IAIK TU Graz

iOS Application Security

Mobile Security 2022

Florian Draschbacher
florian.draschbacher@iaik.tugraz.at

Some slides based on material by **Johannes Feichtner**

Reminder: Assignment 1

- **Submission on April 10th**

- If you haven’t started yet:
  - Start now!

- Any questions?
  - Discord channel for anything relevant for others as well
  - Send me an email

Outline

- App Internals
  - Application Format
  - Sandbox
  - Code Signing
- App Distribution
- App-Level Security on iOS
- iOS Malware & Jailbreaking
- App Analysis on iOS

APP STALKING —

Dozens of iOS apps surreptitiously share user location data with tracking firms

Applications don't mention that they're selling your precise location to third parties.

SEAN GALLAGHER - 9/10/2018, 9:11 PM

Source: https://goo.gl/FjCesH

What?

Location data of popular apps leaked to 12 known monetarization firms

- Bluetooth LE Beacon Data
- GPS Longitude and Latitude
- Wi-Fi SSID (Network Name) and BSSID (Network MAC Address)
- Further device data
  - Accelerometer, Cell network MCC/MNC, Battery Charge % and status (Battery or charged via USB)

Problem?

Users *agree* on sharing their location for different purposes, e.g. „Location based social networking for meeting people nearby"

Application Security

Even on a perfectly hardened platform

- Malicious applications may compromise sensitive data
- Insecure applications can open doors to attackers!

iOS Platform limits potential attack surface to a minimum

- Code Signing
- Sandbox

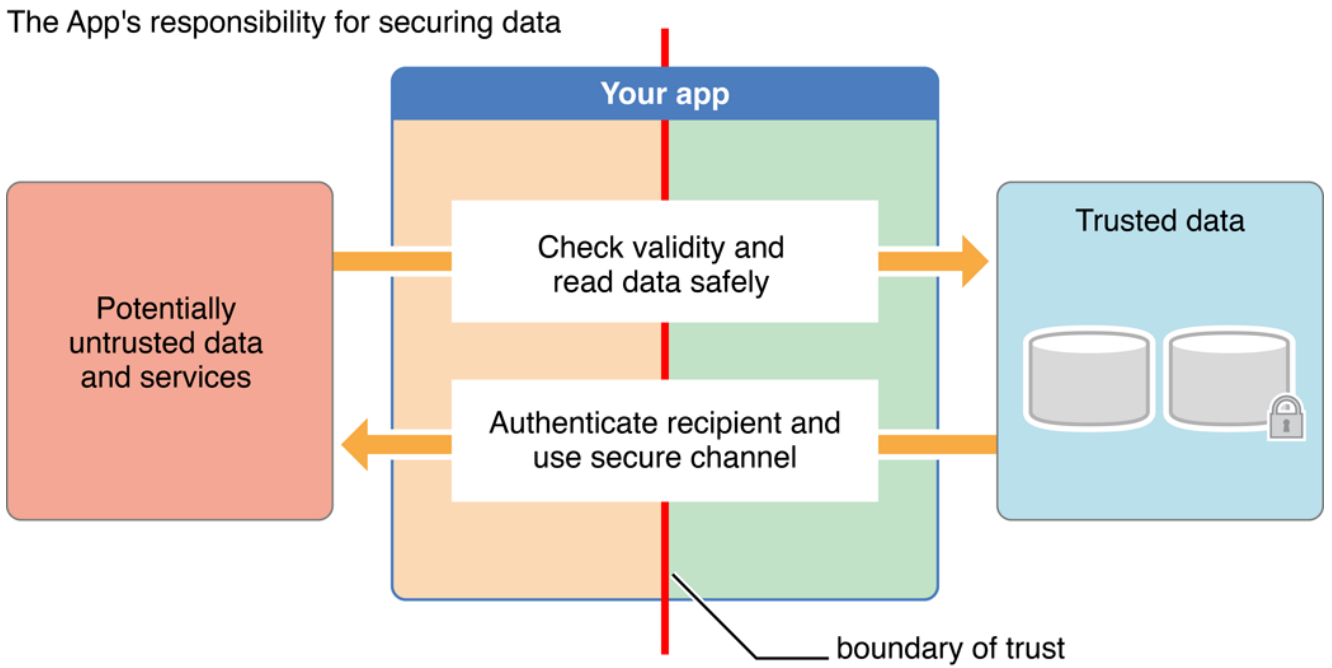
App developers need to

- Submit applications to Apple for review before publishing
- Follow security guidelines

Application Security

From Apple's Developer Documentation:

"The most important thing to understand about security is that it is not a bullet point item. You cannot bolt it on at the end of the development process. You must consciously design security into your app or service **from the very beginning**, and make it a conscious part of the entire process from design through implementation, testing, and release."



Source: apple.com

App Internals

App Files

- Distributed in IPA format (“iOS App Store Package”)
- ZIP archive with all code + resources

```
$ unzip SuperPassword.ipa –d acndemo
$ ls -R acndemo/

/Payload/SuperPassword.ipa/          App itself + static resources
       -> SuperPassword              Binary executable (ARM-compiled code)
       -> Info.plist                 Bundle ID, version number, app name to display
       -> MainWindow.nib             Default interface to load when app is started
       -> Settings.bundle            App-specific preferences for system settings
       -> _CodeSignature             Signatures of resource files
       -> further resources          Language files, images, sounds, more GUI layouts (nib)
/iTunesArtwork                       512x512 pixel PNG image -> app icon
/iTunesMetadata.plist                Developer name + ID, bundle identifier,
                                     copyright information, etc.
```

FairPlay DRM

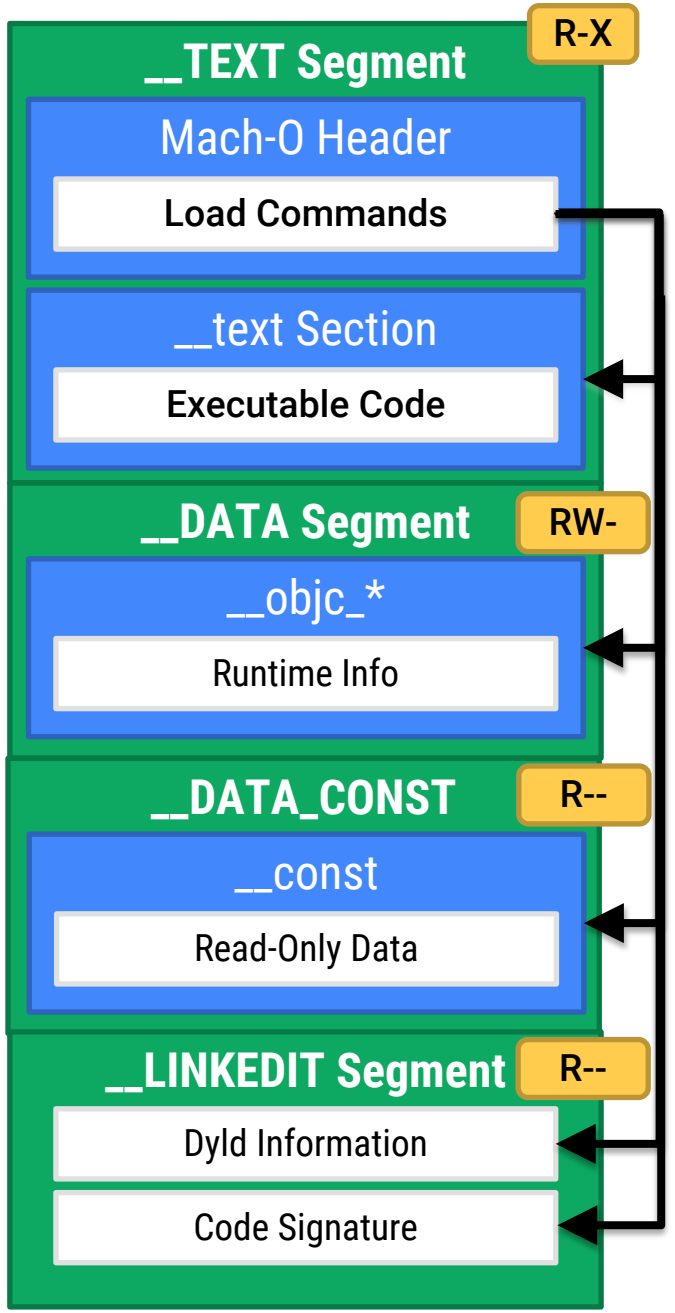
Not to be confused with Code Signing!
(Covered in a few minutes)

- The executable binary inside the IPA file is DRM-protected
  - Encrypted using Apple's FairPlay DRM scheme
- At runtime, it is transparenly decrypted by the kernel
  - Apple Protect Pager: Transparenly decrypts file when mapping into memory
  - FairPlay DRM system is heavily obfuscated and only partly reverse-engineered
- Encryption is carried out by Apple, and only affects App Store apps (*)
  - (*) TestFlight (Beta-Test) distribution is also affected
- DRM can be removed by using a Jailbroken device
  - Dump the application's memory at runtime

Sources: J. Levin: "*OS Internals", Junzhi Lu et al.: "Play with Fire: Uncover FairPlay DRM and Obfuscation for Fun and Profit"

iOS Executables

- Binaries are in **Mach-O** format (once decrypted)
- Contains ***segments*** of one or multiple ***sections***
  - Header
    - Architecture
    - Load Commands
      - Virtual Memory Layout
      - Libraries
      - Encryption
  - Data
    - Executable code
    - Read / write data
    - Objective C runtime information
    - Code signature



Source: J. Levin: "*OS Internals",

App Installation

- The application and its data are spread across multiple file system locations
  - `/private/var/mobile/Containers/Bundle/Application/<APP_UUID>/`
    - Extracted IPA contents
  - `/private/var/mobile/Containers/Data/Application/<CONTAINER_UUID>/`
    - User-generated app data. Container UUID changes with every new launch.
    - Subfolder „Library“: Cookies, caches, preferences, configuration files (plist)
    - Subfolder „tmp“: Temp files for current app launch only (not persisted)
    - Subfolder „Documents“: Visible through iTunes File Sharing and Files app (if enabled)
  - `/private/var/mobile/Containers/Shared/AppGroup/<APP_UUID>/`
    - To share with other apps & extensions of same app group

Source: J. Levin: "*OS Internals",

Application Sandbox

Application Sandbox

- Isolate apps from each other and the system
  - Restricts resource access and system integration of third-party applications
  - App must hold *Entitlements* for advanced interactions with system
  - Apps may request access to some system-wide data by asking user permission

- Limits file system access to app's container
  - `/var/mobile/Containers`
- Disallows most system calls
  - Prevent sandbox escape

Source: J. Levin: "*OS Internals"

Recall: Mandatory Access Control (MACF)

- Various hooks scattered throughout syscall implementations in kernel
- Hooks call out to Policy Modules for checking if operation permitted
- Foundation for central iOS security features
  - Code Signing Policy Module: AppleMobileFileIntegrity.kext
  - **Sandbox Policy Module: Sandbox.kext**

Picture: Google / Apache 2.0

Source: J. Levin: "*OS Internals"

Sandbox.kext

MACF Policy Module that implements the application sandbox

- Can be configured through *Profiles*
  - Compiled from proprietary Sandbox Profile Language (SBPL)
  - Specifies what is allowed and what not
  - iOS only supports profiles hard-coded into the kernel extension
  - Dynamically extended
    - Depending on user-granted access (e.g. Media Library)
    - Depending on app entitlements
- Profiles enforced in hooks of > 100 system calls

Source: J. Levin: "*OS Internals"

Code Signing

Code Signing

All code executed on iOS must be signed

- Protects the integrity of applications
- Ensures that Apple had a chance to screen developer and/or application
- Signature also contains and protects app entitlements

- **Exceptions for some Apple apps**
  - Holding a special entitlement (discussed later)
  - E.g. Javascript JIT in Safari

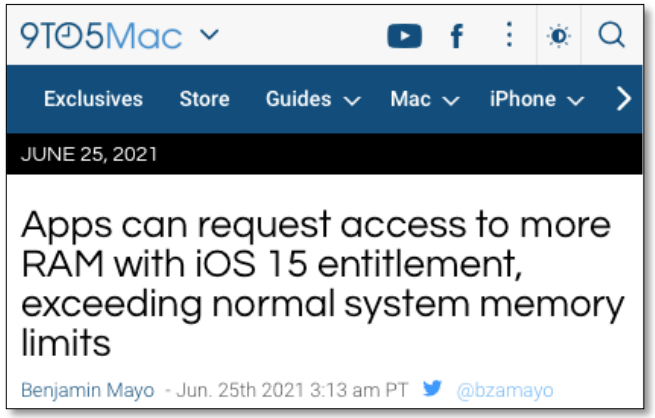
- **Exceptions for apps controlled by a debugger**
  - Development!

Source: J. Levin: "*OS Internals",

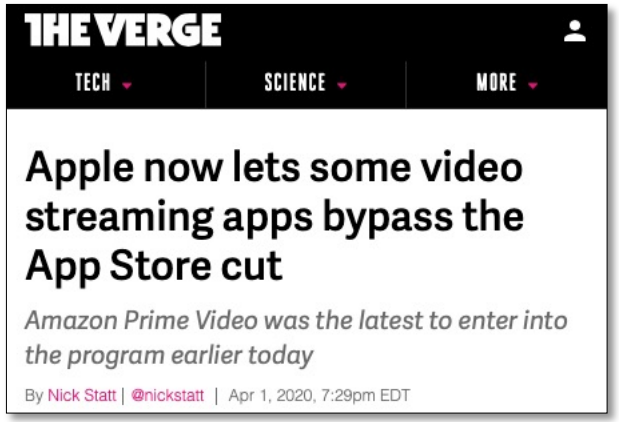
Entitlements

- Define degree to which application can integrate and interact with system
- Enforced by kernel and system before sensitive operations
- Granted by Apple

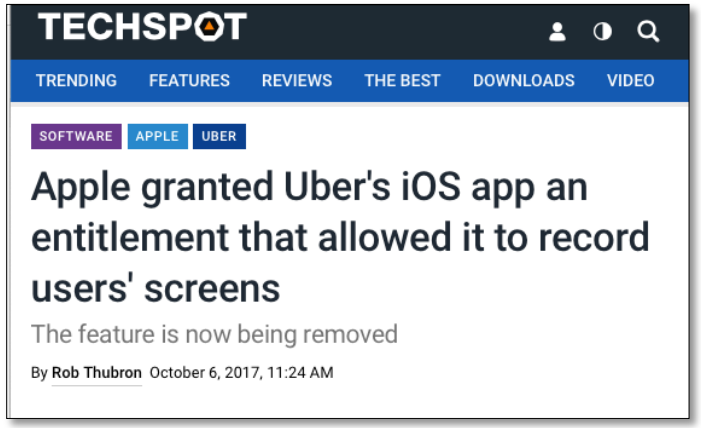
- More than 3000 entitlements defined throughout subsystems on iOS 15
  - Only a fraction are officially documented and allowed to normal third-party apps



Source: 9to5mac.com



Source: theverge.com



Source: techspot.com

Sources: J. Levin: "*OS Internals", newosxbook.com

Code Signatures

- Two parts
  - Application Seal: `_CodeSignature/CodeResources`: Hashes of all resources
  - Embedded Signature: Actual *code* signature

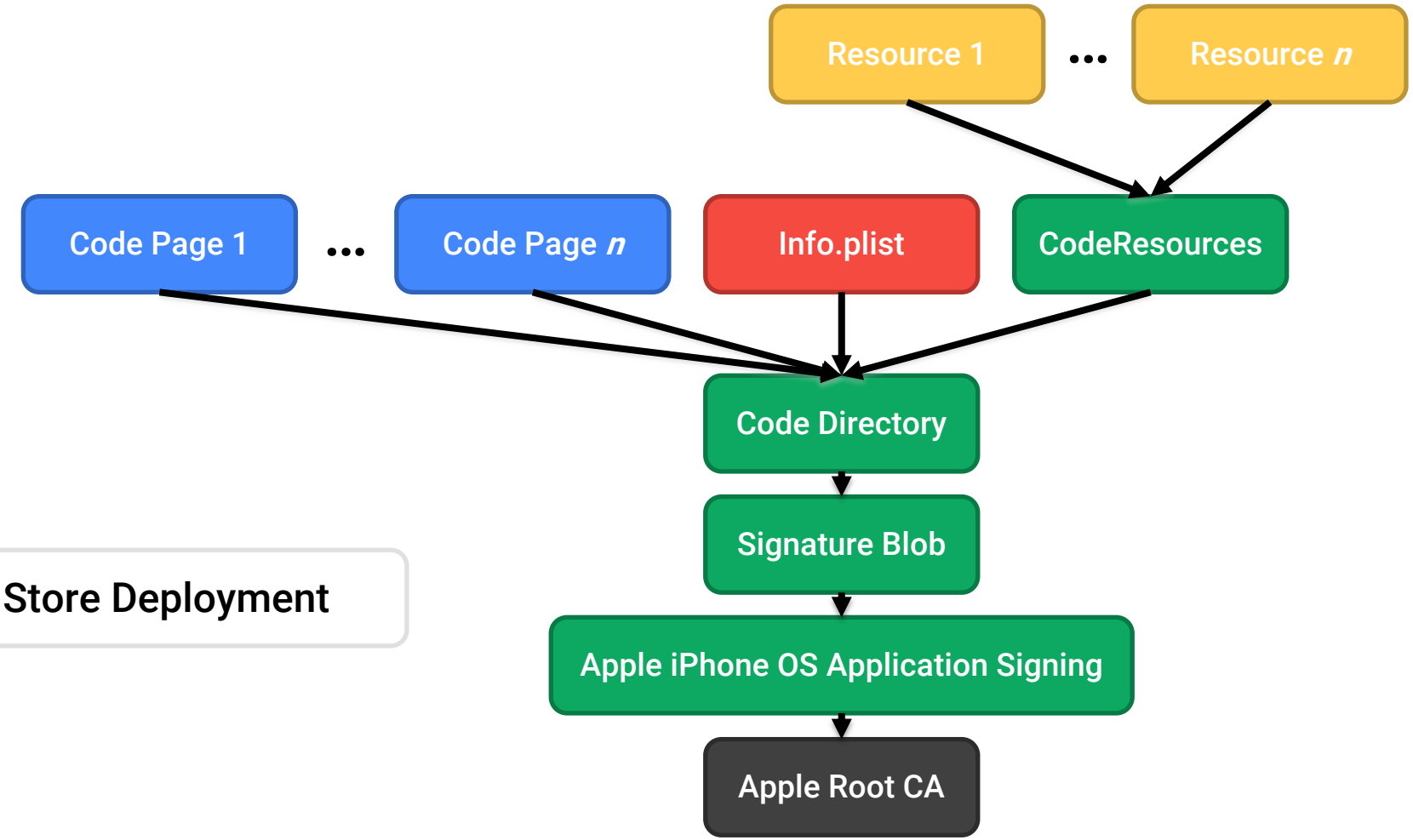
The Embedded Signature

- Stored in `__LINKEDIT` segment of the MACH-O binary
- Consists of Codesigning Blobs:
  - **Entitlements Blob**: List of app's entitlements
  - **Requirements Blob**: Specify rules for validating the app signature
  - **Code Directory Blob**: Hash of code pages, App Seal and Codesigning Blobs
  - **Signature Blob:** Signs all these hashes

Sources: J. Levin: "*OS Internals", Umang Raghuvanshi: "A Deep Dive into iOS Code Signing"

Code Signatures

Code Signature forms a **signed tree of hashes**, rooted at Apple CA certificate



App Store Deployment

Code Signature Enforcement

But how is it implemented?

Before starting a process (in the `exec` system call)

- Kernel extracts the Code Signature from the binary
- Stores it in special *Unified Buffer Cache*

On page faults

- Handler checks whether page belongs to a code-signed object
- Requests MACF policies to validate the signature of the page
  - **AppleMobileFileIntegrity.kext!**

Source: J. Levin: "*OS Internals"

AppleMobileFileIntegrity.kext (AMFI)

- Basic validation of Code Signature format and hashes
- Check CodeDirectory Hash *(CDHash)* against Trust Cache
  - Preinstalled system applications
- Third-party apps: pass to user-space amfid daemon
  - Don’t parse complex signature format in kernel
- Also hooks into `mmap` and `mprotect` system calls
  - Ensure requested memory protections do not allow execution

Source: J. Levin: “*OS Internals”

AMFI Userspace Daemon (amfid)

- Enforces rules from Requirements Blob
- Inspects certificate chain in the Signature Blob
  - Complex PKI parsing
- Queries installed Provisioning Profiles
  - To complete chain from Developer Certificate to Apple CA
- This is the weakest point in Code Signing Enforcement
  - Most jailbreaks manipulate amfid to circumvent code signing

Source: J. Levin: "*OS Internals"

Entitlements Vulnerability (“Psychic Paper”)

- A vulnerability in iOS <13.5 enabled apps to gain arbitrary entitlements
- Exploited differences between XML parsers in kernel and user space

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <!-- these aren't the droids you're looking for -->
    <!--><!-->
    <key>platform-application</key>
    <true/>
    <key>com.apple.private.security.no-container</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
    <!-- -->
</dict>
</plist>
```

User Space (amfid):

No entitlements

Kernel (AMFI.kext):

- task_for_pid-allow: true
- platform-application: true
- com.apple.private.security.no-container: true

Source: blog.siguza.com

App Distribution

Distribution Options

- Apple tightly restricts the possibilities for installing software on iOS
  - Jailbroken devices: Code signing usually disabled

Distribution	Developer Account	Review	Devices
App Store	Paid (99$/yr)	Yes	All
TestFlight	Paid (99$/yr)	Yes (if *public* beta test)	Limited
Enterprise	Enterprise (*) (299$/yr)	No	All that have *Provisioning Profile*
Development / Ad-Hoc	Free	No	Limited, Preregistered

(*) Eligible only companies of more than 100 employees, for in-house distribution of proprietary software

Source: apple.com

Provisioning Profiles

- Apps that do not go through a review process cannot be signed by Apple
  - Developers sign them using a *Development Certificate* issued by Apple
- How to restrict the power of this development certificate?
  - Restrict it to certain application, devices, entitlements
- How?
  - Provisioning Profiles

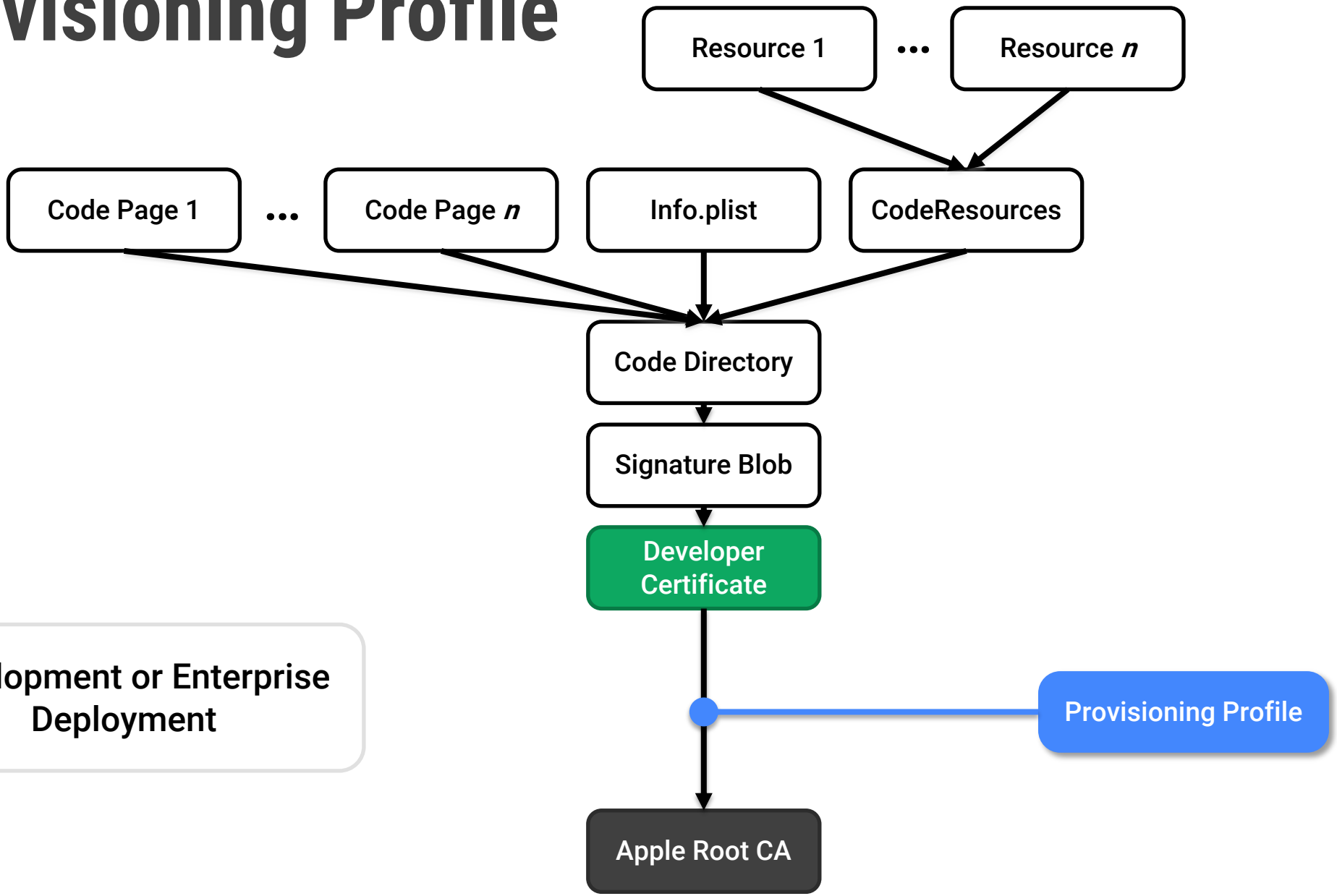
Source: J. Levin: "*OS Internals"

Provisioning Profile

- Link between developer certificate and Apple CA
  - Must be installed on the device (may be embedded in IPA)
  - Only needed for development and enterprise distribution
    - App Store or TestFlight distribution: Signed by Apple after review
- Contains:
  - Application Identifier: Dev. Certificate can only sign specified app **Wildcard possible!**
  - Device UDIDs: Profile may only be installed on specified devices
  - Entitlement Restrictions: The entitlements a signed app may have at most
  - Developer Certificate: The corresponding private part signs the application
- Signed and issued by Apple

Source: J. Levin: "*OS Internals"

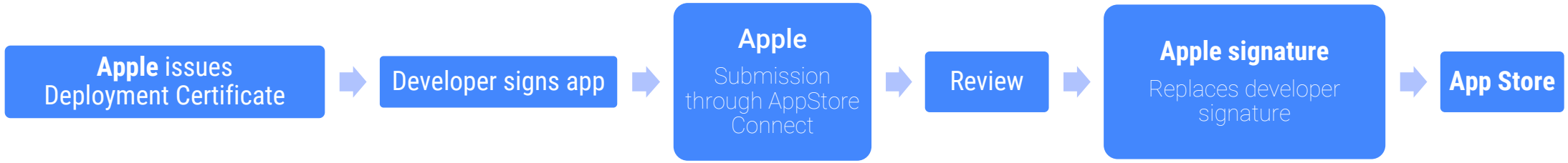
Provisioning Profile



Development or Enterprise Deployment

Application Signing

App Store Distribution:



Development Distribution:

Please note the key pair for the development and deployment certificates must be supplied by the developer in both cases
Signing an app involves using the private key for the development/deployment certificate.

App Store Review

> “On average, 50% of apps are reviewed in 24 hours and over 90% are reviewed in 48 hours.”

Source: apple.com

Process:

1. Developer uploads app
2. Enter queue for review (on re-upload: back to start)
3. After review
   - On reject: Notification with reason
   - On success: App release

\+ Quality control and nearly no evil apps

\- Not possible to fix bugs / security issues quickly (2 expedited reviews / yr)

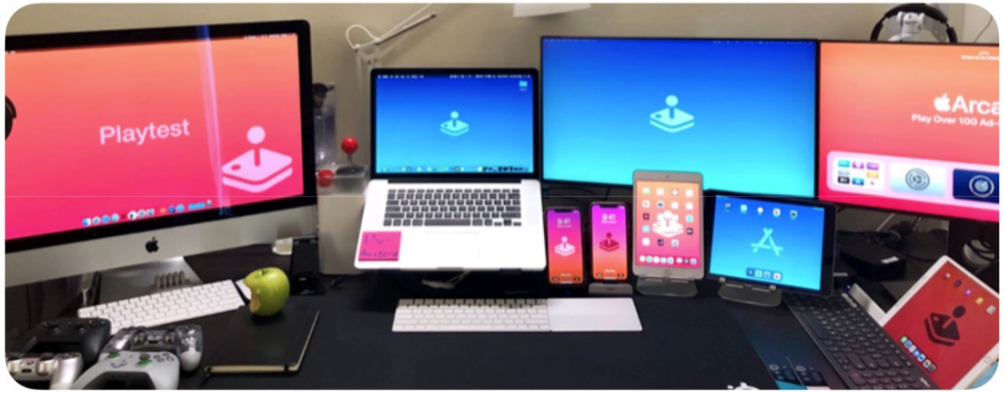
- Used to be a very opaque process
  - Some details leaked during Apple vs Epic lawsuit

App Review Process

Multiple steps

- **Automated Static Analysis**
  - Analyse application binary
- **Automated Dynamic Analysis**
  - Detect runtime behavior using random user input
- **Manual Reviews**
  - Manually check for guideline violations

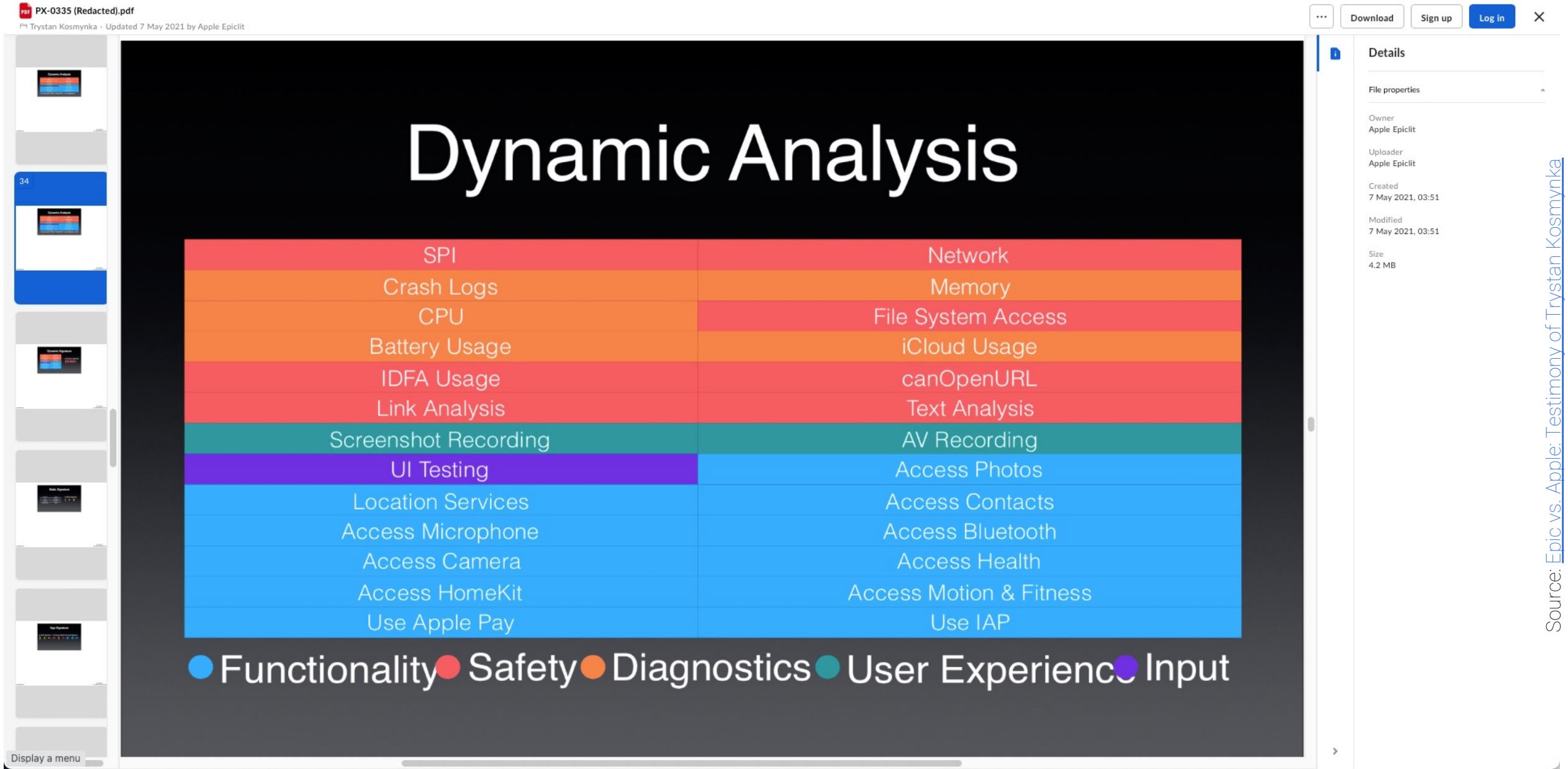
Dynamic Analysis



Manual Analysis

Source: Epic vs. Apple: Testimony of Trystan Kosmynka

App Review Process: Dynamic Analysis



Source: Epic vs. Apple: Testimony of Trystan Kosmynka

App Review Process: Static Analysis

PX-0335 (Redacted).pdf
Trystan Kosmynka · Updated 7 May 2021 by Apple Epiclit

Download | Sign up | Log in

Details

File properties

Owner
Apple Epiclit

Uploader
Apple Epiclit

Created
7 May 2021, 03:51

Modified
7 May 2021, 03:51

Size
4.2 MB

Source: Epic vs. Apple: Testimony of Trystan Kosmynka

App Review Process: Manual Analysis

- More than 500 people review 100,000 apps per week
- Process is assisted by automation
  - E.g. automatically identifying changes in app updates
- Decisions regarding high-profile apps may be overruled by ERB
  - Executive Review Board
    - Phil Schiller, VP of Marketing at Apple

Sources: 9to5mac.com, cnbc.com

App Distribution: Future

- Several ongoing lawsuits and initiatives
- Breaking Apple's uncompetitive control over app distribution

- Apple vs Epic
  - USA: Apple must allow external IAP payment options

- EU: Digital Markets Act expected to be finalized this year
  - Allow sideloading
  - Alternate purchase methods

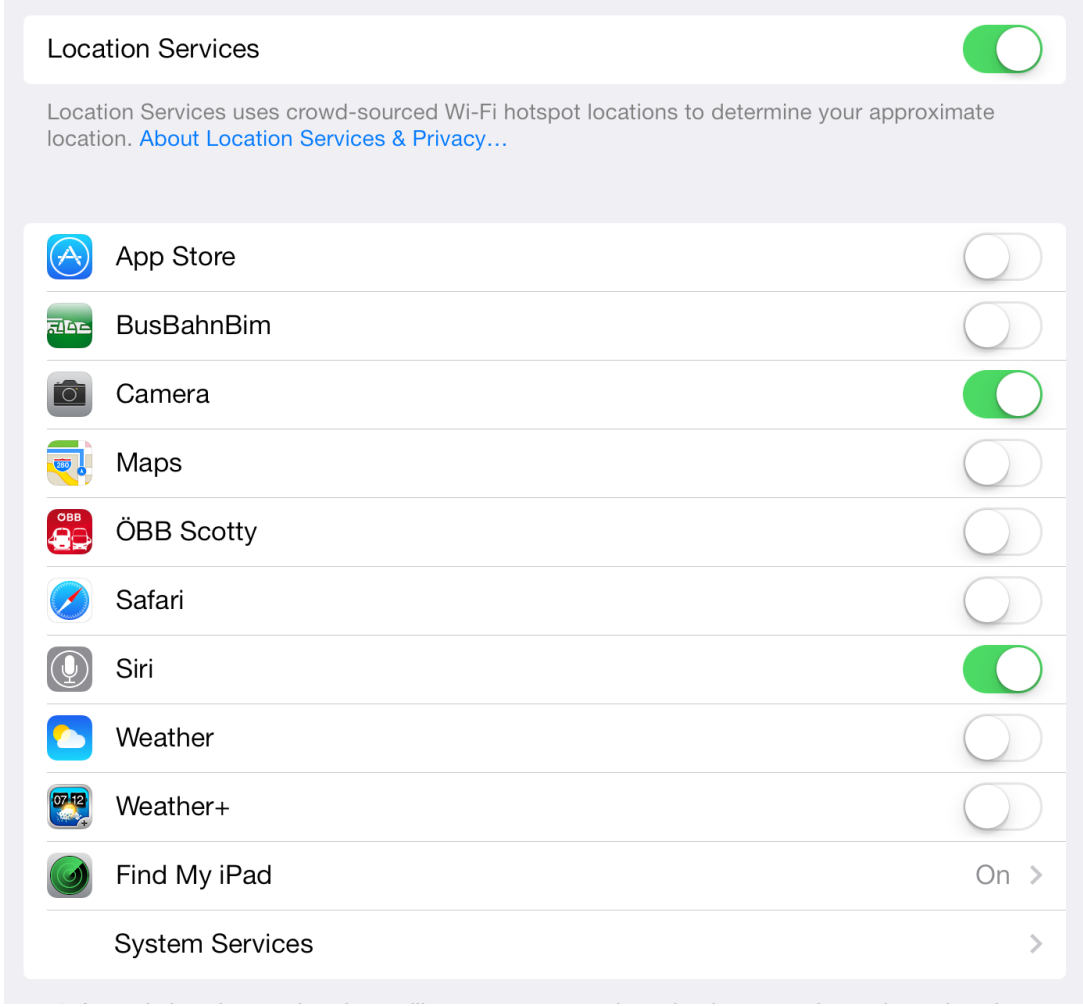
Sources: macrumors.com, techcrunch.com

App-Level Security

iOS Permissions

- Users can grant certain permissions
  - Apps show permission dialog at runtime
- Can be revoked in app settings
- Workflow
  - First API access: Request user permission
  - Further API access:
    Refer to saved permission state

Note: Only way to remove internet access for app
→ Turn off your WiFi / LTE connection…



iOS Permissions

- Apps do not *directly* request permissions
  - Developers do not have to specify which they want to use
  - Depending on use of sensitive APIs

- **Example:** App wants to access user's contacts
  - App calls method from CNContactStore class
  - Since iOS 10: Apps must present description how requested data is used
  - API access blocked until permission granted / denied

- ***Sensitive APIs***

Contacts, Microphone, Calendar, Camera, Reminders, Photos, Health, Motion Activity & Fitness, Speech Recognition, Location Services, Bluetooth Sharing, Media Library, Social Media Accounts

iOS Cryptography APIs

- CommonCrypto iOS 2+
  - Low-level C library for symmetric encryption, message digests, KDF, HMAC
- CryptoKit iOS 13+
  - High-level Swift library for asymmetric & symmetric crypto, MAC, digests
- Security Framework iOS 2+
  - Low-level C library for cryptographically secure random numbers
- Network Framework iOS 12+
  - Low-level Swift library for TLS (and TCP, UDP)
- URLSession API iOS 7+
  - High-level ObjC/Swift library for HTTPS (and HTTP, FTP, …)

Source: developer.apple.com

App Transport Security (ATS)

iOS 9+

- **Requires that all URLSession requests are made over HTTPS** (instead of HTTP)
  - And that the connection employs modern TLS standards

- **Configurable in Info.plist dictionary**
  - Specify exceptions
    - For specific domains
    - For specific contents
  - Exceptions must be justified for App Review!

Certificate Pinning or Self-Signed Certificates still relatively difficult!

Source: developer.apple.com

iOS Malware & Jailbreaking

Malware?

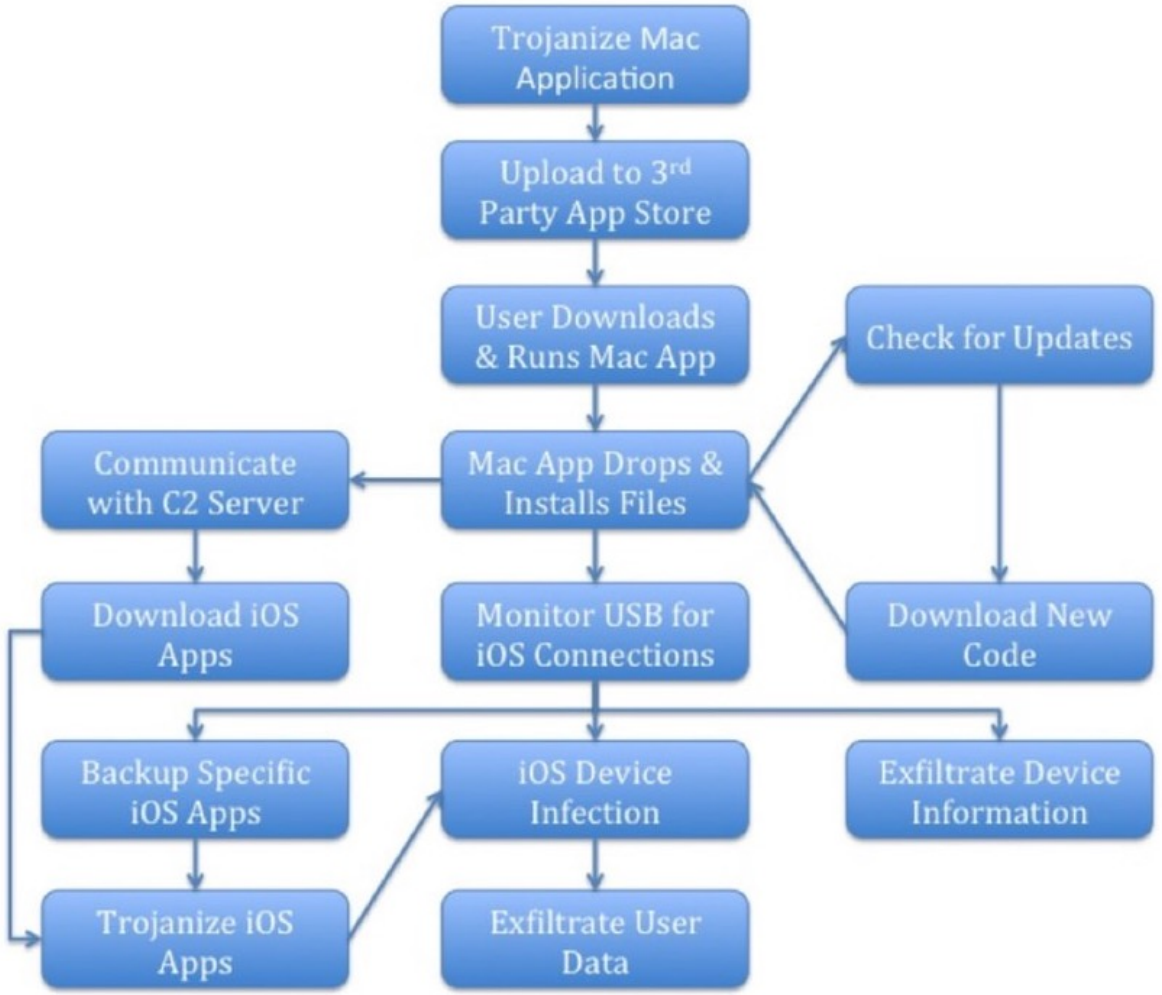
- Advanced protections
  - Code Signing
  - Sandbox
- Reduced attack surface → stripped down OS
  - Lots of useful binaries missing, e.g. no `/bin/sh` → no „shell“ code ☹
  - Even if shell → no `ls`, `rm`, `ps`, etc.
  - With code execution, what could you do?
- Privilege separation
  - Most processes run as user „**mobile**“
    - Mobile Safari, Mobile Mail, Springboard, etc
  - Many resources require **root** privileges

Wirelurker Malware (2014)

- Maiyadi App Store
  - 3rd Party Mac AppStore in China
  - Hosts „free“ apps
- Code signatures can be disabled on macOS

Attack scenario

1. macOS infection
2. App installed via cable on iPhone, signed with enterprise app store cert (User has to trust Provisioning profile!)



Source: paloaltonetworks.com

Source: paloaltonetworks.com

XcodeGhost (2015)

- Maliciously modified version of the Xcode compiler
- Added backdoors to apps during compilation
- Particularly wide-spread in Chinese applications

- Infected applications could be remotely controlled
  - Steal device information
  - Hijack opening of URLs

- Affected more than 128 million users
  - According to Apple's estimation

Source: macrumors.com

Source: paloaltonetworks.com

Pegasus (2016-now)

- Spyware exploits zero-click vulnerabilities for essentially jailbreaking device
  - Location tracking
  - Application monitoring
  - Intercepting messages
  - Recording calls

- Sold by NSO Group to nation state actors for surveiling suspects
  - Also used by some authoritarian governments against political opponents

- Supports very recent iOS versions (up to iOS 14.8)

Sources: googleprojectzero.blogspot.com, vice.com, amnesty.org

Jailbreak

All third-party applications on iOS are jailed

- Must be signed by registered developer or Apple
- Restricted to very few syscalls
- Can only access its own data container

We want to use the device to its full potential

- Run arbitrary unsigned apps
- Use all syscalls, access full file system, …
- Example: Run Emulator with JIT

How?

- We sneak out of the jail and open the doors for others to escape

Source: J. Levin: "*OS Internals"

Jailbreak Variants

- **Untethered Jailbreak**
  - Persists across reboots
  - Hardest to achieve
- **Tethered Jailbreak**
  - Requires USB connection to host for rebooting
  - Jailbreak is accomplished by manipulating the USB stack of BootROM or iBoot
- **Semitethered Jailbreak**
  - Manually run app on device after reboot
  - Bootstrap re-jailbreaking from a normal sandboxed app

Source: J. Levin: "*OS Internals"

Jailbreaking: General procedure

1. Run code on device
   - Install enterprise app **or** exploit built-in app **or** exploit Lockdown (iTunes) services
2. Bypass code signing
   - Run any code we need
3. Escape Sandbox
   - Execute arbitrary syscalls, access full file system
   - Exploit unprotected built-in service or allowed kernel interface
4. Elevate privileges
   - Obtain root acess to modify system files
5. Kernel patching
   - Disable AMFI and Sandbox for other processes

Source: J. Levin: "*OS Internals"

From code execution to kernel

- Usually involves exploiting multiple vulnerabilities
  - In built-in services or kernel interfaces
- Hindered by code signing!
  - Use Return Oriented Programming (ROP) to chain gadgets of existing functions
- Additional challenge posed by Pointer Authentication (Apple A12+)
  - Pointers are signed to prevent modifications

Sources: J. Levin: "*OS Internals", googleprojectzero.blogspot.com

Kernel Patching

Kernel Address Space Layout Randomization (KASLR) iOS 6+

Problem: Kernel loaded at different random offsets for each boot

Solution: Find patch targets by scanning kernel memory

- Look for unique instruction sequences or strings

Kernel Patch Protection (KPP) iOS 9+

Problem: Program in protection level EL3 checks for kernel modifications

Solution: Quickly patch and unpatch between checks

- Obtain task port for kernel_task (tfp0)

Kernel Text Readonly Region (KTRR) A10 / iPhone 7+

Problem: Modern chips catch write attempts to protected kernel pages in HW

Solution: Attack before KTRR is set up (iBoot) or find r/w kernel struct

Sources: J. Levin: "*OS Internals", blog.siguza.net

Full Jailbreak Writeup

- Full jailbreaks are complex to find and take years of experience
  - The more countermeasures, the harder it gets

- For the interested: Have a look at the early modern jailbreaks
  - Evasi0n:
    - iOS 6 Jailbreak (2013)
    - The first to deal with KASLR
    - Source Code Released in 2017 Source: github.com
    - Writeups for User Space Source: www.accuvant.com
    - And Kernel Patches Source: blog.azimuthsecurity.com

iOS App Analysis

IAIK TU Graz

Application Analysis

→ Traditionally two approaches

- Dynamic Analysis: Monitor live file access using jailbroken device
- Static Analysis: Look for file API calls + parameters in binary dump
  - Still needs jailbroken device to obtain decrypted application binary

Challenge?

- iOS apps are compiled down to native code
  - Analysis on disassembly, e.g. using Ghidra or Hopper
  - Compilation removes high-level information
  - Still, the dynamic nature of Objective-C is helpful here!
    - Swift is a little more difficult to reverse!

Case Study: Viber

Source: apps.apple.com

Objective-C Selectors Visible!

```
-[VIBEncryptionContext initWithContext:]
-[VIBEncryptionContext context]
-[VIBEncryptionContext params]
-[VIBEncryptionContext setParams:]
-[VIBEncryptionContext .cxx_destruct]
-[VIBEncryptionManager initWithInjector:]
-[VIBEncryptionManager dealloc]
-[VIBEncryptionManager checkEncryptionAbilityForAttachment:completion:]
-[VIBEncryptionManager checkEncryptionForConversation:completion:]
-[VIBEncryptionManager beginEncryptionWithContext:]
-[VIBEncryptionManager encryptData:length:withContext:]
-[VIBEncryptionManager endEncryptionWithContext:]
-[VIBEncryptionManager popEncryptionParamsForContext:]
-[VIBEncryptionManager encryptData:encryptionKey:]
-[VIBEncryptionManager calculateMD5ForAttachment:]
-[VIBEncryptionManager decryptAttachment:completion:]
-[VIBEncryptionManager decryptData:withEncryptionParams:]
-[VIBEncryptionManager decryptFile:withEncryptionParams:]
-[VIBEncryptionManager handleSecureStateChanged:]
-[VIBEncryptionManager supportedMediaTypes]
-[VIBEncryptionManager .cxx_destruct]
```

Case Study: Viber

```
000632fa        str         r4, [sp, #0x100 + var_100]
000632fc        movw        r2, #0x412e                             ; @"Viber can not verify this number. This may be the result of an error or a breach.\\nPlease verify %@ agai
00063300        movt        r2, #0xd9                               ; @"Viber can not verify this number. This may be the result of an error or a breach.\\nPlease verify %@ agai
00063304        mov         r1, r6                                  ; argument #2 for method imp___picsymbolstub4__objc_msgSend
00063306        add         r2, pc                                  ; @"Viber can not verify this number. This may be the result of an error or a breach.\\nPlease verify %@ agai
00063308        mov         r3, r8
0006330a        mov
0006330c        blx         imp___picsymbolstub4__objc_msgSend
00063310        mov
00063312        blx         imp___picsymbolstub4__objc_retainAutoreleasedReturn
00063316        str         r0, [sp, #0x100 + var_C8]
00063318        mov         r0, r5
0006331a        blx         imp___picsymbolstub4__objc_release
0006331e        ldr.w       r0, [fp]                                ; objc_cls_ref_NSBundle,_OBJC_CLASS_$_NSBundle, argument #1 for method imp___picsymbolstub4__objc_msgSend
00063322        mov         r1, sl
00063324        blx         imp___picsymbolstub4__objc_msgSend
00063328        mov         r7, r7
0006332a        blx         imp___picsymbolstub4__objc_retainAutoreleasedReturnValue
0006332e        str         r4, [sp, #0x100 + var_100]
00063330        movw        r2, #0x410a                             ; @"Messages sent by participants in this conversation are encrypted and %@ is Verified", :lower16:(cfstring_
00063334        movt        r2, #0xd9                               ; @"Messages sent by participants in this conversation are encrypted and %@ is Verified", :upper16:(cfstring_
00063338        mov         r1, r6                                  ; argument #2 for method imp___picsymbolstub4__objc_msgSend
0006333a        add         r2, pc                                  ; @"Messages sent by participants in this conversation are encrypted and %@ is Verified"
0006333c        mov         r3, r8
0006333e        mov         r5, r0
00063340        blx         imp___picsymbolstub4__objc_msgSend
00063344        mov         r7, r7
00063346        blx         imp___picsymbolstub4__objc_retainAutoreleasedReturnValue
0006334a        str         r0, [sp, #0x100 + var_B8]
0006334c        mov         r0, r5
0006334e        blx         imp___picsymbolstub4__objc_release
00063352        ldr.w       r0, [fp]                                ; objc_cls_ref_NSBundle,_OBJC_CLASS_$_NSBundle, argument #1 for method imp___picsymbolstub4__objc_msgSend
00063356        mov         r1, sl
00063358        blx         imp___picsymbolstub4__objc_msgSend
0006335c        mov         r7, r7
0006335e        blx         imp___picsymbolstub4__objc_retainAutoreleasedReturnValue
00063362        str         r4, [sp, #0x100 + var_100]
00063364        movw        r2, #0x40e6                             ; @"This conversation cannot be encrypted. This may be the result of an error or a geo-location limitation",
00063368        movt        r2, #0xd9                               ; @"This conversation cannot be encrypted. This may be the result of an error or a geo-location limitation",
0006336c        mov         r1, r6                                  ; argument #2 for method imp___picsymbolstub4__objc_msgSend
0006336e        add         r2, pc                                  ; @"This conversation cannot be encrypted. This may be the result of an error or a geo-location limitation"
00063370        mov         r3, r8
00063372        mov         r5, r0
```

Method calls have to go through objc_msgSend

Facilitates reverse-engineering

Outlook

- 29.04.2022
  - Android Platform Security

- 06.05.2022
  - Application Security on Android