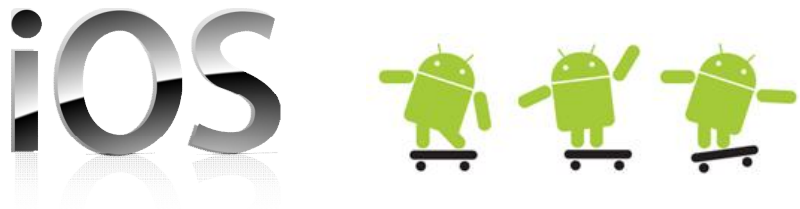
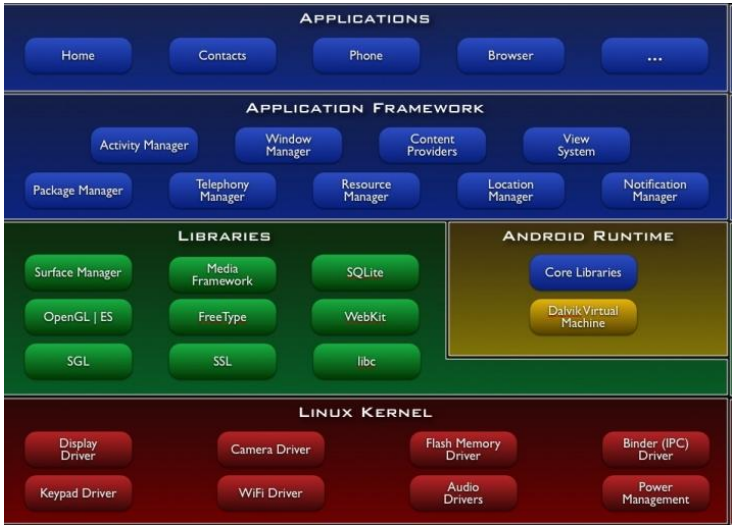
CAN WE TRUST ANDROID AND IOS?



Overview

- Android and iOS: vad is it?
- Security in Android
  - Foundation for security
  - Interaction and access-controle in Android
  - Permissions
- iOS: differences (security wise) compared to Android
- Androids platforms security
- Can we trust smartphones?

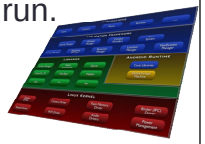
Android components and stack



"THE BASICS" IN ANDROID

Android security – basics

- Applications get each a user id and OS sees to that these are isolated.
- Interaction between applications is controlled via a special API and the aforementioned isolation.
- Applications compiled for other platforms do not run.
- Through privileges application get access to other resources.



Fundament: Android Package (APK) file structure

/META-INF

- **MANIFEST.mf** - sha-1 digest for each file of the applicationen
- **HelloWorld.sf** - sha-1 digest of each file (based on info from Manifest.mf)
- **HelloWorld.rsa** - PKCS#7 RSA signature of HelloWorld.sf

/res

- **Application's resource files**

/

- **AndroidManifest.xml** – app information to Android system
- **classes.dex** - compiled Android Dalvik app
- **resources.arsc** – table on all resources in the application

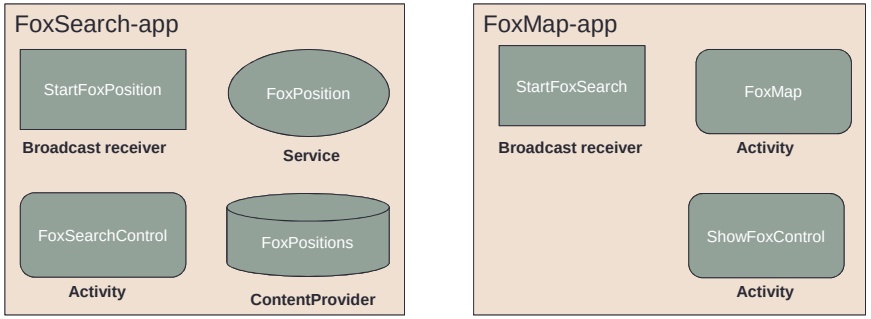
Fundament: Processes and Threads

- Each application gets it's own Linux process
- Each component runs per default in the main thread of the process
- One can create threads for long running processes

Fundament: Android applications are built according a common design pattern

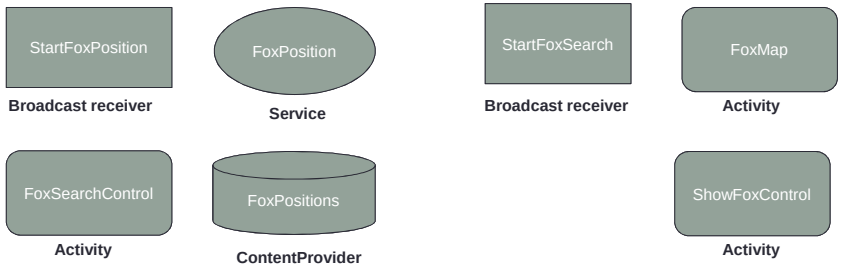
- **Activities** form the presentation layer; one for each screen and Views make the UI for an activity.
- **Intents** specify what is to be done
- **Services** are background processeses with no UI: update data and trigger events
- **Broadcast receivers** offer message mailboxes from other applications and can trigger intents that start an application
- **Content providers** provide data/resources

Android Applications - Example



FoxSearchControl : UI for starta/stop searchfunction
FoxPosition: Contacts external localisation server
FoxPositions: Save last positions

Android Applications - Example

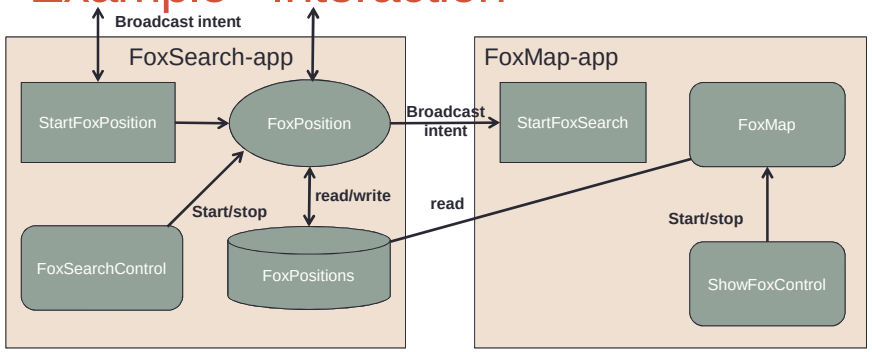


FoxSearchControl : UI for starta/stop searchfunction
FoxPosition: Contacts external localisation server
FoxPositions: Save last positions

Program interaction in Android

- Most interactions in Android are inter-component interactions (ICC):
  - Intents: message object with target address and data
  - Actions: actual process for inter-component interaction
- Android's philosophy is that applications can make (part of) their APIs accessible for other applications

Example - Interaction



Most interactions in Android are inter-component interactions

APPLICATION SECURITY

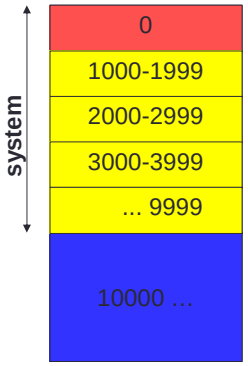
Access control

Runtime protection

Access control in Android

- Android protects applications att
  - (Linux) system level and
  - inter-component communication level: Permissions
- Each application runs as a unique Linux user under its own identity (user ID) which limits the damage an application program error can inflict on other applications :
  - Linux access control (as usual) keeps apps apart
  - User ID is (locally) assigned during app installation

User ID Table



Root: Only few root processeses e.g. init, runtime, Zygote,

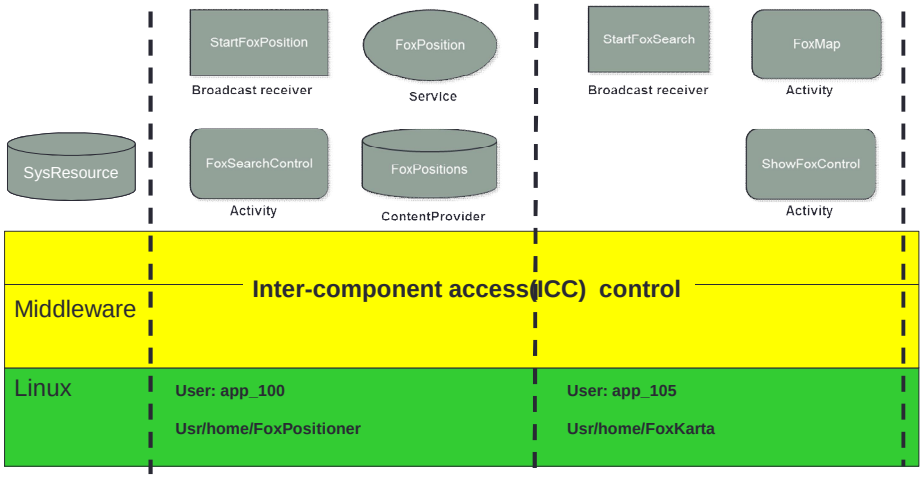
System services

ADB and other debug services

Android services that the kernal is aware

Apps get user ID ≥ 10000

Access control



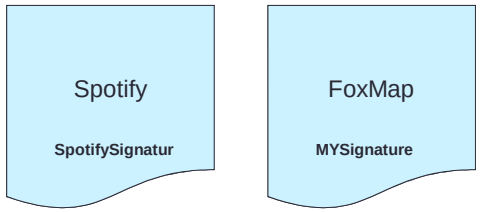
Access control in Android

- Android has implemented a comprehensive mechanism for access control.
- The use of signed applications is an important ingredient in this BUT NOT to assure the (correct) origin of the applications but to assure that the assignment of unique user IDs cannot be spoofed.
- We look at signed applications first

Signed applications

- To identify an application application signing by self-signed certificates is used.
- Androids Distribution does not accept app signed with the default key in the SDK
  - Hence it is a policy issue if known (SDK) keys are accepted for an app.
- Two applications kan share the same user ID if they are signed by using the same key/certificate (should normally be avoided)

PKI structure and access control
without signature permissions



- In vanilla Android the signatures have no security value if signature permissions are not used.
- Remember: signature is in this case only used to identify an application.

PKI structure and access control
with signature permissions

Android distinguishes four core platforms components each having their own key

- **Platform:** a key for packages that are part of the core platform.
- **Shared:** a key for things that are shared in the home/contacts process.
- **Media:** a key for packages that are part of the media/download system.
- **Testkey/releasekey (Application) :** the default key to sign with if not otherwise specified

Access control – the problem

- The Android API consists of 1665 classes with a total of 16732 public and private methods
- Number of Permissions Checks. There are 1244 Android API calls with permission checks, which is 6% of all API methods (including hidden and private methods). In addition the manufacturer might have added APIs with permission checks.

Using Permissions Configuration of manifest

- A basic Android application has no permissions associated with it, meaning it can not do anything that would adversely impact the user experience or any data on the device.
- To get access to protected features of the device the application needs to declare which permission it needs. This is done in the manifest.xml file of the application.
- Applications not declaring needed permission will in most cases get a SecurityException when exercising a not allowed permission failure.

For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >

    <uses-permission android:name="android.permission.RECEIVE_SMS"
     android:permissionGroup="android.permission-group.COST_MONEY"
     android:protectionLevel="dangerous" />

</manifest>
```

Permissions (1/2)

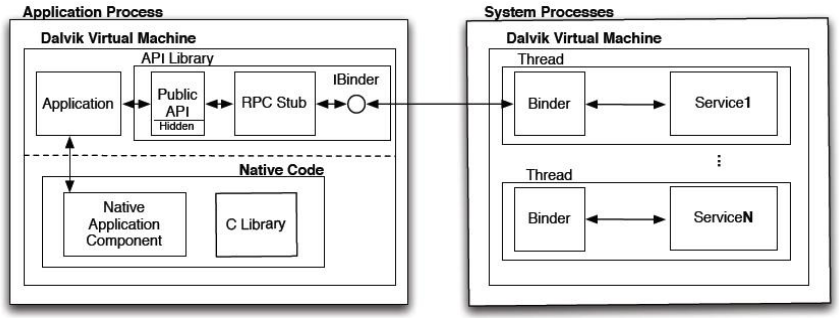
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS"
     android:permissionGroup="android.permission-group.COST_MONEY"
     android:protectionLevel="dangerous" />
</manifest>
```

- Permissions are defined in a text string in the manifest file
  Very flexible but doubtful if user realy understands what is going to happen. There are meny predefined permissions e.g. Manifest.permission_group
- The application gets its requested permision at time of installation.
- Depending on the API, its permissions and who has signed the application the user has to approve the use of the access to the requested API..

Permissions (2/2)

- ICC is set att installation time and can not be altered after that: compare to mandatory access control (MAC).
- When updating an application the same key/certificate must be used
- Access errors can be signaled to the application but not allows. Normally they will appear in the system log.

Permissions (3/3)



Permission control happens in the system processes

Permission model: protection levels

Android 2.2: 134 permissions devided in 4 levels

- **normal:**
  gives applications automatically access, e.g. SET_WALPAPPER
- **dangerous:**
  use of the API is with risk and requires users consent
- **signature:**
  access allowed only if the app is signed with same certificate as the app that has declared the permission
- **signatureOrSystem:**
  as above but can be used freely by packages in the system partition (careful here)

User ID and File access

- Files normally belong to a user ID. (on SD cards it gets tricky though)
- Remember Max two different Android packages kan get the same user ID (signerad against same certificate and in the manifestet we have ”sharedUserId” attribute).
- One can set MODE_WORLD_READABLE and/or MODE_WORLD_WRITEABLE flags on a file so att alla apps are allowed to read/write from/to the file.

Use of permissions- investigation

- In a study of 940 Android applications from the Android Market one found that about one-third of applications are overprivileged.
- The overprivileged applications generally request few extra privileges:
  - more than half only contain one extra permission,
  - and only 6% request more than four unnecessary permissions.

Android Permissions Demystified
Adrienne Porter Felt, et al 2011.

Common permissions

Permission	Usage
BLUETOOTH	85
BlUETOOTH_ADMIN	45
READ_CONTACTS	38
ACCESS_NETWORK_STATE	24
WAKE_LOCK	24
ACCESS_FINE_LOCATION	22
WRITE_SETTINGS	21
MODIFY_AUDIO_SETTINGS	21
ACCESS_COARSE_LOCATION	18
CHANGE_WIFI_STATE	16

Table 1: Android's 10 most checked permissions.

Overpriviledged apps

Permission	Usage
ACCESS_NETWORK_STATE	16%
READ_PHONE_STATE	13%
ACCESS_WIFI_STATE	8%
WRITE_EXTERNAL_STORAGE	7%
CALL_PHONE	6%
ACCESS_COARSE_LOCATION	6%
CAMERA	6%
WRITE_SETTINGS	5%
ACCESS_MOCK_LOCATION	5%
GET_TASKS	5%

Table 2: The 10 most common unnecessary permissions and the percentage of overprivileged applications that request them.

Runtime protection

- **Address Space Layout Randomization (ASLR)** is a known technique that aims to reduce the possibility of mounting attacks that exploit ways to get access to desired memory locations through randomization. Attackers are with ASLR in place faced with a much harder problem to find thus address locations. This reduces, for example, the possibility to make a buffer overflow condition into something useful in an attack. ASLR is introduced in Android 4.0
- **ARM NX** Since many Android devices use ARM type CPUs the not execute (NX) feature for data reduces the risk of exploits where stored data is executed. Using the NX bit data can be tagged as not executable until, say, a successful verification has been carried out on which the NX status is removed. Android supports HW based NX since the Ice cream sandwich release.
- **SELINUX:** Starting with Android 4.3, Security-Enhanced Linux (SELinux) is used to harden the boundaries of the Android application sandbox.

Secure boot and dm-verity

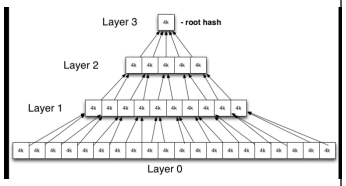
In Android 4.4 there is support to verify boot through the optional device-mapper-verity (dm-verity) kernel feature.

It provides integrity checking of block devices.

A root hash is computing by using SHA256 in a hash tree that covers the block device data.

The root hash is singed to allow it to be verified using a public key burned/programmed into the device

Hash Tree -



IMPLEMENTATION

Android's platform security

All Android product vendors do make adoptions to Android when integrating Androiod to their hardware

- Secure boot process: which components that are verified and how differ among vendors and products
- Vendors try to limit access to root: some suceed better than others
- Most products that use a mobile network modem have modem hardware that is isolated from the application processor subsystem. The modem is controled via the RIL (Radio Interface Layer) which is in practice a proprietary library with a standard API

Weak and strong points

- Very good isolation of APPs.
- Very good control of what can be permitted and what not but it is questionable if the user can really handle it. Careful user may select good apps based on reputation.
- Recent Android version have file system encryption (but not always SD card file encryption is supported)
- Freedom to download reduces incentive to root the device but also increases risk of downloading malware.
- APPs via Google play are only screened by a tool. This is not a device but more a ecosystem issue.
- Security of an Android device (even with same OS version) varies depending on manufacturer's integration and choice of platform

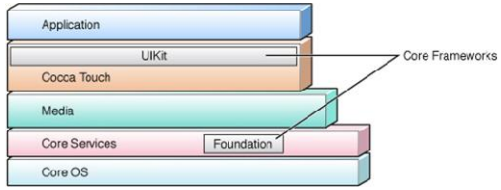
IOS

iOS – remark on material

- Since Apple has not revealed all iOS details we have less exact knowledge how things work and in some cases we rely on people that reported on their analysis and reverse engineering work
- Here we threat iOS through comparison with Android

iOS Stack

- *iOS uses XNU* a free and open source operating system kernel also utilized in Mac OS X. XNU is a hybrid kernel based on the Mach 3.0 microkernel modified for performance and other reasons to include components from Free BSD
- The core (Core OS) of IOS/Mac OS X is called *Darwin* (=open source) and besides the kernel XNU includes facilities like the I/O Kit – a modular dynamic device driver framework, the Virtual File System (VFS), and networking support.
- The main application environment for both IOS and Mac OS X is called Cocoa, and this includes the Objective-C libraries (a.k.a. *frameworks*), APIs and runtimes.
- Both IOS and Mac OS X require the use of the *Foundation* framework which includes basic object management such as memory management and notifications, object wrappers for programmatic primitives, etc. The main difference between applications in iOS and Mac OS X lies in the user interface frameworks.
- The *AppKit* and *UIKit* frameworks contain the objects used to implement a graphical event-driven user interface in Cocoa applications .



FreeBSD: POSIX API, basic security policies, user and group ids, permissions, cryptographic framework, mandatory access control, among others.

iOS sandboxing

- All applications in iOS run as user "*mobile*". Therefore iOS applies fine-grained process runtime security policies specifying file and system access restrictions for the applications.
- The XNU Sandbox kernel extension (a.k.a. "Seatbelt") is implemented as a policy module for the TrustedBSD Mandatory Access Control (MAC) framework. The Sandbox extension is a loadable kernel module that installs MAC policy hooks for accessing system objects and resources.
- iOS assigns each installed third party application (incl. preferences and data) a home catalogue[1] (or *container*) of the file system. By default the application may only read and write files within their container, but more specific permissions are detailed in the sandbox profile. When an application is launched, its sandbox profile is determined by so-called profile entitlements.
- Some apps are built-in/preinstalled into the device. These apps have the same User ID (501, mobile) as the 3rd party application but in different sandboxes.
- IOS only supports static sandbox profiles, there are 35 custom made profiles usually named by the single built-in application that uses them (MobileMail, MobileSafari, MobileSMS, …). Third-party applications are placed in a container sandbox.

[1] /var/mobile/Applications/UUID, where the application Universally Unique IDentifier (UUID) is randomly generated when the application is installed.

Entitlements and Provisioning Profile

- The execution of apps is controlled by a Provisioning Profile (PP), which is an *Apple signed* .plist ("property list") .
- The PP defines which entitlements the developer is permitted to give to apps he/she signs. In this way access conditions can be modified from their defaults
- Examples of entitlements are:
  - *get-task-allow* which allows debugging (not applicable to applications that are being distributed)
  - *keychain-access-groups* which defines a list of Keychain namespaces the applications are allowed to access. This allows apps to share Keychain items
  - *seatbelt-profile* which specifies which built-in sandbox profile to apply to the process

Code signing and verification

All IOS executable binaries and applications must be signed against a trusted certificate.

While Apple's certificates are inherently trusted, any other certificates must be installed via a properly signed Provisioning Profile (PP). The PP also contains any specific entitlements deviating from default for the App.

Verification: The CodeDirectory (CD) is a resource in the application's master directory consisting of a versioned header followed by an array of hashes, including pages of the main executable. Before an application is launched, the kernel verifies the signature of the CD against the trust caches.

The kernel contains a static set of known CD hashes for built in system binaries (static trust cache) and a dynamic list (cache) of CD hashes of all verified executables executed since boot.

iOS Access control

Access control is based on sandboxing and MAC (TrustedBSD) profiles.

The "container" sandbox profile defines access permissions of 3rd party applications and is a white-list that in general restricts access to the application's own home directory but includes permissions to access

- some necessary system files, e.g.
  - random number generation & crypto functions
- address book (read and write)
- photos and music (read). Note: Geotagged photos require app to have **user granted** permission to access location data.
- customizations related to carriers including voice mail numbers, MMS and APN settings etc. (read)
- the Documents/Inbox directory (read and delete files related to the app, e.g. viewing mail attachment documents with the associated application)
- keyboard cache (read), presumably to enable auto-completion → Privacy concerns

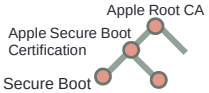
Furthermore
- outbound network connections are allowed,
- applications are allowed to execute binaries from within their application bundle, and
- applications are allowed to create sockets to receive kernel events.

Built-in trusted boot

```
ADC7 US
ADD4 Apple Inc.
ADE9 Apple Certification Authority
AE11 Apple Root CA
AE22 060425214036Z
AE31 350209214036Z
AE58 Apple Inc.
AE6D Apple Certification Authority
AE95 Apple Root CA
B062 https://www.apple.com/appleca/
B093 Reliance on this certificate by any party assumes
     acceptance of the then applicable standard terms
     and conditions of use, certificate policy and
     certification practice statements.
```

The iPhone products implement a trusted boot scheme. The root of trust is an Apple root certificate stored in the boot ROM. Firmware images are stored encrypted and signed.

The PKI scheme in use has three **levels**



There are three different boot **modes**:

- In the **Normal boot** mode the trusted boot sequence is:
  Boot ROM → LowLevelBootloader (LLB) → Iboot → Kernel.
- **Recovery mode** is a fail-safe option in the Iboot bootloader that allows uploading of a RAM disk and reflashing of the device.
- **DFU mode** is a fall-back option that allows to restore from a given boot state. DFU has a slightly different boot sequence.

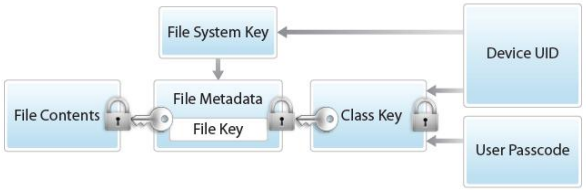
DFU and Recovery modes are accessible over the USB interface.

iPhone jailbreaking targets mostly the circumvention of this trusted boot (see, e.g., iPhone Dev Team)

.

File encryption

- iOS implements a comprehensive file encryption scheme called DataProtection using an hw AES engine
- By setting up a device passcode, the user automatically enables Data Protection.
- Every time a file on the data partition is created a new 256-bit key (the "per-file" key) is created and gives it to the AES engine, which uses the key to encrypt (AES CBC) the file as it is written to flash memory. The CBC initialization vector is the output of a linear feedback shift register (LFSR) calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.
- More details are known



iPhone iOS

Similarities

- Unix/Linux based, ASLR, NX protection
- Use of profiled open source libs
- SDK tools freely available
- APP installation packages must be signed
- APPs and their data are isolated
- Permission model present
- Native browser uses webKit

Differences

- Kernel protection differences
- Programming languages differ
- Signing is used to check origin of APP installation packages (e.g. limit to come from specific source)
- iOS uses sandboxing for isolation, most apps have same user id (501, mobile), 35 sandbox profiles.
- iOS permission are more handled through fixed policies. Very few by user consent.
- Screening of APPs before releasing for deployment (is not a device issue though)
- iOS has more developed data/file encryption support
- Trusted boot in iPhones which in Android phones is vendor specific
- iPhone APPs are DRM protected during delivery and on the device
- Fingerprint sensor

AND THE RESULT IS ....

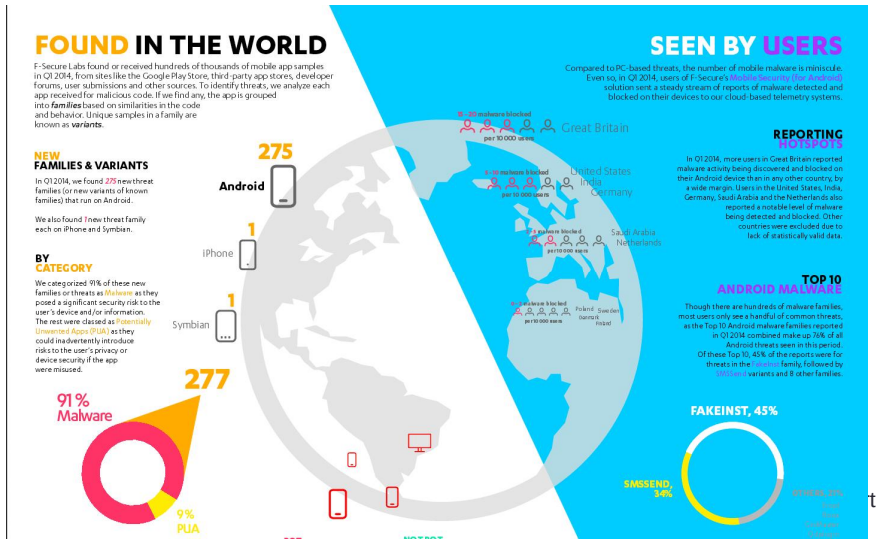
Rank of Top Mobile Threats

October 4, 2012 – The Cloud Security Alliance (CSA) Mobile Working Group today released findings from a new survey that calls out the specific security concerns enterprise executives say are the real and looming threats as it relates to mobile device security in the enterprise

1. Data loss from lost, stolen or decommissioned devices
2. Information-stealing mobile malware
3. Data loss and data leakage through poorly written third-party applications
4. Vulnerabilities within devices, OS, design and third-party applications. Insecure Wifi network or rogue access points
5. Insecure WiFi, network access and rogue access points.
6. Insecure or rogue marketplaces
7. Insufficient management tools, capabilities and access to APIs (includes personas).
8. NFC and proximity-based hacking.

Malware statistic

Apple's screening of apps before placing them in App Store pays off.



References

- Android developer site pages: (too many to list here)
- Understanding Android Security, William Enck, et. al, Security & Privacy January/February 2009
- Developing Secure Mobile Applications For Android, iSEC Partners 2008
- Android Permissions Demystified, Adrienne Porter Felt, et al 2011.
- iOS Security, Apple Inc, May 2012
- https://media.blackhat.com/bh-us 11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf
- http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf