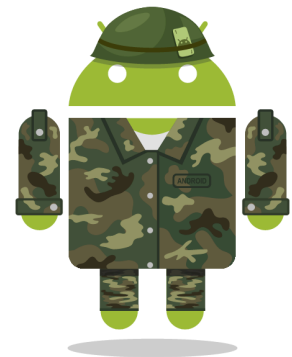


Deep Dive into Android Security

by Aleksandar Gargenta, Marakana Inc.



video/slides at
<http://mrkn.co/andsec>



AN JA AP V6 JQ PY TC SE JS RB JB H5  marakana



About Aleksandar (Saša) Gargenta

- Developing in Java since 1996 – mostly server-side
- Hacking Android since 2008 – from the SDK to the kernel
- Teaching Java, Android, etc. at Marakana since 2005
 - <http://marakana.com/>
- Founder & Organizer of San Francisco Java User Group
 - <http://www.sfjava.org/>
- Founder & Organizer of San Francisco Android User Group
 - <http://www.sfandroid.org/>
- Co-founder & co-organizer of San Francisco HTML5 User Group
 - <http://www.sfhtml5.org/>
- Writing *Android Internals* for O'Reilly (ETA? yesterday)
- Worked on SMS, MMS, WAP Push, but also Linux and system administration in past life





Overview

- Why care?
- Android Security Model
- Permissions on Android
- Encryption on Android
- Device Admin
- Rooting Android Devices
- Anti-rooting? ASLR? SE-Linux? Locking bootloaders?
- Tap-jacking
- Developer Best Practices
- Other concerns

Why Care?

- "Scary Android security hole in 99% of phones: PANIC!" – Computerworld
- "HTC promises fix for massive Android security flaw" – MobileBeat
- "Android users are two and a half times as likely to encounter malware today than 6 months ago..." – Lookout Mobile Threat Report
- "Today's mobile devices are a mixed bag when it comes to security... still vulnerable to many traditional attacks...." - Carey Nachenberg, Symantec
- "Android Security Will Be Big News in 2011: 10 Reasons Why" - eWeek
- "The growth rate in malware within Android is huge; in the future there will definitely be more." - Nikolay Grebennikov, CTO of Kaspersky
- "Any time a technology becomes adopted and popular, that technology will be targeted by the bad guys." - Jay Abbott, PricewaterhouseCoopers LLP

