

Notice that `<receiver>` element now includes required `android:permission="android.permission.BIND_DEVICE_ADMIN"` permission declaration.

We also must include the appropriate intent action filter `android.app.action.DEVICE_ADMIN_ENABLED` as well as the `<meta-data/>` element that specifies that this receiver users `@xml/device_admin_sample` resource, which we’ll look at next.

XML Resource File

This XML resource file, referenced from `AndroidManifest.xml` specifies what policies we are interested in.

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
        <uses-policies>
                <limit-password />
                <watch-login />
                <reset-password />
                <force-lock />
                <wipe-data />
                <expire-password />
                <encrypted-storage />
        </uses-policies>
</device-admin>
```

In this example, we ask for most of the available policies merely to illustrate what is available. In a real-world example, you should only ask for policies that you really require.

Device Admin Receiver Component

This is the main device administration component. It is basically a specialized `BroadcastReceiver` class that implements some callbacks specific to device administration.

```
package com.marakana.android.devicepolicydemo;

import android.app.admin.DeviceAdminReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.widget.Toast;

/**
 * This is the component that is responsible for actual device administration.
 * It becomes the receiver when a policy is applied. It is important that we
 * subclass DeviceAdminReceiver class here and to implement its only required
 * method onEnabled().
 */
public class DemoDeviceAdminReceiver extends DeviceAdminReceiver {
    static final String TAG = "DemoDeviceAdminReceiver";

    /** Called when this application is approved to be a device administrator. */
    @Override
    public void onEnabled(Context context, Intent intent) {
        super.onEnabled(context, intent);
        Toast.makeText(context, R.string.device_admin_enabled,
                Toast.LENGTH_LONG).show();
        Log.d(TAG, "onEnabled");
    }
```

```
    /** Called when this application is no longer the device administrator. */
    @Override
    public void onDisabled(Context context, Intent intent) {
        super.onDisabled(context, intent);
        Toast.makeText(context, R.string.device_admin_disabled,
                Toast.LENGTH_LONG).show();
        Log.d(TAG, "onDisabled");
    }

    @Override
    public void onPasswordChanged(Context context, Intent intent) {
        super.onPasswordChanged(context, intent);
        Log.d(TAG, "onPasswordChanged");
    }

    @Override
    public void onPasswordFailed(Context context, Intent intent) {
        super.onPasswordFailed(context, intent);
        Log.d(TAG, "onPasswordFailed");
    }

    @Override
    public void onPasswordSucceeded(Context context, Intent intent) {
        super.onPasswordSucceeded(context, intent);
        Log.d(TAG, "onPasswordSucceeded");
    }



}
```

Notice that we subclass `DeviceAdminReceiver` class. This is the required for this component to be able to receive policy notifications.

We also must implement the required `onEnabled()` method that is called when the policy administration is first enabled.

We don’t really do much here other than log what happened to visually illustrate the execution of this code.

Activity

The activity acts as our demo client in this case. The significant methods are `onClick()`, `onCheckedChanged()` and `onActivityResult()`.

```
package com.marakana.android.devicepolicydemo;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.Toast;
import android.widget.ToggleButton;

public class DevicePolicyDemoActivity extends Activity implements
        OnCheckedChangeListener {
    static final String TAG = "DevicePolicyDemoActivity";
    static final int ACTIVATION_REQUEST = 47; // identifies our request id
    DevicePolicyManager devicePolicyManager;
    ComponentName demoDeviceAdmin;
    ToggleButton toggleButton;

    /** Called when the activity is first created. */
    @Override
```

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    toggleButton = (ToggleButton) super
            .findViewById(R.id.toggle_device_admin);
    toggleButton.setOnCheckedChangeListener(this);

    // Initialize Device Policy Manager service and our receiver class
    devicePolicyManager = (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);
    demoDeviceAdmin = new ComponentName(this, DemoDeviceAdminReceiver.class);
}

/**
 * Called when a button is clicked on. We have Lock Device and Reset Device
 * buttons that could invoke this method.
 */
public void onClick(View v) {
    switch (v.getId()) {
    case R.id.button_lock_device:
        // We lock the screen
        Toast.makeText(this, "Locking device...", Toast.LENGTH_LONG).show();
        Log.d(TAG, "Locking device now");
        devicePolicyManager.lockNow();
        break;
    case R.id.button_reset_device:
        // We reset the device - this will erase entire /data partition!
        Toast.makeText(this, "Locking device...", Toast.LENGTH_LONG).show();
        Log.d(TAG,
                "RESETing device now - all user data will be ERASED to factory settings");
        devicePolicyManager.wipeData(ACTIVATION_REQUEST);
        break;
    }
}
```

```java
/**
 * Called when the state of toggle button changes. In this case, we send an
 * intent to activate the device policy administration.
 */
public void onCheckedChanged(CompoundButton button, boolean isChecked) {
    if (isChecked) {
        // Activate device administration
        Intent intent = new Intent(
                DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
        intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN,
                demoDeviceAdmin);
        intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                "Your boss told you to do this");
        startActivityForResult(intent, ACTIVATION_REQUEST);
    }
    Log.d(TAG, "onCheckedChanged to: " + isChecked);
}

/**
 * Called when startActivityForResult() call is completed. The result of
 * activation could be success of failure, mostly depending on user okaying
 * this app's request to administer the device.
 */
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
    case ACTIVATION_REQUEST:
        if (resultCode == Activity.RESULT_OK) {
            Log.i(TAG, "Administration enabled!");
            toggleButton.setChecked(true);
        } else {
            Log.i(TAG, "Administration enable FAILED!");
            toggleButton.setChecked(false);
```

```
            }
            return;
        }
        super.onActivityResult(requestCode, resultCode, data);
    }

}
```

onCheckedChanged() is invoked when the toggle button changes state. It sends an intent requesting that this application be granted device administration permissions on this device. User has to allow this request. The result of user's action is then passed to onActivityResult() method.

onClick() method processes button clicks for lock and reset buttons. Note that its calls to DevicePolicyManager **will** lock and wipe out the device user data partition, respectively.

Lab (Device Administration)

Device-admin-enable an existing Android app:

1. Download http://marakana.com/static/courseware/android/SecureNote.zip
2. Expand into your (Eclipse) workspace
3. Import into Eclipse
4. Using Device Admin APIs
    1. Require that the screen-lock password be set (of at least 6 characters with at least one digit)
    2. Automatically wipe the device after 5 invalid login attempts
5. Check against the solution available at http://marakana.com/static/courseware/android/DeviceAdministeredSecureNote.zip

Malware

Lessons From The Field

- Dissecting Android Malware: Characterization and Evolution paper by Yajin Zhou and Xuxian Jiang from North Carolina State University
- Collected and analyzed over 1200 malware samples from August 2010 to October 2011 for the Android Malware Genome Project
- From these 1200+ samples, they found that:
  - 86% are repackaged versions of existing apps with malicious payloads
    - These include paid apps, popular games, utilities (including security), as well as porn-related apps
    - The malicious payload often "hidden" with class-file names that look legitimate (e.g. `com.sec.android.provider.drm`, `com.google.ssearch`, or `com.google.update`)
    - Some take advantage of compromised platform-level keys (for some popular custom ROMs)
  - 7% look benign because they don't *contain* the malicious code, but instead offer the user an option to install the *updated* version of itself, which does contain the malicious code
    - This is done to avoid detection
    - The update may be downloaded or packaged as an asset or a resource
  - Some take advantage of Dalvik's run-time class-loading capabilities via dalvik.system.DexClassLoader
    - Avoid updating the whole app (which requires user confirmation)
    - Download (encrypted) DEX code, load it into the Dalvik VM, and execute it

```java
DexClassLoader classLoader = new DexClassLoader(
  new File(context.getFilesDir(), "bad.jar").getAbsolutePath(), // contains classes.dex
  context.getDir("dex", 0).getAbsolutePath(), // create a directory for optimized DEX code
  null, // we could also specify a path for native libraries!
  context.getClassLoader());
Class<?> badClass = classLoader.loadClass("com.malicious.app.Bad");
Bad bad = (Bad) badClass.newInstance();
bad.run();
```

- Some use *drive-by-download* to send the user to a site, which "analyzes" the user's device, finds a problem, and attempts to trick the user into installing a malware app that claims to address the problem
- 13.8% are straight-out spyware/malware that hide their malicious behavior behind some seemingly useful functionality
- 83.3% are activated by `BOOT_COMPLETED` broadcast intent (among other events, like `SMS_RECEIVED`)
- Compared to normal apps, malware apps request more (~ 20) permissions than popular non-malware apps (~ 4)
    - `INTERNET`: 98% vs 89%
    - `RECEIVE_BOOT_COMPLETED`: 53.7% vs 10.9%
    - `READ_SMS`: 62.7% vs 2.6%
    - `WRITE_SMS`: 52.2% vs ?% (not in the top 20)
    - `RECEIVE_SMS`: 39.6% vs ?% (not in the top 20)
    - `SEND_SMS`: 43.9% vs 3.4%
    - `READ_CONTACTS`: 36.3% vs 5.6%
    - `RESTART_PACKAGES`: 26.4% vs 2.6%
- 36.7% leverage root-level exploits

    - Common payloads include exploid, RageAgainstTheCage, Zimperlich, GingerBreak, and asroot
    - 29.5% come with multiple root exploits
    - Some encrypt the root-exploit code and store it as asset/resource files to avoid detection
  - 93% turn the compromised phones into a botnet controlled over the network via C&C servers
    - The names of C&C servers are often obfuscated (or encrypted) to avoid detection
    - Some store the names of C&C servers in obfuscated posts/comments on public blog/forum sites
    - Some use SMS for control
  - 45.3% have the built-in support of sending out SMS messages to premium-rate numbers or making phone calls without user awareness
    - The numbers to SMS/call often come from C&C servers
    - Some subscribe users to premium services via outbound SMS, trap incoming SMS replies, and send out confirmation SMS messages to "complete the transaction"
  - 51.1% harvest user's information (e.g. user accounts, SMS messages, and dialed numbers)
  - Some come with "shadow payloads" that get installed (via root), so that the payload remains even if the original app gets discovered and removed
  - Some use obfuscation/"encryption" or native/JNI code to make it harder to detect and analyze
- The also tested major anti-virus/anti-malware applications with the following rates of detection:
  - AVG Antivirus Free - 54.7%
  - Lookout Security & Antivirus - 79.6%
  - Norton Mobile Security Lite - 20.2%

- Trend Micro Mobile Security Personal Edition - 76.7%

Detection

- Use `android.content.pm.PackageManager.getInstalledPackages(int flags)` for the full app scan
  - By package-name (e.g. check against a black/white-list):

```
for (PackageInfo packageInfo : getPackageManager().getInstalledPackages(0)) {
  if (packageInfo.packageName.startsWith("com.malware")) {
     // found malware!
  }
}
```

  - By permission (e.g. check against a black-list of inappropriate permissions):

```
for (PackageInfo packageInfo : getPackageManager().getInstalledPackages(PackageManager.GET_PERMISSIONS)) {
  if ((packageInfo.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0) {
    continue; // skip system apps
  }
  if (packageInfo.requestedPermissions != null) {
    for (String requestedPermission : packageInfo.requestedPermissions) {
      if (requestedPermission.equals(android.Manifest.permission.SEND_SMS)) {
        // found malware!
      }
    }
  }
}
```

  - By signature (e.g., check against a black/white-list of issuers):

```java
CertificateFactory certificateFactory = CertificateFactory.getInstance("X509");
PublicKey trustedIssuerPublicKey = // get some trusted key
for (PackageInfo packageInfo : getPackageManager().getInstalledPackages(PackageManager.GET_SIGNATURES)) {
  if ((packageInfo.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0) {
    continue; // skip system apps
  }
  if (packageInfo.signatures != null) {
    for (Signature signature : packageInfo.signatures)  {
      InputStream input = new ByteArrayInputStream(signature.toByteArray());
      try {
        X509Certificate cert = (X509Certificate) certificateFactory.generateCertificate(input);
        cert.checkValidity();
        cert.verify(trustedIssuerPublicKey);
        if (cert.getIssuerDN().getName().equals("CN=Some One,O=Bad,C=US")) {
          // found malware!
        }
      } catch (CertificateException e) {
        // found malware!
      }
    }
  }
}
```

- By code:

```java
for (PackageInfo packageInfo : getPackageManager().getInstalledPackages(0)) {
  if ((packageInfo.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0) {
    continue; // skip system apps
  }
  String apk = packageInfo.applicationInfo.sourceDir;
  FileInputStream in = new FileInputStream(apk); // this is legal!
  try {
    // scan for malicious code (unzip first) or upload to remote server for analysis
  } finally {
    in.close();
  }
  String nativeDir = packageInfo.applicationInfo.nativeLibraryDir; // these are readable by all!
  // scan native libraries for malicious code or upload to remote server for analysis
}
```

- Listen for `android.intent.action.PACKAGE_ADDED` broadcasts and verify new/updated apps:

AndroidManifest.xml:

```xml
<manifest … >
  …
  <application … >
    …
    <receiver android:name=".ApplicationInstallReceiver" >
      <intent-filter>
        <action android:name="android.intent.action.PACKAGE_ADDED" />
        <action android:name="android.intent.action.PACKAGE_REPLACED" />
        <data android:scheme="package" />
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

ApplicationInstallReceiver.java:

```java
…
public class ApplicationInstallReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Uri packageUri = intent.getData();
    String packageName = packageUri.getEncodedSchemeSpecificPart();
    int flags = PackageManager.GET_PERMISSIONS | PackageManager.GET_SIGNATURES;
    try {
      PackageInfo packageInfo =
        context.getPackageManager().getPackageInfo(packageName, flags);
      // verify packageInfo
    } catch (NameNotFoundException e) {
      // handle
    }
  }
}
```

Removal

- Once a malicious app is found, offer the user a chance to delete it

```
Uri packageURI = Uri.parse("package:com.malicous.app");
Intent uninstallIntent = new Intent(Intent.ACTION_DELETE, packageURI);
startActivity(uninstallIntent);
```

- Cannot use `PackageManager.deletePackage(String packageName, IPackageDeleteObserver observer, int flags)`
    - Requires `signatureOrSystem` permission
    - The API call is hidden (unless we use the framework APIs or reflection)

- Sample source-code available
    - As a ZIP archive: https://github.com/marakana/AntiMalware/zipball/master
    - By Git: `git clone https://github.com/marakana/AntiMalware.git`

Google’s App Verification Service

- Introduced in Android 4.2
- Apps installed via Google Play are already verified on the "server side"
- Meant to protect against malicious apps installed via alternative channels:
    - Side-loaded from SDCARD, remote HTTP server, message attachment, etc.
    - Alternative app markets (such as Amazon app-store)
    - ADB installed (including through and IDE such as Eclipse)

- Enabled via Setting → Security → Verify apps
- The actual implementation is a combination of Google Play client-side app, and Google Play cloud infrastructure
- When an app is about to be installed:
    1. Google Play client collects the information about the app
        - Package/app name
        - App version
        - SHA1 digest of the app contents
        - The location (e.g. URL) from where the app is being installed
        - Information about the device (ID, IP, etc)

    2. Google Play cloud uses this information to *verify* the app based on some internal heuristics and responds to the client

3. When it receives the response, the client:
    1. Blocks the installation if the app is found to be "dangerous"
    2. Gives user a warning if the app is found to be "potentially harmful or dangerous", at which point the user can cancel the installation or proceed at their own risk
    3. Resumes the installation without user's intervention if the app appears to be safe

- As of November 30, 2012, the Google's App Verification Service was found to have a detection rate of only 15-20% - on a sample of 1260 apps belonging to 49 families of known Android malware (compared to 50-100% for established AV vendors)
    - Most malware is distributed such that it is mutated on every download, so digest-only-based detection offers weak protection

- For more info, see An Evaluation of the Application ("App") Verification Service in Android 4.2 from NC State University

SE Android (SE-Linux on Android)

- Discretionary Access Control (DAC) vs. Mandatory Access Control (MAC)
- DAC on Android
    - Default form of access control (also true for most Linux systems)
    - Access to data is controlled by the app developers
        - Except for root
    - Based on user/group identity associated with each app and its data
    - Coarse-grained decentralized control
        - No easy way to establish a system-wide policy
- MAC with SE-Linux
    - System-wide security policy applies to all processes, data, and system operations
    - Based on security labels
    - Confines flawed/malicious apps as well as system processes (including those that run as root!)
        - Prevent privilege escalation
    - Centralized/manageable device-wide policy
- See SEAndroid website on how to build it

- From Stephen Smalley's "The Case for Security Enhanced (SE) Android":
- SE Android brings MAC to Android and has the following goals:
  - Prevent privilege escalation by apps
  - Prevent data leakage by apps
  - Prevent bypass of security features
  - Enforce legal restrictions on data
  - Protect integrity of apps and data
  - Targeted at consumers, businesses, and government

- What SE Android *can* help with:
  - 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 SE Android *can not* help with:
  - Kernel vulnerabilities (in general)
    - May block exploitation of specific vulnerabilities

    - Anything allowed by security policy
        - Good policy is important
        - Architecture of system applications matters
            - Decomposition, least privilege

- Challenges
    - Kernel
        - No support for per-file security labeling (yaffs2)
        - Unique kernel subsystems lack SELinux support
    - Userspace
        - No existing SELinux support
        - Sharing through framework services
    - Policy
        - Existing policies unsuited to Android

Other Security Concerns

- Unintentional app vulnerabilities
    - For example (CVE-2011-1717) "Skype for Android stored sensitive user data without encryption in sqlite3 databases that have weak permissions, which allows local applications to read user IDs, contacts, phone numbers, date of birth, instant message logs, and other private information."

- Push-based installation of apps from Market (based on the Google account)
    - If the Google account associated with the device is compromised, malicious applications could be pushed directly to affected owner's devices

- Social-engineering vectors of attack
    - Lack of the developer-user trust model
    - Reactive vs preventative security of Market-installed apps

- Firewall
    - Android uses all-or-nothing access to networking
        - Requested via the INTERNET permission, mapped to `inet` group, enforced via `ANDROID_PARANOID_NETWORK` kernel extension
        - `iptables` is available, but not exposed to the user
        - No easy way to setup a firewall policy controlling/limiting access to network resources

    - WhisperMonitor is a 3rd party custom ROM that tries to address this shortcoming, providing a full-fledged application firewall manageable by the end user

- Encryption of communication
  - Whisper Systems' RedPhone application uses ZRTP-encrypted SMS messages to establish calls over a VOIP connection (hidden behind an alternative dialer)
    - ZRTP was designed by PGP inventor Phillip Zimmerman

- Rogue applications signed using platform keys (targeting custom ROMs): jSMSHider
  - http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/

- App obfuscation
  - Proguard

- Recovery mode
  - State of the device while in the recovery mode

- Controlling access to private content with a privacy manager
  - North Carolina State University's TISSA (Taming Information-Stealing Smartphone Applications) privacy manager controls access to user's private data (Location, Phone Identity, Contacts, and Call Log)
    - On an app-by-app basis, users decide how their data is shared: Trusted (unlimited), Bogus (false), Anonymized (filtered), or Empty (pretend that there is no data)
    - http://mobile.engadget.com/2011/04/19/ncsu-teases-tissa-for-android-a-security-manager-that-keeps-per/

- Security issues with device skins (e.g. HTC Sense): "Security hole in HTC phones gives up e-mail addresses, location"

- http://arstechnica.com/gadgets/news/2011/10/security-hole-in-htc-phones-gives-up-e-mail-addresses-location.ars
- http://www.dnaindia.com/scitech/report_new-android-security-flaw-can-wipe-all-data-from-smart-phones_1747879

- Dual-mode phones (work and pleasure) - e.g. AT&T Toggle (a.k.a. Enterproid Divide)
  - http://www.technologyreview.com/communications/38865/?p1=A1

- Non-market installations
  - Server-side polymorphism to generate unique variants and avoid detection by anti-malware
  - http://www.infoworld.com/d/security/symantec-warns-of-android-trojans-mutate-every-download-185664?source=fssr

- Google Bouncer
  - Upon application upload to Market, Google Bouncer scans it for known malware, spyware and trojans
  - Application is then run in a simulated environment (inside Google's cloud) and tested for hidden and malicious behavior (comparing it to previously analyzed apps)
  - New developer accounts are checked against previously known offenders
  - Already-installed malicious apps can be automatically removed (remote kill-switch)
  - http://googlemobile.blogspot.com/2012/02/android-and-security.html

- Security of `WebView` - calling `addJavaScriptInterface()` can lead to XSS, CSRF especially if used over un-encrypted HTTP

Summary

We talked about the following:

- Android Stack
- Security Essentials
  - Security Architecture
  - Application Signing
  - User IDs
  - File Access
  - Multiuser Support
  - Permissions
- Advanced Security
  - Encryption
  - Rooting
  - Device Admin
  - Malware
  - SE Android
  - Other Security Concerns

Questions?

Thank you for your attention!

Slides and screencast from this class will be posted to: http://mrkn.co/cgcsn

You can follow me here:

- @MarkoGargenta
- +Marko Gargenta
- http://marakana.com/s/author/2/marko_gargenta

Legal Disclaimer