6.858 Lecture 20
Android Security

Why this paper?
- Real system, widely used.
- Careful security design (more so than for web or desktop applications).
  - Principals = Applications (not users)
  - Policy separate from code (manifests)
- Some problems inevitable, and instructive to see where problems come up.
- But also interesting to see how to design a reasonable security plan.

Threat model
- Goal: Anyone can write an app that anyone can install
- Threats:
  - Apps may have bugs
  - Apps may be malicious

CVE database
- http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html
- Some bugs but not overwhelming---is the security plan working?
  - buffer overrun (still happens ....)
- Of course, Android runs on Linux, and this includes Linux kernel problems

Overall plan:
- First understand how Android applications look like and work.
- Then discuss security mechanisms and policies.

What does an Android application look like?
- Four types of components:
  - Activity: UI component of app, typically one activity per "screen".
  - Service: background processing, can be invoked by other components.
  - Content provider: a SQL database that can be accessed by other components.
  - Broadcast receiver: gets broadcast announcements from other components.
- Each application also has private file storage.
- Application typically written in Java.
- Runs on a Linux kernel + Android "platform" (will get to it shortly).
- Application also has a manifest declaring its permissions (later).
- Entire application is signed by the developer.

Activity: can draw on the screen, get user input, etc.
- Only one activity is running at a time.
- Helps users reason about security of inputs.

- If user is running bank app (activity), no other activity gets user's input.

Intent: basic messaging primitive in Android.
- Represents app's intent to do something / interact with another component.

Intent fields:
- Component: name of component to route the request to (just a string).
    - E.g., com.google.someapp/ComponentName
- Action: the opcode for this message (just a string).
    - E.g., android.intent.action.MAIN, android.intent.action.DIAL, ..
- Data: URI of data for the action (just a string).
    - E.g., tel:16172536005, content://contacts/people/1 (for DIAL).
    - Also includes the MIME type of the data.
- Category: a filtering mechanism for finding where to send intent.
    - E.g., android.intent.category.BROWSABLE means safe to invoke from browser, for action android.intent.action.VIEW, which views the URI in data.

- Explicit intents: component name specified.
- Implicit intents: no component name, so the system must figure it out.
    - Looks at action, data, category.
    - Could also ask the user what app to use, if multiple components match.
    - E.g., user clicks on an address -- what map application to open?

RPC to services.
- Initial communication to a service happens by sending an intent.
- Service can also define an RPC protocol for clients to use.
    - More efficient than sending intents each time.
    - Client "binds" a connection to a service.

Networking -- accessing the Internet.
- Work just as in any other Linux system.
- Application can use sockets directly, or via Java's networking libraries.

Why do we need a new app model? (Or, what's wrong with existing models?)
- Desktop applications:
    - −: Not much isolation between applications.
    - −: Every app has full privileges, any one malicious app can take over.
    - +: Applications can easily interact with one another, share files.
    - +: User can choose app for each task (email app, image viewer, etc).
- Web/browser-based applications:
    - +: No need to install applications or worry about local state.
    - −: Requires a server in the typical model (hard to use offline).
    - −: Limited interactions between applications.
    - −: Interactions that do exist are typically hard-wired to particular URLs.

        - E.g., links to a contact manager app's URL: user cannot choose new one.
        - Getting better: "Web intents" are trying to solve this problem.
    - −: Somewhat limited functionality for purely client-side applications.
        - Getting better: camera, location info, local storage, worker threads…

How does Android's application model handle app interaction, user choosing app?

- Mostly based on intents.
- If multiple apps could perform an operation, send implicit intent.
- Android framework decides which app gets the intent; could ask user.

How does Android's application model handle app isolation?

- Each application's processes run under a separate UID in Linux.
    - Exception: one developer can stick multiple applications into one UID.
- Each application gets its own Java runtime (but that's mostly by convention).
- Java interpreter not trusted or even required; kernel enforces isolation.

What are per-app UIDs good for?

- One app cannot directly manipulate another app's processes, files.
- Each app has private directory (/data/data/appname).
    - Stores preferences, sqlite DBs for content providers, cached files, etc.