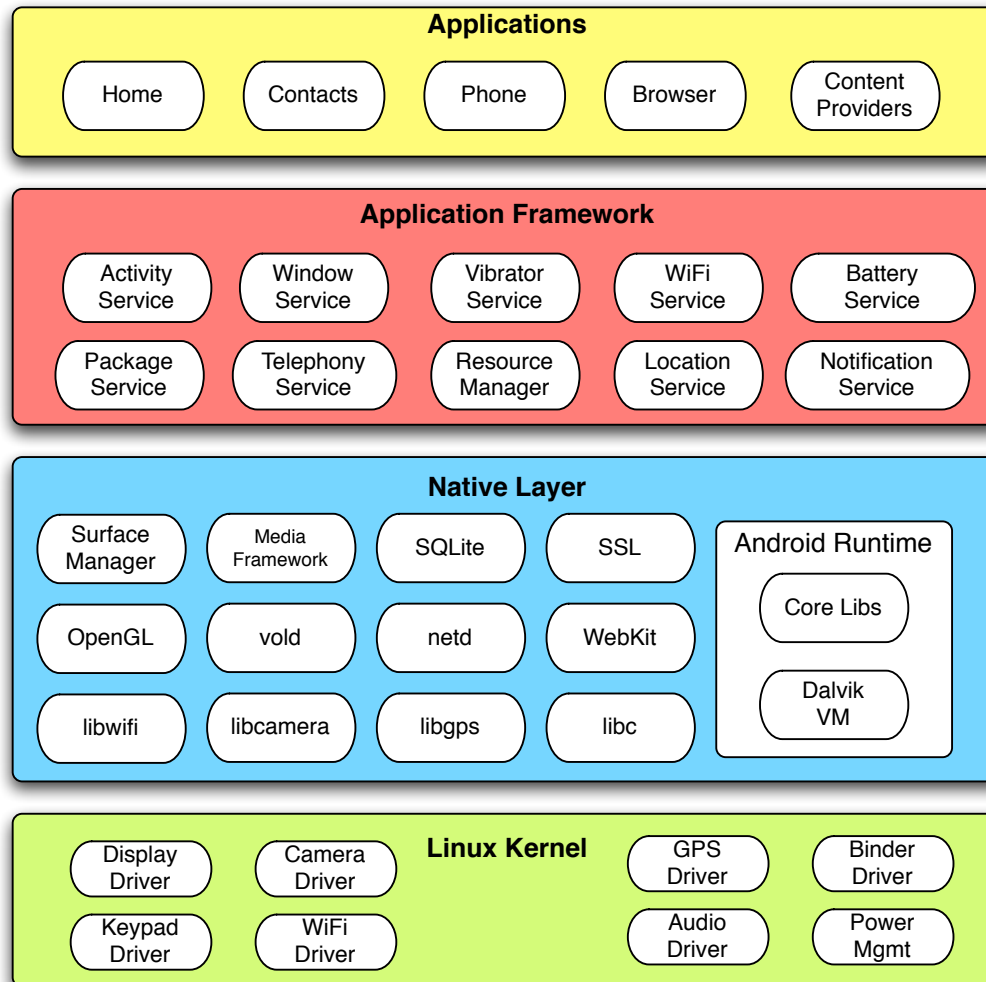




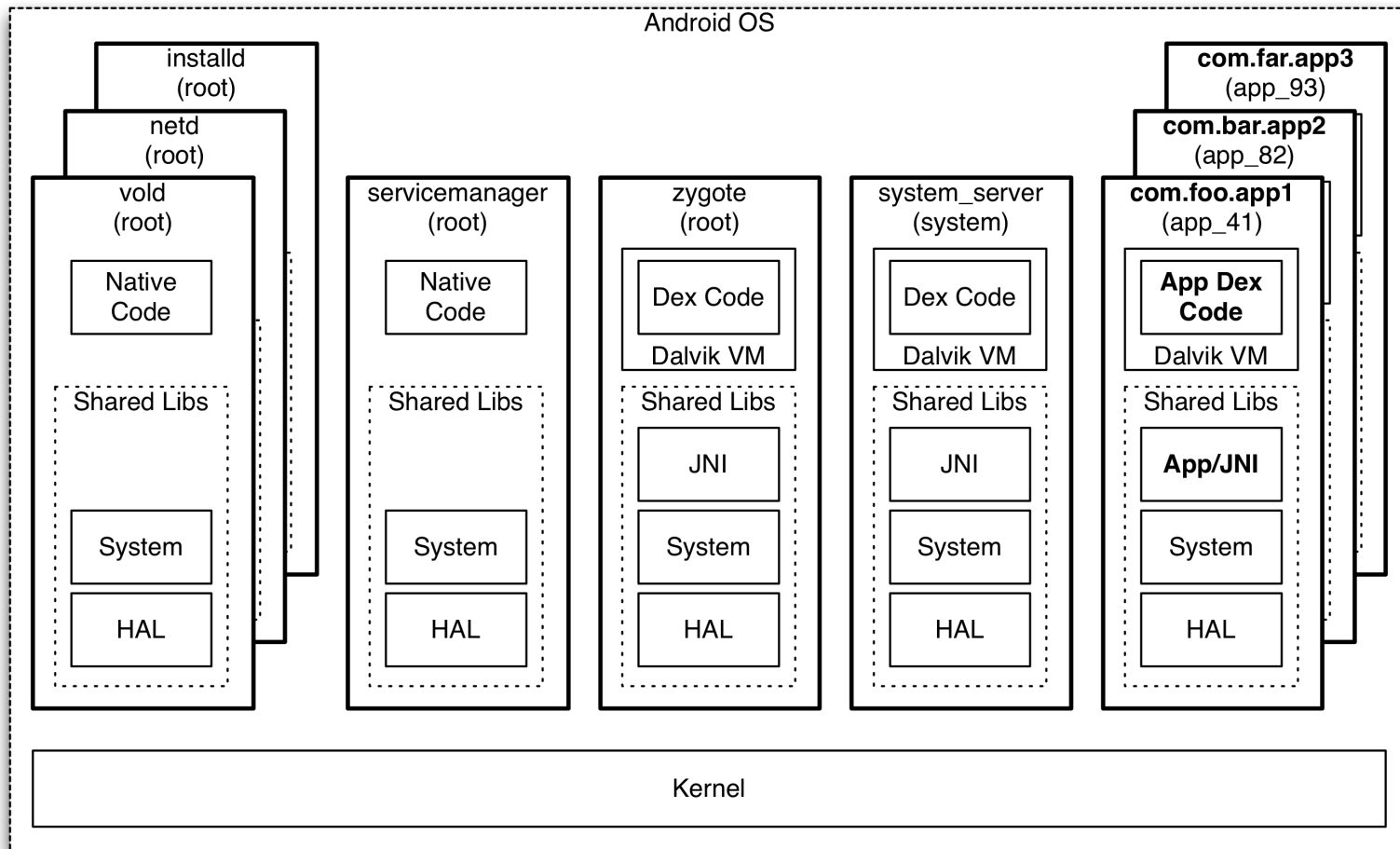
Foundations of Android Security

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

Android Stack (revisited)



Android Application Isolation

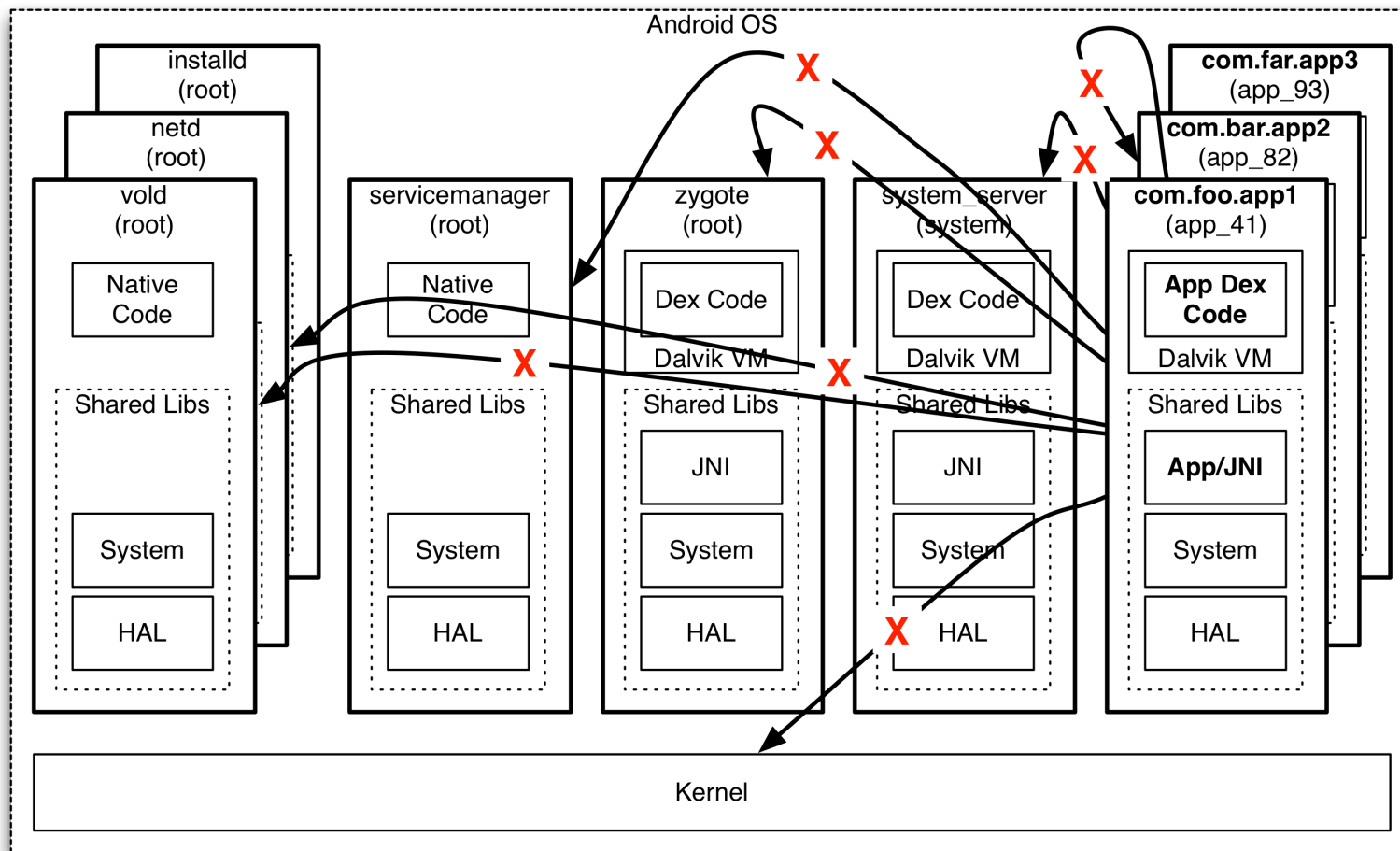




Android Application Isolation

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

Default Android Permissions Policy



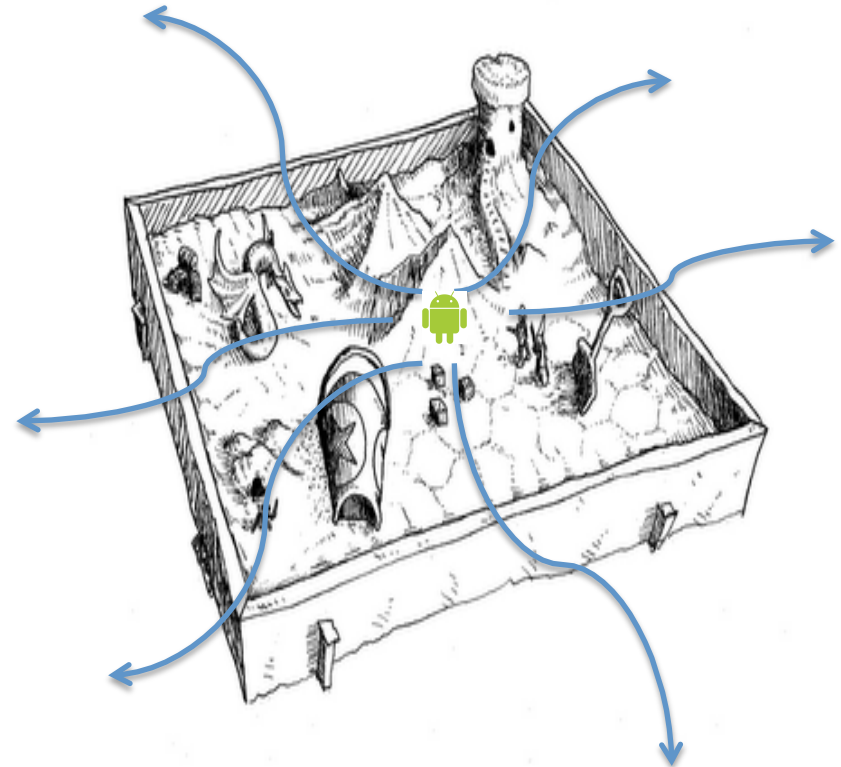


Default Android Permissions Policy

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

Escaping The Sandbox

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





Built-in Android Permissions

ACCESS_FINE_LOCATION, ACCESS_NETWORK_STATE,
ACCESS_WIFI_STATE, ACCOUNT_MANAGER,
BLUETOOTH, BRICK, CALL_PHONE, CAMERA,
CHANGE_WIFI_STATE, DELETE_PACKAGES,
INSTALL_PACKAGES, INTERNET, MANAGE_ACCOUNTS,
MASTER_CLEAR, READ_CONTACTS, READ_LOGS,
READ_SMS, RECEIVE_SMS, RECORD_AUDIO,
SEND_SMS, VIBRATE, WAKE_LOCK, WRITE_CONTACTS,
WRITE_SETTINGS, WRITE_SMS, ...

<http://developer.android.com/reference/android/Manifest.permission.html>



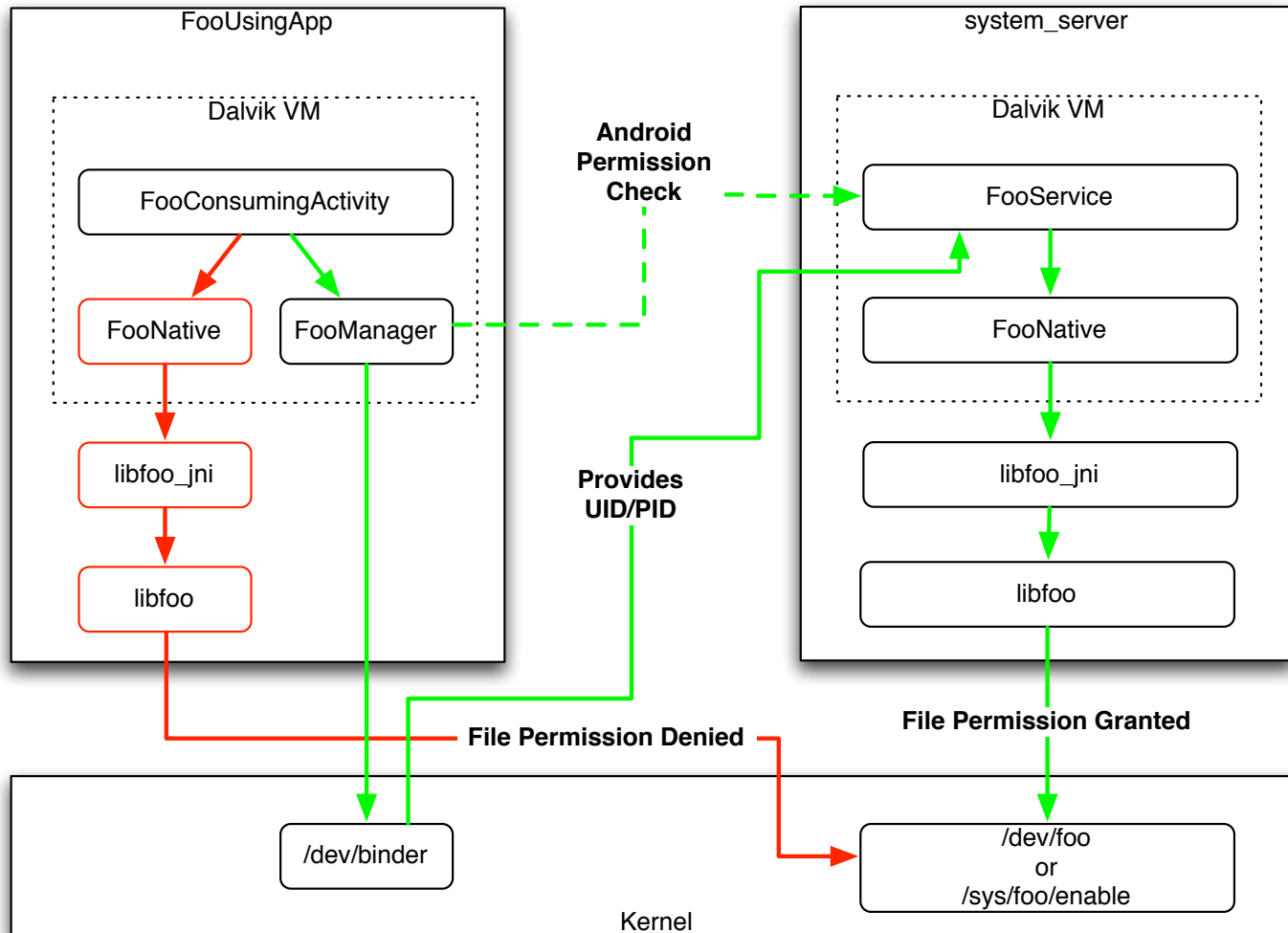
Example: Buddy Tickler App

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

- **App's** `AndroidManifest.xml`:

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

Logical Permission Enforcement



Permission Enforcement Example

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

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

- Only apps with `android.permission.VIBRATE` permissions can access `VibratorService.vibrate(...)` method:
`frameworks/base/services/java/com/android/server/VibratorService.java`

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



Kernel Permission Enforcement

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

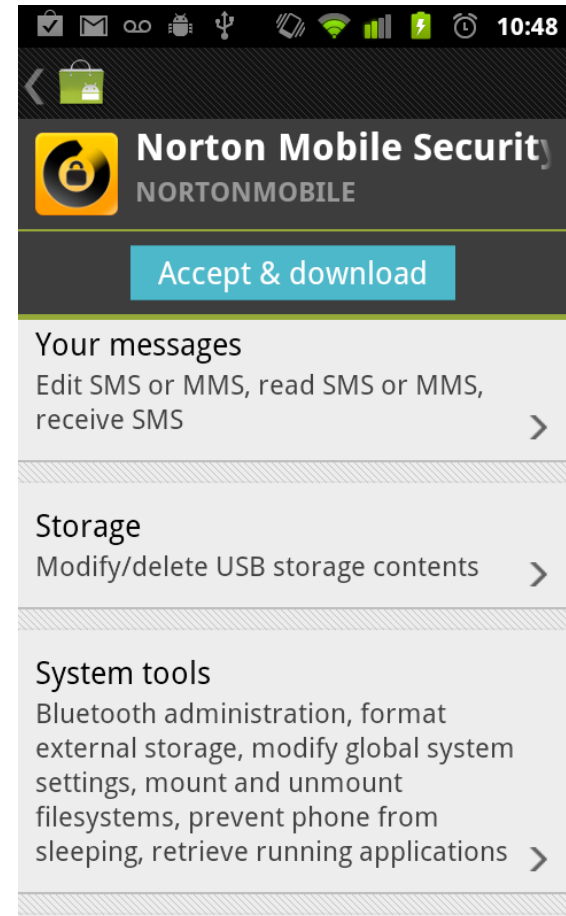
/system/etc/permissions/platform.xml:

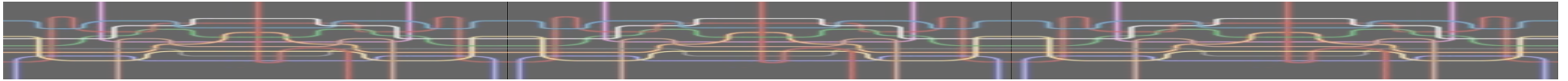
```
<permissions>
...
<permission name="android.permission.INTERNET" >
  <group gid="inet" />
</permission>
<permission name="android.permission.CAMERA" >
  <group gid="camera" />
</permission>
<permission name="android.permission.READ_LOGS" >
  <group gid="log" />
</permission>
<permission name="android.permission.WRITE_EXTERNAL_STORAGE"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

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

Permission Granting

- Permissions are granted **once**, at the application install time
 - Ok, updates too
 - One exception, URI permissions
- All-or-nothing!
- But, can a ~~novice~~ any user tell whether the combination of requested permissions is OK? (Can you?)
 - Permissions marked as "normal" are hidden behind "See all"
- What about combo of permissions across different apps from the same (malicious) author? (Apps can share)





Permission Granting, Alternatives?

- Switch to dynamically granting permissions on use or on start of each app ("session")?
 - Annoying
 - Hard to provide seamless app-switching
 - Over-prompting leads to a conditioned-response
 - Users already committed to the app
- Cannot make informed-decisions on whether to grant permissions? Let app ratings + comments from "sophisticated" users on Market help



Application Provenance

- Can we trust the developer of an application we are about to install? (mostly, no)
- Can we trust that our apps are resistant to tampering once installed? (mostly, yes)
- To get onto Android Market, a developer just needs to register with Google and pay \$25 with a valid credit card
 - A mild deterrent against authors of malicious apps
- Apps can also be side-loaded (not on AT&T)

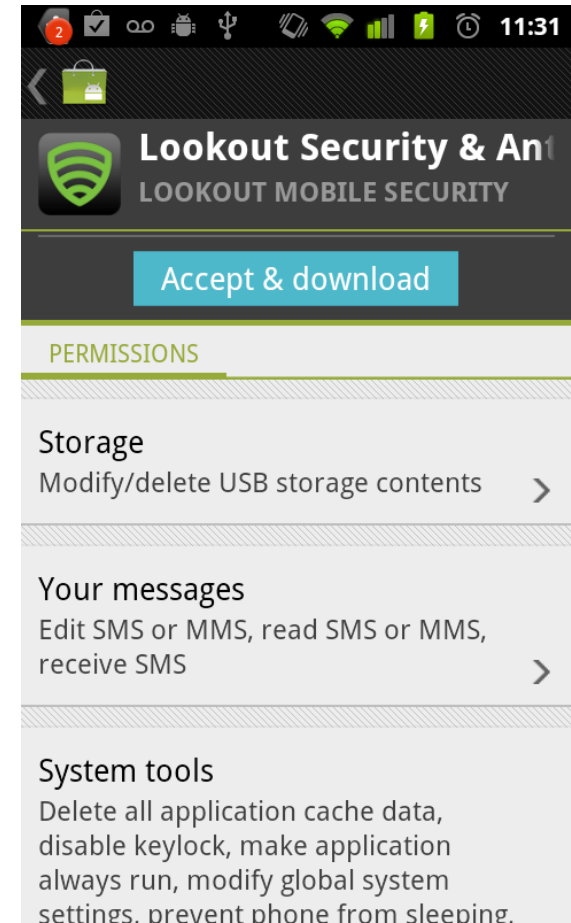


Application Provenance (Signing)

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

Application Provenance (Signing)

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





Application Provenance (Signing)

- The result?
 - ☛ Android.Rootcager
 - ☛ Android.Pjapps
 - ☛ Android.Bgserv
- All took advantage of weak trust relationship
 - ① Take an existing (popular) app
 - ② Inject malicious code (e.g. a trojan)
 - ③ Re-package and re-sign with a new key/cert
 - ④ Upload to market (or distribute via web)
 - ⑤ Wait for the "sheep" to come (not really our fault)

Safeguarding Apps' Data

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



Data Encryption

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



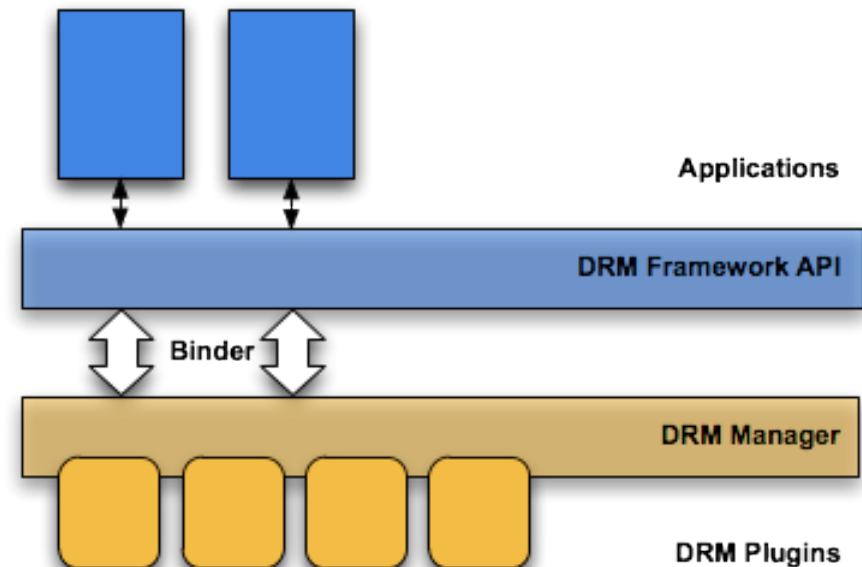


Whole Disk Encryption

- *Settings* → *Location & Security* → *Encryption* → *Encrypt tablet*
 - Requires screen-lock password
 - Encrypts `/data` partition with AES128 with CBC and ESSIV:SHA256 (password combined with salt then SHA1'd)
 - Disabling encryption requires device master reset
- Based on Linux' dm-crypt kernel feature
 - `/data` as an encrypted block device (`/dev/block/dm-0`)
- User-password used directly (change requires re-encrypt!)
- Not hardware-accelerated: 54% degradation in I/O read performance on Samsung Galaxy Tab 10.1
- Vulnerable to "Evil maid attack" and cold-boot attacks

Digital Rights Management

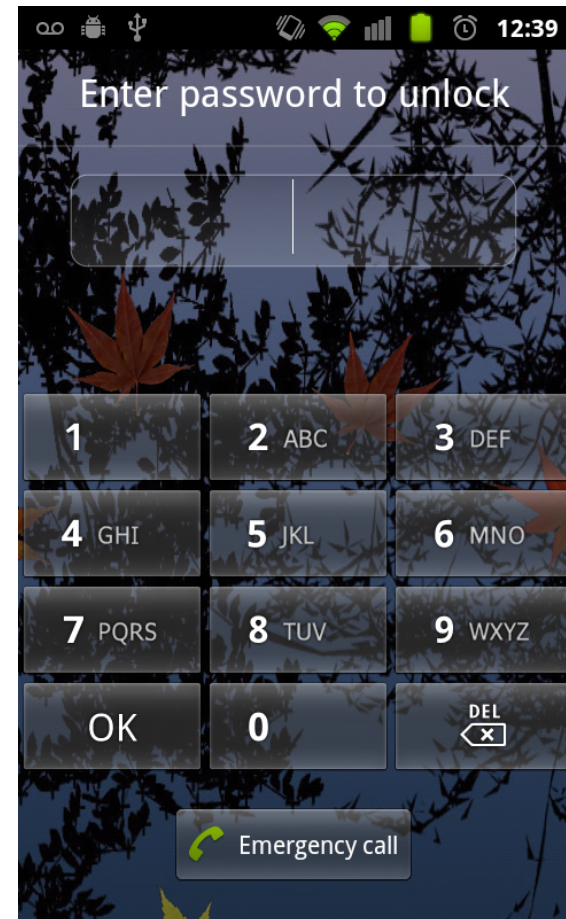
- Android provides a pluggable DRM framework (API ≥ 11)
- Actual schemes provided by OEMs
- Hides complexity of DRM when accessing rights-protected (or plain) content under various schemes





[Physical] Access Control

- Screen unlock pattern, pin, password
- More options with device admin (including password expiration, encryption, auto-device-wipe, etc.)
- Low-level access to SIM card is not available to apps
- But:
 - SIM/SD Card can be simply ejected, bypassing screen unlock
 - Cold-boot attacks



Taking Android To Work: Device Admin





Rooting

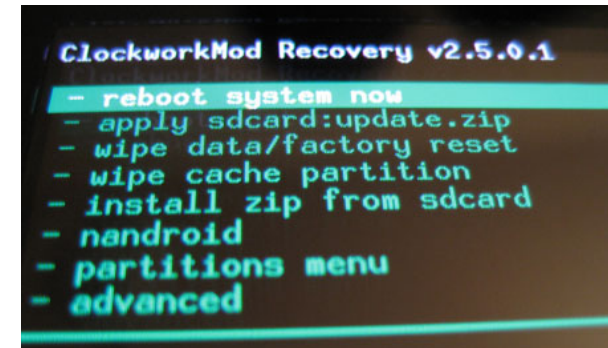
- Why root?
 - Access to custom ROMs
 - Reuse old hardware
 - Remove offending system apps
 - Get more speed
 - Get better looks
 - Because it's cool
 - Rootkit ☹
- But, it comes at a price





Rooting: How-to

1. Exploit a weakness of the existing ROM to gain root
2. Flash the recovery partition with an alternative image
3. Download an alternative compatible ROM (already rooted) onto the `/sdcard`
4. Reboot into recovery, and flash the new ROM
5. Get root at any time with `Superuser.apk` + `/system/bin/su`
 - Or, as easy as: `$ fastboot oem unlock`





Getting Root

- *exploid*: exploit a bug in `udev` (on Android `init/ueventd`) to pass a fake message (`NETLINK_KOBJECT_UEVENT`) with executable `FIRMWARE` code to run as root
- *rageagainstthecage*: exploit a race-condition in `adbd` to preempt its call to `setuid()` (to shell user) leaving it running as root
- *softbreak/gingerbread*: exploit a buffer-overflow condition in `vold` (which runs as root) to execute arbitrary code as root
- ...

Dangers of Rooting

- App isolation
- System/app permissions
- Data-safeguards + encryption
- Device administration
- ...

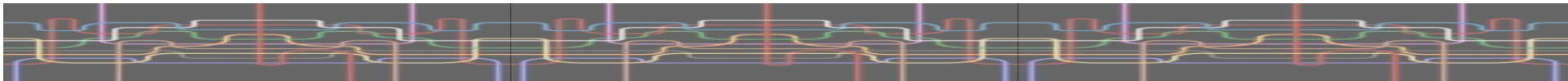
all **fall-apart** when we allow un-trusted code to run as root (this is what malicious apps do)





Memory Security Protection

- Hardware-based No eXecute (NX) to prevent code execution on the stack and heap
- ProPolice to prevent stack buffer overruns
- `safe_iop` to reduce integer overflows
- Extensions to OpenBSD `dldmalloc` to prevent double `free()` vulnerabilities and to prevent chunk consolidation attacks (against heap corruption)
- OpenBSD `calloc` to prevent integer overflows during memory allocation
- Linux `mmap_min_addr()` to mitigate null pointer dereference privilege escalation
- But, what about shared libraries?



Address Space Layout Randomization

- Shared libraries on Android are pre-linked*: their address are fixed, for performance reasons
- Successful memory corruption attacks can easily return to libc (i.e. execute arbitrary code)
- ASLR on Android (just a proposal at this time):
 - Randomize offsets to shared libs and executables at system upgrade-time
 - Record offsets to undo randomization for OTA updates
 - Detect brute-force guessing with cloud-based analysis
- <http://bojinov.org/professional/wisec2011-mobileaslr-paper.pdf>
- ASLR is finally a standard in ICS/4.0 (* no pre-linking?)

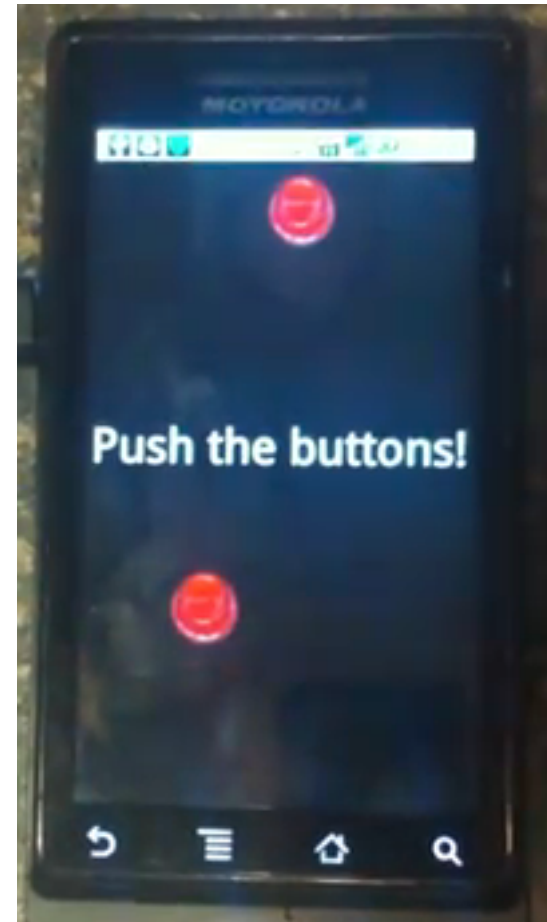


SE-Linux on Android

- SELinux allows us to run OS services with minimum privileges (i.e. not root)
 - Heavy use on the desktop/server-side
- SELinux on Android is possible, but hard
 - Slow
 - Requires rethinking on the security model for easier configuration
 - Does not support yaffs2
- Folks at Hitachi got it to work, but it seems stalled

Tap-Jacking on Android

- A malicious app starts a security-sensitive (e.g. system settings) activity
- It then overlays a full-screen custom notification dialog on top of the targeted activity (like a game) – works like `Toasts`
- User interacts with the custom notification dialog, but her touch events are passed down to the legitimate activity
- In API ≥ 9 prevent with XML attr on UI `filterTouchesWhenObscured` (or programmatically)





Developer Best Practices

- Avoid building apps that require root
- If you are using encryption, be sure to know what you are doing (e.g. use IVs)
- Mark your application's components as `android:exported="false"` unless you are specifically building them for public use
 - Don't trust Intent inputs/results (especially pending)
 - Don't leak broadcast events you are sending out
 - Use custom permissions to control access



Custom Permissions

```
<manifest ... package="com.marakana.myapp" >
  <permission
    android:name="co.mrkn.perm.GET_PASSWORD"
    android:label="@string/get_password_label"
    android:description="@string/get_password_desc"
    android:permissionGroup=
      "android.permission-group.PERSONAL_INFO"
    android:protectionLevel="dangerous" />
  ...
</manifest>
```



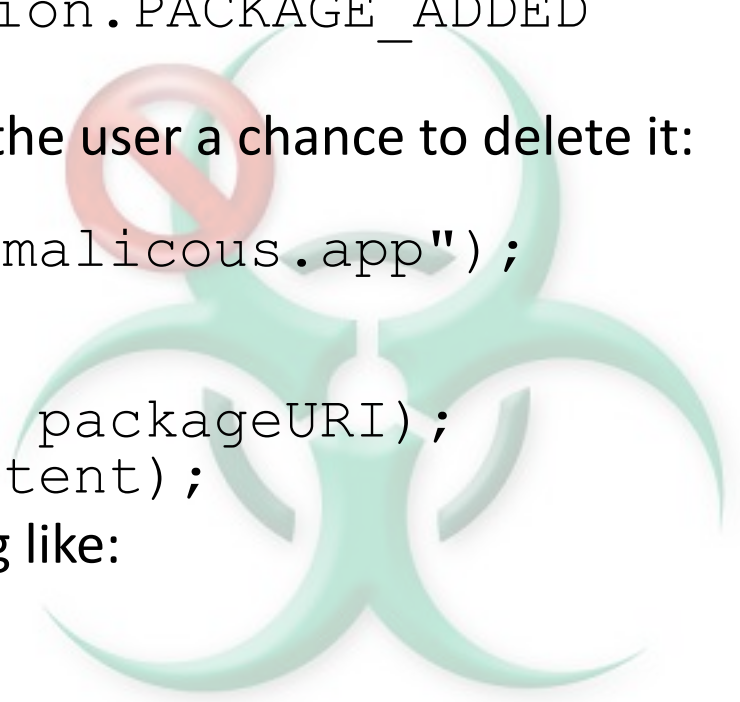
Requiring Permissions

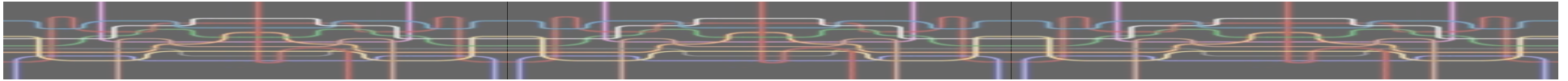
- **Statically, in `AndroidManifest.xml` on our application components via attributes**
 - `android:permission`
 - `android:readPermission`
 - `android:writePermission`
- **Dynamically, on broadcast senders via**
 - `aContext.registerReceiver(BroadcastReceiver, String, Handler)`
- **Dynamically, in bound-services via**
 - `aContext.checkCallingPermission(String)`
 - `aContext.enforceCallingOrSelfPermission(String)`



Anti-malware

- Use `PackageManager.getInstalledPackages(int)` for the initial scan of apps/packages against a known black-list
 - E.g. check for package names, permissions, signatures
 - Listen for `android.intent.action.PACKAGE_ADDED` broadcasts and verify new apps
 - Once a malicious app is found, offer the user a chance to delete it:

```
Uri packageURI =  
    Uri.parse("package:com.malicious.app");  
Intent uninstallIntent =  
    new Intent(  
        Intent.ACTION_DELETE, packageURI);  
startActivity(uninstallIntent);
```
 - For personal use, consider something like:
 - *Lookout Security & Antivirus*
 - *Norton Mobile Security*
- 



Other Security Concerns

- Push-based install from Android Market (GMail)
- Social-engineering
- Firewall
- Encryption of communication
- Compromised platform keys
- App obfuscation
- Protecting bootloader/recovery
- Security of skins
- OEM/Carrier OS upgrade cycles

Thank You!

- Questions?
- More info:
 - <http://mrkn.co/andsec> (video of this talk)
 - http://source.android.com/tech/encryption/android_crypto_implementation.html
 - http://www.symantec.com/about/news/release/article.jsp?prid=20110627_02
- Contact Info:
 - <http://marakana.com/>
 - sasa@marakana.com
 - @agargenta on Twitter
 - aleksandar.gargenta@gmail.com on Google+

