MOBILE PLATFORM SECURITY

Prof. Dr. Bernhard Tellenbach

Content

- Introduction to Security of Mobile Devices
- iOS Security
- Android Security

Goals

For both the iOS and Android security model:

- You know the different security features that are used
- You understand the purpose of the security features and can explain their basic principle
- You know what jailbreaking / rooting is and what it means with respect to weakening the security models

Introduction

What is Special about Mobile Devices? (1)

- Fundamentally, they are not so much different, hardware and software architectures are similar
- But their usage differs from standard computers
  - People carry them around => they are often lost and easily stolen
  - They are more and more becoming *the* central device that is used for all kind of activities, e.g., as wallets, second authentication factor, and so on
  - They contain a lot of personal and sensitive data
  - They are used in various networks, including non-trusted Wi-Fi APs
  - Apps usually store credentials of users as entering long password is cumbersome
  - Easy app installation mechanisms => users install all kinds of apps
  - Even users that don’t use real computers use them, so the security awareness is even lower
- Mobile devices are very attractive attack targets

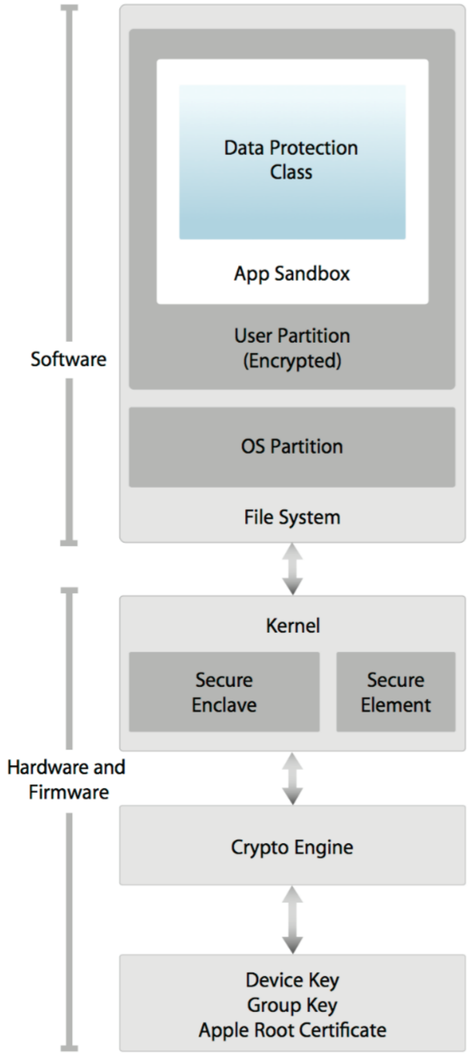
What is Special about Mobile Devices? (2)

- As a result, the device and OS manufacturers have integrated protection mechanisms that go beyond those of standard computers
- These mechanisms should provide the following protection:
  - Make it difficult to load malicious apps onto the device
  - If they get onto the device, they shouldn't be able to access arbitrary data on it
  - A thief cannot use the phone or access the data on it
  - Users do not have administrative rights to avoid accidentally harming themselves
  - ...
- We look at two examples here
  - iOS, which provides a strong but also restrictive security model
  - Android, which provides a less restrictive but also less secure model

iOS Security

iOS Security Principles

- iOS puts a strong focus on security and provides several security features
  - Secure boot process
  - File encryption and data protection
  - Passcode and Face ID (or Touch ID)
  - Keychain
  - App code signing and app installation only from official App Store
  - Runtime security features such as sandboxing of apps with limited inter-app communication
  - Permission model to grant access to general system functions
- The iOS security model benefits that Apple controls hardware and software



Secure Boot Chain (1)

- The goal of a secure boot process is to make sure that the original, untampered iOS kernel is run
  - Therefore, the integrity of the kernel must be checked
- This is solved with a secure boot chain that consists of three steps
  - For systems with processors ≤ A9, there is a fourth step, the Low-Level-Bootloader (LLB)
  - Merged with iBoot step as its code was anyway shared with iBoot
- Note that Apple refers to the BootROM also as SecureROM

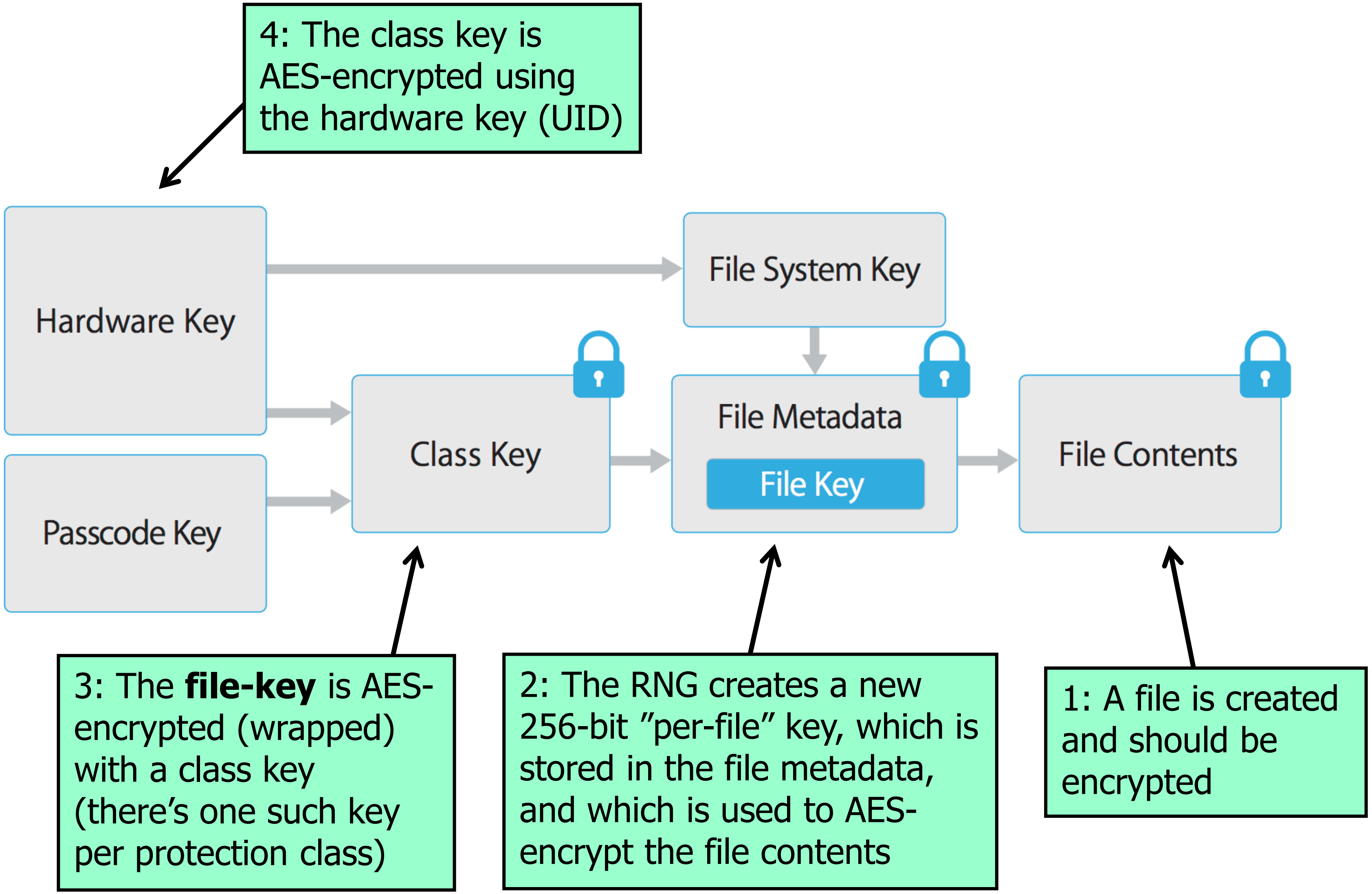
Secure Boot Chain (2)

- The hardware contains code stored in the Boot ROM (read-only)
  - Code is executed when the device is turned on
  - Code contains the Apple Root CA public key
  - Code cannot be changed (not even by Apple)
- iBoot loads, verifies (signature), and runs the iOS Kernel
- If everything is ok, we can trust the running iOS kernel to be the original one
- The trust root is that there is Boot ROM code that cannot be tampered with
  - Are there also disadvantages that this code is not updatable at all?

File Encryption and Data Protection

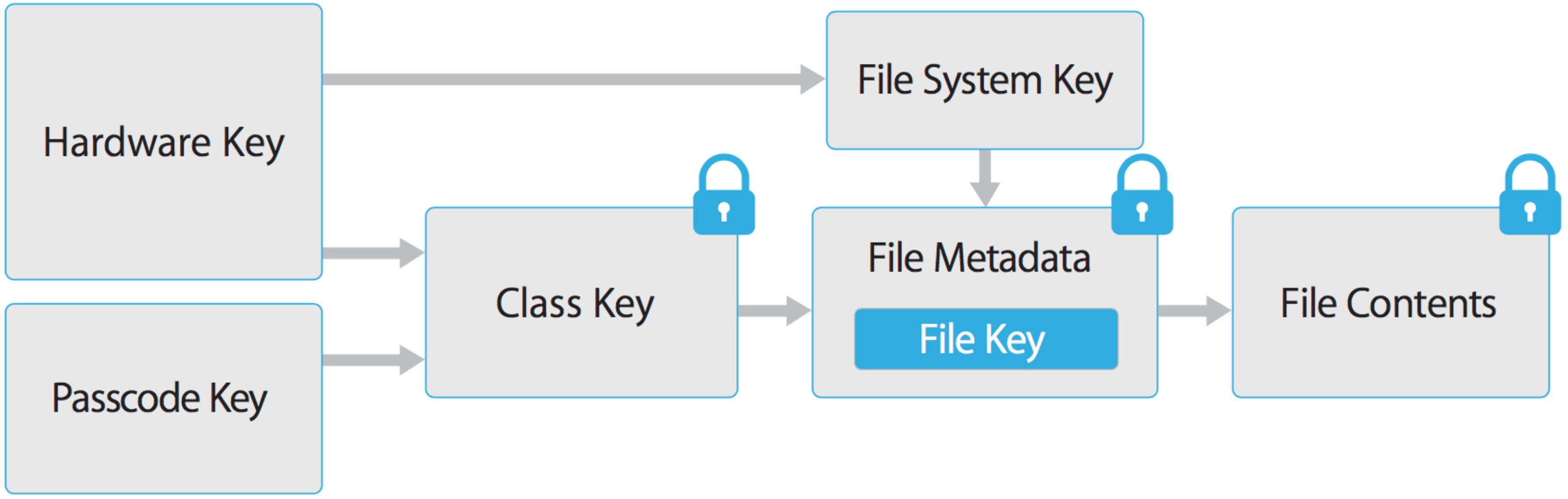
- Every iOS device includes a hardware AES 256 crypto engine between system memory and flash storage
  - As a result, the file system is encrypted (including all user data)
- The hardware includes two IDs, which are used as keys and which can not be read by software
  - A unique ID (UID), which is unique per device
  - A device group ID (GID), which is the same for a specific processor class (e.g., all devices using the A13 processor)
  - The GID is used for tasks that are not (directly) related to the security of user-data such as to deliver system software during installation/restore
- The hardware also includes a random number generator (RNG), which is used to generate further cryptographic keys

File Encryption (1)



File Encryption (2)

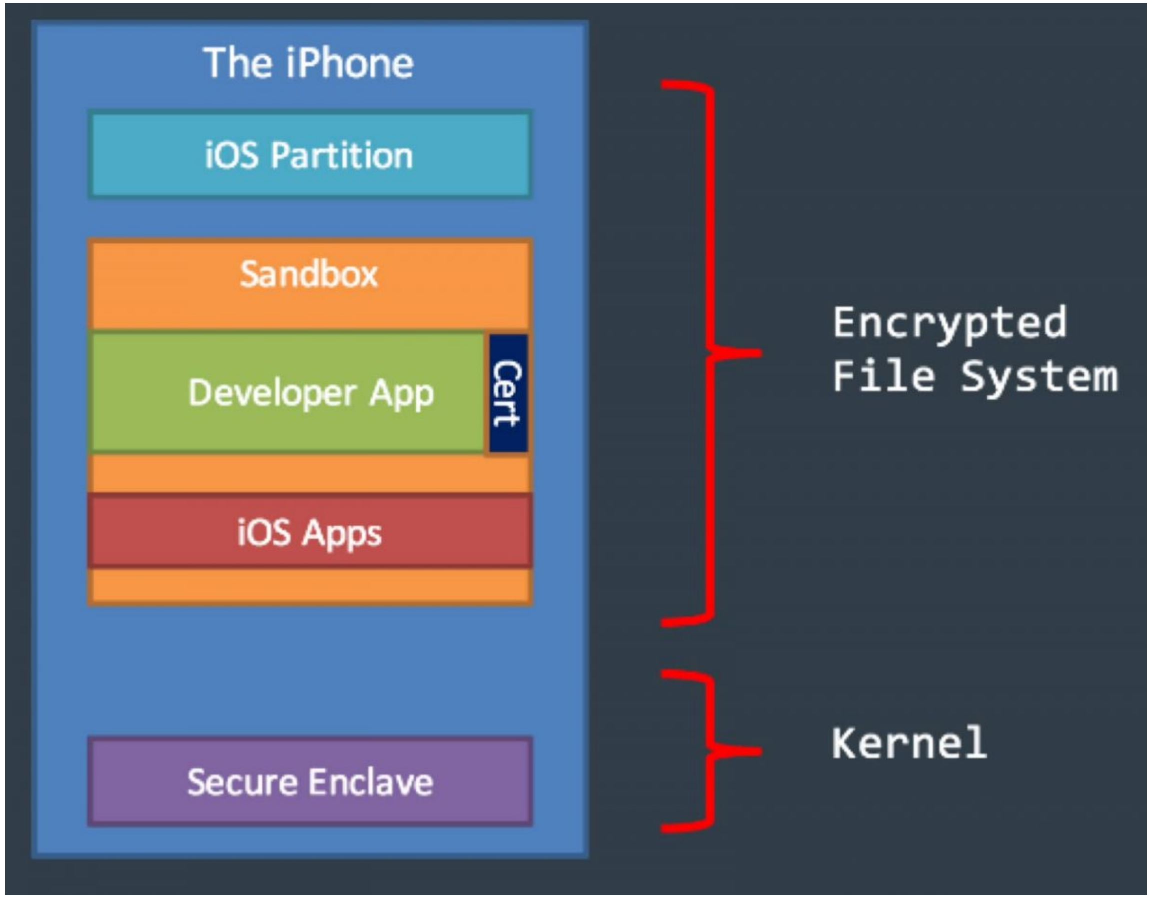
5: In addition, the file metadata (incl. the protection class info) is encrypted with a file system key (created after iOS installation) stored in effaceable storage => This is used to quickly wipe the device by deleting this key



6: If a passcode is used, this is used in addition to the hardware key (UID) as keying material to AES-encrypt the class key (depending on the protection class)

Secure Enclave – Data Protection

- Secure Enclave stores keys in a secure environment to prevent their exposure to the CPU or main memory
- "The Secure Enclave is a hardware-based key manager that's isolated from the main processor to provide an extra layer of security"
- Runs on a separate OS, based on Apples L4 microkernel
- It is also responsible for processing face and fingerprint data to detect if there is a match to grant access



Source: https://www.macobserver.com/analysis/everything-need-know-apple-secure-enclave-hack/

Passcodes (1)

- iOS supports 4-digits, 6-digits and arbitrary-length alphanumeric passcodes
- Used to prevent other people from easily accessing the device and also as additional key material for file protection
- Brute forcing the passcode on the lock screen
  - When entering the passcode on the lock screen, the user is delayed after a few wrong attempts => even a 4-digit code is relatively effective
  - Option: Wipe device after 10 failed attempts

Delays between passcode attempts

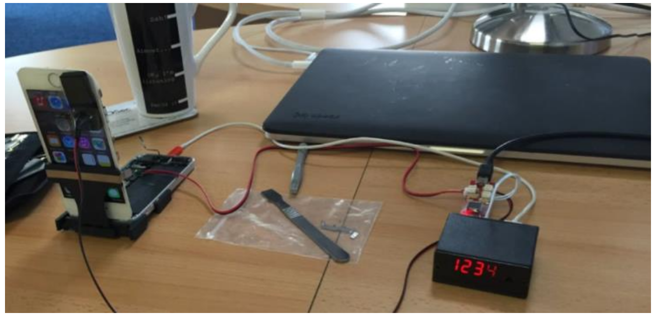
Attempts	Delay Enforced
1-4	none
5	1 minute
6	5 minutes
7-8	15 minutes
9	1 hour

- Other brute force attacks?
  - The passcode check involves AES and the UID => must be performed on the actual device
  - Checking a passcode on the device takes 80 ms
  - In the best case, testing all 4- and 6-digit code therefore takes 13.5 minutes / 22 hours

Passcodes (2)

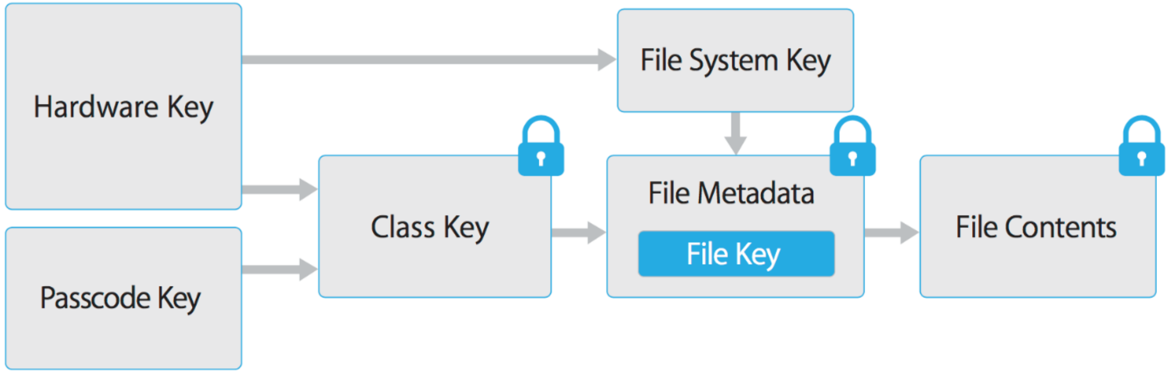
- So far, some attacks to brute force the passcode have been demonstrated
  - So the “ideal” 13.5 minutes / 22 hours have not been reached
- 2018: Gray Box breaks passcode in 72h (2h for 4-digit)
  - All iPhones until iOS 11 (including iPhone X) exploit used is (yet) unknown, but it seems to be some form of brute-force hacking.

- 2015: 4-digit passcode brute forced in 110 hours
  - Exploited a bug in iOS 8.1.1 to simulate PIN-entry over USB and cutting power after every attempt
  - Allows to test one passcode every 40 seconds



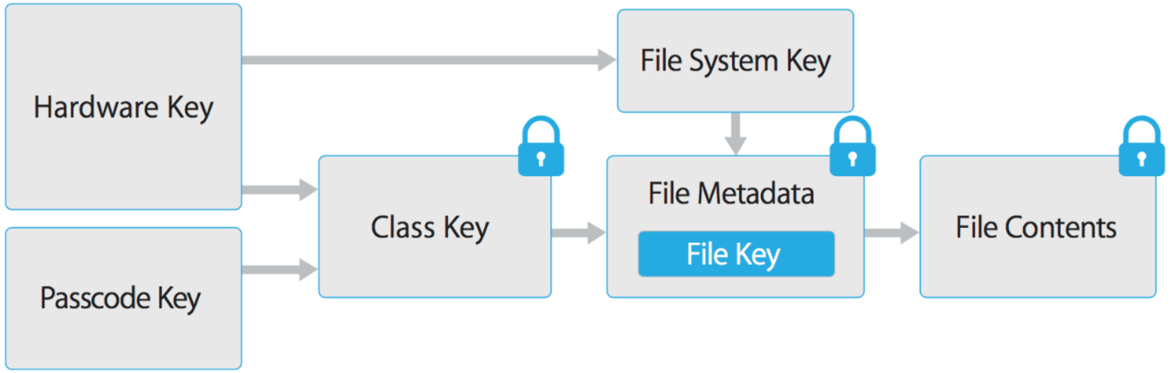
Data Protection Classes (1)

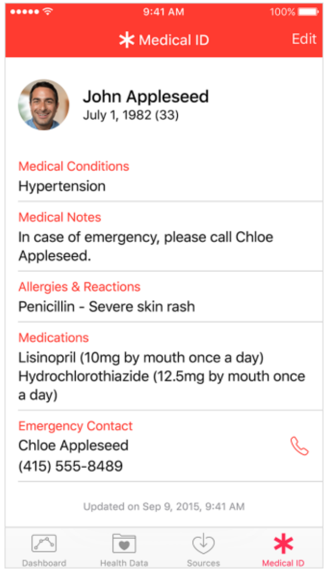
- Each file has a data protection class
  - The class defines when a file is accessible
- Complete Protection
  - The class key is protected with UID and passcode
  - When the device is unlocked, UID and passcode decrypt the class key and the passcode is removed from memory
  - The class key is kept in memory, which allows file access
  - As soon as the device is locked, the class key is removed from memory
  - Example: Mail and attachments and the health data stored on the device



Data Protection Classes (2)

- Protection Until First User Authentication
  - Just like Complete Protection, but the class key is not removed from memory when the device is locked
  - Default data protection used by the apps you develop, unless another class is set
- No Protection
  - Class key is only protected with the UID, and files are accessible even if the user never enters his passcode
  - Example: Used for the Medical ID so this is accessible by everyone without having to unlock the device





File Encryption and Data Protection Summary

- Every file in the file system is encrypted with AES 256
- Encryption includes a device-specific key (UID), which cannot be (easily) extracted from the device
  - All encryption / decryption must therefore happen on the device
  - As an additional feature, this allows to quickly wipe the device
- The passcode provides reasonable protection to prevent easy access to the device and allows different protection classes
  - So, files are only decryptable when needed
  - Using a strong password (alpha-numeric code) is much better than a 6-digit or a 4-digit code
  - So far, no efficient ways to brute force the passcode have been found

File Encryption – Exercise

* assuming that the iCloud activation lock can be circumvented

Consider the following threats. Will they succeed*?

- Alice has stolen your iPhone and wants to possess the device and use it herself
- Bob has stolen your iPhone and wants to use it (e.g., access your e-mail, browse the web with the stored passwords, …)
- Carol has managed to exploit a vulnerability in your iPhone and could install malware that runs with root rights. She wants to access all files on the device

Touch ID (1)

- Fingerprint-based system to unlock the device
  - False acceptance rate: 1:50’000 (according to Apple)
  - It has been shown that having a good copy of the fingerprint of the owner of the device allows to break Touch ID, but it’s far from trivial
  - Using Touch ID is better than using a short (4-digit) passcode
- In some cases, the passcode must be used instead of the fingerprint (which limits attack window / time)
  - After restarting or if the device hasn’t been unlocked for 48 hours
  - After five unsuccessful attempts to match a fingerprint
- Additional features
  - Authentication in some built-in apps (e.g., App Store app, Pay app)
  - Can be used by 3rd-party apps to authenticate users or to access keychain entries (see later)

Touch ID (2)

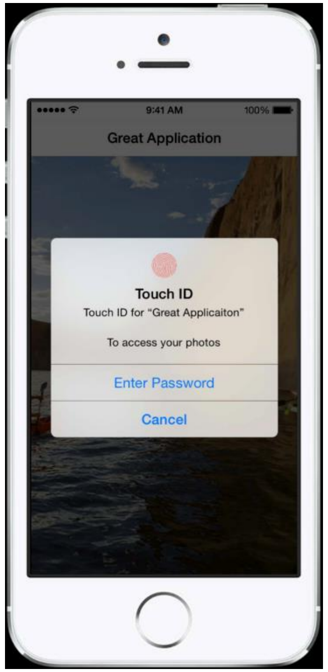
- Touch ID uses an additional coprocessor, the Secure Enclave
  - Has its own secure boot chain and built-in AES engine and UID (key)
  - Stores the fingerprint details of the user
  - Verifies the "submitted" fingerprint by the user
- Does not replace the passcode, it's just a more user-friendly way to access it
  - File encryption and data protection still uses the passcode
- What happens if the device with Touch ID is locked / unlocked?
  - First unlock (after restart) requires passcode => unlocks class key for Complete Protection
  - When locking, the class key is not deleted from memory
  - The class key is encrypted by the Secure Enclave => files can no longer be decrypted
  - When unlocking, the class key is decrypted again

Keychain (1)

- The iOS Keychain allows to store passwords and keys in a secure way
  - Implemented as a single SQLite database in the file system
  - Apps can only access their own data; the App ID is used to assign keychain entries to apps
    - e.g., A1B2C3D4E5.ch.zhaw.supercoolapp
  - Access control is provided by the security daemon of the iOS kernel
- Keychain entries are subject to a similar data protection scheme as files
  - Always (= No Protection)
    - Used e.g., for the "Find My iPhone token"
  - AfterFirstUnlock (= Protection Until First User Authentication)
    - Used e.g., for Wi-Fi and mail account passwords
  - WhenUnlocked (= Complete Protection)
    - Used e.g., by most 3rd party apps that store login credentials

Keychain (2)

- In addition, there are some special protection classes
- WhenUnlockedThisDeviceOnly
  - Like WhenUnlocked, but is backed up (iCloud) in encrypted form (based on UID), so it cannot be restored on another device
- WhenPasscodeSetThisDeviceOnly
  - Like WhenUnlocked, but user must use Touch ID (or enter the passcode) when accessing the entry
  - Entry is not backed up at all
- Keychain summary
  - Similar concept like with files, limits exposure of sensitive entries
  - But just like with data protection, access to the items is possible for malware with root rights or user root on a jailbroken device



App Code Signing

- Once the iOS kernel is running, it controls which apps can run
- On iOS, only apps that are directly signed by Apple or apps that are signed using a certificate issued by Apple can run
  - Built-in apps (Safari, Mail, …) are directly signed by Apple
  - Third-party apps are signed by the developer
- To do this, developers must join Apple‘s Developer Program
  - Required to get a certificate for code signing
- App code signing extends the chain of trust from the OS to apps
  - The secure boot chain “guarantees” that only an official iOS kernel is running
  - The kernel knows the root certificates of Apple and enforces that only signed apps can run

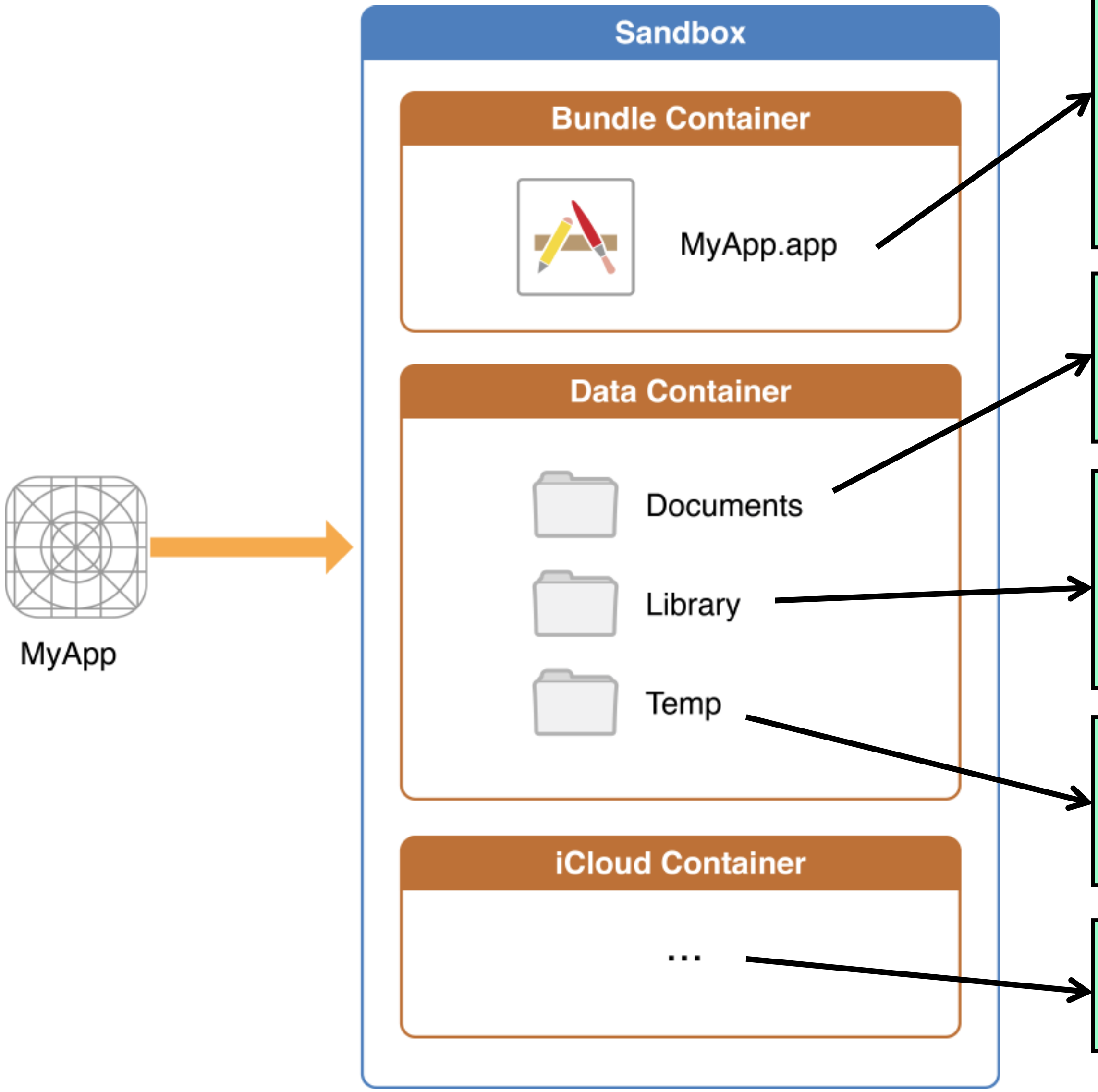
App Store and App Reviewing

- Third-party apps can only be installed via the App Store
  - Exceptions: developers can install apps on a limited number of devices for testing, by specifying their unique device ID (UUID) in a so-called Ad Hoc Provisioning Profile
- Before publishing an app in the app store, Apple…
  - …checks that the app has been signed by the developer with a valid key
  - …reviews the app to check if they "operate as described and if they don't contain obvious bugs and problems"
- It's not entirely clear how this review process works (manual, automated, …), but the following is typically rejected:
  - Apps using private APIs (not officially documented), e.g., to get the UUID
  - Apps that interprete code (i.e., Emulators)
- It happens from time to time that an app "slipps through"...
  - Apple pulls it from the app store once it realizes its mistake

Runtime Security and Sandboxing (1)

- iOS is based on OSX, which itself is based on BSD Unix
- Apps are written in Objective C or Swift and are compiled to native code
- iOS has two users: root and mobile
  - All 3rd party apps run as user *mobile*
- The directories used by an app are as follows:
  - Application folder with static content and compiled binaries
    /var/mobile/Containers/Bundle/Application/<AppID>/MyApp.app/
  - Three additional directories to store persistent and temporary data and support files
    /var/mobile/Containers/Data/Application/<AppID>/Documents/
    /var/mobile/Containers/Data/Application/<AppID>/Library/
    /var/mobile/Containers/Data/Application/<AppID>/tmp/
  - <AppID> is created randomly during app installation

Runtime Security and Sandboxing (2)



- Compiled binary and static app content (e.g., images)
- Is never written, only read
- Is protected by the app signature

- For persistent data storage, can contain e.g., a DB file
- Is backed up by iTunes

- Configuration files, preferences, caches, cookies
- Backed up, except Caches subdirectory

- Temporary files, may be deleted between app usages, not backed up

- App may access additional containers such as iCloud

Runtime Security and Sandboxing (3)

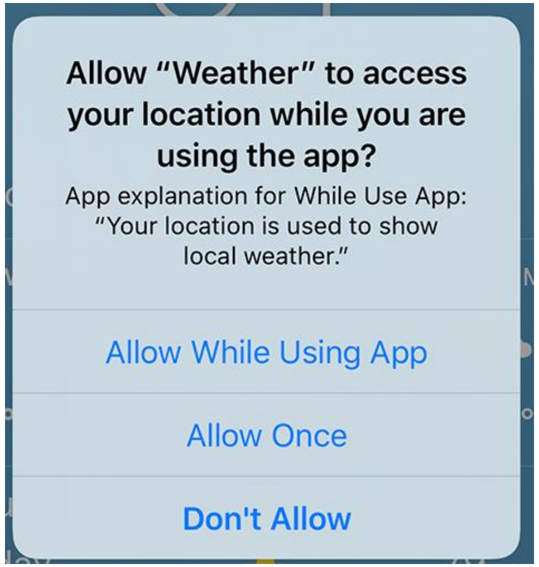
- iOS enforces strict sandboxing for apps
  - E.g., apps cannot access files of other apps
  - Since apps run as user mobile, DAC-based access control can obviously not be used
- To enforce sandboxing, iOS uses the TrustedBSD Mandatory Access Control (MAC) Framework
  - For every app, a policy file is created that defines what parts of the file system and what system calls may be accessed
  - The standard policy of every app defines for instance that
    - Read access to the Bundle Container is allowed
    - Read/write access to the Data Container is allowed
  - The iOS kernel enforces the policy – malicious apps cannot elevate their privileges

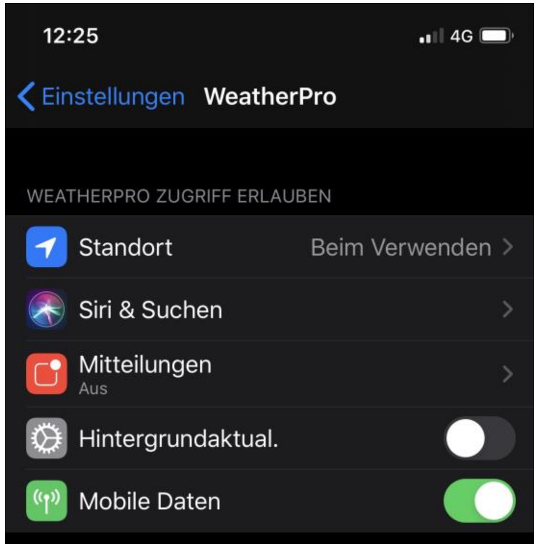
Additional Runtime Security Features

- Enforcing that only Signed Code is Executed
  - When (parts of) an app's code is loaded into a memory page, iOS makes sure that this code is signed code
  - iOS enforces that this code remains unchanged by disallowing memory pages that are both executable and writeable
  - Exception: Safari and WebViews in apps
    - Since CSE would restrict any code generation, iOS added an exception to web applications so that they can use just-in-time (JIT) code generation
- Buffer overflow protection mechanisms
  - Address Space layout Randomization (ASLR) to randomize all memory location upon launch
  - Execute Never (XN) feature of ARM CPUs to mark memory pages such as the stack as non-executable

Permissions

- An app can access specific data or functions, but to do so it must ask the user for permission
  - E.g., location service, camera, contacts, calendar, photos, …
- At launch or when the app requires access, it asks the user for the necessary permissions
- Permissions can be reviewed in the settings and revoked at any time





Inter-App Communication

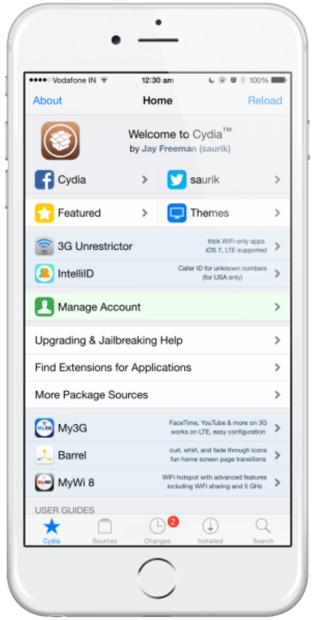
- Although apps are strictly separated from each other, they can communicate with each other using a URL based scheme
- An app ZHAWReceiverApp that allows receiving data from other apps has to register an URL scheme, e.g., zhaw
- Any other app can then use this URL scheme to send data to that app
  - zhaw://this-is-great-data-i-want-to-send-to-you
  - ZHAWReceiverApp receives the request, is put into the foreground, and processes its content
- The receiving app can also restrict the apps that are allowed to send data, this may be reasonable for security reasons
  - To do this, the receiving app checks the application ID of the sender before processing it
- There are also some pre-defined URL schemes in iOS
  - E.g., http (Safari), tel (Phone), mailto (Mail)

Jailbreaking (1)

- Jailbreaking removes the strict protection mechanisms of iOS and grants the user root-level access
- Well… that shouldn’t be possible, right?
  - In theory – yes, as the strong security model should prevent this
  - In practice – no, as vulnerabilities were found to get root-level access, which allowed to manipulate the kernel to deactivate security measures
- There are different vulnerability/exploit types, some examples:
  - Bootrom level
    - Most powerful version, cannot be patched => modify the whole boot chain
  - iBoot level
    - Very powerful, still early in the boot process, can be fixed by a software upgrade
  - User space
    - Less powerful, easier to fix, needs two exploits (one for code execution, one to escalate privileges)

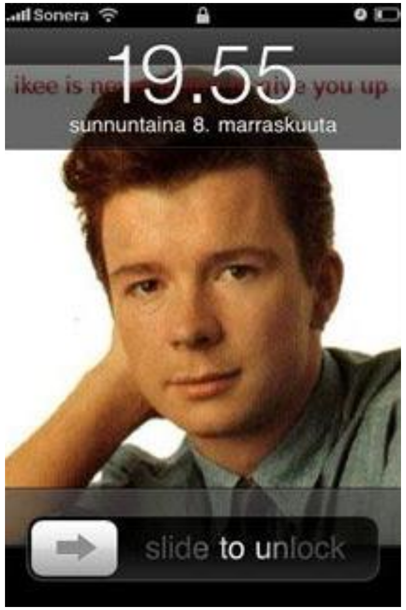
Jailbreaking (2)

- Why would you want to jailbreak an iOS device?
  - You want to understand what is going on under the hood
  - To do security / penetration testing of mobile apps
  - To install apps not allowed into the App Store (e.g., emulators)
  - To customize the look and feel of iOS
  - Software piracy (use apps that cost money for free…)
- Cydia and Sileo are the "App Stores" (packet managers)
  - Very easy to use, enables installation of apps/tweaks from the main and third-party repositories
  - End of 2018, Cydia discontinued the built-in payment methods for paid apps/tweaks in the main repos
  - One of the first tools one typically installs is OpenSSH, which installs an SSH server so low-level access to the device can be done from a terminal on a "real" computer



Jailbreaking (3)

- Of course, jailbreaking significantly reduces the security of iOS devices
  - Arbitrary apps can be installed, unsigned apps are not sandboxed, code that runs with root-rights can be installed, …
  - It's not a good idea to use it on devices you use for productive and security-critical activities
- The risk of malware on jailbroken devices is much higher
  - E.g., IKEE Worm – not much harm done, just set the wallpaper to Rick Astley
  - The vulnerability: The standard root password on iOS is alpine, which most users who jailbreak their phones never change…



Jailbreaking (4)

- Will it ever be stopped by Apple?
  - They try to... (also legally, but in most countries, jailbreaking is legal)
  - So far, virtually all iOS versions were eventually jailbroken
  - Times to jailbreak for new iOS (and HW) versions vary greatly (up to almost 200 days) https://en.wikipedia.org/wiki/IOS_jailbreaking
- iOS 14.5.1 (with A12-A14 CPUs) was the last version with a fully untethered jailbreak and rather an exception, nowadays most jailbreaks are tethered or semi-(un)tethered
  - Untethered: Can boot normally - Boots jailbroken
  - Semi-untethered: Can boot jailed - can jailbreak without PC
  - Semi-tethered: Can boot jailed - can jailbreak with PC
  - Tethered: Can not even boot without connecting to PC and using JB tools
- Status: https://www.theiphonewiki.com/wiki/jailbreak

Android Security

Android Security Principles

- Overall, Android offers similar security features as iOS, but in many cases, they are less strict and not enforced
  - One reason for this is that Android runs on a huge number of different hardware devices
  - For instance, the security of the boot process does not directly depend on Android, but on the implementation of the process on a specific device
- Android includes the following security features
  - Runtime security features such as sandboxing of apps
  - Permission model to grant access to general system functions
  - Code signing
  - Limited inter-app communication
  - Full disk encryption

Runtime Security and Sandboxing (1)

- Android is based on a modified Linux kernel
- Apps are written in Java, Kotlin, native code is written in C/C++
  - Until Android 4.4.x, the Dalvik VM was used to execute apps
  - Since 5.0 the Android Runtime (ART) virtual machine is used
- The compilation process is as follows (hidden to the user)
  - First, the Java compiler is used to create Java bytecode
  - Optional: The bytecode is minimized / obfuscated with ProGuard
  - Then the dx tool converts the Java bytecode to Dalvik bytecode
  - When installing the app on the device, the dex2oat tool is used to convert the bytecode to ART bytecode
  - This guarantees backward compatibility

Runtime Security and Sandboxing (2)

- Apps are distributed as Android Packages (apk / aab files)
  - A zipped archive that contains the bytecode and other resources such as images etc.
- When the app is installed on Android, two directories are created / used
- The apk file is placed in a common directory for all apk files: /data/app/ch.zhaw.myapp.apk
  - The installed apk files get owner system, which is a user used by Android for system tasks (such as starting an app)
- An additional data directory is created to store data that is created and used by the app during runtime: /data/data/ch.zhaw.myapp/
  - For every installed app, a new user is created, and this directory is owned by this user

Runtime Security and Sandboxing (3)

- Example: tripadvisor app, com.tripadvisor.tripadvisor
- Directory /data/app:

```
root@serranolte:/data/app # ls -l com.tripadvisor.tripadvisor-2.apk
-rw-r--r-- system   system   40211333 2015-07-06 16:05 com.tripadvisor.tripadvisor-2.apk
```

  - Read access for everyone (content is not sensitive at all), write access only for user `system`
- Directory /data/data:

```
root@serranolte:/data/data # ls -ld com.tripadvisor.tripadvisor
drwxr-x--x u0_a56   u0_a56            2016-02-04 09:59 com.tripadvisor.tripadvisor
```

  - A specific user (and group) u0_a56 was created, which owns the directory
  - Only that user can write to it
  - Other users can still enter the directory (x), but they usually cannot read / write the files in it (this is used to implement some special sharing options between apps)

Runtime Security and Sandboxing (4)

- What happens when the tripadvisor app runs?

```
root@serranolte:/ # ps | grep tripadvisor
u0_a56    14058 188 _ 618652 65136 ffffffff 400518f0 S com.tripadvisor.tripadvisor
```

  - It runs as user u0_a56
- Only this process can access the data in /data/data/com.tripadvisor.tripadvisor
  - Other (non-root) processes cannot and the tripadvisor app cannot access data of other apps
  - => This is the sandboxing model implemented by Android
- In contrast to iOS, Android has no MAC model, but "simply" relies on discretionary access control with standard Linux file permissions
  - Where all processes run using their own user

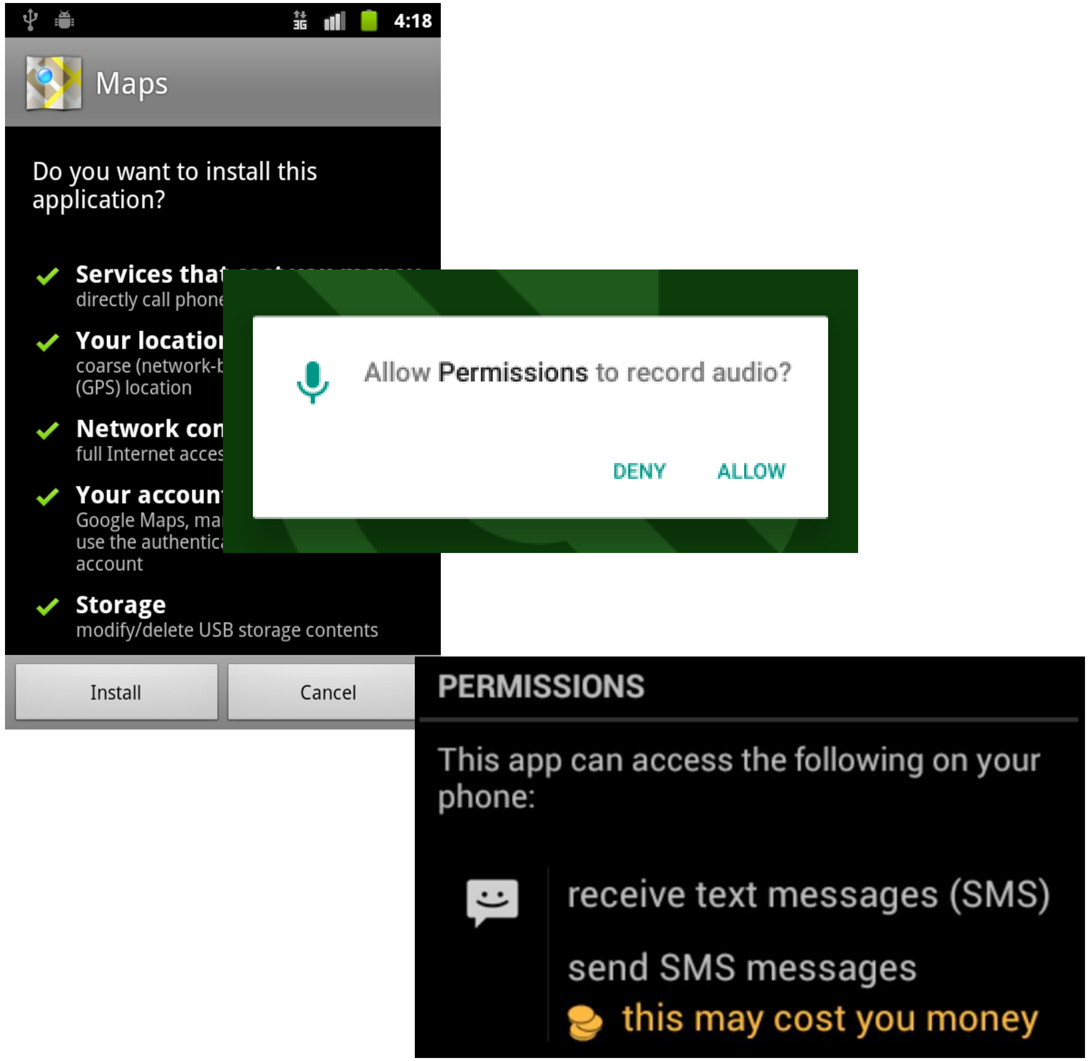
```
u0_a41    13853 188   574484 38600 ffffffff 400518f0 S com.android.mms
u0_a66    13886 188   555656 22616 ffffffff 400518f0 S com.sec.android.widgetapp.alarmwidget
u0_a142   13899 188   556116 23408 ffffffff 400518f0 S com.sec.android.widgetapp.programmonitorwidget
u0_a107   13912 188   557844 22636 ffffffff 400518f0 S com.sec.android.app.controlpanel
u0_a56    14058 188   617612 65132 ffffffff 400518f0 S com.tripadvisor.tripadvisor
```

Additional Runtime Security Features

- Just like iOS, Android provides some additional runtime protection features
- Most of them serve to prevent buffer overflow / memory corruption attacks
  - But this shouldn't be a problem with Java?
  - True, but apps can use native libraries written in C/C++, where corresponding problems may arise
- Buffer overflow protection mechanisms
  - Address Space layout Randomization (ASLR) to randomize memory locations (Android 4.0+)
  - No eXecute (NX) to prevent code running on the stack or heap

Permissions

- Fine-grained permission model
- Android <6.0: User is informed about desired permissions, had to accept all
- Andorid >=6.0: Two permission categories: dangerous and normal
  - Normal – Granted implicitly
  - Dangerous – Ask when needed
  - Permissions organized in groups
  - If one dangerous permission was granted in a group, all are granted*
- Experience shows that users rarely inspect the list and just click "install"
  - In fact, many Android malware cases abuse this, e.g., overlay-attacks



Code Signing and apk distribution

- In contrast to iOS, apps CAN be installed from anywhere
  - There is the Google Play Store, but distribution by e-mail, download etc. is possible too
  - Installing apps from untrusted resources is highly critical, of course
- Apps must be signed, but this is not done with certificates issued by Google
  - Instead, a developer typically uses a self-signed certificate for signing
- This seems pointless… what could be the purpose of this?

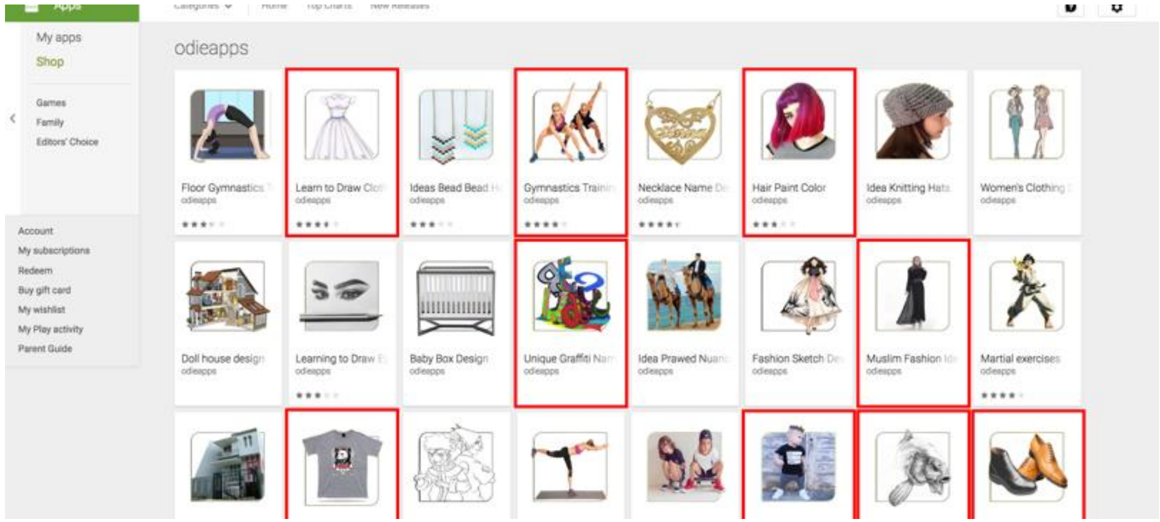
Certificate of tripadvisor App

```
rema:Desktop marc$ openssl pkcs7 -inform DER -in CERT.RSA -text -print_certs
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1278620470 (0x4c363336)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=MA, L=Newton, O=TripAdvisor, OU=Mobile Engineering, CN=Ollie the owl
        Validity
            Not Before: Jul  8 20:21:10 2010 GMT
            Not After : Jun 25 20:21:10 2060 GMT
        Subject: C=US, ST=MA, L=Newton, O=TripAdvisor, OU=Mobile Engineering, CN=Ollie the owl
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:c6:63:0b:e5:90:e7:d2:88:05:6b:f7:ee:37:08:
                    fc:8d:d7:bd:7e:91:dd:0e:50:55:06:c4:d6:11:ef:
                    c0:95:cf:93:57:3f:c9:9e:20:81:32:da:93:ac:ea:
                    45:b7:ba:52:9d:9c:63:4e:c0:e7:c3:4a:fb:ec:35:
                    28:6b:ee:5a:5f:87:7a:34:95:38:b4:2e:2d:be:7b:
                    aa:b6:2e:2f:59:03:36:ba:d6:6d:78:f2:70:6e:6f:
                    46:b8:9f:1e:2a:0b:85:50:70:bb:84:da:08:14:1d:
                    a3:81:89:d5:7a:5d:91:76:88:1b:bc:5f:30:48:81:
                    9d:09:ed:fe:25:d4:79:0b:f9
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
         5f:ce:fb:db:8e:9e:32:8f:3c:49:50:20:ad:92:f6:34:ab:a6:
```

Issuer == Subject
=> Self-signed

Apk Distribution via Google Play Store

- Play Store does not find all malware
- Genuine apps can be updated to malware apps
  - Add malicious features after play store release
  - Malware can try to hide, e.g. detect when it's run on a emulator
  - Load new code at runtime is forbidden by Google, but it can be done
- Social engineering is often used to trick the user to load a malicious app
- Apk repackaging: Genuine apps are repacked with malware features.



- Example: Windows Keylogger - https://www.bleepingcomputer.com/news/security/android-apps-infected-with-windows-keylogger-removed-from-google-play-store/

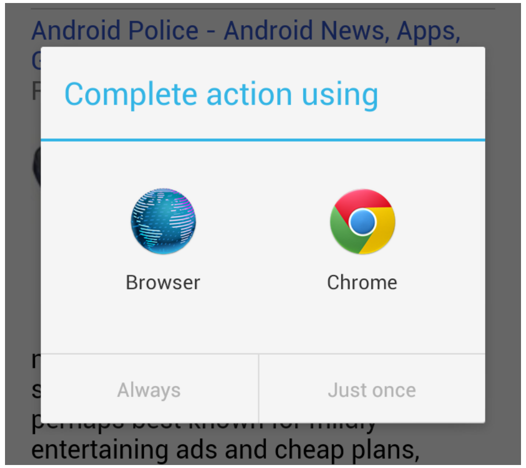
Inter-App Communication (1)

- Android allows apps to communicate with each other
  - The sender communicates with a specific component (an activity) of the recipient
  - The object that is used for messaging is called intent
- If an app wants to receive intents, the activity must be configured with an intent-filter in AndroidManifest.xml
  - Usually, URL schemes are used for communications
  - Therefore, the intent-filter must define the accepted URL schemes (e.g., zhaw)
- Example: Chrome browser

```
<activity android:name="com.google.android.apps.chrome.document.ChromeLauncherActivity"
    android:theme="@android:style/Theme.NoDisplay"/>
    <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="googlechrome"/>
        <data android:scheme="http"/>
        <data android:scheme="https"/>
        <data android:scheme="about"/>
        <data android:scheme="javascript"/>
    </intent-filter>
</activity>
```

Inter-App Communication (2)

- Other apps use this URL scheme to send data to the app
  - https://www.securesite.com (=> Chrome)
  - zhaw://this-is-great-data-i-want-to-send-to-you (=> our own app)
  - Apps using the same scheme: User chooses target app



- Very similar to iOS, but the Android approach offers additional functionality
  - Bi-directional communication: the recipient can send back an intent to the sender
  - Apart from URL schemes (implicit intents), there are also explicit intents
    - The sending app explicitly defines the target activity
    - The sending app specifies the fully qualified activity name (e.g., ch.zhaw.myapp.ReceiverActivity)
    - This prevents that a malware app can hijack intent data by registering for many URL schemes

File Encryption (1)

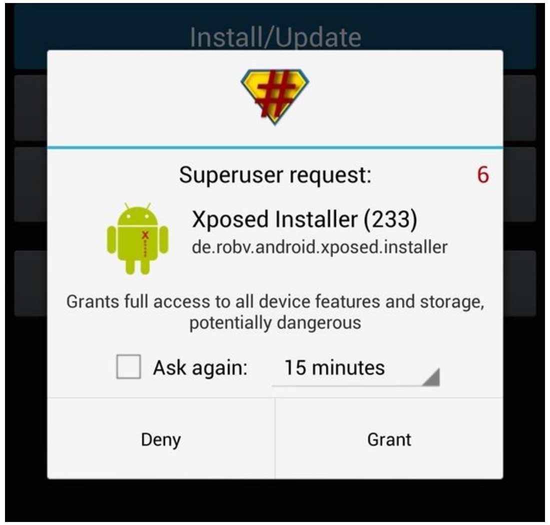
- Android support Full Disk Encryption (FDE) and File-Based Encryption (FBE)
- Full Disk Encryption is the default for Android 5.0 to 9.0
  - Uses the dm-crypt Linux kernel module with AES-CBC with 128-bit key to encrypt /data
- Key management
  - The key is randomly generated when enabling FDE or at first boot since Android 5.0
  - If no PIN / password is set, the key is "protected" with "default_password" :-D
  - If a PIN / password is set, it is "stretched" using scrypt => used to encrypt the FDE key
- File-Based Encryption (FBE) is required for Android 10 and beyond
  - Based on fscrypt with AES-256 in XTS mode
  - FBE (master) key stored in the TEE, unlocked when user logs in (no PIN/PW = no protection)
- Overall, it is complicated* - manufacturers can modify default behavior, there might (not) be a hardware TEE, system apps might not encrypt data,…

File Encryption (2)

- How does FDE/FBE help?
- If someone steals your device, (most) data cannot be easily accessed
- In contrast to iOS, the key is not required to be “in the hardware”, so there’s eventually no need to perform any brute force attacks on the device itself
  - The data can therefore be copied from flash storage and the PIN / password can be brute-forced using “lots of computing power”
  - This also means no built-in mechanisms to delay the attacker after a few wrong attempts can be used
- How difficult is the brute force attack?
  - scrypt is specifically designed to make brute force attacks using special hardware difficult
  - Short PINs (e.g., 4 or 6 digits) can nevertheless be cracked easily

Rooting (1)

- Rooting "more or less" corresponds to Jailbreaking
  - Like with iOS, Android does not give the user the right to do anything as root by default
  - Technically, it's different, but the goal is the same: get root rights on the device…
  - Many Phone manufacturers provide official ways to root the device (if not => vulnerability)
- The approach of rooting typically means getting a su binary onto the device
  - **su** is a standard Linux/Unix tool that allows a user to switch the security context and become another user, e.g., root
  - As the root user on Android does not have a password, simply calling **su** (without a username) is enough to become root
- In addition, an app such as SuperSu is installed, which is launched by the su binary to show the user a dialogue to grant or deny root rights



Rooting (2)

- Just like with jailbreaking iOS, there are various reasons for rooting
  - See what is going on under the hood and for security / penetration testing
  - Run apps that require root rights (e.g. powerful backup solutions or apps to remove pre-installed bloatware)
- There are two main methods to root an Android device
  - Unlock the bootloader
    - Some devices have an unlockable bootloader that allows to flash new firmware onto it
    - This can then be used e.g. to install a pre-rooted kernel that contains **`su`**
  - Use a vulnerability that allows to escalate your privileges to root
    - Android (e.g., the Linux kernel), may contain suitable vulnerabilities to ultimately install **`su`**
- Just like with jailbreaking, most devices can eventually be rooted

Summary

- Mobile devices have a different risk-profile than standard computers
  - Loss, theft, use in non-trusted environments, digital wallet, authentication factor, …
- Therefore, device and OS manufacturers have integrated protection mechanisms that go beyond those of standard computers
- iOS implements a strong and restictive security model
  - Secure boot chain, file encryption, passcodes / Face ID, keychain, apps from Apple‘s app store only, sanboxing, restricted inter-app communication, permissions
- Android implements a less restrictive but also less secure model
  - Sandboxing, permissions, apps from any source*, inter-app communication, file encryption
- Jailbreaking/rooting inactivates some security features and provides root access