What's missing from UID isolation: access control to shared resources.

- Network access.
- Removable sd card.
- Devices (camera, compass, etc).
- Intents: who can send, what intents, to whom?
- And we also need to somehow determine the policy for all of this.

First, mechanism: how does Android control access to all of the above?

- Network access: GIDs.
    - Special group IDs define what apps can talk to the network.

```
GID AID_NET_BT_ADMIN (3001): can create low-level bluetooth sockets
GID AID_NET_BT       (3002): can create bluetooth socket
GID AID_INET         (3003): can create IP socket
GID AID_NET_RAW      (3004): can create raw socket
GID AID_NET_ADMIN    (3005): can change network config (ifconfig, ..)
```

    - Requires kernel changes to do this.
    - Each app gets a subset of these group IDs, depending on its privileges.
    - No finer-grained control of network communication.
        - E.g., could have imagined per-IP-addr or per-origin-like policies.
- Access to removable sd card.
    - Why not use file system permissions?
        - Want to use FAT file system on SD card, to allow access on other devices.

        - FAT file system has no notion of file ownership, permissions, etc.
    - Kernel treats all SD card files as owned by special group sdcard_rw (1015).
    - Apps that should have access to SD card have this GID in their group list.
    - No finer-grained isolation within the entire SD card.
- Devices.
    - Device files (/dev/camera, /dev/compass, etc) owned by special groups.
    - Apps run with appropriate groups in their group list.
- Intents.
    - All intents are routed via a single trusted "reference monitor".
    - Runs in the system_server process.
    - Reference monitor performs intent resolution (where to send intent?), for implicit intents. [ref: ActivityStack.startActivityMayWait]
    - Reference monitor checks permissions, based on intent and who sent it. [ref: ActivityStack.startActivityLocked]
    - Routes intent to the appropriate application process, or starts a new one.
- Why not just use intents for everything, instead of special groups?
    - Efficiency: want direct access to camera, network, SD card files.
    - Sending everything via intents could impose significant overhead.

How does the reference monitor decide whether to allow an intent?
- "Labels" assigned to applications and components.
    - Each label is a free-form string.
    - Commonly written as Java-style package names, for uniqueness.
    - E.g., com.android.phone.DIALPERM.
- Each component has a single label that protects it.
    - Any intents to that component must be sent by app that has that label.
    - E.g., phone dialer service is labeled with ...DIALPERM.
    - For content providers, two labels: one for read, one for write.
- An application has a list of labels it is authorized to use.
    - E.g., if app can dial the phone, ...DIALPERM is in its label set.
- Other permissions (network, devices, SD card) map to special label strings.
    - E.g., android.permission.INTERNET translates to app running w/ GID 3003.

How does an application get permissions for a certain set of labels?
- Each app comes with a manifest declaring permissions (labels) the app needs.
- Also declares the labels that should protect each of its components.
- When app is installed, Android system asks user if it's ok to install app.
- Provides list of permissions that the application is requesting.

At one point, Android allowed users to set fine-grained permission choices.
- Android 4.3 introduced the "permission manager".
- Apparently this was removed in Android 4.4.
- Possible reason: developers want predictable access to things.

Who defines permissions?

- Apps define permissions themselves (recall: just free-form strings).
- Android system defines perms for built-in resources (camera, network, etc).
    - Can list with 'adb shell pm list permissions -g'.
- Built-in applications define permissions for services they provide.
    - E.g., read/write contacts, send SMS message, etc.
- Defining a permission means specifying:
    - User-visible name of the permission.
    - Description of the permission for the user.
    - Grouping permission into some categories (costs money, private data, etc).
    - Type of permission: "normal", "dangerous", and "signature".

What do the three types of permission mean?

- Normal:
    - Benign permissions that could let an app annoy the user, but not drastic.
        - E.g., SET_WALLPAPER.
        - diff $(pm list permissions -g -d) and $(pm list permissions -g)
    - System doesn't bother asking the user about "normal" permissions.
    - Why bother having them at all?
        - Can review if really interested.
        - Least-privilege, if application is compromised later.
- Dangerous:
    - Could allow an app to do something dangerous.
    - E.g., internet access, access to contact information, etc.
- Signature:
    - Can only be granted to apps signed by the same developer.
    - Think ForceHTTPS: want to prevent user from accidentally giving it away.

Why do this checking in the reference monitor, rather than in each app?

- Convenience, so programmers don't forget.
    - Could do it in a library on the application side.
- Intent might be routed to different components based on permissions.
    - Don't want to send an intent to component A that will reject it, if another component B is willing to accept it.
- Mandatory access control (MAC): permissions specified separately from code.
    - Aside: annoyance, MAC is an overloaded acronym.
        - Media Access Control -- MAC address in Ethernet.
        - Message Authentication Code -- the thing that Kerberos v4 lacked.
    - Want to understand security properties of system without looking at code.
- Contrast: discretionary access control (DAC) in Unix.
    - Each app sets its own permissions on files.
    - Permissions can be changed by the app over time.

    - Hard to tell what will happen just by looking at current file perms.
- Apps can also perform their own checks. [ref: checkCallingPermission()]
    - Breaks the MAC model a bit: can't just look at manifest.
    - Necessary because one service may export different RPC functions, want different level of protection for each.
    - Reference monitor just checks if client can access the entire service.

Who can register to receive intents?
- Any app can specify it wants to receive intents with arbitrary parameters.
- E.g., can create activity with an intent filter (in manifest):

```
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT"/>
<category android:name="android.intent.category.BROWSABLE"/>
<data android:scheme="http" android:host="web.mit.edu" />
</intent-filter>
```

- Is this a problem? Why or why not?
- System will prompt user whenever they click on a link to http://web.mit.edu/.
    - Only "top-level" user clicks translate to intents, not web page components.
- Might be OK if user is prompted.
    - Even then, what if your only map app is "bad": steals addresses sent to it?
- Not so great for broadcast intents, which go to all possible recipients.

- Controlling the distribution of broadcast intents.
    - In paper's example, want FRIEND_NEAR intents to not be disclosed to everyone.
    - Solution: sender can specify extra permission label when sending bcast intent.
    - Reference monitor only sends this intent to recipients that have that label.

- How to authenticate the source of intents?
    - Generally using a permission label on the receiving component.
        - Don't necessarily care who sender is, as long as it had the right perms.
    - Turns out apps often forgot to put perm restrictions on broadcast receivers.
        - Paper at Usenix Security 2011: "permission re-delegation attacks".
        - E.g., can create an alarm that beeps and vibrates forever.
        - E.g., can send messages to the settings bcast receiver to toggle wifi, etc.
    - One solution in android: "protected broadcasts" (not complete, but..)

    - Reference monitor special-cases some intent actions (e.g., system bootup).
    - Only system processes can send those broadcast intents.

Can a sender rely on names to route intents to a specific component?
- More broadly, how does android authenticate names? (App names, perm names.)
- No general plan, just first-come-first-served.
- System names (apps, permissions, etc) win in this model.
- Other apps could be preempted by a malicious app that comes first.
- Could send sensitive data to malicious app, by using app's name.
- Could trust intent from malicious app, by looking at its sender name.
- Could set lax permissions by using a malicious app's perm by name.

What happens if two apps define the same permission name?
- First one wins.
- Malicious app could register some important perm name as "normal".
- Any app (including malicious app) can get this permission now.
- Other apps that rely on this perm will be vulnerable to malicious app.
  - Even if victim app defines its own perms and is the only one that uses it. (E.g., signature perms.)
- Possibly better: reject installing an app if perm is already defined.
  - Allows an app to assume its own perms are correctly defined.
  - Still does not allow an app to assume anything about other app/perm names.

If app names are not authenticated, why do applications need signatures?
- Representing a developer.
- No real requirement for a CA.
- Helps Android answer three questions:
  - Did this new version of an app come from the same developer as the old one? (if so, can upgrade.)
  - Did these two apps come from the same developer? (if so, can request same UID.)
  - Did the app come from same developer as the one that defined a permission? (if so, can get access to signature-level perms.)

How to give another app temporary permissions?
- URI delegation.
  - Capability-style delegation of URI read/write access.
  - System keeps track of delegated access by literal string URI.
    - E.g., content://gmail/attachment/7
  - Must remember to revoke delegated access!
    - E.g., URI may mean another record at a later time..
    - ref: grantUriPermission(), revokeUriPermission()

    - Reference monitor keeps granted URIs in memory.
        - ref: ActivityManagerService.mGrantedUriPermissions
    - Grants are ephemeral, only last until a reboot.
- Pending intents.
    - Use case: callbacks into your application (e.g., from alarm/time service).
    - system_server keeps track of pending intents in memory; ephemeral.
        - ref: PendingIntentRecord.java
    - Revocation problem, as with URI delegation.
- "Breaks" the MAC model: can't quite reason about all security from manifest.

Where are apps stored?

- Two options: internal phone memory or SD card.
- Internal memory is always controlled by Android, so can assume it's safe.
- Installing apps on SD card is more complicated, but desirable due to space.
    - Threat models:
        - Worried about malicious app modifying SD card data.
        - Worried about malicious user making copies of a paid app.
    - SD card uses FAT file system, no file permissions.
    - Approach: encrypt/authenticate app code with a per-phone random key.
    - Key stored in phone's internal flash, unique to phone.

How secure is the Android "platform"?

- TCB: kernel + anything running as root.
- Better than desktop applications:
    - Most applications are not part of the TCB.
    - Many fewer things running as root.
- Some vulnerabilities show up in practice.
- Bugs in the Linux kernel or in setuid-root binaries allow apps to get root.
    - How to do better?
    - Syscall filtering / seccomp to make it harder to exploit kernel bugs?
    - Not clear.
- Users inadvertently install malware applications with dangerous permissions.
    - Actual common malware: send SMS messages to premium numbers.
    - Attackers directly get money by deploying such malware.
    - Why do users make such mistakes?
        - One cause: some permissions necessary for both mundane + sensitive tasks.
        - E.g., accessing phone state / identity required to get a unique device ID.
        - Causes unnecessary requests for dangerous permissions, de-sensitizes user.
        - Another cause: apps ask for permissions upfront "just in case".
        - E.g., might need them later, but changing perms requires manual update.
        - Another cause: cannot say "no" to certain permissions.

        - Another cause: copies of existing Android apps containing malware.
    - How to fix?
        - Find ways to allow more permissions "non-dangerous" without asking user.
        - Allow user to selectively disable certain permissions. (Some research work on this, see refs below.)
        - Static/runtime analysis and auditing -- implemented by Google now.
            - Looks for near-identical clones of existing popular apps.
            - Runs apps for a little bit to determine what they do.
            - Security researchers got a (non-root) shell on Google's app scanner.
            - Reasonably expected in retrospect: app scanner just runs the app..
        - Android's app market (Google Play) allows Google to remotely kill an app.

Other model for security in mobile phone apps: iOS/iPhone.

- Security mechanism: all apps run two possible UIDs.
    - One UID for Apple apps, another for all other apps.
    - Historically made sense: only one app was active at a time.
    - With switch to multi-tasking apps, didn't change the UID model.
    - Instead, isolate apps using Apple's sandbox ("Seatbelt"?).
    - Apple applications not isolated from each other originally (unclear now?).
    - Thus, exploit of vulnerability in browser left all Apple apps "exposed".
- Prompt for permissions at time of use.
    - Users can run app and not give it permissions (unlike Android).
    - "Normal" permissions not very meaningful in this model.
- Apple approves apps in its app store, in part based on security eval.
    - "Reputation-based" system: hard to exploit many phones and avoid detection.

References:

- http://developer.android.com/guide/topics/security/security.html
- http://research.microsoft.com/pubs/149596/AppFence.pdf
- http://css.csail.mit.edu/6.858/2012/readings/ios-security-may12.pdf
- http://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf