Mobile applications security (Android and iOS)

Philippe BITON
pbiton@prox-ia.com
Twitter: @proxia

Mobile Security Newsletter – December 2018

Apple

Widespread Apple ID Phishing Attack Pretends to be App Store Receipts

A widespread and sneaky phishing campaign is underway that pretends to be a purchase confirmation from the Apple App store.

Company Says Its Software Allows Anyone to Hack into an iPhone

A company called DriveSavers, which is specialized in data recovery services, promises that anyone can hack into an iPhone, even if it's protected with a super-long passcode.

Related:

•theverge.com DriveSavers claims it has a way to break into locked iPhones with 100 percent success

Google

Google Patches 11 Critical RCE Android Vulnerabilities

Remote code-execution (RCE) vulnerabilities dominated Google's December Android Security Bulletin.

Google Beefs Up Android Key Security for Mobile Apps

Changes to how data is encrypted can help developers ward off data leakage and exfiltration.

We Broke Into A Bunch Of Android Phones With A 3D-Printed Head

We tested four of the hottest handsets running Google's operating systems and Apple's iPhone to see how easy it'd be to break into them. We did it with a 3D-printed head. All of the Androids opened with the fake. Apple's phone, however, was impenetrable.

Android Clickfraud Op Impersonates iPhones to Bump Ad Premiums

A mobile clickfraud campaign used 22 Android apps to trick online advertisers into paying the higher price for advertising on iPhone 5 to 8 Plus devices.

Cryptography

Crypto basics ?

Symmetric key encryption
Asymmetric key encryption
Hashing
Message Authentication Code (HMAC-AES256)
Signature
Key Derivation Functions

Security testing

Defensive not offensive

Attack path

Static Analysis (false positive from tools)
Dynamic Analysis (false positive from tools)
Penetration testing (White box, Black box, Gray box)

- **Preparation** - defining the scope of security testing
- **Intelligence Gathering** - analyzing the **environmental** and **architectural** context of the app
- **Mapping the Application** - automated scanning and manually exploring the app.
- **Exploitation** - exploiting the vulnerabilities
- **Reporting** - essential to the client

During SDLC (Waterfall verus Agile)

Attack Path

Network Communication

Certificate

TLS

VM network config

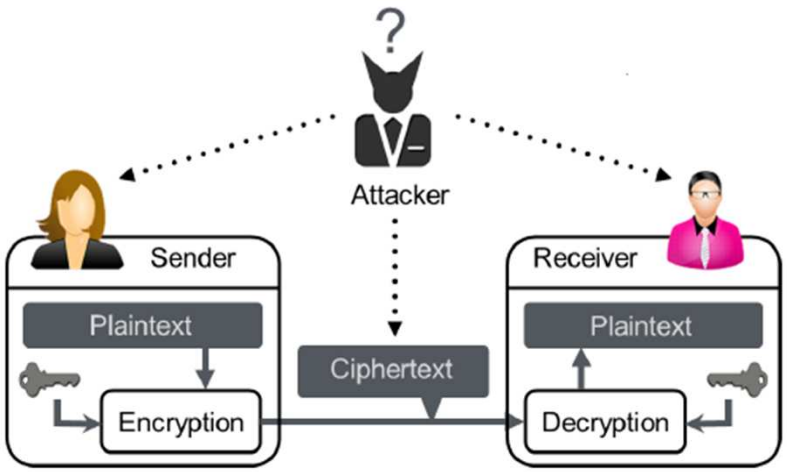
Intercepting network traffic

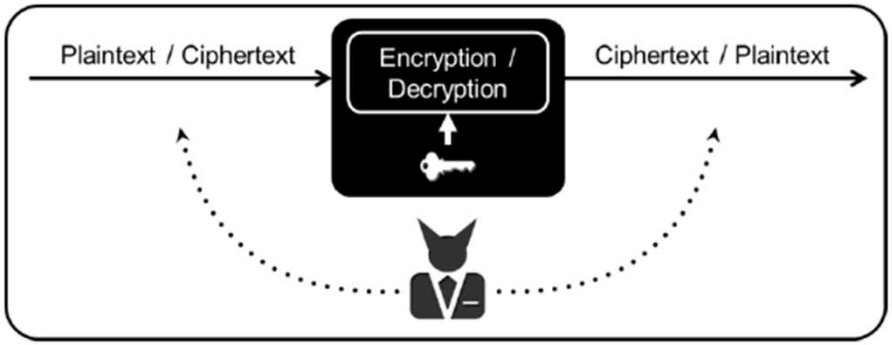
- Burp Suite (HTTPS)
- Wireshark
- tcpdump

MITM

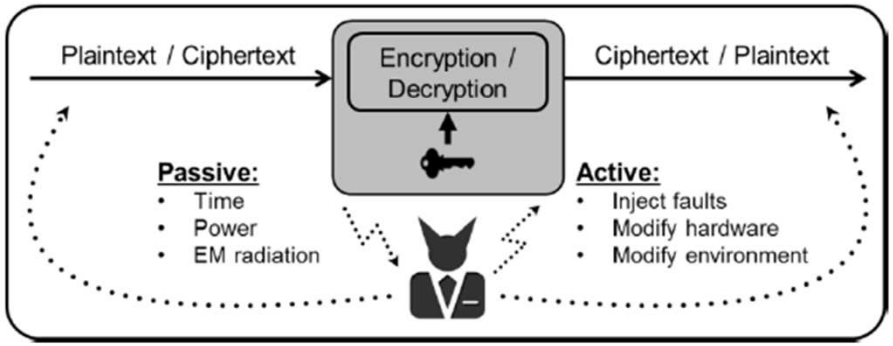
- ettercap (ARP poisoning)
- Rogue Wifi AP
- proxy via runtime instrumentation (rooted or jailbroken device) using hooking tools like Inspeckage or code injection frameworks like frida and cycript

Typology of attacks

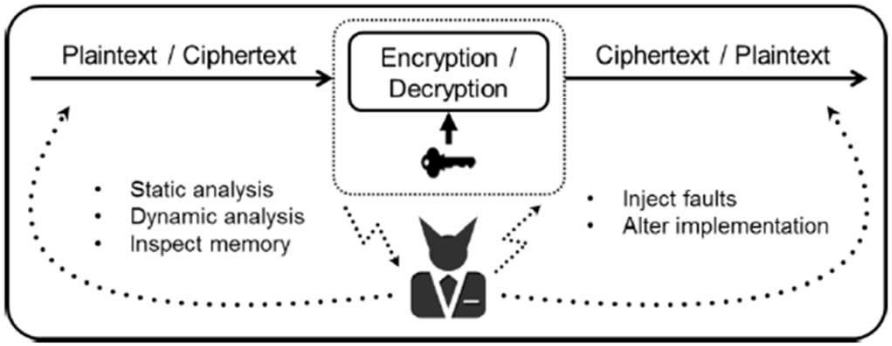




Endpoints are trusted parties
Attacker "observes" data being transferred



Hardware implementations tend to leak key-correlated information



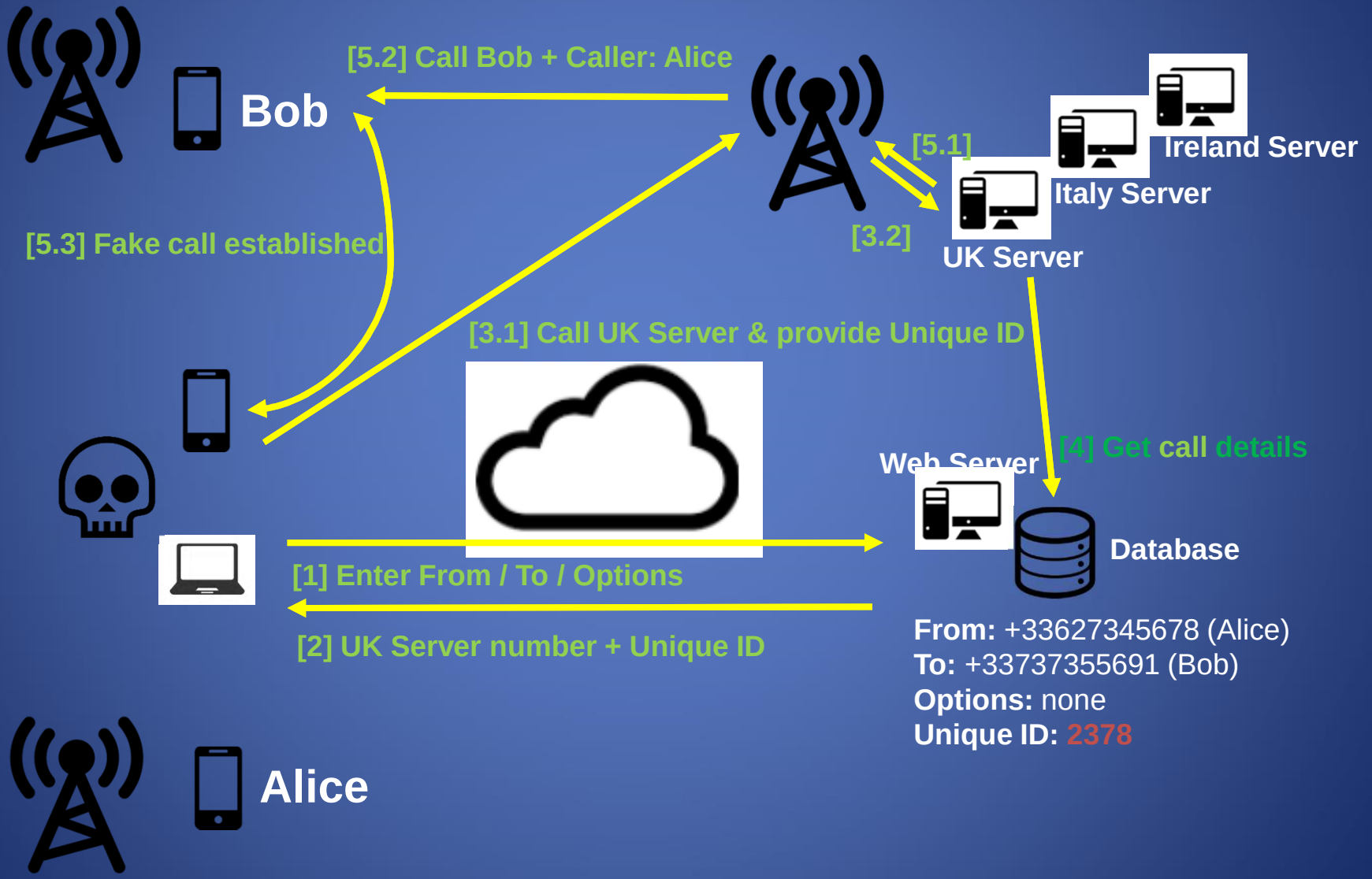
Adversary owns the device running the application

Typology of attacks

Top 10 2016

Source: https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10

Typology of attacks



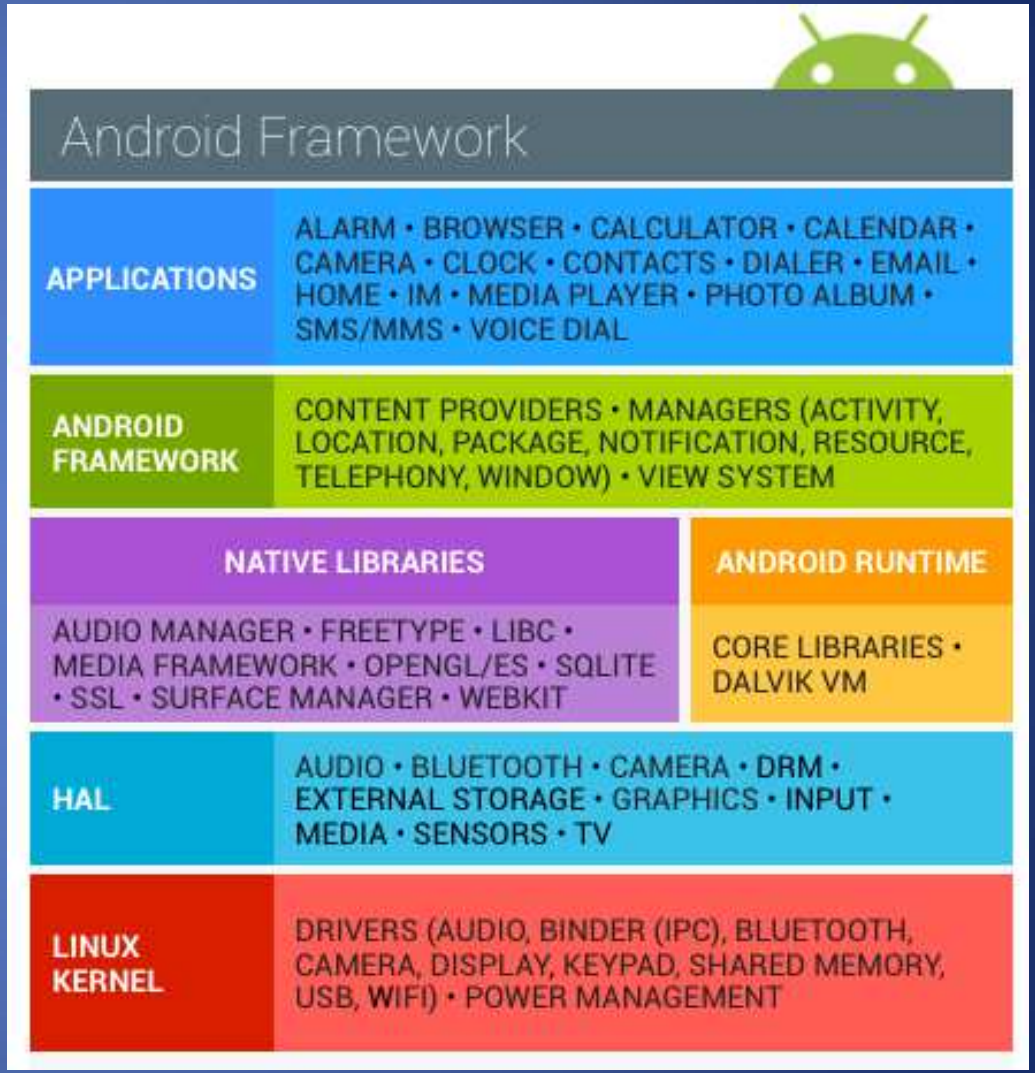
Typology of attacks

Android Security

1. Platform Overview
2. Testing Data Storage on Android
3. Android Cryptographic APIs
4. Local Authentication on Android
5. Android Network APIs
6. Android Platform APIs
7. Code Quality and Build Settings for Android Apps
8. Tampering and Reverse Engineering on Android
9. Android Anti-Reversing Defenses

Android Security: Platorm Overview

Android security architecture



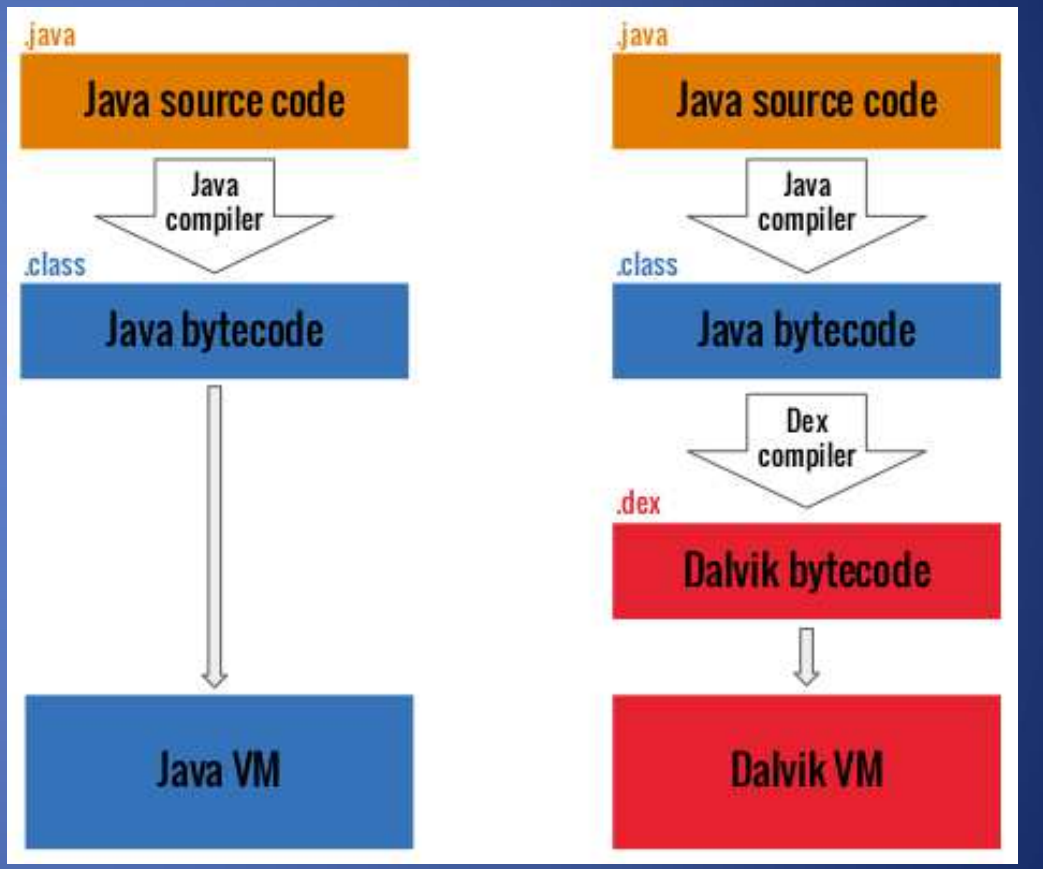
HAL : hardware Abstraction layer (standard interface for interacting with built-in hardware components) - Shared library modules that the Android system calls

Android Security: Platorm Overview

Android security architecture

ART is the successor to Android's original runtime, the Dalvik VM. **Dalvik** bytecode is translated into machine code at execution time, a process known as ***just-in-time*** (JIT) compilation. JIT compilation adversely affects performance: the compilation must be performed every time the app is executed.

ART introduced ***ahead-of-time*** (AOT) compilation. **Apps are precompiled before they are executed for the 1st time**. This precompiled machine code is used for all subsequent executions. AOT improves performance by a factor of two while reducing power consumption.



Android Security: Platorm Overview

Android security architecture

Android users and groups

With a few exceptions, each app runs as though under a separate Linux user, effectively isolated from other apps and the rest of the operating system.

For example, Android Nougat defines the following system users (system/core/include/private/android_filesystem_config.h):

```
#define AID_ROOT 0 /* traditional unix root user */
#define AID_SYSTEM 1000 /* system server */ ...
#define AID_SHELL 2000 /* adb and debug shell user */ ...
#define AID_APP 10000 /* first app user */
```

Android Security: Platorm Overview

Inter-Process Communication (IPC)

Android apps interact with system services via the Android Framework, an abstraction layer that offers high-level Java APIs.

The majority of these services are invoked via normal Java method calls and are translated to IPC calls to system services that are running in the background

Examples of **system services** include:

- Connectivity (Wi-Fi, Bluetooth, NFC, etc.)
- Cameras
- Geolocation (GPS)
- Microphone

Android Security: Platorm Overview

Inter-Process Communication (IPC)

The oldest Android version supported at the time of writing is 4.4 (KitKat), API level 19, and the current Android version is 7.1 (Nougat), API level 25.

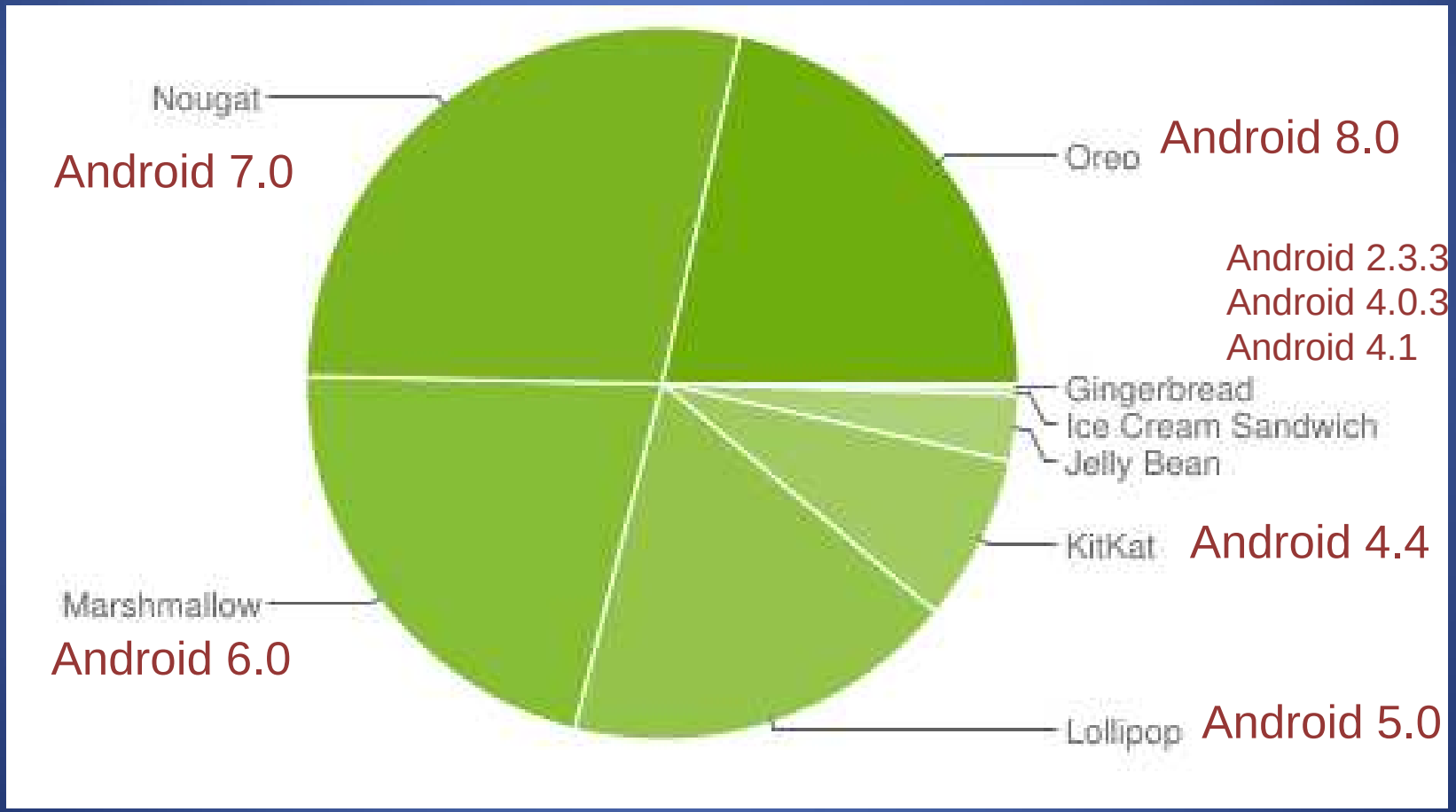
API versions:

- ...
- Android 4.2 **Jelly Bean** (API 16) in November 2012 (introduction of SELinux)
- Android 4.3 Jelly Bean (API 18) in July 2013 (SELinux became enabled by default)
- Android 4.4 **KitKat** (API 19) in October 2013 (several new APIs and **ART** introduced)
- Android 5.0 **Lollipop** (API 21) in November 2014 (**ART used by default** and many other features added)
- Android 6.0 **Marshmallow** (API 23) in October 2015 (many new features and improvements, including granting; **detailed permissions setup at run time rather than all or nothing during installation**)
- Android 7.0 **Nougat** (API 24-25) in August 2016 (new JIT compiler on **ART**)
- Android 8.0 **Oreo** (major security fixes)
- Android 9.0 **Pie** released on **August 6, 2018** and was initially available for Google Pixel devices and the Essential Phone. The Sony Xperia XZ3 was the first device with Android Pie pre-installed.

Android Security: Platorm Overview

Inter-Process Communication (IPC)

https://developer.android.com/about/dashboards/



Data collected during a 7-day period ending on October 26, 2018

Android Security: Platorm Overview

Android application structure

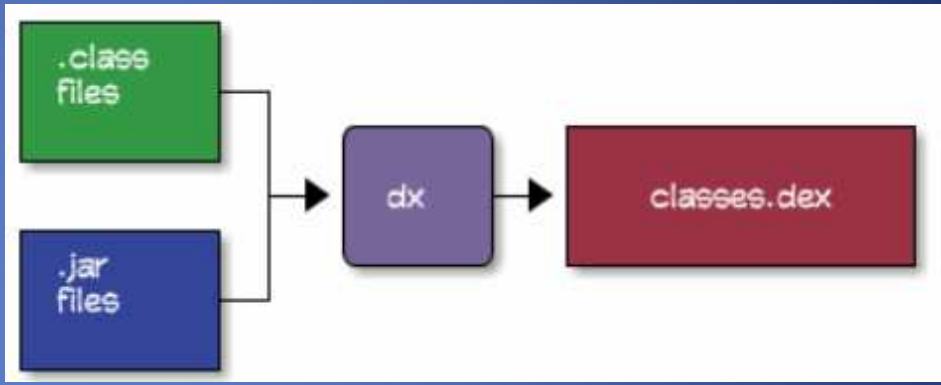
```
$ unzip base.apk
$ ls -lah
-rw-r--r-- 1 sven staff 11K Dec 5 14:45 AndroidManifest.xml
drwxr-xr-x 5 sven staff 170B Dec 5 16:18 META-INF
drwxr-xr-x 6 sven staff 204B Dec 5 16:17 assets
-rw-r--r-- 1 sven staff 3.5M Dec 5 14:41 classes.dex
drwxr-xr-x 3 sven staff 102B Dec 5 16:18 lib
drwxr-xr-x 27 sven staff 918B Dec 5 16:17 res
-rw-r--r-- 1 sven staff 241K Dec 5 14:45 resources.arsc
```

Android Security: Platorm Overview

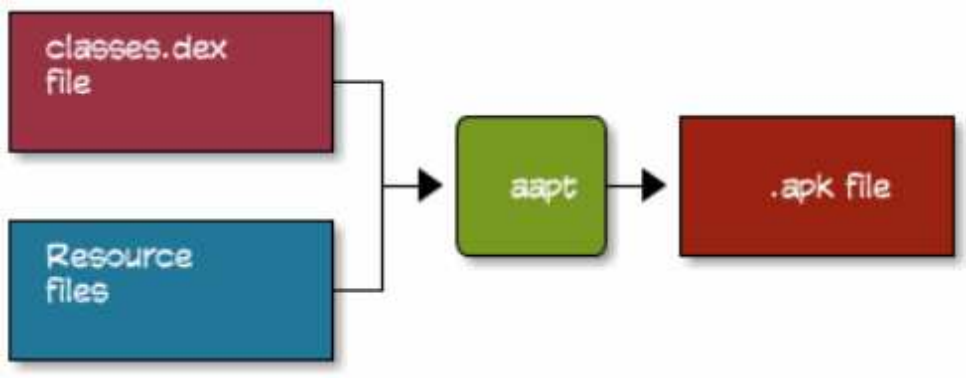
Android application build

Java compilation

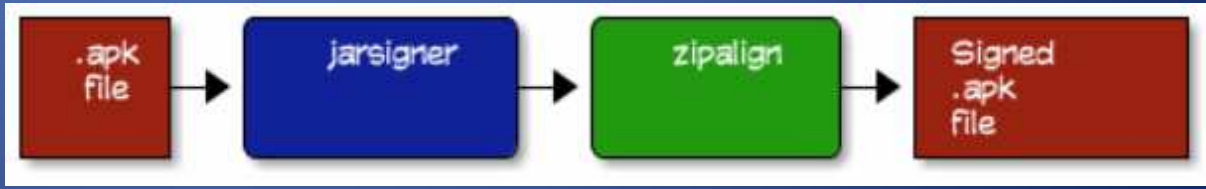
Conversion to Dalvik bytecodes



Put classes.dex and resources into a package file



Sign the .apk file



https://github.com/dogriffiths/HeadFirstAndroid/wiki/How-Android-Apps-are-Built-and-Run

Android Security: Platorm Overview

Android application installation

Following are the step by step installation process:

1) “AndroidManifest.xml” is parsed, information is extracted and stored into “/data/system/packages.xml” and “/data/system/packages.list”

2) XML parsing, resource analysis and “.apk” file copying are done by "PackageManageService.java“. However, directory creation is done by "installd.c“

3) “PackageManageService.java” communicates with “installd.c” via a local socket, located at “/dev/socket/installed”

4) Package where “.apk” file got copied is different for system apps and user apps. for system apps it is "/system/app/“. Where as for user app “.apk” file is copied to "/data/app“

5) “.dex” file, which is extracted from the “.apk” file, is copied to “/data/dalvik-cache/”.

6) “Package Manager” creates data directory “/data/data//” to store database, shared preferences, native library and cache data. A directory for data storage is created for this app.

Android Security: Platorm Overview

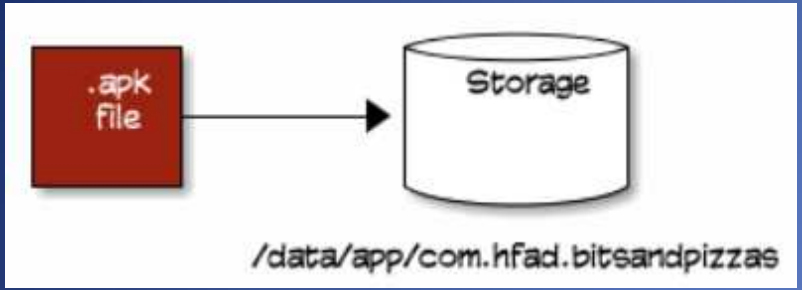
Android application structure

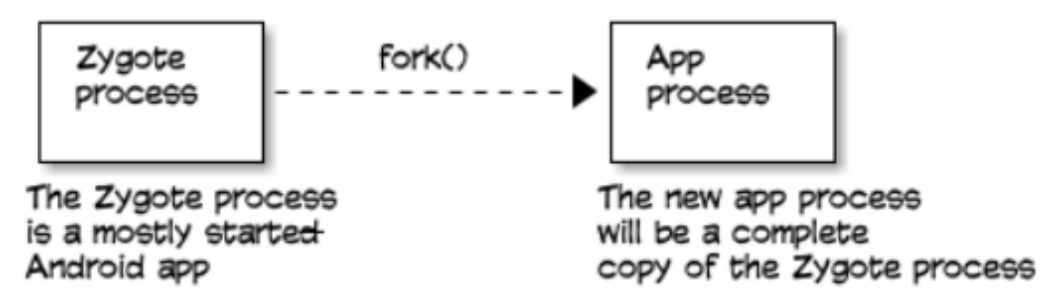
Zygote

- The process Zygote starts up during Android initialization.
- Zygote is a system service for launching apps.
- The Zygote process is a "base" process that contains all the core libraries the app needs.
- Upon launch, Zygote opens the socket /dev/socket/zygote and listens for connections from local clients.
- When it receives a connection, it forks a new process, which then loads and executes the app-specific code

Android Security: Platorm Overview

Android application launch





OAT is a file format produced by compiling a DEX file with AOT compilation.

Android Security: Platorm Overview

How ART works (1/2)

ART uses ahead-of-time (AOT) compilation, and starting in Android 7.0 (Nougat), it uses a hybrid combination of AOT, just-in-time (JIT) compilation, and profile-guided compilation.
As an example, Pixel devices are configured with the following compilation flow:

1) An application is initially installed without any AOT compilation. The first few times the application runs, it will be interpreted, and methods frequently executed will be JIT compiled.
2) When the device is idle and charging, a compilation daemon runs to AOT-compile frequently used code based on a profile generated during the first runs.
3) The next restart of an application will use the profile-guided code and avoid doing JIT compilation at runtime for methods already compiled. Methods that get JIT-compiled during the new runs will be added to the profile, which will then be picked up by the compilation daemon.

https://source.android.com/devices/tech/dalvik/configure

Android Security: Platorm Overview

How ART works (2/2)

ART comprises a compiler (the dex2oat tool) and a runtime (libart.so) that is loaded for starting the Zygote. The dex2oat tool takes an APK file and generates one or more compilation artifact files that the runtime loads. The number of files, their extensions, and names are subject to change across releases, but as of the Android O release, the files being generated are:

- .vdex: contains the uncompressed DEX code of the APK, with some additional metadata to speed up verification.
- .odex: contains AOT compiled code for methods in the APK.
- .art (optional): contains ART internal representations of some strings and classes listed in the APK, used to speed application startup.

https://source.android.com/devices/tech/dalvik/configure

Android Security: Platorm Overview

Android application structure

- **AndroidManifest.xml**: app's package name, user-granted permissions, etc.
- **META-INF**: contains the app's metadata
  - MANIFEST.MF: stores hashes of the app resources
  - CERT.RSA: the app's certificate(s)
  - CERT.SF: list of resources and the SHA-1 digest of the corresponding lines in the MANIFEST.MF file
- **assets**: allows the developer to place files in this directory that he would like bundled with the application
- **classes.dex**: classes compiled in the **DEX (Dalvik EXecutable)** file format, the Dalvik virtual machine/Android Runtime can process.
- **lib**: directory containing libraries that are part of the APK, for example, the third-party libraries that aren't part of the Android SDK
- **res**: contains all the application activity layouts, images used, and any other files that the developer would like accessed from code in a structured way. These files are placed in the raw/ subdirectory.
- **resources.arsc**: resources can be compiled into this file instead of being put into the “res” folder. Also contains any application strings

Android Security: Platorm Overview

Android application structure

```
$ cd base
$ ls -alh
total 32
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 .
drwxr-xr-x 5 sven staff 170B Dec 5 16:29 ..
-rw-r--r-- 1 sven staff 10K Dec 5 16:29 AndroidManifest.xml
-rw-r--r-- 1 sven staff 401B Dec 5 16:29 apktool.yml
drwxr-xr-x 6 sven staff 204B Dec 5 16:29 assets
drwxr-xr-x 3 sven staff 102B Dec 5 16:29 lib
drwxr-xr-x 4 sven staff 136B Dec 5 16:29 original
drwxr-xr-x 131 sven staff 4.3K Dec 5 16:29 res
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 smali
```

Android Security: Platorm Overview

Android application structure

- **AndroidManifest.xml**: The decoded Manifest file, which can be opened and edited in a text editor.
- **apktool.yml**: file containing information about the output of apktool
- **original**: folder containing the MANIFEST.MF file, which contains information about the files contained in the JAR file
- **res**: directory containing the app's resources
- **smali**: directory containing the disassembled Dalvik bytecode in Smali. Smali is a human-readable representation of the Dalvik executable.

Android Security: Platorm Overview

Android application structure

Android reverse engineering

- Android binaries stored in APK format
- Dalvik executables stored as « classes.dex »
  - Bytecode interpreted code
  - Dalvik can be reverse-engineered and recompiled into working examples
- « apktool » used to decompile an APK file to Dalvik bytecode (and recompile)

Java decompiling

- Convert Dalvik file into Java JAR file
  - « Dex to Jar » translates Dalvik files into JAR files
  - JAR file can be viewed with a Java decompiler
- Java decompiler gives readability to source code
  - JD-GUI
  - Jadx
- We cannot recompile from Java

Android App Manipulation

- Best option is to decompile app to an intermediate format
  - Preserves functionality
  - Does not provide readability

TP 0001

Using **fr.poulpage_com.apk**

- Get an apk from mobile
- Study the apk unzipped
- Decompile apk with apktool and study
- Install and uninstall apk

Check Runtime:

$ cd /data/property
$ vi persist.sys.dalvik.vm.lib
 # if libart.so => ART
 # if libdvm.so => Dalvik

Android Security: Platorm Overview

Android application structure

Every app also has a data directory for storing data created during run time.

This directory is at /data/data/[package-name] and has the following structure:

```
drwxrwx--x u0_a65 u0_a65 2016-01-06 03:26 cache
drwx------ u0_a65 u0_a65 2016-01-06 03:26 code_cache
drwxrwx--x u0_a65 u0_a65 2016-01-06 03:31 databases
drwxrwx--x u0_a65 u0_a65 2016-01-10 09:44 files
drwxr-xr-x system system 2016-01-06 03:26 lib
drwxrwx--x u0_a65 u0_a65 2016-01-10 09:44 shared_prefs
```

Android Security: Platorm Overview

Android application structure

cache: data caching. For example, the WebView cache is found in this directory.
code_cache: file system's application-specific cache directory designed for storing cached code.
databases: SQLite database files generated by the app at run time, e.g., user data files.
files: regular files created by the app.
lib: native libraries written in C/C++. These libraries can have one of several file extensions, including .so and .dll (x86 support). This folder contains subfolders for the platforms the app has native libraries for, including

- armeabi: compiled code for all ARM-based processors
- armeabi-v7a: compiled code for all ARM-based processors, version 7 and above only
- arm64-v8a: compiled code for all 64-bit ARM-based processors, version 8 and above based only
- x86: compiled code for x86 processors only
- x86_64: compiled code for x86_64 processors only
- mips: compiled code for MIPS processors

shared_prefs: This folder contains an XML file that stores values saved via the SharedPreferences APIs

Android Security: Platorm Overview

Android application structure

// Normal mobile phone
$ id
uid=**10188**(u0_a188) gid=10188(u0_a188) groups=10188(u0_a188),
3003(**inet**), 9997(everybody), 50188(all_a188)
context=u:r:untrusted_app:s0:c512,c768

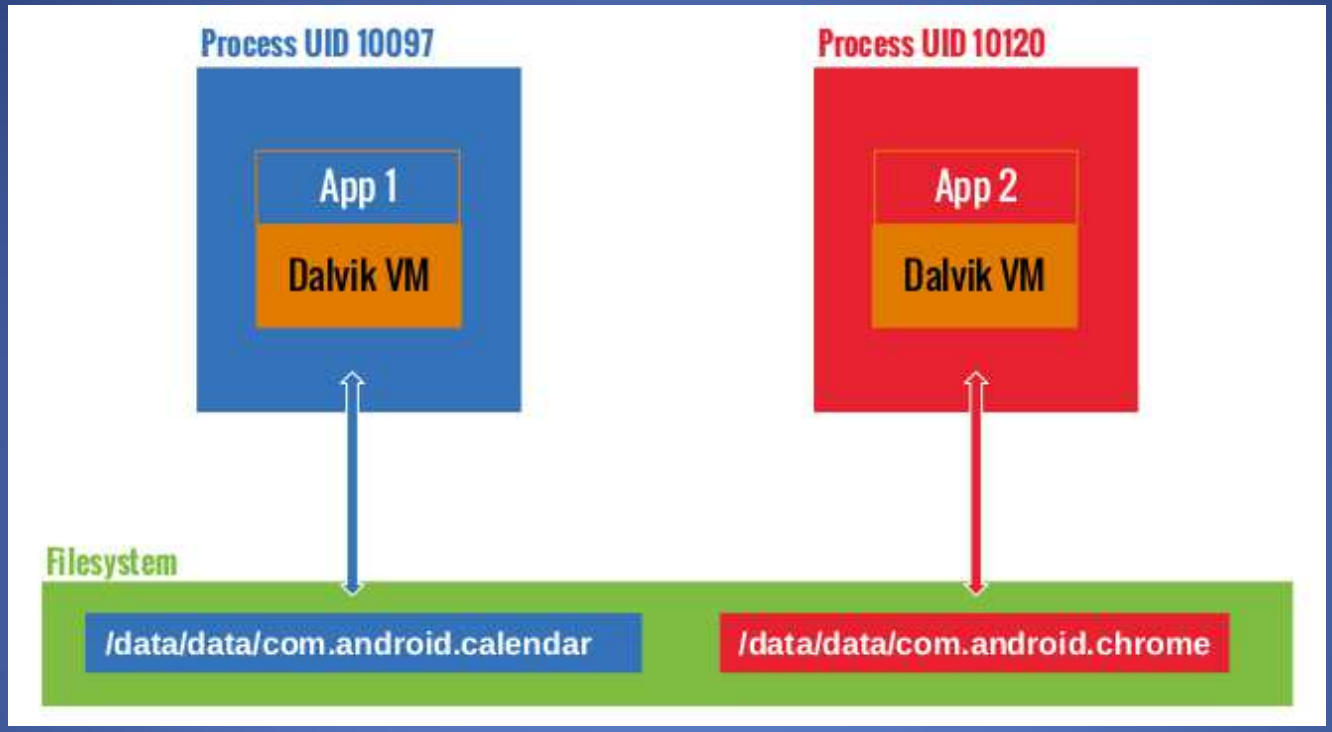
// adb and debug shell user
$ id
uid=**2000**(shell) gid=2000(shell) groups=1003(graphics), 1004(input),
1007(**log**), 1011(adb), 1015(sdcard_rw), 1028(sdcard_r),
3001(net_bt_admin), 3002(net_bt), 3003(inet), 3006(net_bw_stats)

// Traditional unix root user
$ id
uid=**0**(root) gid=0(root)

Look at **/system/etc/permissions/platform.xml** and search **inet** or **log**

Android Security: Platorm Overview

Android application structure



Application Sandboxing separates the app data and code execution from other apps

Installation of a new app creates a new directory named '/data/data/[package-name]'

$ cd /data/data/

$ ls

drwx------ 4 u0_a97 u0_a97 4096 2017-01-18 14:27 com.android.calendar

drwx------ 6 u0_a120 u0_a120 4096 2017-01-19 12:54 com.android.chrome

TP 0002

Using **com.Slack.apk**

- Understand the id (user and group)
- Connect to device with a shell using adb
- Study the system and app file structure
- Get disclosed data with slack

Android Security: Platorm Overview

Android application structure

- **Activities**—Activities represent visual screens of an application with which users interact. For example, when you launch an application, you see its main activity
- **Services—**Services are components that do not provide a graphical interface. They provide the facility to perform tasks that are long running in the background and continue to work even when the user has opened another application or has closed all activities of the application that contains the service

  $ adb shell service list
- **Broadcast receivers—**Broadcast receivers are non-graphical components that allow an application to register for certain system or application events. For instance, an application that requires a notification when receiving an SMS would register for this event using a broadcast receiver. This allows a piece of code from an application to be executed only when a certain event takes place.
- **Content providers—**These are the data storehouses of an application that provide a standard way to retrieve, modify, and delete data. The terminology used to define and interact with a content provider is similar to SQL: query, insert, update, and delete.

TP 0013

Using **sieve.apk**

- Find vulnerabilities in smali
- Find vulnerabilites in java
- Check apk permissions with aapt
- Explore permissions in AndroidManifest.xml
- Exploit vulnerability in Content provider
- Exploit SQL injection in Content provider with projection
- Exploit directory traversal in Content provider
- Exploit vulnerability on Activity

M4-Insecure Authentication

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: Threat agents that exploit authentication vulnerabilities typically do so through automated attacks that use available or custom-built tools.

Exploitability: Once the adversary understands how the authentication scheme is vulnerable, they fake or bypass authentication by submitting service requests to the mobile app's backend server and bypass any direct interaction with the mobile app. This submission process is typically done via mobile malware within the device or botnets owned by the attacker.

Frequency: Weaker authentication for mobile apps is fairly prevalent due to a mobile device's input form factor. The form factor highly encourages short passwords that are often purely based on 4-digit PINs.

M4-Insecure Authentication

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Detectability: In traditional web apps, users are expected to be online and authenticate in real-time with a backend server. In mobile apps, users are not expected to be online at all times during their session. Mobile apps may have uptime requirements that require offline authentication.

To detect poor authentication schemes, testers can perform binary attacks against the mobile app while it is in 'offline' mode. Through the attack, the tester will force the app to bypass offline authentication and then execute functionality that should require offline authentication

As well, testers should try to execute any backend server functionality anonymously by removing any session tokens from any POST/GET requests for the mobile app functionality.

M4-Insecure Authentication

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Technical Impact: The technical impact of poor authentication is that the solution is unable to identify the user performing an action request. Immediately, the solution will be unable to log or audit user activity because the identity of the user cannot be established.

This will contribute to an inability to detect the source of an attack, the nature of any underlying exploits, or how to prevent future attacks.

Authentication failures may expose underlying authorization failures as well. If an attacker is able to anonymously execute sensitive functionality, it may highlight that the underlying code is not verifying the permissions of the user issuing the request for the action.

Business Impact: The business impact of poor authentication will typically result in the following at a minimum:

- Reputational Damage / Information Theft
- Unauthorized Access to Data.

M4-Insecure Authentication

Am I Vulnerable ?

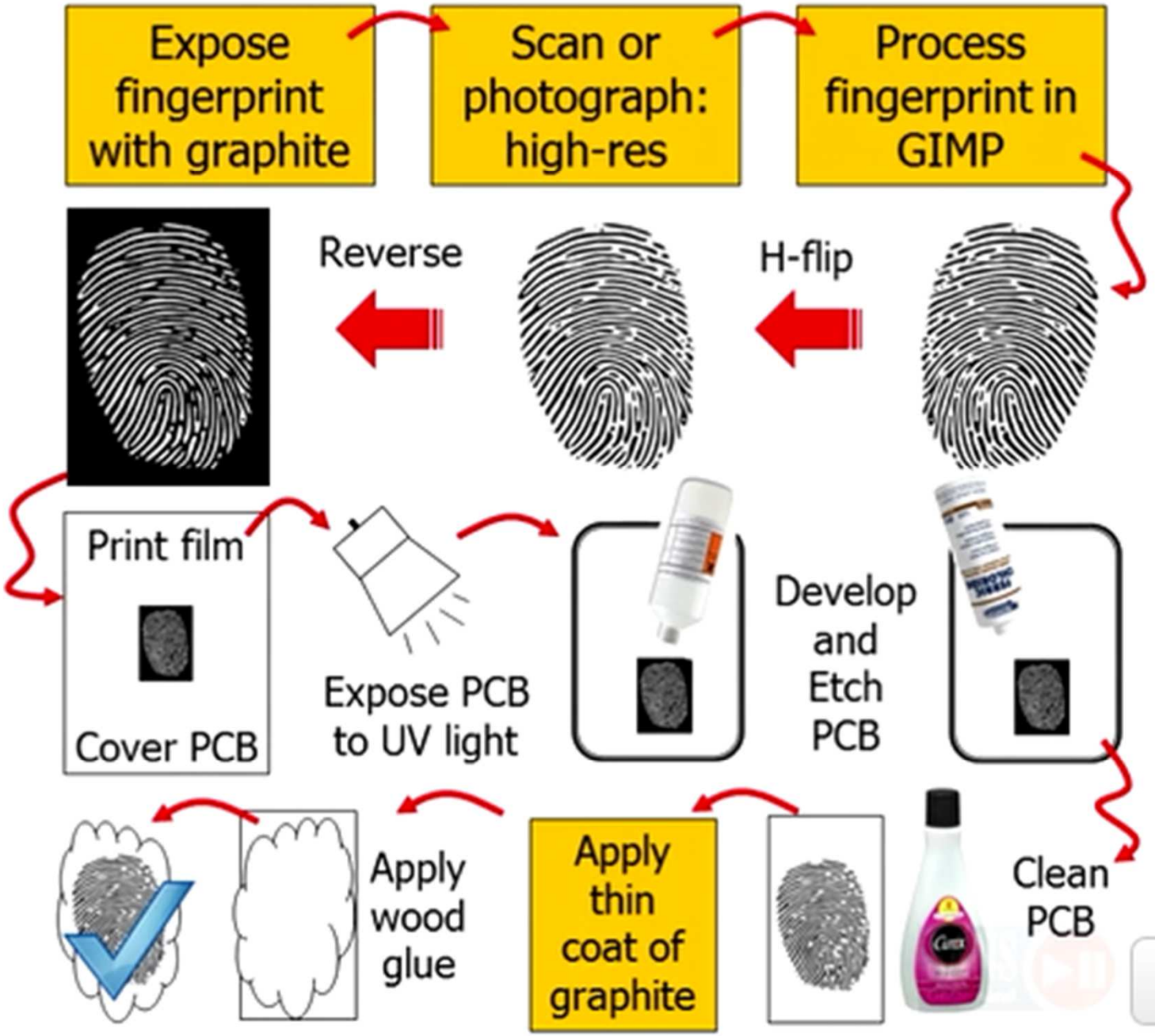
There are many different ways that a mobile app may suffer from insecure authentication:

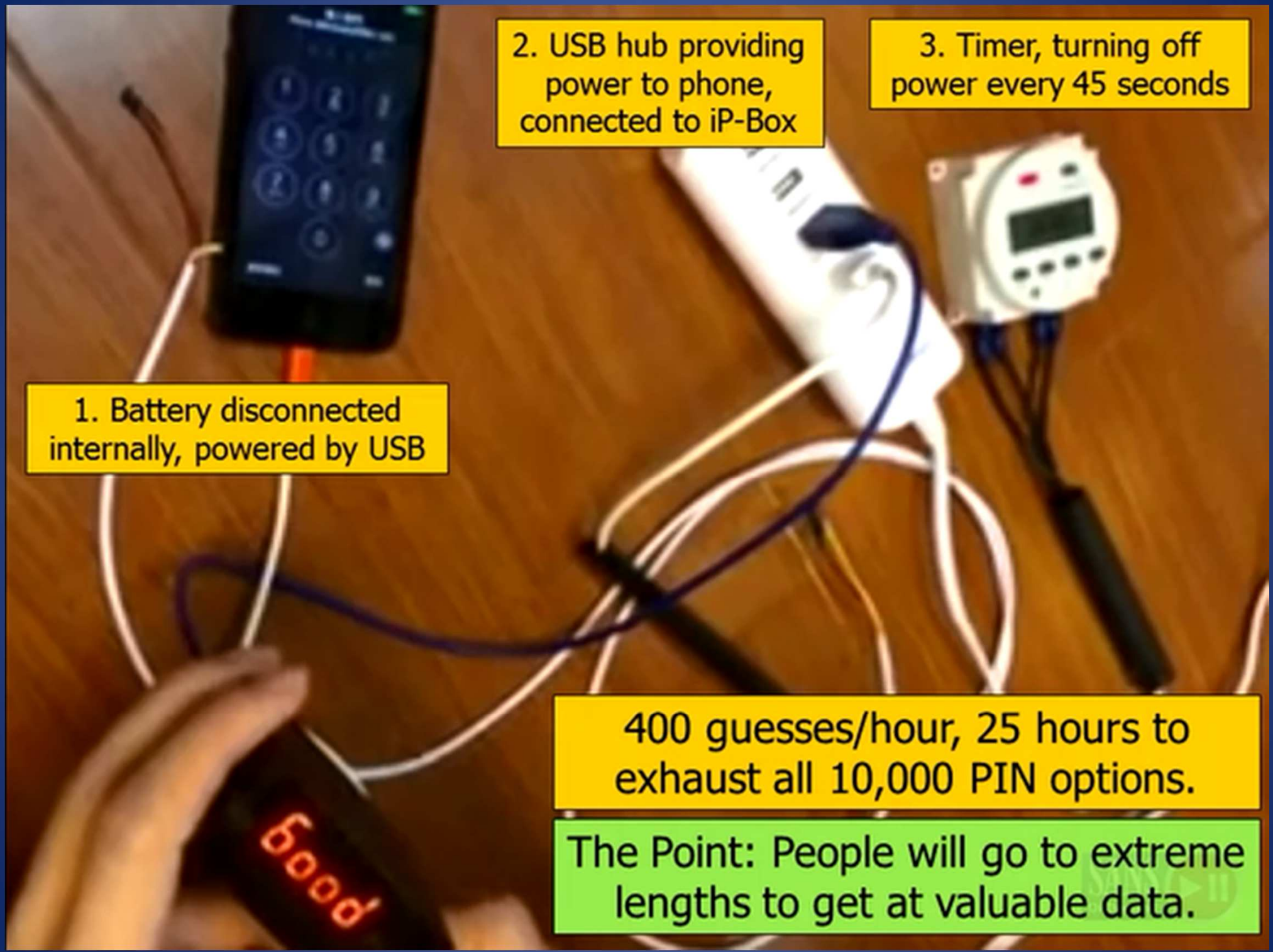
- If the mobile app is able to anonymously execute a backend API service request without providing an access token, this application suffers from insecure authentication
- If the mobile app stores any passwords or shared secrets locally on the device, it most likely suffers from insecure authentication
- If the mobile app uses a weak password policy to simplify entering a password, it suffers from insecure authentication
- If the mobile app uses a feature like TouchID, it suffers from insecure authentication.

Hacking Touch ID

Materials

- Fingerprint: phone screen or other hard, smooth surface
- Graphite powder or superglue
- Scanner or high-resolution picture
- GIMP (or Photoshop)
- Transparent film and a high-res laser printer
- Photo sensitive PCB material
- UV light box
- PCB developer solution
- Ferric chloride
- Nail polish remover
- Wood glue




2. USB hub providing power to phone, connected to iP-Box
3. Timer, turning off power every 45 seconds
1. Battery disconnected internally, powered by USB
400 guesses/hour, 25 hours to exhaust all 10,000 PIN options.
The Point: People will go to extreme lengths to get at valuable data.

M4-Insecure Authentication

How Do I Prevent?

Avoid Weak Patterns

- It should not be possible to authenticate with less authentication factors than the web browser
- Ensure that application data will only be available after successful authentication
- If client-side storage of data is required, the data will need to be encrypted using an encryption key that is securely derived from the user's login credentials.
- Persistent authentication (Remember Me) functionality implemented within mobile applications should never store a user's password on the device
- If possible, do not allow users to provide 4-digit PIN numbers for authentication passwords

Reinforce Authentication

- Authorization and authentication controls must be re-enforced on the server-side whenever possible.
- Due to offline usage requirements, mobile apps may be required to perform local authentication or authorization checks within the mobile app's code. If this is the case, developers should instrument local integrity checks within their code to detect any unauthorized code changes.

M4-Insecure Authentication

How Do I Prevent?

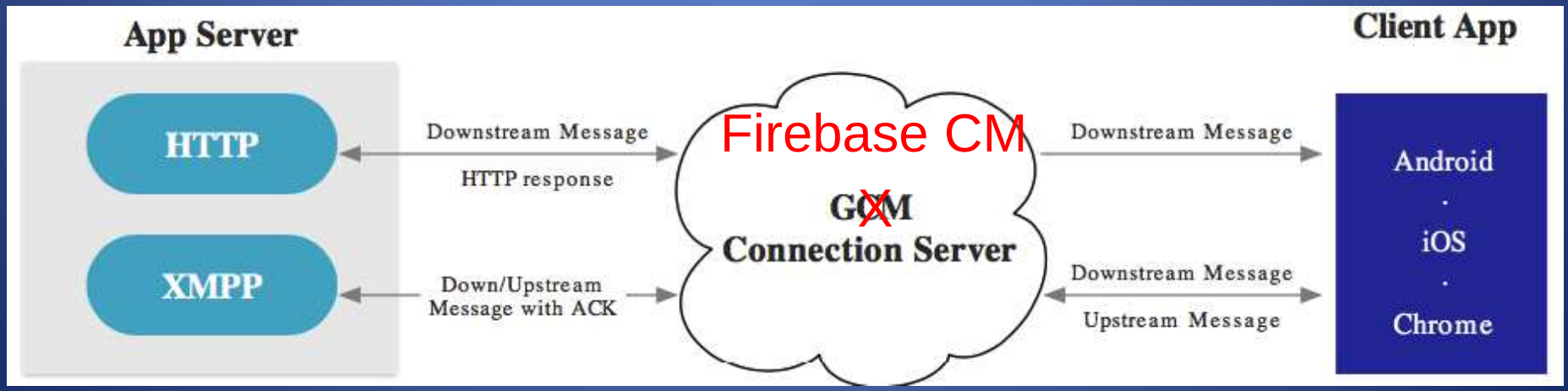
Reinforce Authentication

- 2FA / **Step-up authentication**
  - One-Time password via SMS (SMS-OTP)
  - One-time Code via phone call
  - Hardware or software token
  - Push notifications in combination with PKI and local authentication. FCM (Firebase Cloud Messaging for Android, Chrome and iOS)
- **Contextual**
  - Geolocation
  - IP address
  - Time of day

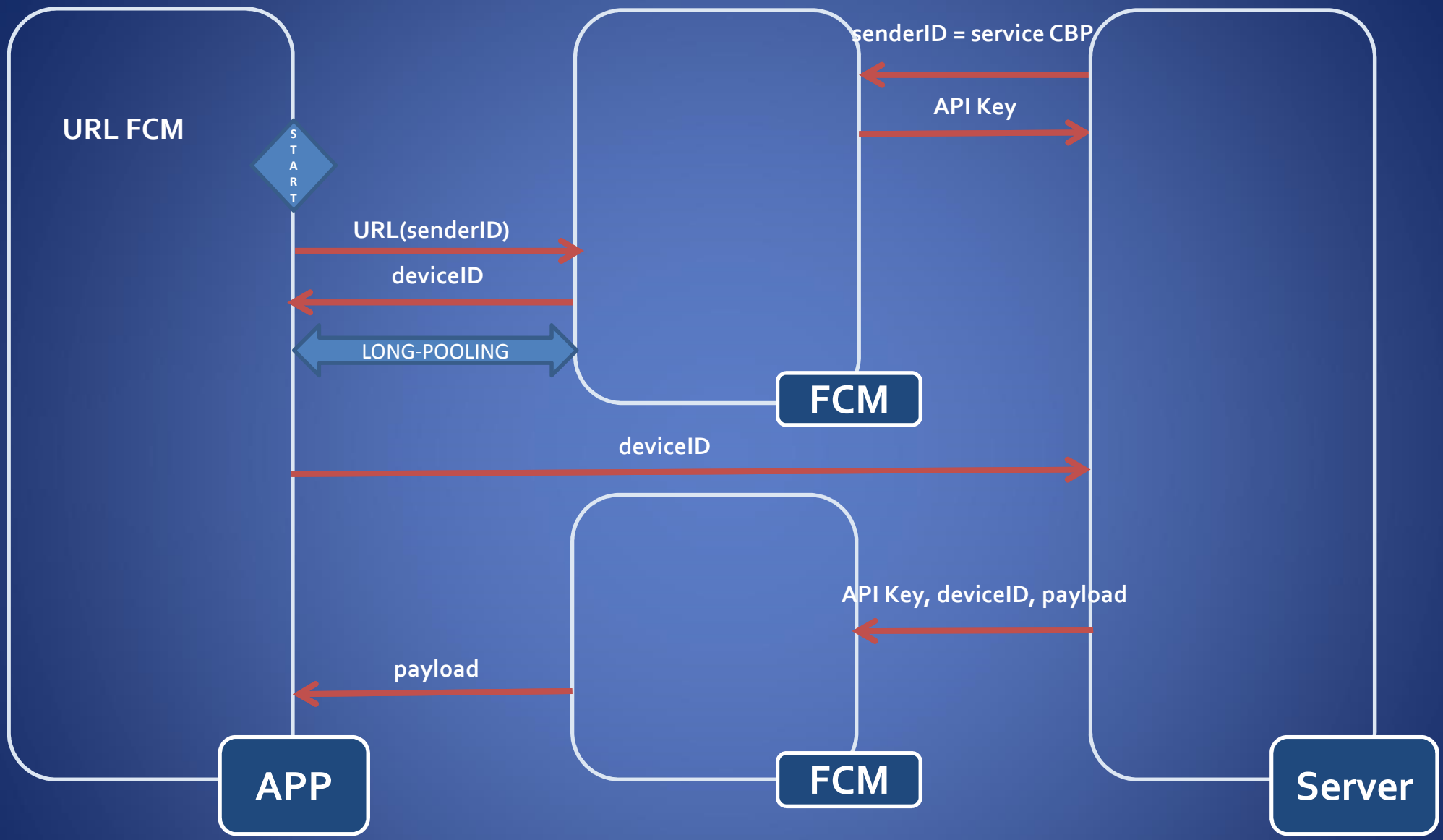
Push notification

Transaction Signing with Push Notifications and PKI

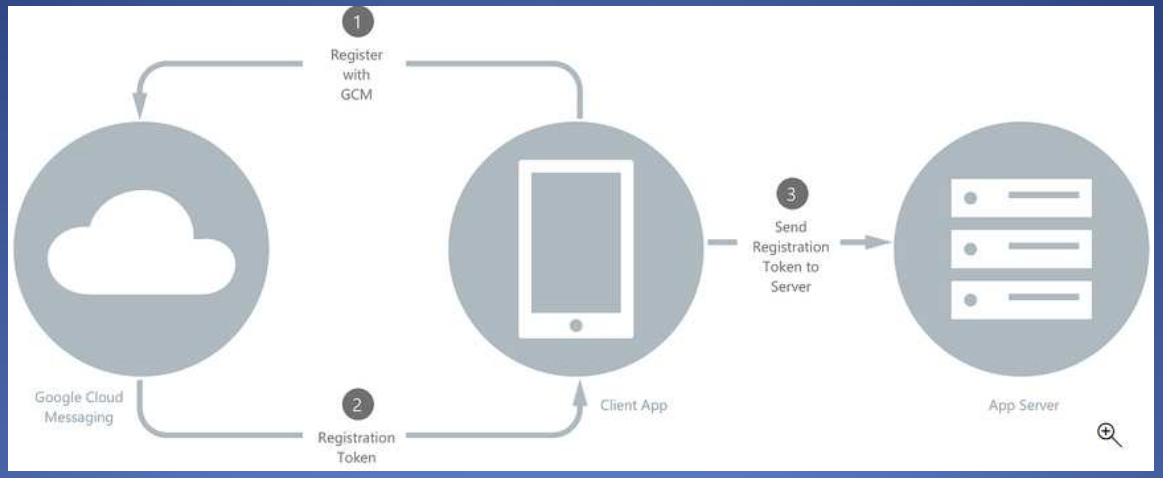
- The app will generate a public/private key pair when the user signs up, then registers the public key on the back end.
- The private key is securely stored in the device keystore.
- To authorize a transaction, the back end sends the mobile app a push notification containing the transaction data.
- After confirmation, the user is prompted to unlock the Keychain (by entering the PIN or fingerprint),
- The data is signed with user's private key.
- The signed transaction is then sent to the server, which verifies the signature with the user's public key.

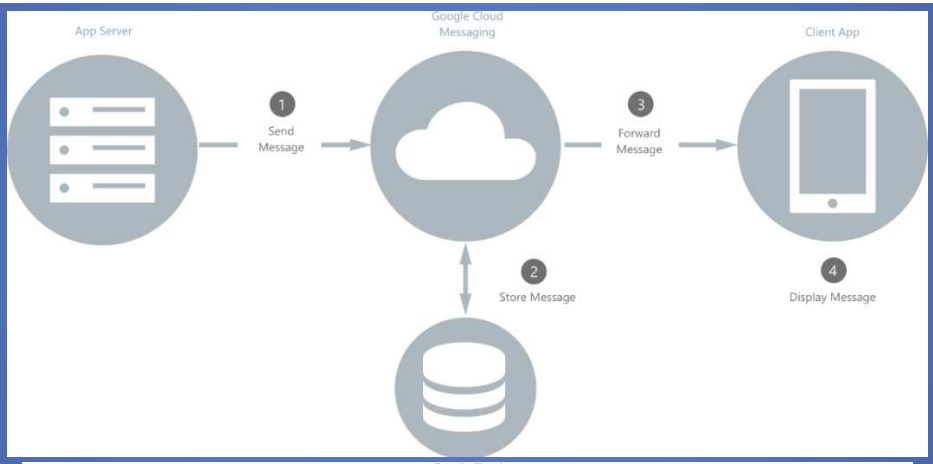


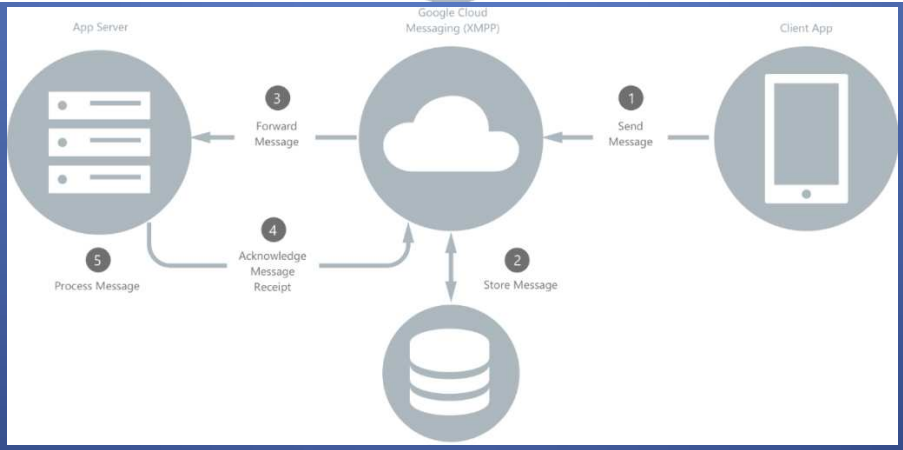
Push notification



FCM (Firebase Cloud Messaging)







https://docs.microsoft.com/fr-fr/xamarin/android/data-cloud/google-messaging/google-cloud-messaging
https://docs.microsoft.com/fr-fr/xamarin/android/data-cloud/google-messaging/firebase-cloud-messaging

M4-Insecure Authentication

Examples

- **Scenario #1:** Hidden Service Requests: Developers assume that only authenticated users will be able to generate a service request that the mobile app submits to its backend for processing. During the processing of the request, the server code does not verify that the incoming request is associated with a known user. Hence, adversaries submit service requests to the back-end service and anonymously execute functionality that affects legitimate users of the solution.
- **Scenario #2:** Interface Reliance: Developers assume that only authorized users will be able to see the existence of a particular function on their mobile app. Back-end code that processes the request does not bother to verify that the identity associated with the request is entitled to execute the service. Hence, adversaries are able to perform remote administrative functionality using fairly low-privilege user accounts.
- **Scenario #3:** Usability Requirements: Due to usability requirements, mobile apps allow for passwords that are 4 digits long. Server code correctly stores a hashed version of the password. However, due to the severely short length of the password, an adversary will be able to quickly deduce the original passwords using rainbow hash tables.

Android Security: Platorm Overview

Android application structure

- An *intent* is a defined object used for messaging that is created and communicated to an intended application component. This communication includes all relevant information passed from the calling application to the desired application component and contains an action and data that is relevant to the request being made.
- A simple example of an application sending a request to open a particular URL in a browser would look as follows in code:

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
startActivity(intent);
```

- An application defines “intent filters” in its manifest, which catches the intents that are appropriate for its components. For example, if an activity in your application can handle HTTP links to websites, then an appropriate intent filter looks as follows:

```
<activity android:name="MyBrowserActivity">
  <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <data android:scheme="http" />
  </intent-filter>
</activity
```

Android Security: Platorm Overview

Android application structure

- A WebView is an embeddable application element that allows web pages to be rendered within an application.
- It makes use of web rendering engines for the loading of web pages and provides browser-like functionality.
- Prior to Android 4.4 it made use of the WebKit (see https://www.webkit.org/) rendering engine; however, it has since been changed to use Chromium (see http://www.chromium.org).

Android Security: Platorm Overview

Android application structure – Permissions 1/2

Android permissions are ranked on the basis of the protection level they offer and divided into four different categories:

- ***Normal***: the lower level of protection. It gives the apps access to isolated application-level features with minimal risk to other apps, the user, or the system. It is granted during app installation and is **the default protection level**: Example: android.permission.INTERNET
- ***Dangerous***: This permission allows the app to perform actions that might affect the user's privacy or the normal operation of the user's device. This level of permission may not be granted during installation; **the user must decide** whether the app should have this permission. Example: android.permission.RECORD_AUDIO
- ***Signature***: This permission is granted only if the requesting app has been signed with the same certificate as the app that declared the permission. If the signature matches, the permission is automatically granted. Example: android.permission.ACCESS_MOCK_LOCATION
- ***SystemOrSignature***: This permission is granted only to apps embedded in the system image or signed with the same certificate that the app that declared the permission was signed with. Example: android.permission.ACCESS_DOWNLOAD_MANAGER

Android Security: Platorm Overview

Android application structure – Permissions 2/2

- **Requesting Permissions**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.permissions.sample" ...>
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<application>...</application>
</manifest>
```

- **Declaring Permissions**

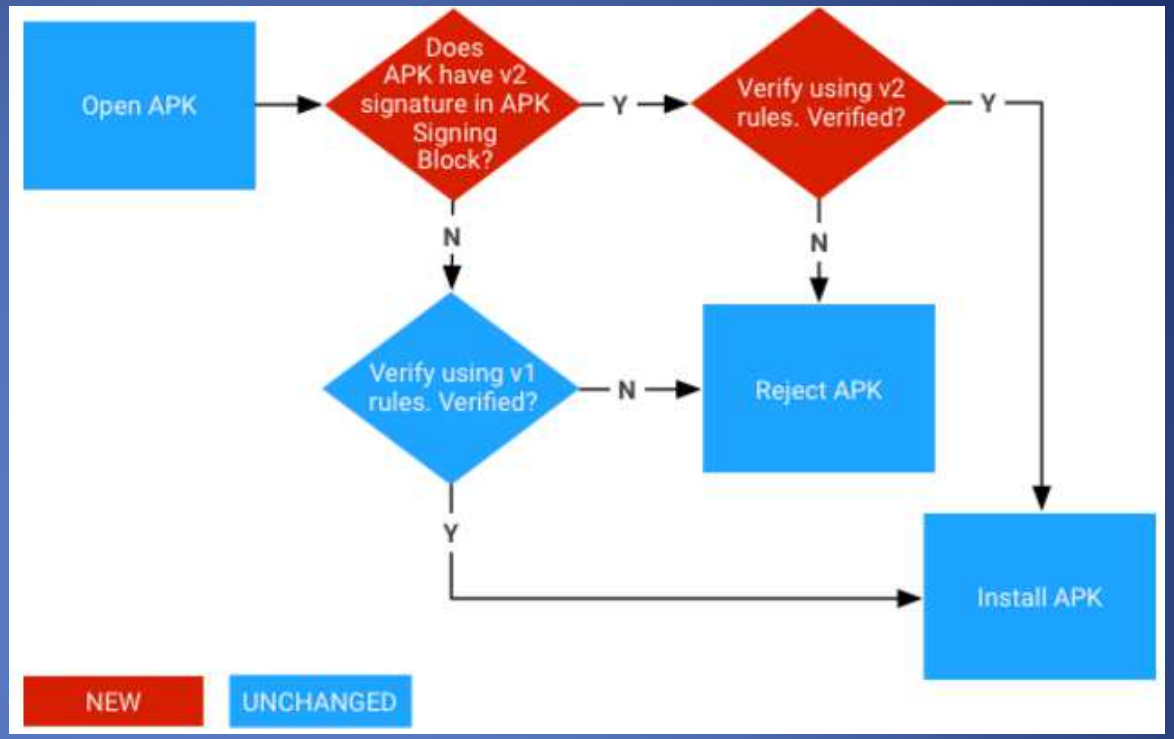
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.permissions.sample" ...>
<permission android:name="com.permissions.sample.ACCESS_USER_INFO"
android:protectionLevel="signature" /> <application>...</application>
</manifest>
```

- **Enforcing Permissions on Android Components**

```
<receiver android:name="com.permissions.sample.AnalyticsReceiver"
android:enabled="true"
android:permission="com.permissions.sample.ACCESS_USER_INFO">
... </receiver>
```

Android Security: Platorm Overview

Android application publishing



JAR Signing (v1 Scheme)

- This scheme does not protect some parts of the APK, such as ZIP metadata.
- The drawback of this scheme is that the APK verifier needs to process untrusted data structures before applying the signature, and the verifier discards data the data structures don't cover.
- The APK verifier must decompress all compressed files

APK Signature Scheme (v2 Scheme)

- The complete APK is hashed and signed, and an APK Signing Block is created and inserted into the APK.
- During validation, the v2 scheme checks the signatures of the entire APK file.
- This form of APK verification is faster and offers more comprehensive protection against modification.

Android Security: Platorm Overview

Android application publishing

Modify apk

$ apktool d fr.poulpage_com.apk
$ apktool b ./fr.poulpage_com
$ cd ./fr.poulpage_com/dist

Creating Your Certificate

$ keytool -genkey -alias myDomain -keyalg RSA -keysize 2048 -validity 7300 -keystore myKeyStore.jks -storepass myStrongPassword

Signing an Application

$ jarsigner -verbose -keystore ./myKeyStore.jks ./fr.poulpage_com.apk myDomain
Note: apksigner can also be used

Check the new signature

$ unzip fr.poulpage_com.apk
$ more META-INF/MYDOMAIN.RSA
$ adb install fr.poulpage_com.apk
$ adb uninstall fr.poulpage_com

TP 0003

Using **fr.poulpage_com.apk**

- Decompile apk
- Change the content
- Recompile
- Install
- Generate cert using keystore
- Sign
- Check the signature
- Install

Android Security: Platorm Overview

Android application publishing

Publishing process:
https://developer.android.com/distribute/googleplay/start.html

Software updates can be distributed in 2 ways:

- OTA updates
- Side-loading over USB or SD Card

MO and manufacturer are responsible for update delivery

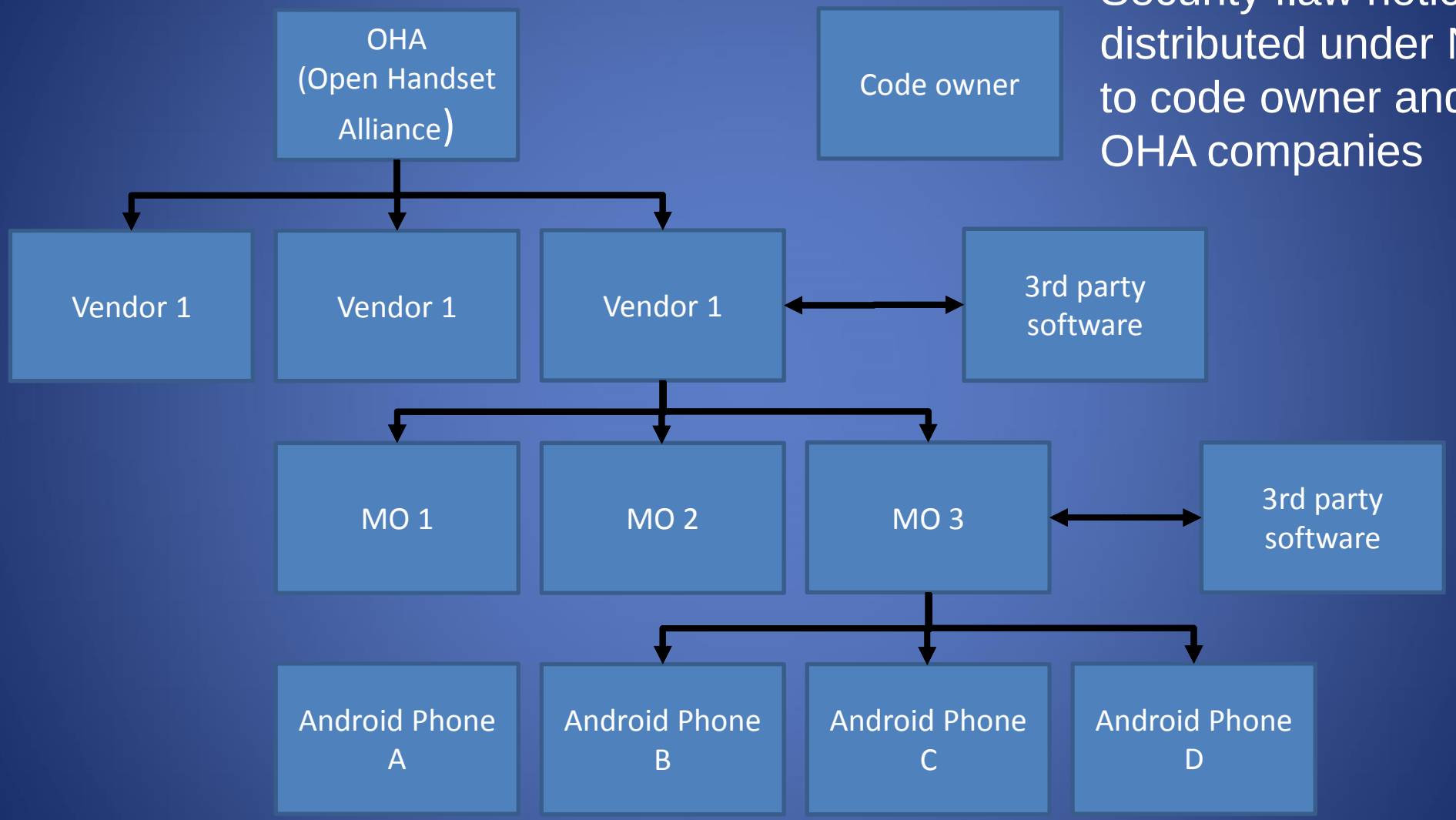
- Initiating OTA delivery
- Providing boot ROM unlock for side-loading

Has led to platform fragmentation among devices

- Commonly, no supported method for users to obtain updates

Android Security: Platorm Overview

Android application publishing



Security flaw notice distributed under NDA to code owner and OHA companies

Android fragmentation forces many users to run vulnerable software with no support or significant delay for security resolution

Android Security: Platorm Overview

Android application publishing

Some vendors are open and communicative about updates

- Motorola, Google: updates from 1 to 3 months

Other vendors distribute updates with less priority

- Samsung, HTC: updates from 6 months to more

Still others abandon recent devices with no communication about updates

- LG, Sony, Acer, Asus: no updates

Those vendors that do not provide updates target feature phones, leaving users of commodity devices without an upgrade path

Android Security: Data Storage

Shared Preferences

The SharedPreferences API is used to permanently save small collections of key-value pairs.
Data stored in a SharedPreferences object is written to a plain-text XML file.
The SharedPreferences object can be declated world-readable (accessible to all apps) or private. Misuse of the SharedPreferences API can often lead to exposure of sensitive data.

Consider the following example:

```
SharedPreferences sharedPref = getSharedPreferences("key",
    MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

Once the activity has been called, the file **key.xml** will be created with the provided data. This code violates several best practices.

Android Security: Data Storage

Shared Preferences

The username and password are stored in clear text in **/data/data/<package-name>/shared_prefs/key.xml**.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
  <map>
    <string name="username">administrator</string>
    <string name="password">supersecret</string>
  </map>
```

MODE_WORLD_READABLE allows all applications to access and read the contents of **key.xml**.

```
root@hermes: #  cd /data/data/sg.vp.owasp_mobile.myfirstapp/shared_prefs
root@hermes: # ls -la
rw-rw-r-- u0_a118 170 2016-04-23 16:51 key.xml
```

TP 0004

Using **MobiSecLab.apk**

- Get the smali code and look for juicy data
- Get the java code and look for juicy data
- Access to file in mobile

Android Security: Data Storage

SQLite Databases

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database,
        "password123", null);
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(
        Username VARCHAR,Password VARCHAR);");
secureDB.execSQL("INSERT INTO Accounts
        VALUES('admin','AdminPassEnc');");
secureDB.close();
```

Asking the user to decrypt the database with a PIN or password once the app is opened (weak passwords and PINs are vulnerable to brute force attacks)

Android Security: Data Storage

Realm Databases

Realm is a lightweight **database** that can replace both SQLite and ORM libraries in your **Android** projects. Compared to SQLite, **Realm** is faster and has lots of modern features, such as JSON support, a fluent API, data change notifications, and encryption support, all of which make life easier for **Android** developers.

```
//the getKey() method either gets the key from the server or from a Keystore,
// or is deferred from a password.
RealmConfiguration config = new RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build();
Realm realm = Realm.getInstance(config);
```

If the database is not encrypted, you should be able to obtain the data. If the database *is* encrypted, determine whether the key is hard-coded in the source or resources and whether it is stored unprotected in shared preferences or some other location.

Note : Sensitive user input: android:inputType="textPassword« (in the definition of EditText)

TP 0005

Using **sieve.apk**

- Understand how to enable backup
- Backup an app
- Get sensitive data from the backup

M5-Insufficient Cryptography

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: Anyone with physical access to data that has been encrypted improperly, or mobile malware acting on an adversary's behalf.

Exploitability: Decryption of data via physical access to the device or network traffic capture, or malicious apps on the device with access to the encrypted data.

Frequency / Detectability: In order to exploit this weakness, an adversary must successfully return encrypted code or sensitive data to its original unencrypted form due to weak encryption algorithms or flaws within the encryption process.

Technical Impact: This vulnerability will result in the unauthorized retrieval of sensitive information from the mobile device.

Business Impact: This vulnerability can have a number of different business impacts. Typically, broken cryptography will result in the following:

- Privacy Violations
- Information Theft / Code Theft / Intellectual Property Theft
- Reputational Damage

M5-Insufficient Cryptography

Am I Vulnerable ?

Poor Key Management Processes

The best algorithms don't matter if you mishandle your keys.

Some examples of problems here include:

- Including the keys in the same attacker-readable directory as the encrypted content
- Use of hardcoded keys within your binary
- Keys may be intercepted via binary attacks

Creation and Use of Custom Encryption Protocols

- Always use modern algorithms that are accepted as strong by the security community, and whenever possible leverage the state of the art encryption APIs within your mobile platform. Binary attacks may result in adversary identifying the common libraries you have used along with any hardcoded keys in the binary. In cases of very high security requirements around encryption, you should strongly consider the use of whitebox cryptography

Use of Insecure and/or Deprecated Algorithms

- These include: RC2, MD4, MD5 and SHA1

M5-Insufficient Cryptography

How Do I Prevent?

It is best to do the following when handling sensitive data:

- Avoid the storage of any sensitive data on a mobile device where possible.
- Apply cryptographic standards that will withstand the test of time for at least 10 years into the future
- Follow the NIST guidelines on recommended algorithms

Table: Comparable strengths

Security Strength	Symmetric key algorithms	FFC (e.g., DSA, D-H)	IFC (e.g., RSA)	ECC (e.g., ECDSA)
≤ 80	2TDEA[21]	$L = 1024$ $N = 160$	$k = 1024$	$f = 160\text{-}223$
112	3TDEA	$L = 2048$ $N = 224$	$k = 2048$	$f = 224\text{-}255$
128	AES-128	$L = 3072$ $N = 256$	$k = 3072$	$f = 256\text{-}383$
192	AES-192	$L = 7680$ $N = 384$	$k = 7680$	$f = 384\text{-}511$
256	AES-256	$L = 15360$ $N = 512$	$k = 15360$	$f = 512+$

NIST Encryption Guidelines

- http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf
- http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

M5-Insufficient Cryptography

Examples

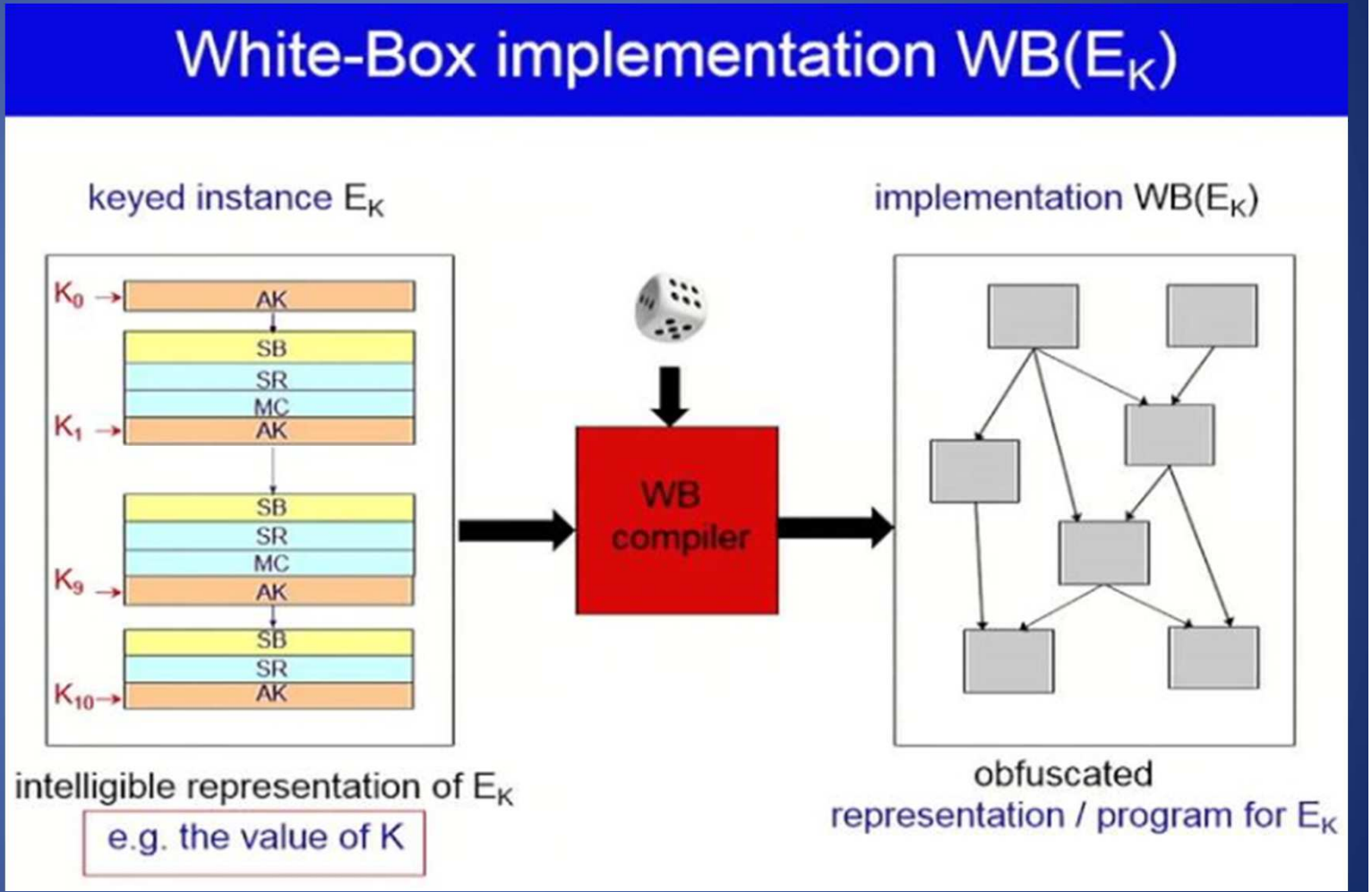
Scenario #1:

- By default, iOS applications are protected (in theory) from reverse engineering via code encryption. The iOS security model requires that apps be encrypted and signed by trustworthy sources in order to execute in non-jailbroken environments. Upon start-up, the iOS app loader will decrypt the app in memory and proceed to execute the code after its signature has been verified by iOS. This feature, in theory, prevents an attacker from conducting binary attacks against an iOS mobile app.
- Using freely available tools like ClutchMod or GBD, an adversary will download the encrypted app onto their jailbroken device and take a snapshot of the decrypted app once the iOS loader loads it into memory and decrypts it (just before the loader kicks off execution). Once the adversary takes the snapshot and stores it on disk, the adversary can use tools like IDA Pro or Hopper to easily perform static / dynamic analysis of the app and conduct further binary attacks.

Source: http://www.idownloadblog.com/2016/07/20/tiff-disabler/

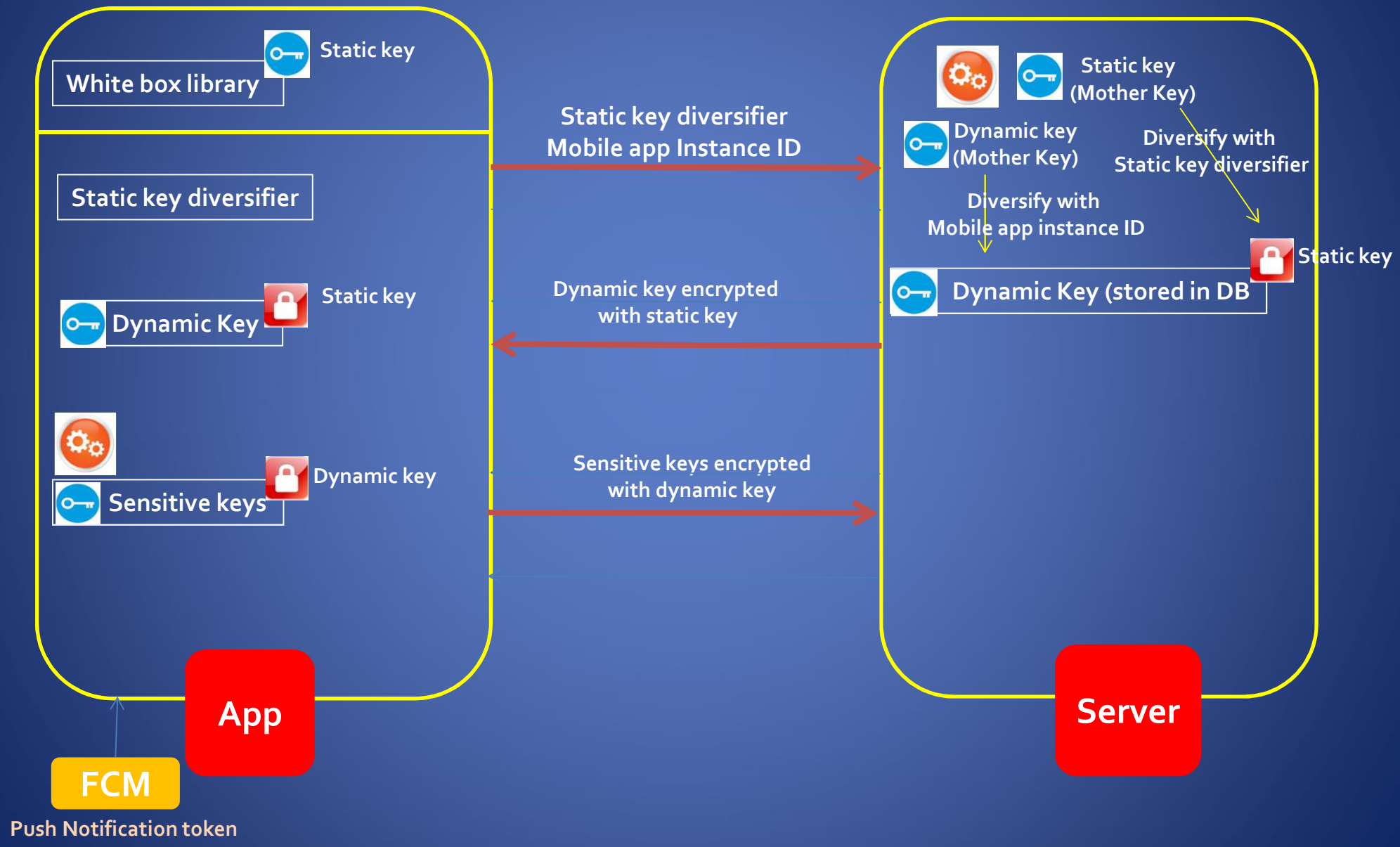
Android Security: Data Storage

Whitebox Cryptography



Android Security: Data Storage

Whitebox Cryptography



Android Security: Data Storage

Internal Storage

You can save files to the device's internal storage. Files saved to internal storage are containerized by default and cannot be accessed by other apps on the device. When the user uninstalls your app, these files are removed. The following code would persistently store sensitive data to internal storage:

```
FileOutputStream fos = null;
try {
        fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
        fos.write(test.getBytes());
        fos.close();
}
catch (FileNotFoundException e)
{
        e.printStackTrace();
}
```

Files stored outside the application folder (data/data/<package-name>/) will not be deleted when the user uninstalls the application

Android Security: Data Storage

External Storage

Every Android-compatible device supports **shared external storage**.
This storage may be removable (such as an SD card) or internal (non-removable).
Files saved to external storage are world-readable.
The user can modify them when USB mass storage is enabled.

You can use the following code to persistently store sensitive information to external storage as the contents of the file **password.txt**:

```
File file = new File (Environment.getExternalFilesDir(), "password.txt");
String password = "SecretPassword";
FileOutputStream fos;
fos = new FileOutputStream(file);
fos.write(password.getBytes());
fos.close();
```

Android Security: Data Storage

KeyStore Security features

Extraction prevention

Key material of Android Keystore keys is protected from extraction using two security measures:

- Key material never enters the application process
- Key material may be bound to the secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)) of the Android device.

Hardware security module

Supported devices running Android 9 (API level 28) or higher installed can have a *StrongBox Keymaster*, an implementation of the Keymaster HAL that resides in a hardware security module. The module contains the following:

- Its own CPU.
- Secure storage.
- A true random-number generator.
- Additional mechanisms to resist package tampering and unauthorized sideloading of apps.

Key use authorizations

Supported key use authorizations fall into the following categories:

- *cryptography*: authorized key algorithm, operations or purposes (encrypt, decrypt, sign, verify), padding schemes, block modes, digests with which the key can be used;
- *temporal validity interval*: interval of time during which the key is authorized for use;
- *user authentication*: the key can only be used if the user has been authenticated recently enough.

Android Security: Data Storage

KeyStore or KeyChain

The Keystore system is used by the KeyChain API as well as the Android Keystore provider feature that was introduced in Android 4.3 (API level 18).

Choose between a keychain or the Android keystore provider

Use the KeyChain API when you want system-wide credentials.

- When an app requests the use of any credential through the KeyChain API, users get to choose, through a system-provided UI, which of the installed credentials an app can access.
- This allows several apps to use the same set of credentials with user consent.

Use the Android Keystore provider to let an individual app store its own credentials that only the app itself can access.

- This provides a way for apps to manage credentials that are usable only by itself while providing the same security benefits that the KeyChain API provides for system-wide credentials.
- This method requires no user interaction to select the credentials.

Android Security: Data Storage

KeyStore

You can protect keys stored in the Android KeyStore with user authentication. The user's lock screen credentials (**pattern, PIN, password, or fingerprint**) are used for authentication.

You can use stored keys in one of two modes:

- Users are authorized to use keys for a **limited period of time after authentication**. In this mode, all keys can be used as soon as the user unlocks the device. You can customize the period of authorization for each key. You can use this option only if the secure lock screen is enabled. If the user disables the secure lock screen, all stored keys will become permanently invalid.
- Users are authorized to use a **specific cryptographic operation that is associated with one key**. In this mode, users must request a separate authorization for each operation that involves the key. Currently, fingerprint authentication is the only way to request such authorization.

Android Security: Data Storage

KeyStore

The level of security afforded by the Android KeyStore depends on its implementation, which depends on the device.

- Most modern devices offer a **hardware**-backed KeyStore implementation: keys are generated and used in a Trusted Execution Environment (**TEE**) or a Secure Element (**SE**), and the operating system can't access them directly. This means that the encryption keys themselves can't be easily retrieved, even from a rooted device.
- The keys of a **software**-only implementation are encrypted with a **per-user** encryption master key. An attacker can access all keys stored on rooted devices that have this implementation in the folder **/data/misc/keystore/**. Because the user's lock screen pin/password is used to generate the master key, the Android KeyStore is unavailable when the device is locked.

Android Security: Data Storage

KeyChain

The **KeyChain** class is used to store and retrieve ***system-wide*** private keys and their corresponding certificates (chain) - every application can access the materials stored in the KeyChain.

The user will be prompted to set a lock screen pin or password to protect the credential storage if something is being imported into the KeyChain for the first time.

https://developer.android.com/reference/android/security/KeyChain
https://code.tutsplus.com/tutorials/keys-credentials-and-storage-on-android--cms-30827
https://nelenkov.blogspot.com/2012/12/certificate-pinning-in-android-42.html
https://nelenkov.blogspot.com/2011/11/using-ics-keychain-api.html
https://github.com/Miserlou/Android-SDK-Samples/tree/master/KeyChainDemo

/data/misc/keychain
/data/misc/keychain/pins: contains default entries for Google services (certificate pinning)

Android Security: Data Storage

Android backup

Given its diverse ecosystem, Android supports many backup options

When USB debugging is enabled, you can use the adb backup command to create full data backups and backups of an app's data directory.
$ adb backup
Google provides a "Back Up My Data" feature that backs up all app data to Google's servers.

- Two Backup APIs are available to app developers:Key/Value Backup (Backup API or Android Backup Service) uploads to the Android Backup Service cloud.Auto Backup for Apps: With Android 6.0 (>= API level 23). This feature automatically syncs at most 25MB of app data with the user's Google Drive account.

OEMs may provide additional options.

In AndroidManifest.xml,
android:allowBackup=“true” // local
android:fullBackupOnly=“true” // cloud : backup agent for Android 6.0 and above
android:backupAgent=“true” // cloud : backup agent for others

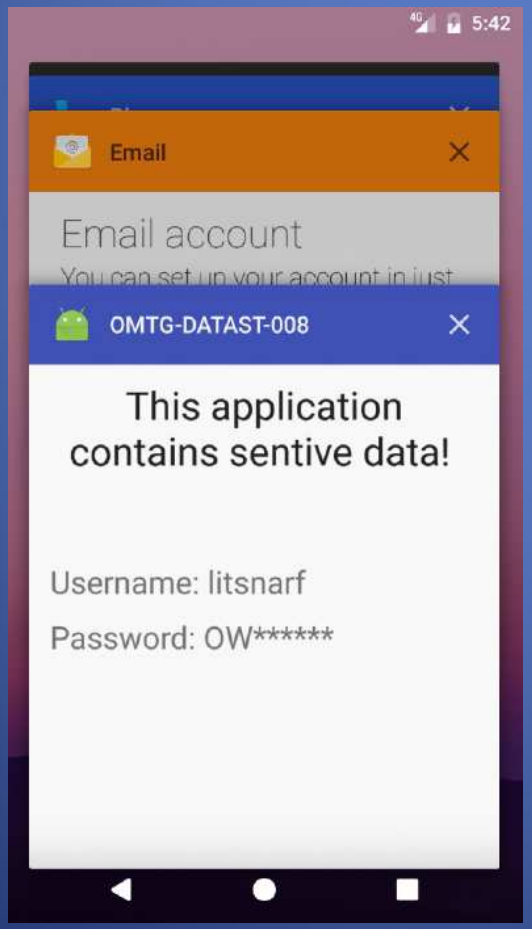
Android Security: Data Storage

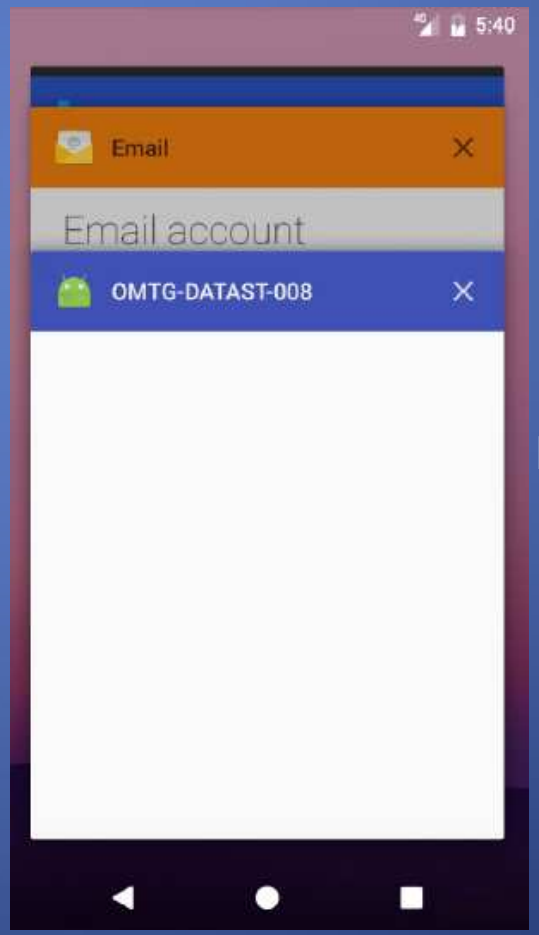
Android FLAG_SECURE

- Navigate to any screen that contains sensitive information and click the home button to send the app to the background
- Then press the app switcher button to see the snapshot.

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
WindowManager.LayoutParams.FLAG_SECURE);
```

FLAG_SECURE has not been set





FLAG_SECURE is set

Android Security: Data Storage

Data in RAM

```
byte[] nonSecret = somePublicString.getBytes("ISO-8859-1"); //default character encoding
byte[] secret = null;
try{
      //get or generate the secret, do work with it, make sure you make no local copies
}
finally {
if (null != secret) {
            // Overwrite value in memory
            // Perform additional calculations (e.g., MODULUS the data into a buffer)
            for (int i = 0; i < secret.length; i++) {
                  secret[i] = nonSecret[i % nonSecret.length];
            }
            // To optimize the bytecode, the compiler will analyze and decide not to
            // overwrite data because it will not be used afterwards
            FileOutputStream out = new FileOutputStream("/dev/null");
            out.write(secret);
            out.flush(); out.close();
} }
```

- No cross-context handling of sensitive data. Each copy of the key can be cleared from within the scope in which it was created.
- The local copy is cleared according to the recommendations given above.

Android Security: Cryptographic APIs

Random Number Generator

Weak

```
import java.util.Random; // Use SecureRandom
// ...
Random number = new Random(123L); // Use no arg constructor (128 byte-long rnd number)
//...
For (int i = 0; i < 20; i++) {
// Generate another random integer in the range [0, 20]
int n = number.nextInt(21); // no need
System.out.println(n); }
```

Stronger

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
// ...
public static void main (String args[]) {
SecureRandom number = new SecureRandom();
// Generate 20 integers 0..20
for (int i = 0; i < 20; i++) {
System.out.println(number.nextInt(21)); } }
```

Android Security: Cryptographic APIs

Protect the key

```
import javax.crypto.spec.*;
import javax.crypto.*;
import java.security.*;
import android.util.*;

public class Encrypt
{
    private static byte[] keyBytes;

    static {
        Encrypt.keyBytes = new byte[] { 7, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 9, 20, 21, 15, 1, 10, 11, 12, 13, 14,
    }

    public static String decrypt(final String s) throws Exception {
        final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
        final Cipher instance = Cipher.getInstance("AES");
        instance.init(2, secretKeySpec);
        return new String(instance.doFinal(Base64.decode(s.getBytes(), 0)));
    }

    public static String encrypt(final String s) throws Exception {
        final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
        final Cipher instance = Cipher.getInstance("AES");
        instance.init(1, secretKeySpec);
        return new String(Base64.encode(instance.doFinal(s.getBytes()), 0));
    }
}
```

Android Security: Local Authentication

Biometric Authentication

Fingerprint hardware must be available:

```
FingerprintManager fingerprintManager = (FingerprintManager)
context.getSystemService(Context.FINGERPRINT_SERVICE);
fingerprintManager.isHardwareDetected();
```

The user must have a protected lockscreen:

```
KeyguardManager keyguardManager = (KeyguardManager)
context.getSystemService(Context.KEYGUARD_SERVICE);
keyguardManager.isKeyguardSecure();
```

At least one finger should be registered:

fingerprintManager.hasEnrolledFingerprints(); The application should have permission to ask for a user fingerprint:

```
context.checkSelfPermission(Manifest.permission.USE_FINGERPRINT) ==
PermissionResult.PERMISSION_GRANTED;
```

If any of the above checks fail, the option for fingerprint authentication should not be offered.

It is important to remember that not every Android device offers hardware-backed key storage. The KeyInfo class can be used to find out whether the key resides inside secure hardware such as a TEE or SE.

Android Security: Local Authentication

Fingerprint Authentication with Symmetric Key

Fingerprint authentication may be implemented by creating a new AES key using the KeyGenerator class by adding setUserAuthenticationRequired(true) in KeyGenParameterSpec.Builder.

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
KEYSTORE);
generator.init(new KeyGenParameterSpec.Builder (KEY_ALIAS,
            KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
            .setUserAuthenticationRequired(true) .build() );
generator.generateKey();
```

To perform encryption or decryption with the protected key, create a Cipher object and initialize it with the key alias.

```
SecretKey keyspec = (SecretKey)keyStore.getKey(KEY_ALIAS, null);
if (mode == Cipher.ENCRYPT_MODE) {
            cipher.init(mode, keyspec);
```

Android Security: Local Authentication

Fingerprint Authentication with Symmetric Key

Keep in mind, a new key cannot be used immediately - it has to be authenticated through the FingerprintManager first. This involves wrapping the Cipher object into FingerprintManager.CryptoObject which is passed toFingerprintManager.authenticate() before it will be recognized.

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, this, null);
```

When the authentication succeeds, the callback method onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) is called at which point, the authenticated CryptoObject can be retrieved from the result.

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult result) {
        cipher = result.getCryptoObject().getCipher();
        (... do something with the authenticated cipher object ...) }
```

TP 0006

Bruteforce the security pattern

Android Security: Network APIs

Server Certificate Verification

Two key issues should be addressed:

- Verify that a certificate comes from a trusted source (CA).
- Determine whether the endpoint server presents the right certificate.

The following code snippet will accept any certificate, overwriting the functions checkClientTrusted, checkServerTrusted, and getAcceptedIssuers.

```
TrustManager[] trustAllCerts = new TrustManager[] {
      new X509TrustManager() {
       @Override public X509Certificate[] getAcceptedIssuers() {
                  return new java.security.cert.X509Certificate[] {}; }
       @Override public void checkClientTrusted(X509Certificate[] chain, String authType)
throws CertificateException { }
       @Override public void checkServerTrusted(X509Certificate[] chain, String authType)
throws CertificateException { } } };

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```

Note: check the Security Provider (e.g. OpenSSL) like every third-party library that have their own vulns

Android Security: Network APIs

Webview Verification

```
// Certificate verification
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
        @Override
        public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError
error) {
                //Ignore TLS certificate errors and instruct the WebViewClient to load the
website
                handler.proceed();
        }
});


// Hostname verification
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
        public boolean verify(String hostname, SSLSession session) {
                return true;
        }
};
```

Android Security: Network APIs

Burp Testing

Dynamic analysis requires an interception proxy. To test improper certificate verification, check the following controls:

Self-signed certificate

- In Burp, go to the Proxy -> Options tab, then go to the Proxy Listeners section, highlight your listener, and click Edit. Then go to the Certificate tab, **check Use a self-signed certificate**, and click Ok. Now, run your application. If you're able to see HTTPS traffic, your application is accepting self-signed certificates.

Accepting invalid certificates

- In Burp, go to the Proxy -> Options tab, then go to the Proxy Listeners section, highlight your listener, and click Edit. Then go to the Certificate tab, check **Generate a CA-signed certificate with a specific hostname, and type in the backend server's hostname**. Now, run your application. If you're able to see HTTPS traffic, your application is accepting all certificates.

Accepting incorrect hostnames

- In Burp, go to the Proxy -> Options tab, then go to the Proxy Listeners section, highlight your listener, and click Edit. Then go to the Certificate tab, check **Generate a CA-signed certificate with a specific hostname, and type in an invalid hostname, e.g., example.org**. Now, run your application. If you're able to see HTTPS traffic, your application is accepting all hostnames.

DEMO 0016

Man In The Middle (Wifi)

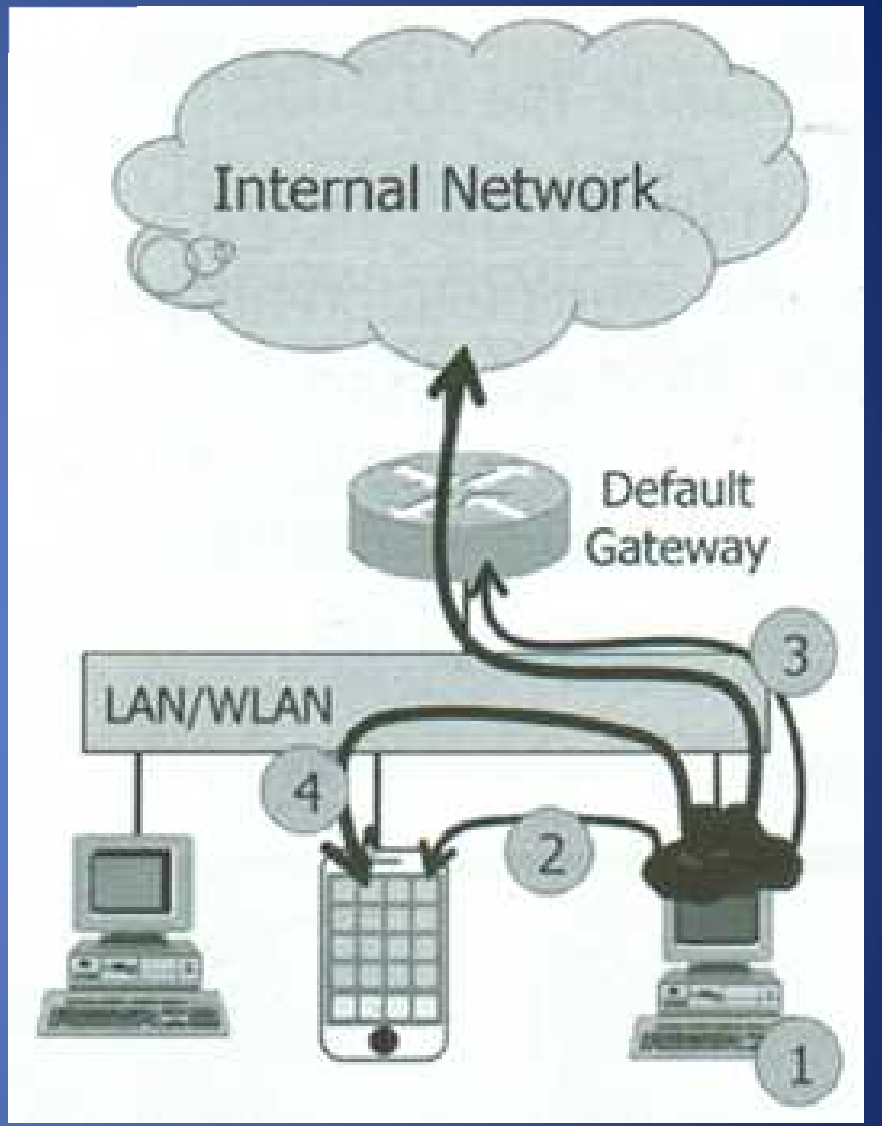
- Sniff network with Wireshark
- Intercept and tamper HTTP requests and responses with Burp Suite

Man In The Middle with Ettercap

Android Security: Network APIs

ARP Spoofing / Poisoning

TCP/IP model	Protocols and services
Application	HTTP, FTTP, Telnet, NTP, DHCP, PING
Transport	TCP, UDP
Network	IP, ARP, ICMP, IGMP
Network Interface	Ethernet



1. Attacker enumerates on the network (multiple ARP requests or other discovery techniques)
2. Attacker sends to a LAN mobile device an ARP reply indicating he is the default gateway (arp table update)
3. Attacker sends to the defualt gateway an ARP reply indicating he is the mobile device (arp table update)
4. All traffic originating from the mobile device or upstream networks bridged through the default gateway are delivered to the attacker, who forwards patcket after inspection or logging or tampering

Android Security: Network APIs

ARP Spoofing / Poisoning

00:18:8b:ad:2a:c7 (172.16.0.1) : **Gateway**
00:23:4d:da:22:23 (172.16.0.111): **Mobile**
00:06:dc:42:18:24 (172.16.0.106): **Attacker**

From: 00:06:dc:42:18:24 (172.16.0.106)
To: 00:18:8b:ad:2a:c7 (172.16.0.1)
Info: i am 00:23:4d:da:22:23 (172.16.0.111)
gateway updates its ARP table ($ arp)

From: 00:06:dc:42:18:24 (172.16.0.106)
To: 00:23:4d:da:22:23 (172.16.0.111)
Info: i am 00:18:8b:ad:2a:c7 (172.16.0.1)
Mobile updates its ARP table

M3-Insecure Communication

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: When designing a mobile application, data is commonly exchanged in a client-server fashion. When the solution transmits its data, it must traverse the mobile device's carrier network and the internet. Threat agents might exploit vulnerabilities to intercept sensitive data while it's traveling across the wire. The following threat agents exist:An adversary that shares your local network (compromised or monitored Wi-Fi);

- Carrier or network devices (routers, cell towers, proxy's, etc); or
- Malware on your mobile device

Exploitability: The exploitabilty factor of monitoring a network for insecure communications ranges. Monitoring traffic over a carrier's network is harder than that of monitoring a local coffee shop's traffic. In general, targeted attacks are easier to perform.

M3-Insecure Communication

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Frequency / Detectability: Mobile applications frequently do not protect network traffic. They may use SSL/TLS during authentication but not elsewhere. This inconsistency leads to the risk of exposing data and session IDs to interception. The use of transport security does not mean the app has implemented it correctly. To detect basic flaws, observe the phone's network traffic. More subtle flaws require inspecting the design of the application and the applications configuration.

Technical Impact: This flaw exposes an individual user's data and can lead to account theft. If the adversary intercepts an admin account, the entire site could be exposed. Poor SSL setup can also facilitate phishing and MITM attacks

Business Impact: At a minimum, interception of sensitive data through a communication channel will result in a privacy violation. The violation of a user's confidentiality may result in:

- Identity theft;
- Fraud, or
- Reputational Damage.

M3-Insecure Communication

Am I Vulnerable ?

This risk covers all aspects of getting data from point A to point B, but doing it insecurely. It encompasses mobile-to-mobile communications, app-to-server communications, or mobile-to-something-else communications. This risk includes all communications technologies that a mobile device might use: TCP/IP, WiFi, Bluetooth/Bluetooth-LE, NFC, audio, infrared, GSM, 3G, SMS, etc.

The usual risks of insecure communication are around data integrity, data confidentiality, and origin integrity.

If the data can be changed while in transit, without the change being detectable (e.g., via a man-in-the-middle attack) then that is a good example of this risk.

That's also an insecure communication problem if confidential data can be exposed, learned, or derived by:

- **observing the communications** as it happens (i.e., eavesdropping)
- **recording the conversation as it happens and attacking it later** (offline attack).

Failing to properly setup and validate a TLS connection (e.g., certificate checking, weak ciphers, other TLS configuration problems) are all here in insecure communication.

M3-Insecure Communication

How Do I Prevent?

General Best Practices

- Assume that the network layer is not secure and is susceptible to eavesdropping.
- Apply SSL/TLS to transport channels that the mobile app will use to transmit sensitive information, session tokens, or other sensitive data to a backend API or web service.
- Account for outside entities like third-party analytics companies, social networks, etc. by using their SSL versions when an application runs a routine via the browser
- Use strong, industry standard cipher suites with appropriate key lengths.
- Use certificates signed by a trusted CA provider.
- Never allow self-signed certificates, and consider certificate pinning for security conscious applications.
- Always require SSL chain verification.

M3-Insecure Communication

How Do I Prevent?

General Best Practices

- Only establish a secure connection after verifying the identity of the endpoint server using trusted certificates in the key chain.
- Alert users through the UI if the mobile app detects an invalid certificate.
- Do not send sensitive data over alternate channels (e.g, SMS, MMS, or notifications).
- If possible, apply a separate layer of encryption to any sensitive data before it is given to the SSL channel. In the event that future vulnerabilities are discovered in the SSL implementation, the encrypted data will provide a secondary defense against confidentiality violation.
- Newer threats allow an adversary to eavesdrop on sensitive traffic by intercepting the traffic within the mobile device just before the mobile device's SSL library encrypts and transmits the network traffic to the destination server. See M10 for more information on the nature of this risk.

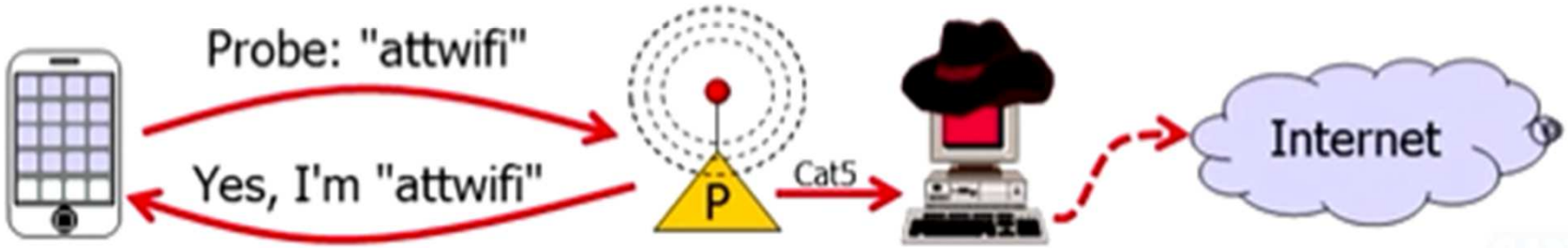
M3-Insecure Communication

Examples

- There are a few common scenarios that penetration testers frequently discover when inspecting a mobile app's communication security:
- **Lack of certificate inspection** The mobile app and an endpoint successfully connect and perform a TLS handshake to establish a secure channel. However, the mobile app fails to inspect the certificate offered by the server and the mobile app unconditionally accepts any certificate offered to it by the server. This destroys any mutual authentication capability between the mobile app and the endpoint. The mobile app is susceptible to man-in-the-middle attacks through a TLS proxy.
- **Weak handshake negotiation** The mobile app and an endpoint successfully connect and negotiate a cipher suite as part of the connection handshake. The client successfully negotiates with the server to use a weak cipher suite that results in weak encryption that can be easily decrypted by the adversary. This jeopardizes the confidentiality of the channel between the mobile app and the endpoint.
- **Privacy information leakage** The mobile app transmits personally identifiable information to an endpoint via non-secure channels instead of over SSL. This jeopardizes the confidentiality of any privacy-related data between the mobile app and the endpoint.

WiFi Pineapple Mark V

- Commercial tool from HakShop.com for $99
- OpenWrt Linux and integrated Karma on Atheros 400 MHz CPU with USB
- Uses Jasager web UI to manage Karma attack and client management
- Bridge to attacker LAN with Internet access for MitM attack



© SANS,
All Rights Reserved

Android Security: Network APIs

Certificate Pinning

- If the app implements certificate pinning, X.509 certificates provided by an interception proxy will be declined and the app will refuse to make any requests through the proxy. To perform an efficient white box test, use a debug build with deactivated certificate pinning.
- There are several ways to bypass certificate pinning for a black box test, for example, SSLUnpinning and Android-SSL-TrustKiller. Certificate pinning can be bypassed within seconds, but only if the app uses the API functions that are covered for these tools. If the app is implementing SSL Pinning with a framework or library that those tools don't yet implement, the SSL Pinning must be manually patched and deactivated, which can be time-consuming.
- There are two ways to manually deactivate SSL Pinning:
  - Dynamic Patching with **Frida** or ADBI while running the app
  - Identifying the SSL Pinning logic in **smali** code, patching it, and reassembling the APK
- Deactivating SSL Pinning satisfies the prerequisites for dynamic analysis, after which the app's communication can be investigated.

Android Security: Network APIs

Certificate Pinning

The Network Security Configuration feature lets apps customize their network security settings in a safe, declarative configuration file without modifying app code.

Network Security Configuration (NSC) that Android provides for versions 7.0 and above (or backport of NSC: https://github.com/datatheorem/TrustKit-Android or TrustKit for iOS)

Specification of the **NSC file reference** in the **Android application manifest** via the "**android:networkSecurityConfig**" attribute on the application tag:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="myserver.com.app">
        <application android:networkSecurityConfig="@xml/network_security_config"> ...
        </application>
</manifest>
```

Android Security: Network APIs

Certificate Pinning

Contents of the NSC file stored in "**res/xml/network_security_config.xml**":

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
        <domain-config>
        <!-- Use certificate pinning for OWASP website access including sub domains -->
        <domain includeSubdomains="true">owasp.org</domain>
        <pin-set>
                <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of the
Intermediate CA of the OWASP website server certificate -->
                <pin digest="SHA-256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
                <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of the Root
CA of the OWASP website server certificate -->
                <pin digest="SHA-
256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
        </pin-set>
        </domain-config>
</network-security-config>
```

Android Security: Network APIs

Certificate Pinning

Applications that use a WebView component may utilize the WebViewClient's event handler for some kind of "certificate pinning" of each request before the target resource is loaded. The following code shows an example **verification of the Issuer DN of the certificate sent by the server**:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
        private String expectedIssuerDN = "CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US;";
        @Override
        public void onLoadResource(WebView view, String url) {
                //From Android API documentation about "WebView.getCertificate()":
                //Gets the SSL certificate for the main top-level page
                //or null if there is no certificate (the site is not secure).
                //Available information on SslCertificate class are "Issuer DN", "Subject DN"
                //and validity date helpers
                SslCertificate serverCert = view.getCertificate();
                if(serverCert != null){
                        //Apply check on Issuer DN against expected one
                        SslCertificate.DName issuerDN = serverCert.getIssuedBy();
                        if(!this.expectedIssuerDN.equals(issuerDN.toString())){
                                //Throw exception to cancel resource loading... } }} });
```

Android Security: Platform APIs

Principle of Least Privilege

Check permissions in AndroidManifest.xml or

$ /opt/mobisec/Android/sdk/build-tools/21.1.2/aapt d permissions ~/Desktop/APKS/com.Slack.apk

package: com.Slack

When Android applications expose IPC components to other applications, they can define permissions to control which applications can access the components. For communication with a component protected by a normal or dangerous permission, Drozer can be rebuilt so that it includes the required permission:

 # If the command below fails you can decompile, change manifest and recompile

$ drozer agent build --permission android.permission.REQUIRED_PERMISSION

Note that this method can't be used for signature level permissions because Drozer would need to be signed by the certificate used to sign the target application.

Android Security: Platform APIs

URL Schemes

Both Android and iOS allow inter-app communication via **custom URL schemes**. These custom URLs allow other applications to perform specific actions within the application that offers the custom URL scheme. Custom URIs can begin with any scheme prefix, and they usually define an action to take within the application and parameters for that action.

Consider this example:
sms:*//compose/to=****your.boss@company.com****&message=****I****%20****QUIT!****&sendImmediately=true.*

When a victim clicks such a link on a mobile device, the vulnerable SMS application will send the SMS message with the maliciously crafted content.

Android Security: Platform APIs

URL Schemes

```
<activity android:name=".MyUriActivity">
    <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="myapp" android:host="path" />
    </intent-filter>
</activity>
```

The example above specifies a new URL scheme called **myapp://**. The category browsable will **allow the URI to be opened within a browser**.

Data can then be transmitted through this new scheme with, for example, the following URI: **myapp://path/to/what/i/want?keyOne=valueOne&keyTwo=valueTwo**.
Code like the following can be used to retrieve the data:

```
Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    Uri uri = intent.getData();
    String valueOne = uri.getQueryParameter("keyOne");
    String valueTwo = uri.getQueryParameter("keyTwo");
}
```

TP 0007

Using **org.owasp.goatdroid.fourgoats.apk**

- First usage of drozer
- Get attack surface
- Find clues in the code
- Exploit vulnerability in Broadcast receiver
- Get app information
- Attempt to exploit vulnerability in URL schemes with Slack

M6-Insecure Authorization

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: Threat agents that exploit authorization vulnerabilities typically do so through automated attacks that use available or custom-built tools.

Exploitability: Once the adversary understands how the authorization scheme is vulnerable, they login to the application as a legitimate user. They successfully pass the authentication control. Once past authentication, they typically force-browse to a vulnerable endpoint to execute administrative functionality. This submission process is typically done via mobile malware within the device or botnets owned by the attacker.

Frequency / Detectability: To test for poor authorization schemes, testers can perform binary attacks against the mobile app and try to execute privileged functionality that should only be executable with a user of higher privilege while the mobile app is in 'offline' mode. As well, testers should try to execute any privileged functionality using a low-privilege session token within the corresponding POST/GET requests for the sensitive functionality to the backend server. Authorization requirements are more vulnerable when making authorization decisions within the mobile device instead of through a remote server. This may be a requirement due to mobile requirements of offline usability.

M6-Insecure Authorization

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Technical Impact: The technical impact of poor authorization is similar in nature to the technical impact of poor authentication. The technical impact can be wide ranging in nature and dependent upon the nature of the over-privileged functionality that is executed. For example, over-privileged execution of remote or local administration functionality may result in destruction of systems or access to sensitive information.

Business Impact: In the event that a user (anonymous or verified) is able to execute over-privileged functionality, the business may experience the following impacts:

- Reputational Damage
- Fraud
- Information Theft

M6-Insecure Authorization

Am I Vulnerable ?

There are a few easy rules to follow when trying to determine if a mobile endpoint is suffering from insecure authorization:

- **Presence of Insecure Direct Object Reference (IDOR) vulnerabilities** - If you are seeing an Insecure Direct Object Reference Vulnerability (IDOR), the code is most likely not performing a valid authorization check
- **Hidden Endpoints** - Typically, developers do not perform authorization checks on backend hidden functionality as they assume the hidden functionality will only be seen by someone in the right role
- **User Role or Permission Transmissions** - If the mobile app is transmitting the user's roles or permissions to a backend system as part of a request, it is suffering from insecure authorization

M6-Insecure Authorization

How Do I Prevent?

In order to avoid insecure authorization checks, do the following:

- Verify the roles and permissions of the authenticated user using only information contained in backend systems. Avoid relying on any roles or permission information that comes from the mobile device itself
- Backend code should independently verify that any incoming identifiers associated with a request (operands of a requested operation) that come along with the identify match up and belong to the incoming identity

M6-Insecure Authorization

Examples

Scenario #1: Insecure Direct Object Reference:

- A user makes an API endpoint request to a backend REST API that includes an actor ID and an oAuth bearer token. The user includes their actor ID as part of the incoming URL and includes the access token as a standard header in the request. The backend verifies the presence of the bearer token but does not validate the actor ID associated with the bearer token. As a result, the user can tweak the actor ID and attain account information of other users as part of the REST API request.

Scenario #2: Transmission of LDAP roles:

- A user makes an API endpoint request to a backend REST API that includes a standard oAuth bearer token along with a header that includes a list of LDAP groups that the user belongs to. The backend request validates the bearer token and then inspects the incoming LDAP groups for the right group membership before continuing on to the sensitive functionality. However, the backend system does not perform an independent validation of LDAP group membership and instead relies upon the incoming LDAP information coming from the user. The user can tweak the incoming header and report to be a member of any LDAP group arbitrarily and perform administrative functionality.

Android Security: Platform APIs

Webview

WebViews may be part of a native app to allow web page viewing.

Every app has its own WebView cache, which isn't shared with the native Browser or other apps.

On Android, WebViews use the WebKit rendering engine to display web pages, but the pages are stripped down to minimal functions, for example, pages don't have address bars.

If the WebView implementation is too lax and allows usage of JavaScript, JavaScript can be used to attack the app and gain access to its data.

JavaScript is disabled by default for WebViews and must be explicitly enabled.

webview.getSettings().setJavaScriptEnabled(true);

Then you should make sure that:

•the communication to the endpoints consistently relies on HTTPS (or other protocols that allow encryption) to protect HTML and JavaScript from tampering during transmission

•JavaScript and HTML are loaded locally, from within the app data directory or from trusted web servers only.

WebView = new WebView(this); webView.loadUrl("file:///android_asset/filename.html");

Android Security: Platform APIs

Webview

Android offers a way for **JavaScript executed in a WebView** to call and use native functions of an Android app: **addJavascriptInterface**

```
WebView webview = new WebView(this);
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);
MSTG_ENV_008_JS_Interface jsInterface = new MSTG_ENV_008_JS_Interface(this);
myWebView.addJavascriptInterface(jsInterface, "Android");
myWebView.loadURL("http://example.com/file.html"); setContentView(myWebView);
```

In Android API levels 17 and above, an annotation called **JavascriptInterface** explicitly allows JavaScript to access a Java method

```
public class MSTG_ENV_008_JS_Interface {
        Context mContext; /** Instantiate the interface and set the context */
        @JavascriptInterface
        public String returnString () { return "Secret String"; }
}
```

TP 0008

Using **flitter.apk**

- Exploit a vulnerability in Webview
- Get the sensitive information in the decompiled app
- Exfiltrate the sensitive information on the attacker web server

Advanced but out of scope:

- Put the attack file in the sdcard (requires to change the app permissions)
- Put the attack file on an external web server

M1-Improper Platform Usage

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: This category covers misuse of a platform feature or failure to use platform security controls. It might include Android **intents**, platform permissions, misuse of TouchID, the Keychain, or some other security control that is part of the mobile operating system.

Exploitability: The attack vectors correspond to the same attack vectors available through the traditional OWASP Top Ten. Any exposed API call can serve as attack vector here.

Frequency: In order for this vulnerability to be exploited, the organization must expose a web service or API call that is consumed by the mobile app. Through the mobile interface, an adversary is able to feed malicious inputs or unexpected sequences of events to the vulnerable endpoint.

Detectability: Refer to OWASP Top 10 exploited vulnerability

Technical Impact: Refer to OWASP Top 10 exploited vulnerability

Business Impact: Refer to OWASP Top 10 exploited vulnerability

M1-Improper Platform Usage

Am I Vulnerable ?

- **Violation of published guidelines.** All platforms have development guidelines for security (c.f., ((Android)), ((iOS)), ((Windows Phone))). If an app contradicts the best practices recommended by the manufacturer, it will be exposed to this risk. For example, there are guidelines on how to use the iOS Keychain or how to secure exported services on Android. Apps that do not follow these guidelines will experience this risk.
- **Violation of convention or common practice.** Not all best practices are codified in manufacturer guidance. In some instances, there are de facto best practices that are common in mobile apps.
- **Unintentional Misuse.** Some apps intend to do the right thing, but actually get some part of the implementation wrong. This could be a simple bug, like setting the wrong flag on an API call, or it could be a misunderstanding of how the protections work.

M1-Improper Platform Usage

How Do I Prevent?

- Secure coding and configuration practices must be used on server-side of the mobile application. For specific vulnerability information, refer to the OWASP Web Top Ten or Cloud Top Ten projects.

Examples

- **App Local Storage Instead of Keychain** The iOS Keychain is a secure storage facility for both app and system data. On iOS, apps should use it to store any small data that has security significance (session keys, passwords, device enrolment data, etc.). A common mistake is to store such items in app local storage. Data stored in app local storage is available in unencrypted iTunes backups (e.g., on the user's computer). For some apps, that exposure is inappropriate.

M1-Improper Platform Usage

Examples

- . Data stored in cloud remote storage is available in unencrypted

Source: https://www.owasp.org/images/4/47/Cloud-Top10-Security-Risks.pdf

M1-Improper Platform Usage

Examples

- Poor Web Services Hardening
  - Business Logic flaws
  - Weak Authentication
  - Weak or no session management
  - Session fixation
  - Sensitive data transmitted using GET method
- Insecure web server configurations
- Injection (SQL, XSS, Command) on both web services and mobile-enabled websites
- Authentication flaws
- Session management flaws
- Access Control vulnerabilities
- Local and Remote File Includes

Android Security: Platform APIs

Object Persistence

The contents of an instance of the BagOfPrimitives is serialized into **JSON**:

```
class BagOfPrimitives {
    private int value1 = 1;
    private String value2 = "abc";
    private transient int value3 = 3;
    BagOfPrimitives() {  // no-args constructor } }
…
// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);
// ==> json is {"value1":1,"value2":"abc"}
```

There are libraries that provide functionality for directly storing the contents of an object in a database and then instantiating the object with the database contents. This is called **Object-Relational Mapping** (ORM). Libraries that use the SQLite database include

- OrmLite,
- SugarORM,
- GreenDAO and ActiveAndroid.

Android Security: Platform APIs

Object Persistence

An object and its data can be represented as a sequence of bytes. This is done in Java via object serialization. **Serialization** is not inherently secure. It is just a binary format (or representation) for locally storing data in a **.ser file**.

Deserializing an object requires a class of the same version as the class used to serialize the object. After classes have been changed, theObjectInputStream can't create objects from older .ser files.

The example below shows how to create a Serializable class by implementing the Serializable interface.

```
import java.io.Serializable;
public class Person implements Serializable {
    private String firstName;
    private String lastName;
    public Person(String firstName, String lastName) {
        this.firstName = firstName; this.lastName = lastName;
} //.. //getters, setters, etc //.. }
```

Now you can read/write the object with **ObjectInputStream/ObjectOutputStream** in another class

Android Security: Platform APIs

Object Persistence

Look for (static analysis):

- import java.io.Serializable
- implements Serializable
- import org.json.JSONObject;
- import org.json.JSONArray;
- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- @Expose, @JsonAdapter,
- @SerializedName,@Since, and @Until
- import com.fasterxml.jackson.core
- import org.codehaus.jackson
- import com.j256.*
- import com.j256.dao
- import com.j256.db
- import com.j256.stmt
- import com.j256.table\
- import com.github.satyan
- extends SugarRecord<Type>
- import io.realm.RealmObject;
- import io.realm.annotations.PrimaryKey;
- ActiveAndroid.initialize(<contextReference>);
- import com.activeandroid.Configuration
- import com.activeandroid.query.*
- import org.greenrobot.greendao.annotation.*
- import org.greenrobot.greendao.database.Database
- import org.greenrobot.greendao.query.Query

Android Security: Code Quality and Build Settings

Applicative vulnerabilities

Signature

*$ apksigner **verify** --verbose Desktop/example.apk*

*$ jarsigner **-verify** -verbose -certs example.apk*

The signing configuration can be managed through Android Studio or the signingConfig block in build.gradle. To activate both the v1 and v2 schemes, the following values must be set:

v1SigningEnabled true v2SigningEnabled true

Insecure storage / Injection / Unintended Data Leakage / Weak crypto / reverse engineer / Poor authentication

Exception Handling

Testing exception handling is about ensuring that the app will handle an exception and transition to a safe state without exposing sensitive information via the UI or the app's logging mechanisms.

Obfuscation

Because decompiling Java classes is trivial, applying some basic obfuscation to the release byte-code is recommended. **ProGuard** offers an easy way to shrink and obfuscate code and to strip unneeded debugging information from the byte-code of Android Java apps. It replaces identifiers, such as class names, method names, and variable names, with meaningless character strings. This is a type of layout obfuscation, which is "free" in that it doesn't impact the program's performance.

TP 0015

Revision

Using com.app.mobshep.csinjection-2:
    Find SQL injection
Using com.android.insecurebankv2
    Discover SQL injection using drozer
Using com.app.mobshep.IDS-2.apk
    Insecure Data Storage
Using com.app.mobshep.UDL-1.apk
    User Data Leakage
Using com.app.mobshep.BC-1.apk
    Broken Cryptography
Using com.mobshep.poorauthentication1-1.apk
    Poor Authentication
Using ReverseEngineer.apk
    Reverse Engineer (find sensitive data)

...and more if you have time (see the list of apks provided with the VM)

M7-Poor Code Quality

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence COMMON	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific

Who: Threat Agents include entities that can pass untrusted inputs to method calls made within mobile code. These types of issues are not necessarily security issues in and of themselves but lead to security vulnerabilities. Poor code-quality issues are typically exploited via malware or phishing scams.

Exploitability: An attacker will exploit vulnerabilities by supplying carefully crafted inputs to the victim. These inputs are passed onto code that resides within the mobile device where exploitation takes place. Typical types of attacks will exploit memory leaks and buffer overflows.

Frequency: Code quality issues are fairly prevalent within most mobile code. The good news is that most code quality issues are fairly benign and result in bad programming practice.

front-end architecture

M7-Poor Code Quality

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence COMMON	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific

Detectability: It is difficult to detect these types of issues through manual code review. Instead, attackers will use third-party tools that perform static analysis or perform fuzzing, to identify memory leaks, buffer overflows. Hackers with extreme low-level knowledge and expertise are able to effectively exploit these types of issues. The typical primary goal is to execute foreign code within the mobile code's address space.

Technical Impact: Most exploitations that fall into this category result in foreign code execution or denial of service on remote server endpoints (and not the mobile device itself). However, in the event that buffer overflows/overruns do exist within the mobile device and the input can be derived from an external party, this could have a severely high technical impact and should be remediated.

Business Impact: The business impact from this category of vulnerabilities varies greatly, depending upon the nature of the exploit. Poor code quality issues that result in remote code execution could lead to the following business impacts:

- Information Theft / Intellectual Property Theft
- Reputational Damage / Degradations in performance

M7-Poor Code Quality

Am I Vulnerable ?

- This is the catch-all for code-level implementation problems in the mobile client. That's distinct from server-side coding mistakes. This captures the risks that come from vulnerabilities like buffer overflows, format string vulnerabilities, and various other code-level mistakes where the solution is to rewrite some code that's running on the mobile device.
- This is distinct from Improper Platform Usage because it usually refers to the programming language itself (e.g., Java, Swift, Objective C, JavaScript). A buffer overflow in C or a DOM-based XSS in a Webview mobile app would be code quality issues.
- The key characteristic of this risk is that it's code executing on the mobile device and the code needs to be changed in a fairly localised way. Fixing most risks requires code changes, but in the code quality case the risk comes from using the wrong API, using an API insecurely, using insecure language constructs, or some other code-level issue. Importantly: this is not code running on the server. This is a risk that captures bad code that executes on the mobile device itself.

M7-Poor Code Quality

How Do I Prevent?

In general, code quality issues can be avoided by doing the following:

- Maintain consistent coding patterns that everyone in the organization agrees upon
- Write code that is easy to read and well-documented
- When using buffers, always validate that the lengths of any incoming buffer data will not exceed the length of the target buffer
- Via automation, identify buffer overflows and memory leaks through the use of third-party static analysis tools
- Prioritize solving buffer overflows and memory leaks over other 'code quality' issues

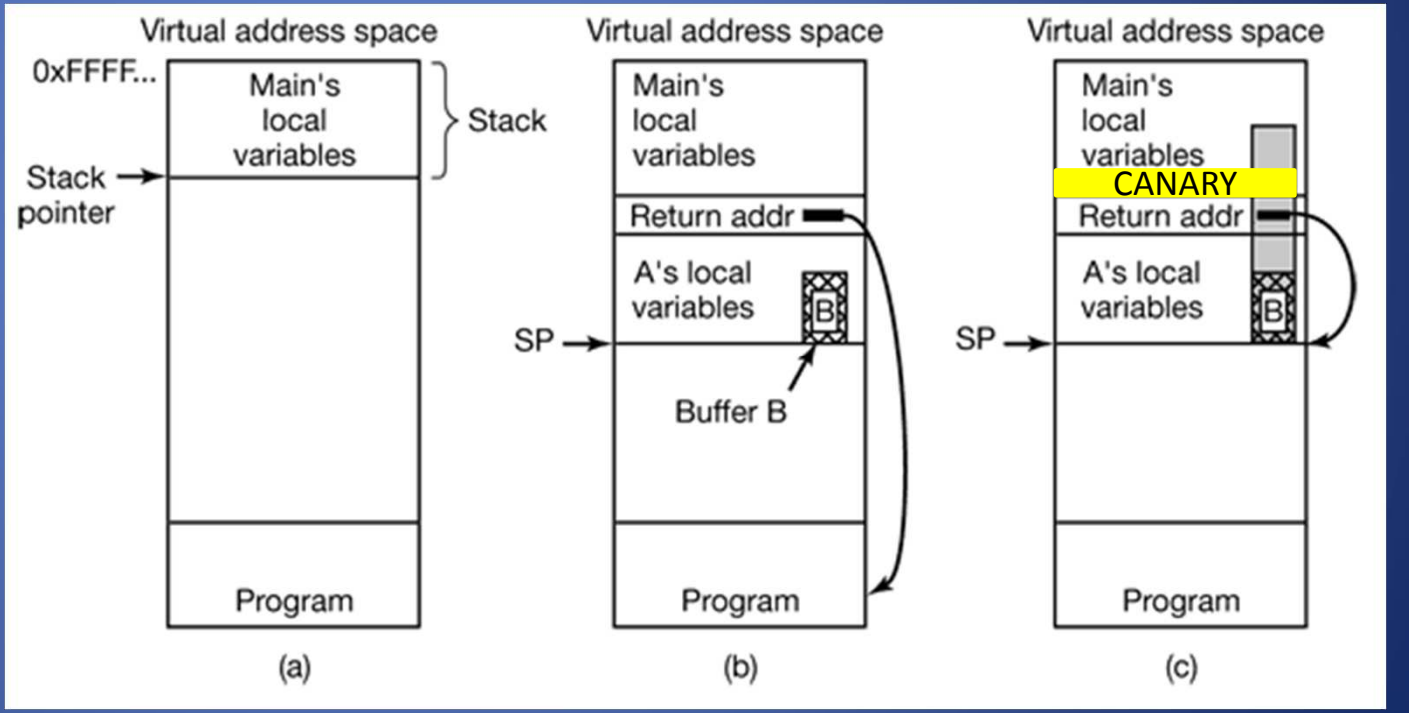
M7-Poor Code Quality

Examples

Scenario #1: Buffer Overflow example:

"f" reads from the standard input an array of the characters, and copies it into the buffer of the char type. The size of this buffer is 10 characters. After that, the application exits.

```
#include <string.h>
void f(char* s) {
    char buffer[10];
    strcpy(buffer, s);
}
void main(void)
{ f("01234567890123456789"); }

[root /tmp]# ./stacktest
Segmentation fault
```



In this example, we should avoid the use of the "*f*" function to avoid a buffer overflow. This is an example of what most static analysis tools will report as a code quality issue

M7-Poor Code Quality

Examples

Scenario #2: CVE-2016-4631

"An exploitable heap based buffer overflow exists in the handling of TIFF images on Apple OS X and iOS operating systems.

A crafted TIFF () document can lead to a heap based buffer overflow resulting in remote code execution.

This vulnerability can be triggered via malicious web page, MMS message, iMessage or a file attachment delivered by other means when opened in applications using the Apple Image I/O API"

Source: http://www.idownloadblog.com/2016/07/20/tiff-disabler/

M10-Extraneous Functionality

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: An attacker seeks to understand extraneous functionality within a mobile app in order to discover hidden functionality in in backend systems. The attacker will exploit extraneous functionality directly from their own systems without any involvement by end-users.

Exploitability: An attacker will download and examine the mobile app within their own local environment. They will examine log files, configuration files, and perhaps the binary itself to discover any hidden switches or test code that was left behind by the developers. They will exploit these switches and hidden functionality in the backend system to perform an attack.

Frequency: There is a high likelihood that any given mobile app contains extraneous functionality that is not directly exposed to the user via the interface. However, some extraneous functionality can be very useful to an attacker. Functionality that exposes information related to back-end test, demo, staging environments should not be included in a production build. Additionally, administrative API endpoints, or unofficial endpoints should not be included in final production builds

M10-Extraneous Functionality

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Detectability:. Detecting extraneous functionality can be tricky. Automated static and dynamic analysis tools can pick up low hanging fruit (log statements). However, some backdoors are difficult to detect in an automated means. As such, it is always best to prevent these things using a manual code review.

Technical Impact: The technical impact from extraneous functionality includes the following:

- Exposure of how backend systems work
- Unauthorized high-privileged actions executed

Business Impact: The business impact from extraneous functionality includes the following:

- Unauthorized Access to Sensitive Functionality
- Reputational Damage
- Intellectual Property Theft

M10-Extraneous Functionality

Am I Vulnerable ?

- Often, developers include hidden backdoor functionality or other internal development security controls that are not intended to be released into a production environment. For example, a developer may accidentally include a password as a comment in a hybrid app. Another example includes disabling of 2-factor authentication during testing.
- The defining characteristic of this risk is leaving functionality enabled in the app that was not intended to be released.

M10-Extraneous Functionality

How Do I Prevent?

The best way to prevent this vulnerability is to perform a manual secure code review using security champs or subject matter experts most knowledgeable with this code. They should do the following:

- Examine the app's configuration settings to discover any hidden switches
- Verify that all test code is not included in the final production build of the app
- Examine all API endpoints accessed by the mobile app to verify that these endpoints are well documented and publicly available
- Examine all log statements to ensure nothing overly descriptive about the backend is being written to the logs

M10-Extraneous Functionality

Examples

Scenario #1: Administrative Endpoint Exposed:

As part of mobile endpoint testing, developers included a hidden interface within the mobile app that would display an administrative dashboard. This dashboard accessed admin information via the back-end API server. In the production version of the code, the developers did not include code that displayed the dashboard at any time. However, they did include the underlying code that could access the back-end admin API. An attacker performed a string table analysis of the binary and discovered the hardcoded URL to an administrative REST endpoint. The attacker subsequently used 'curl' to execute back-end administrative functionality.

Scenario #2: Debug Flag in Configuration File:

An attacker tries manually added "debug=true" to a .properties file in a local app. Upon startup, the application is outputting log files that are overly descriptive and helpful to the attacker in understanding the backend systems. The attacker subsequently discovers vulnerabilities within the backend system as a result of the log.

Android Security: Tampering and Reverse engineering

Debugging Code

Dalvik and **ART** support the **JDWP**, **a protocol for communication between the debugger and the Java virtual machine (VM)** that it debugs.

JDWP is a standard debugging protocol that's supported by all command line tools and Java IDEs, including JDB, JEB, IntelliJ, and Eclipse.

Android's implementation of JDWP also includes hooks for supporting extra features implemented by the Dalvik Debug Monitor Server (DDMS).

A JDWP debugger allows you:

- to step through Java code,
- set breakpoints on Java methods,
- and inspect and modify local and instance variables.

You'll use a JDWP debugger most of the time you debug "normal" Android apps (i.e., apps that don't make many calls to native libraries).

TP 0014

How to solve the “UnCrackable App for Android Level 1” with JDB alone. Note that this is not an *efficient* way to solve this crackme, you can do it much faster with Frida and other methods, which we'll introduce later in the course. This, however, serves as an introduction to the capabilities of the Java debugger.

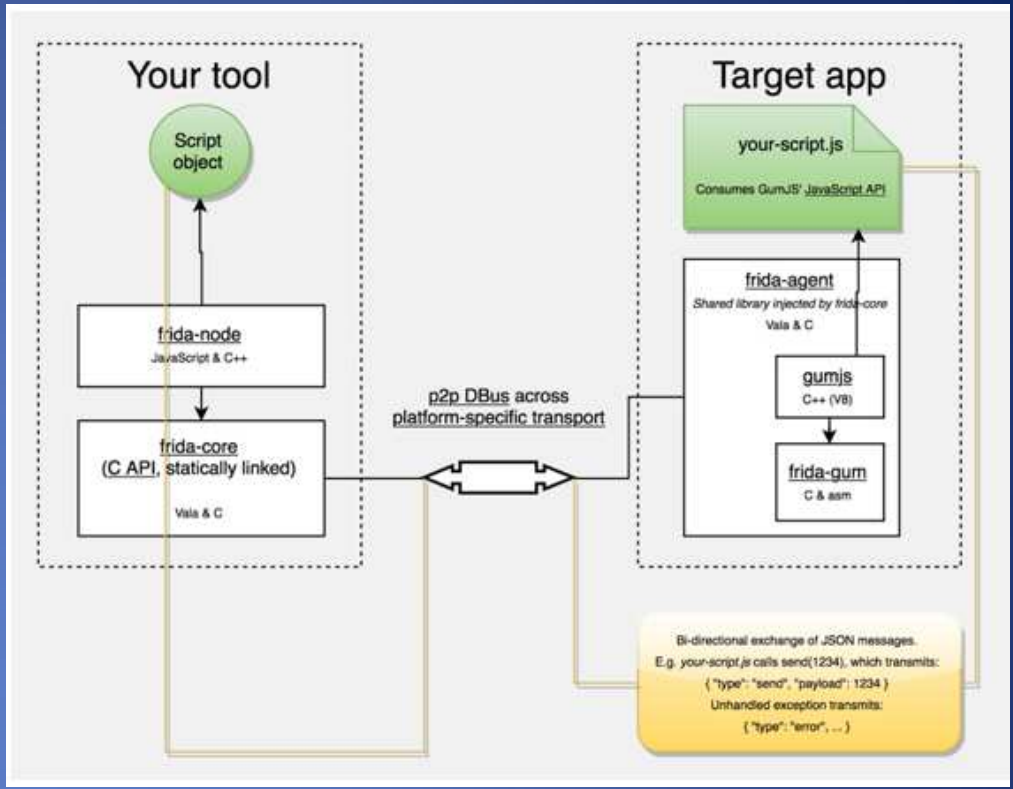
Using **UnCrackable-Level1.apk**

- Generate the code with apkx
- Embed the code in a real IDE (e.g. IDEA or Android Studio if you have one)
- Identify and play with application that are debuggable
- Make the app debuggable
- Use jdb to evade the anti-debug in memory
- Use jdb to get the decrypted key in memory

Android Security: Tampering and Reverse engineering

Debugging Code

Dynamic Instrumention with FRIDA (Javascript Injection)



1) When attached to a running app, Frida uses “ptrace” to hijack a thread of a running process.
2) This thread is used to allocate a chunk of memory and populate it with a mini-bootstrapper.
3) The bootstrapper starts a fresh thread, connects to the Frida debugging server that's running on the device, and loads a dynamically generated library file that contains the Frida agent and instrumentation code.
4) The hijacked thread resumes after being restored to its original state, and process execution continues as usual.
5) Frida injects a complete JavaScript runtime into the process, along with a powerful API that provides a lot of useful functionality, including calling and hooking native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime.

Android Security: Tampering and Reverse engineering

Debugging Code

Debugging native code

To disassemble the code, you can load ”libnative-lib.so” into any disassembler that understands ELF binaries (i.e., any disassembler).

If the app ships with binaries for different architectures, you can theoretically pick the architecture you're most familiar with, as long as it is compatible with the disassembler e.g. IDA Pro: compatible with ARM, MIPS, Java bytecode, Intel ELF (**Executable and Linkable Format**) binaries

TP 0010

Using **UnCrackable-Level1.apk**

- Frida usage to overload app methods in memory

Android Security: Tampering and Reverse engineering

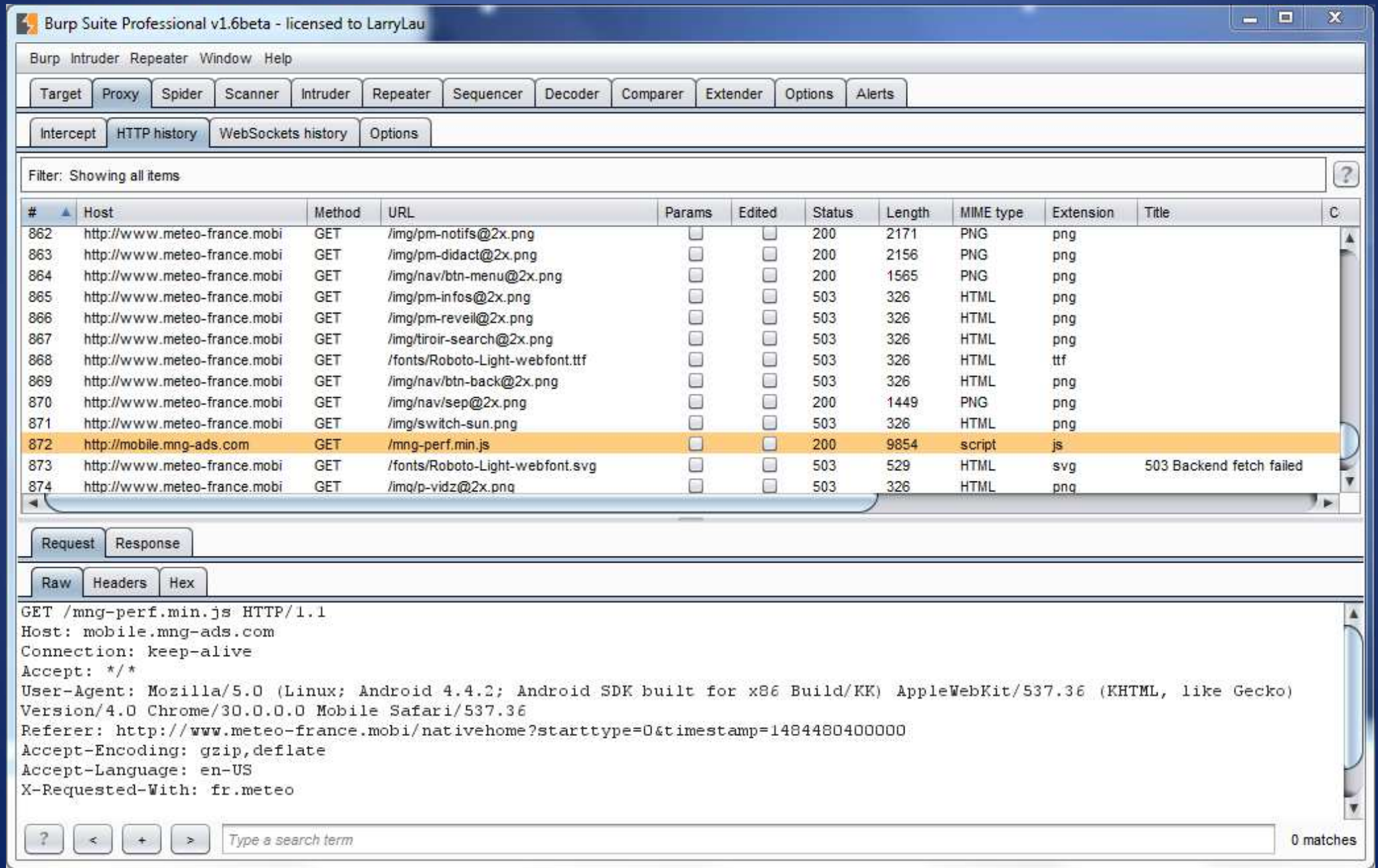
Runtime instrumentation: Patching and repacking

Disabling Certificate Pinning

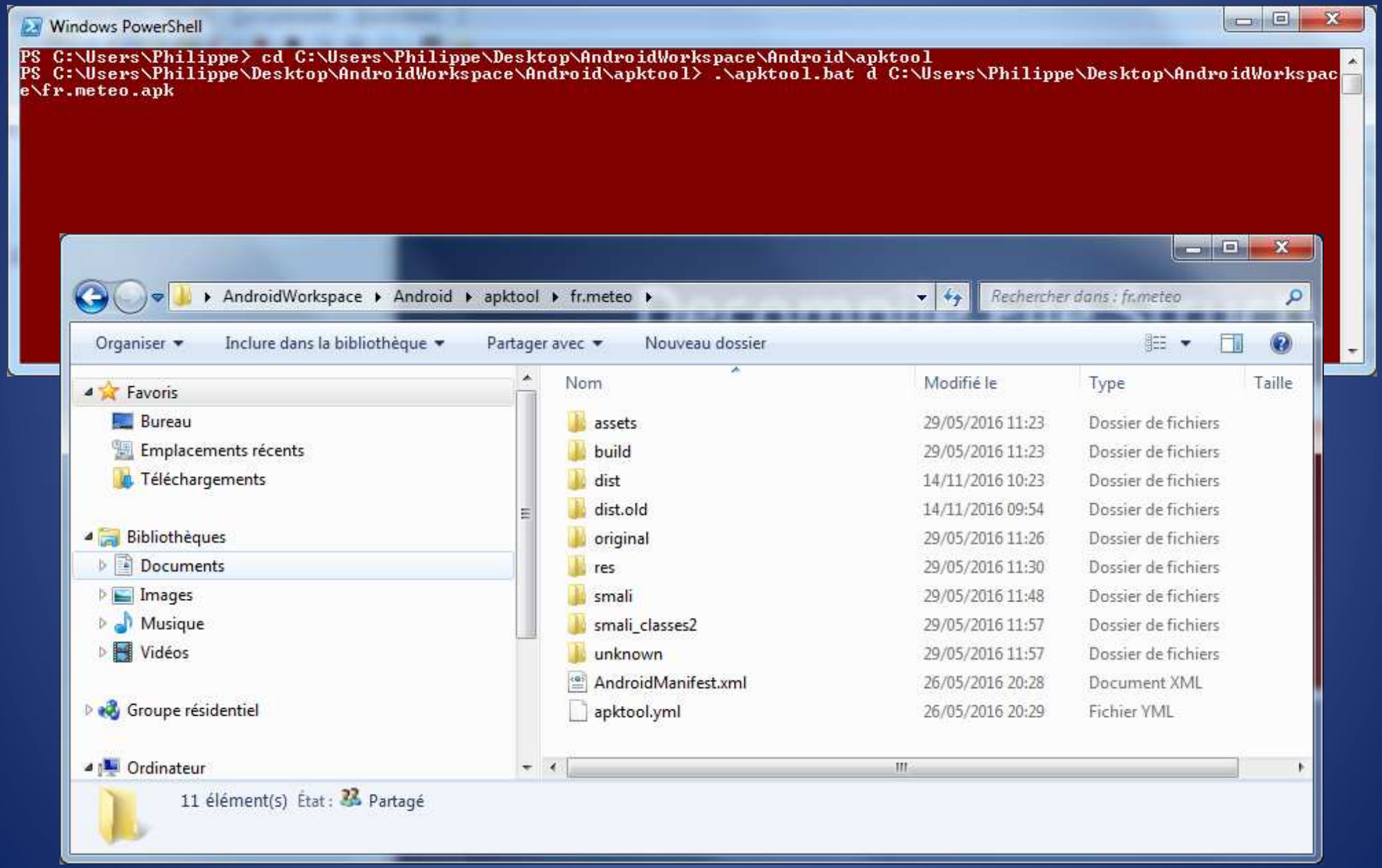
```
.method public checkServerTrusted([LJava/security/cert/X509Certificate;Ljava/lang/String;)V
  .locals 3
  .param p1, "chain" # [Ljava/security/cert/X509Certificate;
  .param p2, "authType" # Ljava/lang/String;
  .prologue
  return-void # <-- OUR INSERTED OPCODE!
  .line 102
  iget-object v1, p0, Lasdf/t$a;->a:Ljava/util/ArrayList;

  invoke-virtual {v1}, Ljava/util/ArrayList;->iterator()Ljava/util/Iterator;

  move-result-object v1

  :goto_0
  invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z
```

Meteo France

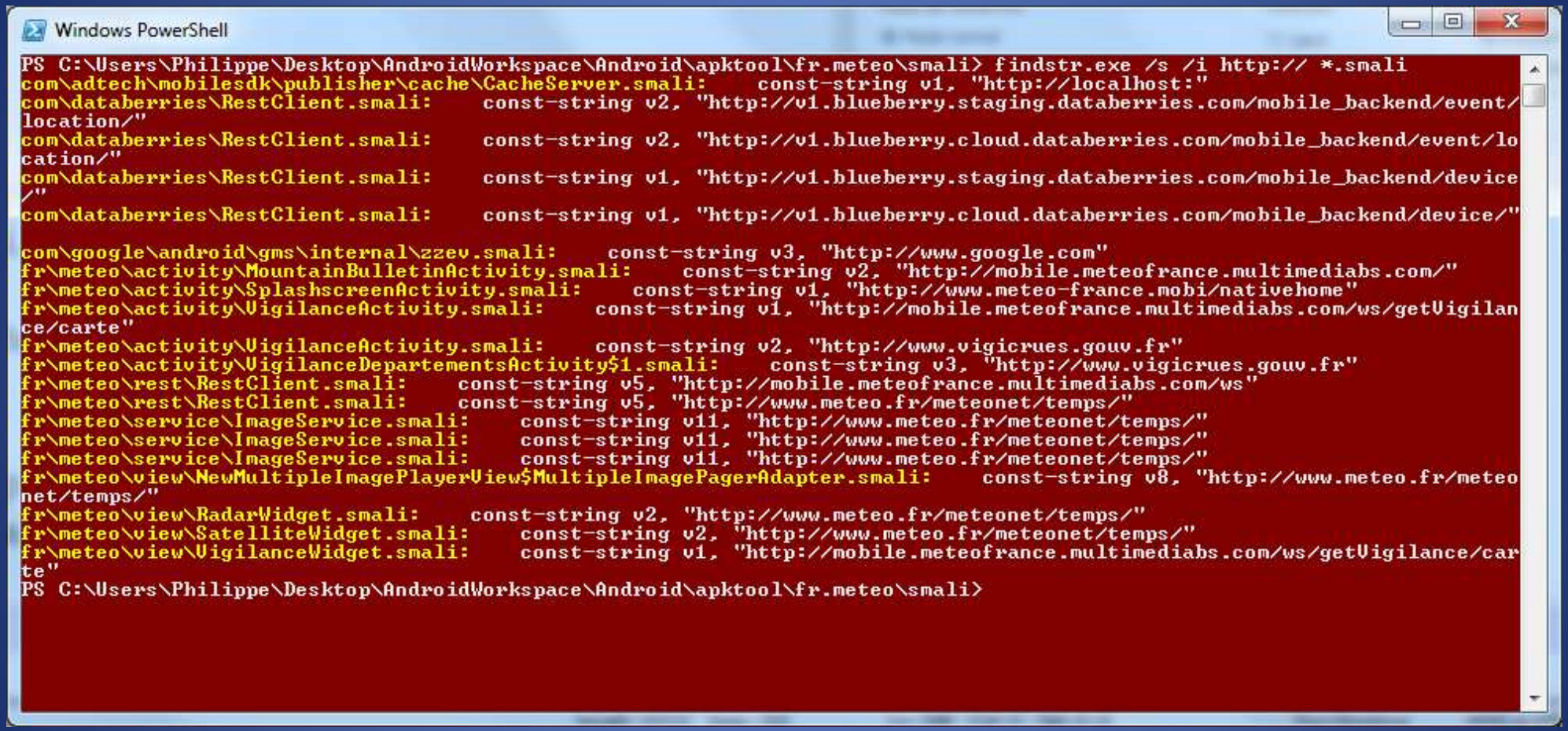
Observe fr.meteo



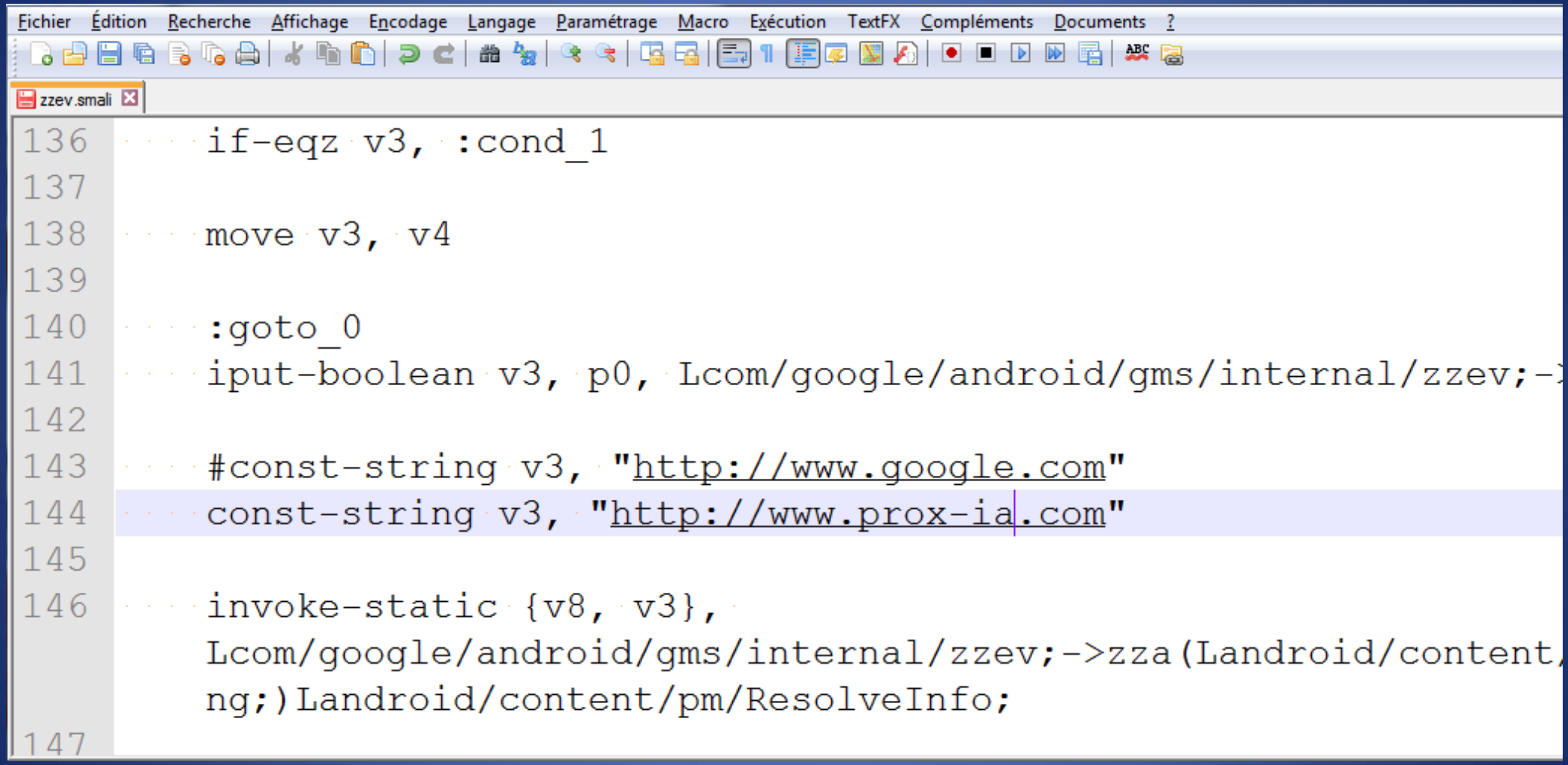
Decompile File Structure



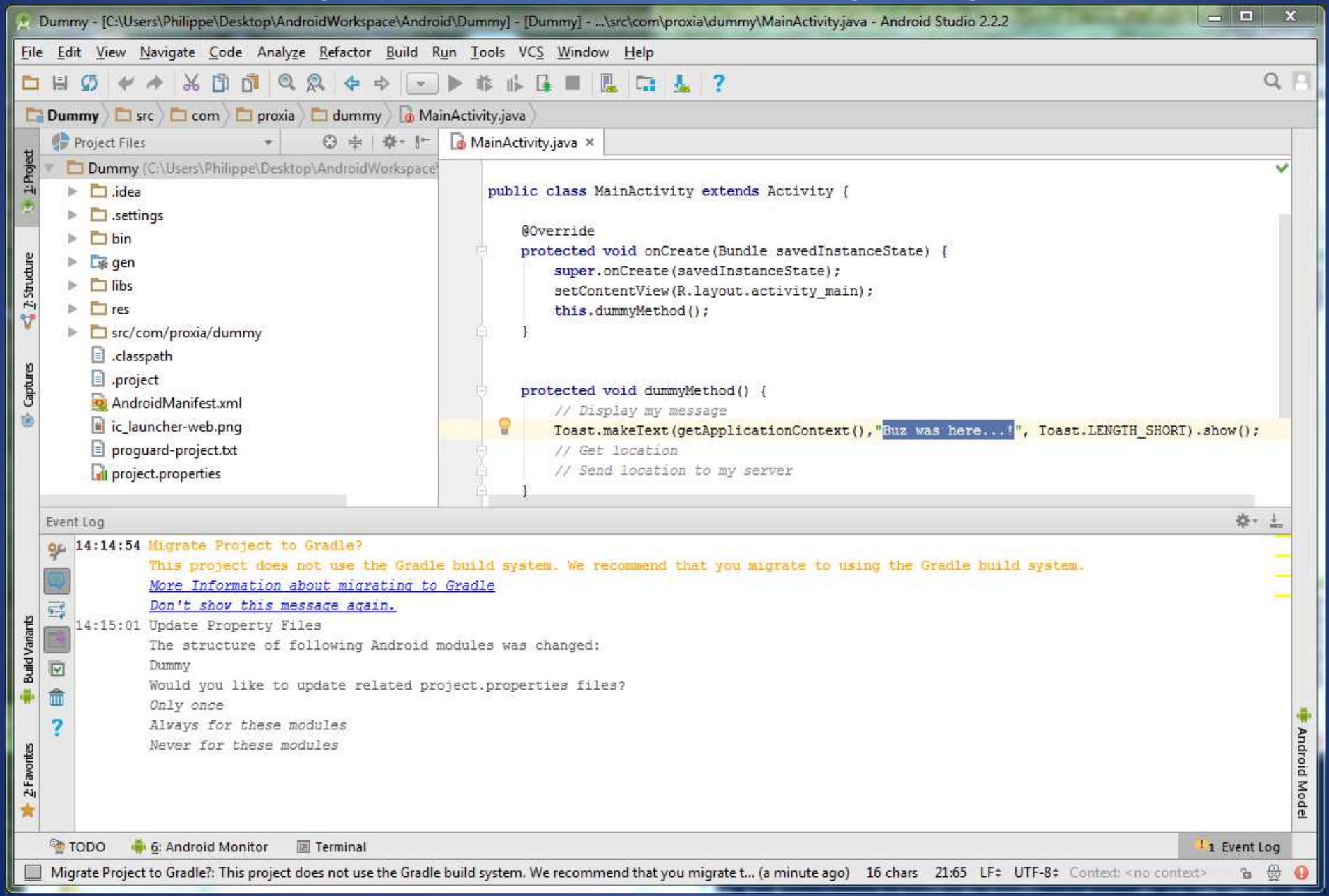
Find strings



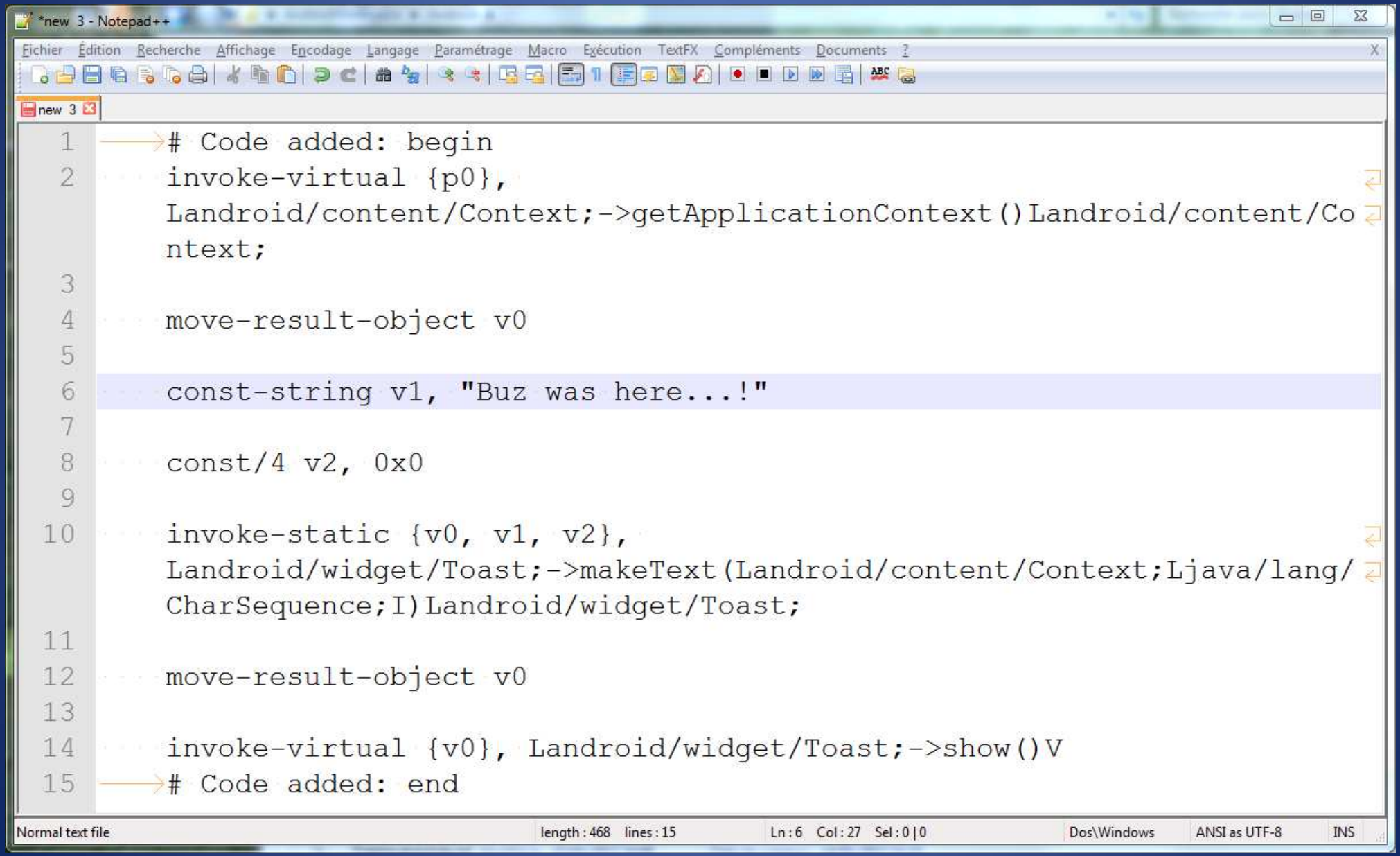
Edit bytecode



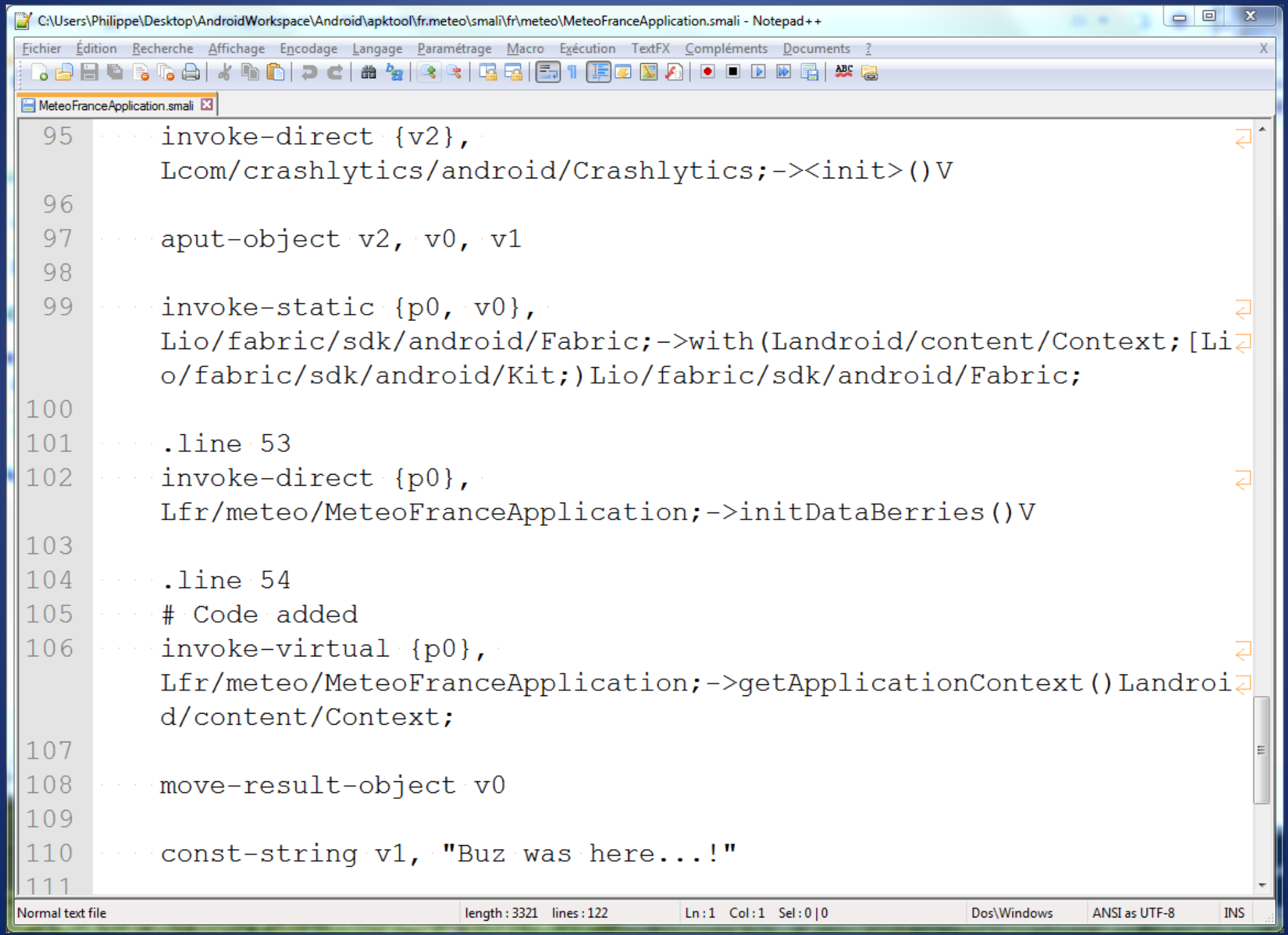
Payload Android project



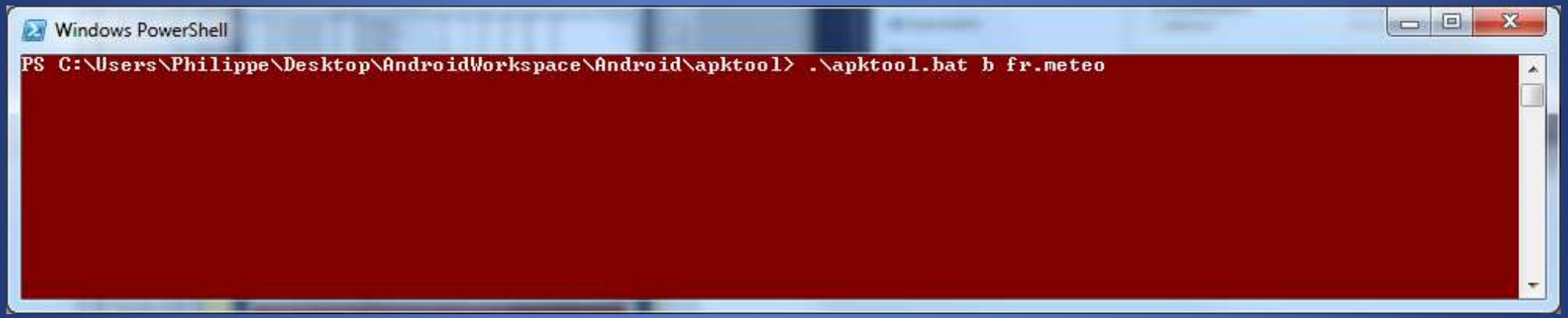
Corresponding bytecode



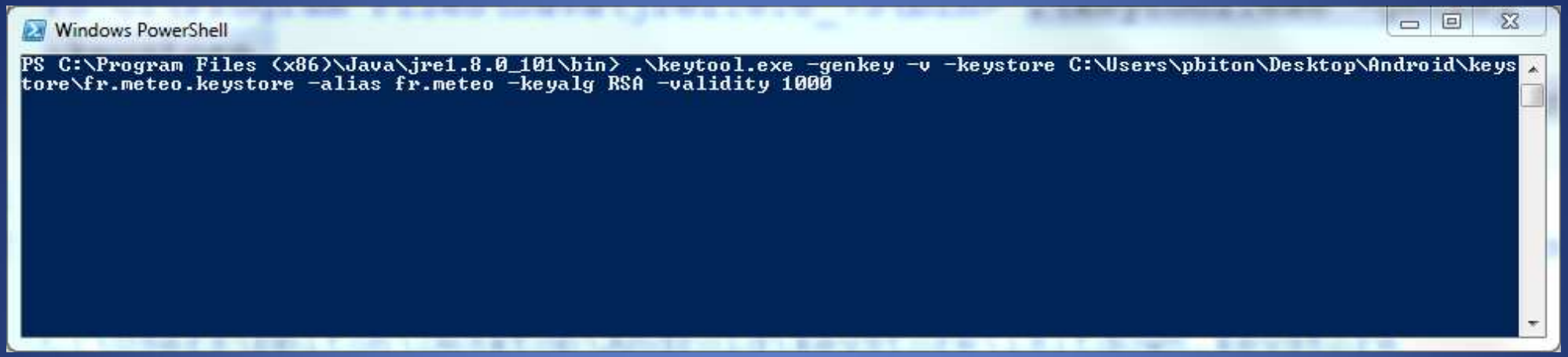
Bytecode added

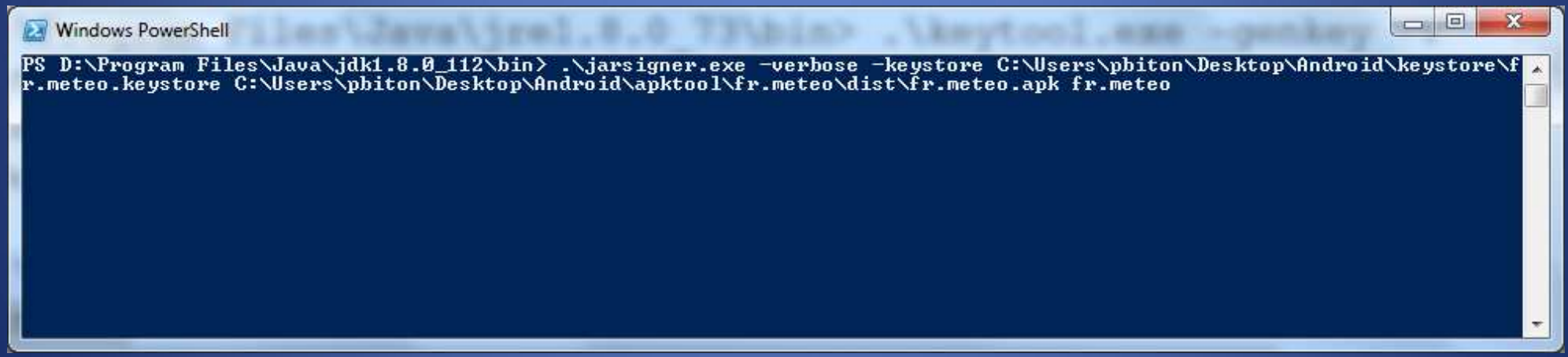


Recompile

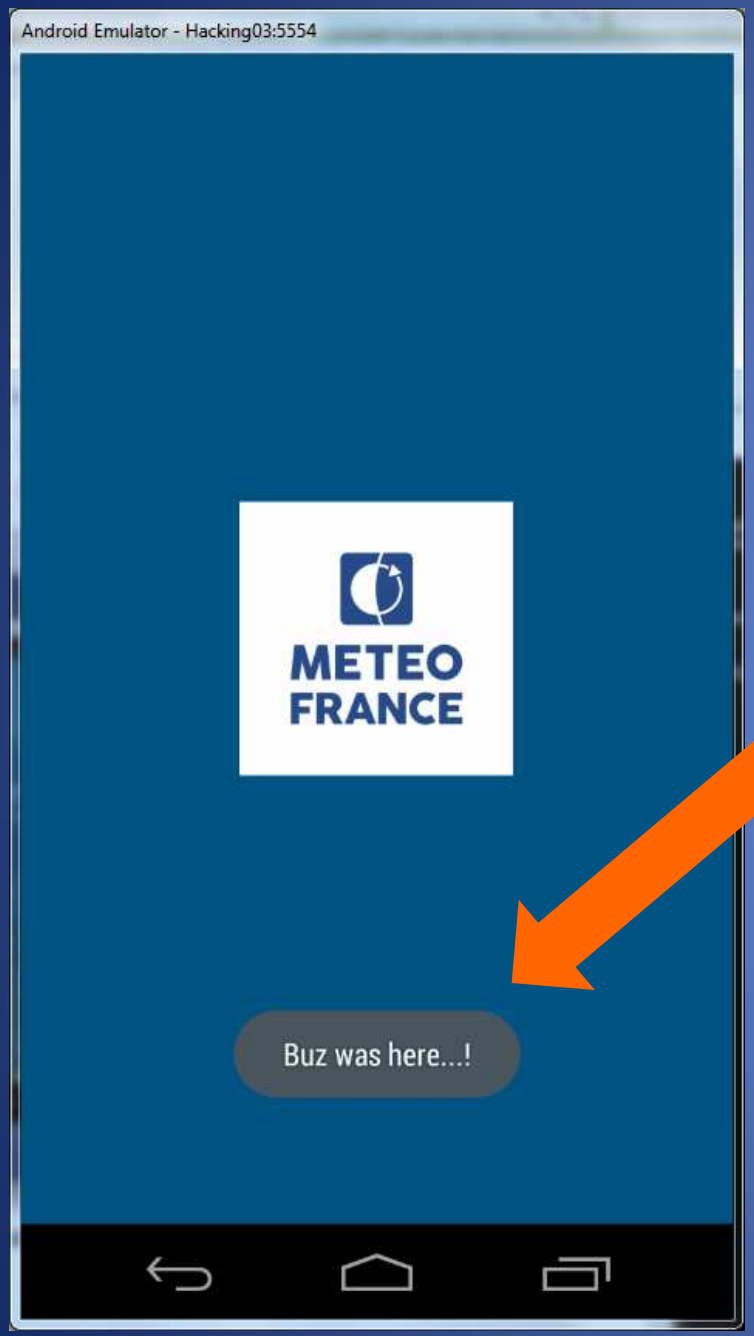

Windows PowerShell
PS C:\Users\Philippe\Desktop\AndroidWorkspace\Android\apktool> .\apktool.bat b fr.meteo

Generate key and sign





Test


Android Emulator - Hacking03:5554
METEO FRANCE
Buz was here...!


Android Emulator - Hacking03:5554
8:48
Aubagne
VIGILANCE JAUNE
Ensoleillé
8°
à 13H
8°
à 16H
dimanche
15 janv. 2017
13H
16H
19H
22H
1H lun.
4H
Prochains jours
dim. 15
lun. 16
mar. 17
mer. 18
jeu. 19
ven. 20

TP 0011

Using **UnCrackable-Level1.apk**

- Evade anti-rooting and anti-debugging by recompiling the app

M8-Code Tampering

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: Typically, an attacker will exploit code modification via malicious forms of the apps hosted in third-party app stores. The attacker may also trick the user into installing the app via phishing attacks.

Exploitability: An attacker will make direct binary changes to the application package's core binary, make direct binary changes to the resources within the application's package, redirect or replace system APIs to intercept and execute foreign code that is malicious

Frequency: Modified forms of applications are surprisingly more common than you think. There is an entire security industry built around detecting and removing unauthorized versions of mobile apps within app stores.

M8-Code Tampering

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Detectability: This category covers binary patching, local resource modification, method hooking, method swizzling (redirect a method to our own code at runtime in iOS Objective C), and dynamic memory modification.

Once the application is delivered to the mobile device, the code and data resources are resident there. An attacker can either directly modify the code, change the contents of memory dynamically, change or replace the system APIs that the application uses, or modify the application's data and resources..

Technical Impact: The impact from code modification can be wide ranging in nature, depending upon the nature of the modification itself. Typical types of impacts include the following:

- Unauthorized new features / Identity theft / Fraud

Business Impact: The business impact from code modification results in the following:

- Revenue loss due to piracy / Reputational damage.

M8-Code Tampering

Am I Vulnerable ?

- Technically, all mobile code is vulnerable to code tampering. Mobile code runs within an environment that is not under the control of the organization producing the code. At the same time, there are plenty of different ways of altering the environment in which that code runs. These changes allow an adversary to tinker with the code and modify it at will.
- Although mobile code is inherently vulnerable, it is important to ask yourself if it is worth detecting and trying to prevent unauthorized code modification. Apps written for certain business verticals (gaming for example) are much more vulnerable to the impacts of code modification than others (hospitality for example). As such, it is critical to consider the business impact before deciding whether or not to address this risk.

M8-Code Tampering

How Do I Prevent?

The mobile app must be able to detect at runtime that code has been added or changed from what it knows about its integrity at compile time. The app must be able to react appropriately at runtime to a code integrity violation.

Typically, an app that has been modified will execute within a Jailbroken or rooted environment. As such, it is reasonable to try and detect these types of compromised environments at runtime and react accordingly (report to the server or shutdown). There are a few common ways to detect a rooted Android device or Jailbroken iOS device:

- **Android Root Detection**
  - Check for test-keys
  - Check for OTA certificates
  - Check for several known rooted apk's
  - Check for SU binaries
  - Attempt SU command directly

- **iOS Jailbreak Detection**

M8-Code Tampering

Examples

There are a number of counterfeit applications that are available across the app stores. Some of these contain malware payloads. Many of the modified apps contain modified forms of the original core binary and associated resources. The attacker re-packages these as a new application and released them into third-party stores.

Scenario #1:

Games are a particularly popular target to attack using this method. The attacker will attract people that are not interested in paying for any freemium features of the game. Within the code, the attacker short-circuits conditional jumps that detect whether an in-application purchase is successful. This bypass allows the victim to attain game artifacts or new abilities without paying for them. The attacker has also inserted spyware that will steal the identity of the user.

Scenario #2:

Banking apps are another popular target to attack. These apps typically process sensitive information that will be useful to an attacker. An attacker could create a counterfeit version of the app that transmits the user's personally identifiable information (PII) along with username/password to a third-party site. This is reminiscent of the desktop equivalent of Zeus malware. This typically results in fraud against the bank.

Android Security: Anti-Reversing Defenses

Root Detection (anti-rooting)

Implemented through libraries

- RootBeer
- Safety (recommended by Google)

https://developers.google.com/android/reference/com/google/android/gms/safetynet/SafetyNet

File existence checks

- /system/app/Superuser.apk
- /system/etc/init.d/99SuperSUDaemon
- /dev/com.koushikdutta.superuser.daemon/
- /system/xbin/daemonsu
- /system/xbin/busybox
- /sbin/su
- /system/bin/su
- /system/xbin/su
- /data/local/su
- /data/local/xbin/su

Executing su and other commands

- Execute it through the Runtime.getRuntime.exec method.
- An IOException will be thrown if su is not on the PATH.

Checking for writable partitions and system directories

- Unusual permissions on system directories may indicate a customized or rooted device.
- System and data directories are normally mounted read-only

Android Security: Anti-Reversing Defenses

Root Detection (anti-rooting)

Checking installed app packages

- com.thirdparty.superuser
- eu.chainfire.supersu
- com.noshufou.android.su
- com.koushikdutta.superuser
- com.zachspong.temprootremovejb
- com.ramdroid.appquarantine

Checking for custom Android builds

Checking for signs of test builds and custom ROMs is also helpful.
One way to do this is to check the BUILD tag for test-keys

Missing Google Over-The-Air (OTA) certificates is another sign of a custom ROM

Android Security: Anti-Reversing Defenses

Rooting tools

To root a mobile device, first unlock its boot loader.

The unlocking procedure depends on the device manufacturer.

However, for practical reasons, rooting some mobile devices is more popular than rooting others, particularly when it comes to security testing: devices created by Google and manufactured by companies like Samsung, LG, and Motorola are among the most popular.

Particularly because they are used by many developers.

The **device warranty is not nullified when the boot loader is unlocked** and Google provides many tools to support the root itself.

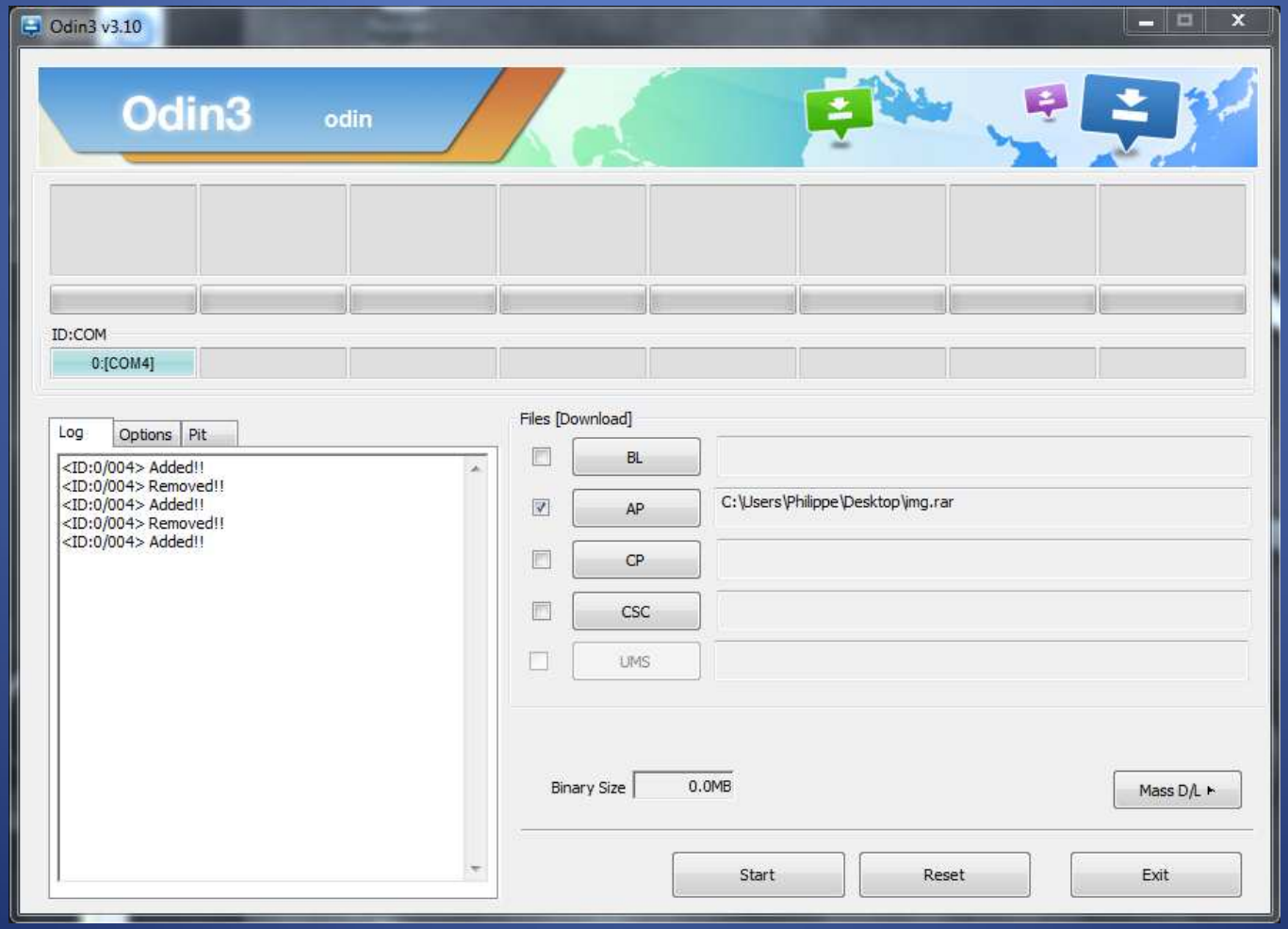
A curated list of guides for rooting all major brand devices is posted on the **XDA forums**.

https://www.xda-developers.com/root/

Android Security: Anti-Reversing Defenses

Rooting tools (for Samsung A7: 2018)

1. Boot into **download** mode
2. Use latest **Odin** and **flash** provided **img.tar** file in AP slot
3. Device will reboot and will say verification failed. Then tap on reset: device will reboot
5. After reset setup device you will find “magisk manager 17.3” in app drawer: open it and install full version. It will ask for additional setup: allow it



DEMO 0012

Rooting of real device

M2-Insecure Data Storage

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Who: an adversary that has attained a lost/stolen mobile device; malware or another repackaged app acting on the adversary's behalf that executes on the mobile device

Exploitability: if an adversary physically attains the mobile device, the adversary hooks up the mobile device to a computer with freely available software. These tools allow the adversary to see all third party application directories that often contain stored personally identifiable information (PII) or other sensitive information assets. An adversary may construct malware or modify a legitimate app to steal such information assets

Frequency / Detectability: Insecure data storage vulnerabilities occur when development teams assume that users or malware will not have access to a mobile device's filesystem and subsequent sensitive information in data-stores on the device. Filesystems are easily accessible. Organizations should expect a malicious user or malware to inspect sensitive data stores. Usage of poor encryption libraries is to be avoided. Rooting or jailbreaking a mobile device circumvents any encryption protections. When data is not protected properly, specialized tools are all that is needed to view application data

M2-Insecure Data Storage

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Technical Impact: This can result in data loss, in the best case for one user, and in the worst case for many users. It may also result in the following technical impacts: extraction of the app's sensitive information via mobile malware, modified apps or forensic tools.

Business Impact: The nature of the business impact is highly dependent upon the nature of the information stolen. Insecure data may result in the following business impacts:

- Identity theft
- Privacy violation
- Fraud
- Reputation damage
- External policy violation (PCI)
- Material loss

M2-Insecure Data Storage

Am I Vulnerable ?

Data stored insecurely includes:

- SQL databases / Log files / XML data stores or manifest files
- Binary data stores / Cookie stores
- SD card / Cloud synced

Unintended data leakage includes vulnerabilities from:

- OS / Frameworks
- Compiler environment / New hardware

In mobile development specifically, this is most seen in undocumented, or under-documented, internal processes such as:

- The way the OS caches data, images, key-presses, logging, and buffers
- The way the development framework caches data, images, key-presses, logging, and buffers
- The way or amount of data ad, analytic, social, or enablement frameworks cache data, images, key-presses, logging, and buffers

M2-Insecure Data Storage

How Do I Prevent?

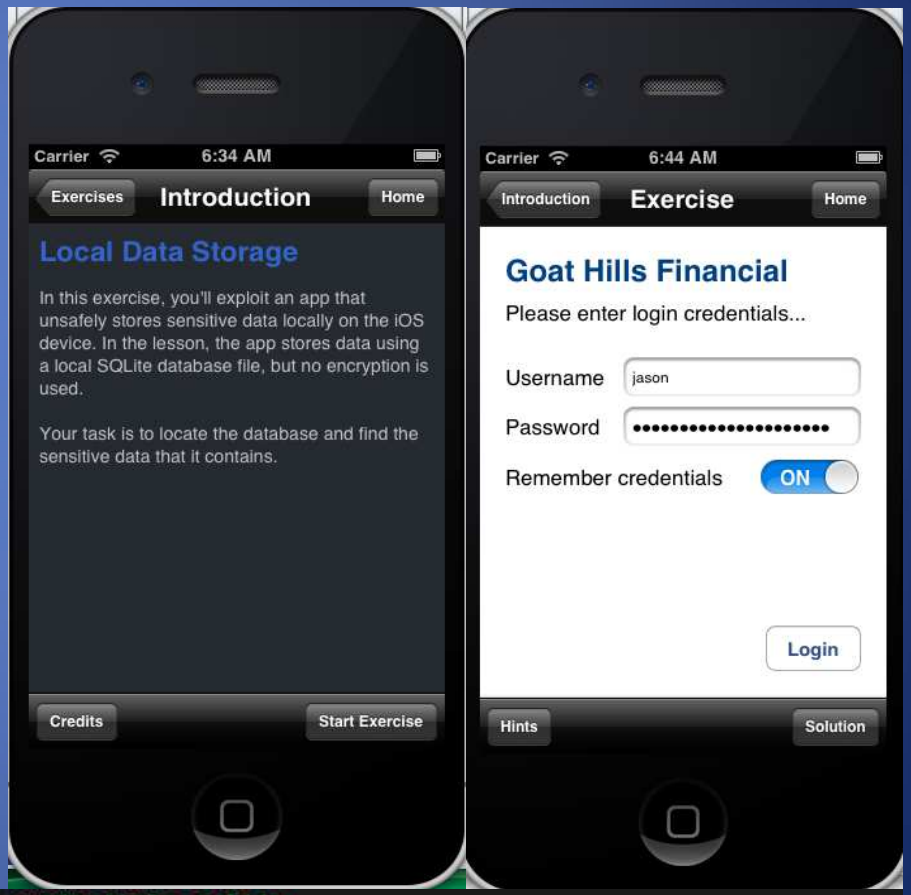
It is important to threat model your mobile app, OS, platforms and frameworks to understand the information assets the app processes and how the APIs handle those assets. It is crucial to see how they handle the following types of features :

- URL caching (both request and response)
- Keyboard press caching
- Copy/Paste buffer caching
- Application backgrounding
- Intermediate data
- Logging
- HTML5 data storage
- Browser cookie objects
- Analytics data sent to 3rd parties

M2-Insecure Data Storage

Examples

iGoat is a purposefully vulnerable mobile app for the security community to explore these types of vulnerabilities first hand. In the exercise below, we enter our credentials and log in to the fake bank app. Then, we navigate to the file system. Within the applications directory, we can see a database called “credentials.sqlite”. Exploring this database reveals that the application is storing our username and credentials (Jason:pleasedontstoremebro!) in plain text.




```
mac:Documents haddix$ strings credentials.sqlite
SQLite format 3
Ytablesqlite_sequencesqlite_sequence
CREATE TABLE sqlite_sequence(name,seq)n
;tablecredscreds
CREATE TABLE creds (id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT, password TEXT)
'jasonpleasedontstoremebro!
```

Android Security: Anti-Reversing Defenses

Anti-Debugging

Debugging is a highly effective way to analyze run-time app behavior. It allows the reverse engineer to step through the code, stop app execution at arbitrary points, inspect the state of variables, read and modify memory, and a lot more.

We have to deal with two debugging protocols on Android: we can debug on the Java level with **JDWP** or on the **native layer**

Anti-debugging features can be preventive or reactive:

- **Prevents the debugger from attaching** in the first place
- **Detecting debuggers and reacting to them** in some way (e.g., terminating the app or triggering hidden behavior).

The "more-is-better" rule applies

Anti-JDWP-Debugging

- **Checking the Debuggable Flag in ApplicationInfo**
- **isDebuggerConnected method from** the Android Debug system class
- **Timer Checks:** Debug.threadCpuTimeNanos indicates the amount of time that the current thread has been executing code. Because debugging slows down process execution, you can use the difference in execution time to guess whether a debugger is attached

You can disable debugging by using similar techniques in ART

Android Security: Anti-Reversing Defenses

Anti-Debugging

Anti-Native-Debugging

Most Anti-JDWP tricks (which may be safe for timer-based checks) won't catch classical, ptrace-based debuggers, so other defenses are necessary.

Many "traditional" Linux anti-debugging tricks are used in this situation.

Checking TracerPid

When the ptrace system call is used to attach to a process, the "**TracerPid**" field in the status file of the debugged process shows the PID of the attaching process.

The default value of "TracerPid" is 0 (no process attached). Consequently**, finding anything other than 0 in that field is a sign of debugging**

Ptrace variations

On Linux, the ptrace system call is used to observe and control the execution of a process (the "tracee") and to examine and change that process' memory and registers.

ptrace is the primary way to implement breakpoint debugging and system call tracing.

Many anti-debugging tricks include ptrace, often exploiting the fact that only one debugger at a time can attach to a process.

You can prevent debugging of a process by **forking a child process and attaching it to the parent as a debugger**

Android Security: Anti-Reversing Defenses

ptrace

Linux provides **ptrace** as a process tracing tool, which can intercept system calls at their entry and exit points, made from another process.

ptrace provides a mechanism by which a parent process may observe and control the execution of another process.

It can examine and change a child process's core image and registers, and is used primarily to implement breakpoint debugging and system call tracing.

ptrace takes the following arguments,

long ptrace(enum __ptrace_request request,

pid_t pid,

*void *addr,*

*void *data);*

where:

request = type of behavior of ptrace. For example, we can attach or detach from a process, read/write registers, read/write code segment and data segment.

pid = process id of the traced process

addr = address

data = data

TP 0017

Using **UnCrackable-Level1.apk**

- Execute code in a jdb session to have a remote shell on the app

M9-Reverse Engineering

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific

Who: An attacker will typically download the targeted app from an app store and analyze it within their own local environment using a suite of different tools.

Exploitability: An attacker must perform an analysis of the final core binary to determine its original string table, source code, libraries, algorithms, and resources embedded within the app. Attackers will use relatively affordable and well-understood tools like IDA Pro, Hopper, otool, strings, and other binary inspection tools from within the attacker's environment.

Frequency: Generally, all mobile code is susceptible to reverse engineering. Some apps are more susceptible than others. Code written in languages / frameworks that allow for dynamic introspection at runtime (Java, .NET, Objective C, Swift) are particularly at risk for reverse engineering.

M9-Reverse Engineering

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific

Detectability: Detecting susceptibility to reverse engineering is fairly straight forward. First, decrypt the app store version of the app (if binary encryption is applied). Then, use the tools outlined in the "Attack Vectors" section of this document against the binary. Code will be susceptible if it is fairly easy to understand the app's control flow path, string table, and any pseudocode/source-code generated by these tools.

Technical Impact: An attacker may exploit reverse engineering to achieve any of the following:

- Reveal information about back end servers or cryptographic constants and ciphers
- Steal intellectual property / Perform attacks against back end systems
- Gain intelligence needed to perform subsequent code modification

Business Impact: The business impacts from reverse engineering are quite varied. They include the following:

- Intellectual Property theft / Identity Theft
- Compromise of Backend Systems / Reputational Damage

M9-Reverse Engineering

Am I Vulnerable ?

Generally, most applications are susceptible to reverse engineering due to the inherent nature of code. Most languages used to write apps today are rich in metadata that greatly aides a programmer in debugging the app. This same capability also greatly aides an attacker in understanding how the app works.

An app is said to be susceptible to reverse engineering if an attacker can do any of the following things:

- Clearly understand the contents of a binary's string table
- Accurately perform cross-functional analysis
- Derive a reasonably accurate recreation of the source code from the binary
- Although most apps are susceptible to reverse engineering, it's important to examine the potential business impact of reverse engineering when considering whether or not to mitigate this risk.

M9-Reverse Engineering

How Do I Prevent?

In order to prevent effective reverse engineering, you must use an obfuscation tool. There are many free and commercial grade obfuscators on the market. Conversely, there are many different deobfuscators on the market. To measure the effectiveness of whatever obfuscation tool you choose, try deobfuscating the code using tools like IDA Pro and Hopper.

A good obfuscator will have the following abilities:

- Narrow down what methods / code segments to obfuscate
- Tune the degree of obfuscation to balance performance impact
- Withstand de-obfuscation from tools like IDA Pro and Hopper
- Obfuscate string tables as well as methods

M9-Reverse Engineering

Examples

- **Scenario #1:** String Table Analysis:

The attacker runs 'strings' against the unencrypted app. As a result of the string table analysis, the attacker discovers a hardcoded connectivity string that contains authentication credentials to a backend database. The attacker uses those credentials to gain access to the database. The attacker steals a vast array of PII data about the app's users.

- **Scenario #2:** Cross-Functional Analysis:

The attacker uses IDA Pro against an unencrypted app. As a result of the string table analysis combined with functional cross-referencing, the attacker discovers Jailbreak detection code. The attacker uses this knowledge in a subsequent code-modification attack to disable jailbreak detection within the mobile app. The attacker then deploys a version of the app that exploits method swizzling to steal customer information.

- **Scenario #3:** Source Code Analysis:

The banking Android application APK file can be easily extracted using 7zip/Winrar/WinZip/Gunzip. Once extracted, the attacker has manifest file, assets, resources and most importantly classes.dex file. Then using Dex to Jar converter, an attacker can easily convert it to jar file. In next step, Java Decompiler (like JDgui) will provide you the code.

Android Security: Anti-Reversing Defenses

File Integrity Checks

App integrity checks

Reverse engineers can easily bypass this check by re-packaging and re-signing an app.

CRC checks can be done on the app byte-code, native libraries, and important data files.

These checks can be implemented on both the Java and the native layer.

File storage integrity checks

Files that the application stores on the SD card or public storage and the integrity of key-value pairs that are stored in SharedPreferences should be protected.

Application Source Code

Integrity checks often calculate a checksum or hash over selected files. Commonly protected files include

- AndroidManifest.xml,
- class files *.dex,
- native libraries (*.so).

Storage

You can either create an HMAC over a given key-value pair (as for the Android SharedPreferences) or create an HMAC over a complete file that's provided by the file system.

Android Security: Anti-Reversing Defenses

Anti-reversing

Reverse engineering refers to methods of analysing a compiled program **without** access to its source code

Here are some APIs **FRIDA** offers on Android:

- Instantiate Java objects and call static and non-static class methods
- **Replace Java method implementations**
- Enumerate live instances of specific classes by scanning the Java heap (Dalvik only)
- Scan process memory for occurrences of a string
- Intercept native function calls to run your own code at function entry and exit (or return fake values)

Detection Methods

An obvious way to detect Frida and similar frameworks is to check the environment for related artifacts, such as package files, binaries, libraries, processes, and temporary files. As an example, look for **frida-server**, the daemon responsible for exposing Frida over TCP.

Android Security: Anti-Reversing Defenses

Emulator Detection (Anti-reversing context)

The goal of emulator detection is to increase the difficulty of running the app on an emulated device, which **impedes some tools and techniques reverse engineers like to use**.

This increased difficulty forces the reverse **engineer to defeat the emulator checks or utilize the physical device**, thereby barring the access required for **large-scale device analysis**.

There are several indicators that the device in question is being emulated. Although all these API calls can be hooked, these indicators provide a modest first line of defense.

The first set of indicators are in the file **/system/build.prop => go and check in emulator**

API Method Value Meaning

Build.ABI armeabi possibly emulator
BUILD.ABI2 unknown possibly emulator
Build.BOARD unknown emulator
Build.Brand generic emulator
Build.DEVICE generic emulator
Build.FINGERPRINT generic emulator
Build.Hardware goldfish emulator
Build.Host android-test possibly emulator

Build.ID FRF91 emulator
Build.MANUFACTURER unknown emulator
Build.MODEL sdk emulator
Build.PRODUCT sdk emulator
Build.RADIO unknown possibly emulator
Build.SERIAL null emulator
Build.TAGS test-keys emulator
Build.USER android-build emulator

You can edit the file **build.prop** on a rooted Android device or modify it while compiling AOSP from source. Both techniques will allow you to bypass the static string checks above

Android Security: Anti-Reversing Defenses

Emulator Detection (Anti-reversing context)

The next set of static indicators utilize the **Telephony manager**. All Android emulators have fixed values that this API can query.

API Value Meaning :

TelephonyManager.getDeviceId() **0**'s emulator

TelephonyManager.getLine1 Number() **155552155** emulator
TelephonyManager.getNetworkCountryIso() **us** possibly emulator
TelephonyManager.getNetworkType() **3** possibly emulator
TelephonyManager.getNetworkOperator().substring(0,3) **310** possibly emulator
TelephonyManager.getNetworkOperator().substring(3) **260** possibly emulator
TelephonyManager.getPhoneType() **1** possibly emulator
TelephonyManager.getSimCountryIso() **us** possibly emulator

TelephonyManager.getSimSerial Number() **89014103211118510720** emulator

TelephonyManager.getSubscriberId() **310260000000000** emulator
TelephonyManager.getVoiceMailNumber() **15552175049** emulator

Keep in mind that a hooking framework, such as Xposed or Frida, can hook this API to provide false data.

Android Security: Anti-Reversing Defenses

Runtime Integrity Checks

Controls in this category verify the **integrity of the app's memory space** to defend the app against memory patches applied during run time.

Such patches include unwanted changes to binary code, byte-code, function pointer tables, and important data structures, as well as rogue code loaded into process memory.

Integrity can be verified by

- comparing the contents of memory or a checksum over the contents to good values,
- searching memory for the **signatures** of unwanted modifications

Detecting tampering with the Java Runtime

```
if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge")
        && stackTraceElement.getMethodName().equals("handleHookedMethod")) {
    Log.wtf("HookDetection", "A method on the stack trace has been hooked using Xposed.");
}
```

Detecting Native Hooks

- Native function hooks can be installed by overwriting function pointers in memory (e.g., Global Offset Table or PLT hooking) **. Check entry points to a legitimately loaded library**
- Inline hooks work by overwriting a few instructions at the beginning or end of the function code. **Check jumps to locations outside the library**

Android Security: Anti-Reversing Defenses

Device Binding

The goal of device binding is to impede an attacker who tries to both copy an app and its state from device A to device B and continue executing the app on device B.

How identifiers can be used for binding?

There are three methods that allow device binding:

- Augmenting the credentials used for **authentication with device identifiers**. This make sense if the application needs to re-authenticate itself and/or the user frequently.
- Obfuscating the data stored on the device by **using device identifiers as keys for encryption methods**. This can help with binding to a device when the app does a lot of offline work or when access to APIs depends on access-tokens stored by the application.
- Use **token-based device authentication (Instance ID)** to make sure that the same instance of the app is used.

Identifiers

Be cautious with data privacy rules

- **Google Instance ID:** If the application is reset, uninstalled,...the Instance ID is reset
- **IMEI & Serial (*)**
- **SSAID (*) -** Settings.Secure.ANDROID_ID;

(*) : Google recommends not using these identifiers unless the application is at a high risk.

Android Security: Anti-Reversing Defenses

Obfuscation

Obfuscation is the process of transforming code and data to make it more difficult to comprehend.

It is an integral part of every software protection scheme.

What's important to understand is that obfuscation isn't something that can be simply turned on or off.

Programs can be made incomprehensible, in whole or in part, in many ways and to different degrees.

ProGuard is a free Java class file shrinker, optimizer, obfuscator, and pre-verifier. It is shipped with Android's SDK tools.

Checklist:

- meaningful identifiers, such as class names, method names, and variable names, have been discarded,
- string resources and strings in binaries are encrypted,
- code and data related to the protected functionality is encrypted, packed, or otherwise concealed.

iOS Security

1. Platform Overview
2. Setting up a Testing Environment for iOS Apps
3. Data Storage on iOS
4. iOS Cryptographic APIs
5. Local Authentication on iOS
6. iOS Network APIs
7. iOS Platform APIs
8. Code Quality and Build Settings for iOS Apps
9. Tampering and Reverse Engineering on iOS

iOS Security

Overview

- iOS apps are isolated from each other at the file system level and are significantly limited in terms of system API access.
- Apple restricts and controls access to the apps that are allowed to run on iOS devices. **Apple's App Store** is the only official application distribution platform
- In the past **sideloading** was possible only with a jailbreak or complicated workarounds. With iOS 9 or higher, it is possible to sideload via Xcode
- iOS apps are isolated from each other via **Apple's iOS sandbox** (historically called Seatbelt), a mandatory access control (MAC) mechanism describing the resources an app can and can't access
- iOS offers very **few IPC** (Inter Process Communication) options, minimizing the potential attack surface
- iOS updates are usually quickly rolled out to a large percentage of users, decreasing the need to support older, unprotected iOS versions.

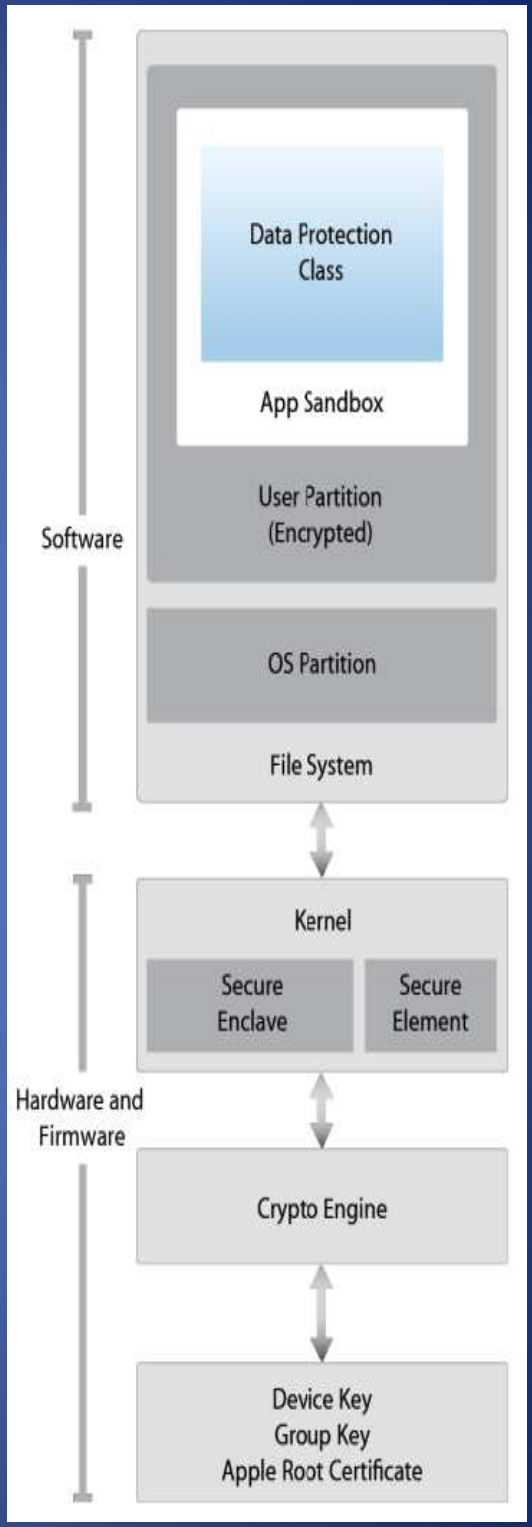
iOS Security

Overview

The **iOS security architecture** consists of six core features:

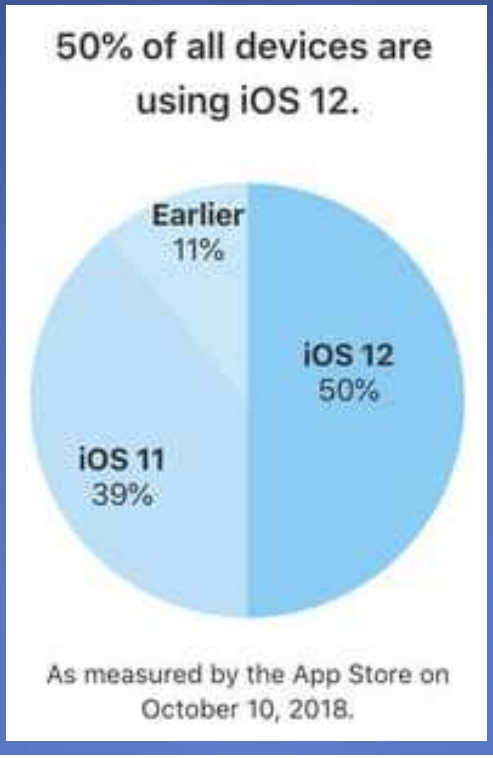
- Hardware Security (keychain)
- Secure Boot
- Code Signing
- Sandbox
- Encryption and Data Protection (file system encryption)

« Data protection, Keychain, Touch ID/Face ID authentication, and network security still leave a large margin for errors »



iOS Security

Overview



Current versions

Version	Build	Processor support	Application support	Kernel	Release date	Device end-of-life		
						iPad	iPhone	iPod Touch
3.1.3	7E18	32-bit ARM			Feb 2, 2010	N/A	1st gen	1
4.2.1	8C148				Nov 22, 2010		3G	2
5.1.1	9B206				May 7, 2012	1st gen	N/A	3
6.1.6	10B500				Feb 21, 2014	N/A	3GS	4
7.1.2	11D257	32/64-bit ARM[1][2]			Jun 30, 2014		4	N/A
9.3.5	13G36				Aug 25, 2016	2, 3, Mini 1	4S	5
10.3.3	14G60				Jul 19, 2017	4	5, 5C	N/A
12.1.3	16D39/16D40	64-bit ARM[3]			Jan 22, 2019	N/A		
12.2 Beta 1	16E5181f				Jan 24, 2019			

Legend: Discontinued Current Beta

iOS Security

Hardware Security

- 2 built-in Advanced Encryption Standard (AES) 256-bit keys
  - UIDs and GID are AES 256-bit keys
  - Stored into the Application Processor (AP) and Secure Enclave Processor (SEP) during manufacturing
- **No direct way to read these keys** with software or debugging interfaces
- Encryption and decryption operations are performed by **hardware AES crypto-engines** that have exclusive access to these keys
- GID is a value **shared by all processors in a class of devices** used to prevent tampering with firmware files and other cryptographic tasks not directly related to the user's private data
- UIDs are unique to each device, are **used to protect the key hierarchy** that's used for device-level file system encryption
- Because UIDs aren't recorded during manufacturing, not **even Apple can restore the file encryption keys for a particular device**

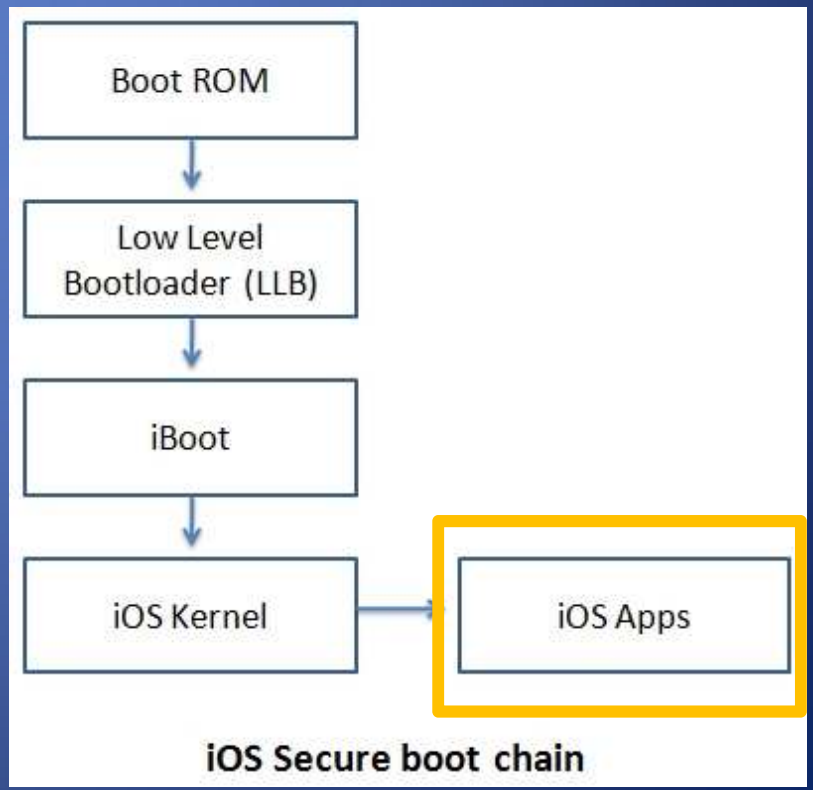
https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

iOS Security

Secure Boot

When an iOS device is powered on, it reads the initial instructions from the **read-only memory** known as Boot ROM, which bootstraps the system. The Boot ROM contains **immutable code** and the **Apple Root CA**, which is etched into the silicon chip during the fabrication process, thereby creating the **root of trust**.

- Each step ensure the next step is **signed** by Apple
- This entire process is called the "**Secure Boot Chain**".
- Its purpose is focused on verifying the boot process **integrity**



iOS Secure boot chain

iOS Security

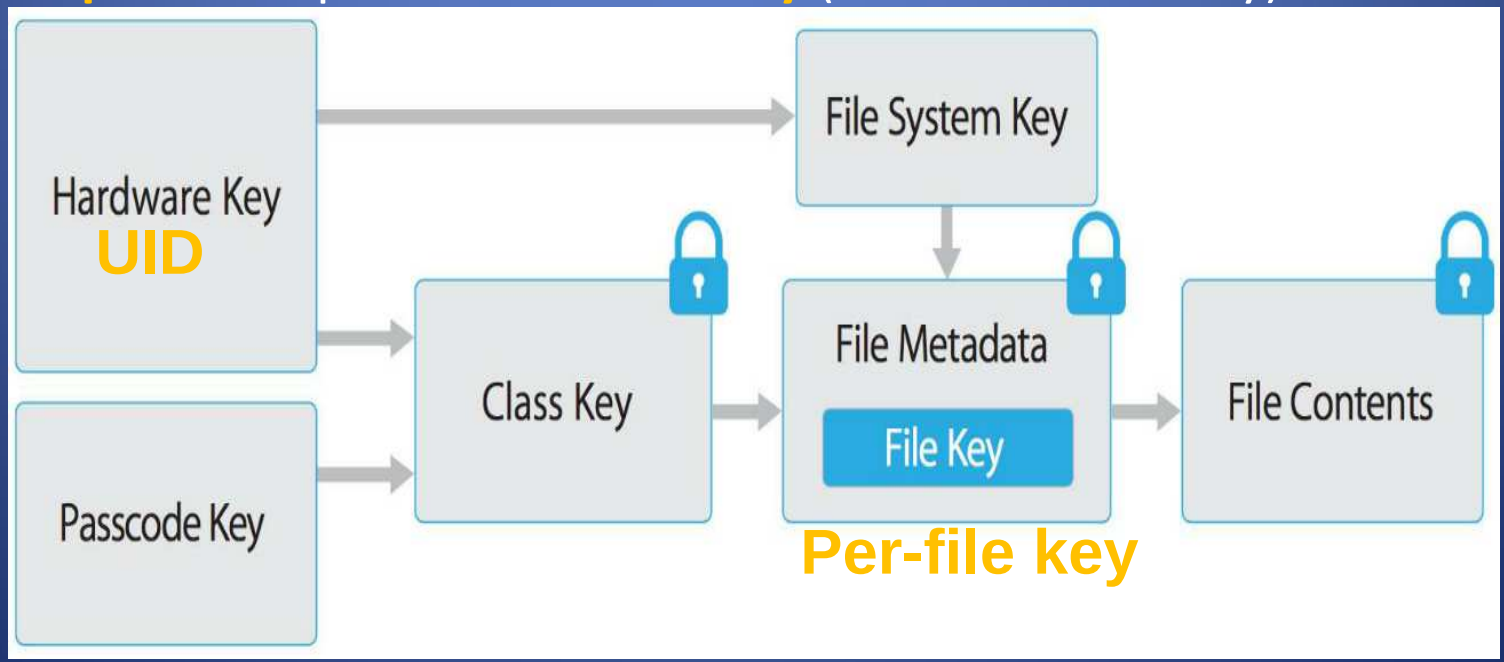
Code Signing

- **Only Apple-approved code runs on their devices**, that is, code signed by Apple
- A developer profile and an **Apple-signed certificate are required** to deploy and run an application
- Developers need to register with Apple, join the Apple Developer Program and **pay a yearly subscription** to get the full range of
- development and deployment possibilities
- There's also a **free developer account** that allows you to compile and deploy apps (but not distribute them in the App Store) via sideloading

iOS Security

Encryption and data protection

- The **passcode** key is derived from the user's passphrase via the PBKDF2 algorithm
- Each data file is associated with a specific protection class **(class key)**. Each **class key** is associated with different device states (e.g., device locked/unlocked).
- The **per-file key** is used to encrypt the **file's contents**.
- The **class key** is wrapped around the **per-file key** and stored in the **file's metadata**.
- The **file system key** is used to encrypt the **metadata**.
- The **UID** and **passcode** protect the **class key** (with a derived key).



iOS Security

Sandbox

- The **appsandbox** is an iOS access control technology. It is enforced at the kernel level
- Its purpose is **limiting system and user data damage** that may occur when an app is compromised.
- All third-party apps run under the same user (**mobile**)
- Only a few system applications and services run as **root** (or other specific system users).
- Regular iOS apps are confined to a **container** that restricts access to the app's own files and a very limited number of system APIs
- **Access to all resources** (such as files, network sockets, IPCs, and shared memory) are controlled by the sandbox.

iOS Security: Platform Overview

IPA Container

- iOS apps are distributed in IPA (iOS App Store Package) archives.
- The IPA file is a **ZIP-compressed archive** that contains all the **code** and **resources** required to execute the app.

IPA files have a built-in directory structure:

- **/Payload/** folder contains all the application data
- **/Payload/Application.app** contains the application data itself (ARM-compiled code) and associated static resources.
- **/iTunesArtwork** is a 512x512 pixel PNG image used as the application's icon.
- **/iTunesMetadata.plist** contains various bits of information, including the developer's name and ID, the bundle identifier, copyright information, genre, the name of the app, release date, purchase date, etc.
- **/WatchKitSupport/WK** is an example of an **extension bundle**. This specific bundle contains the extension delegate and the controllers for managing the interfaces and responding to user interactions on an **Apple watch**.

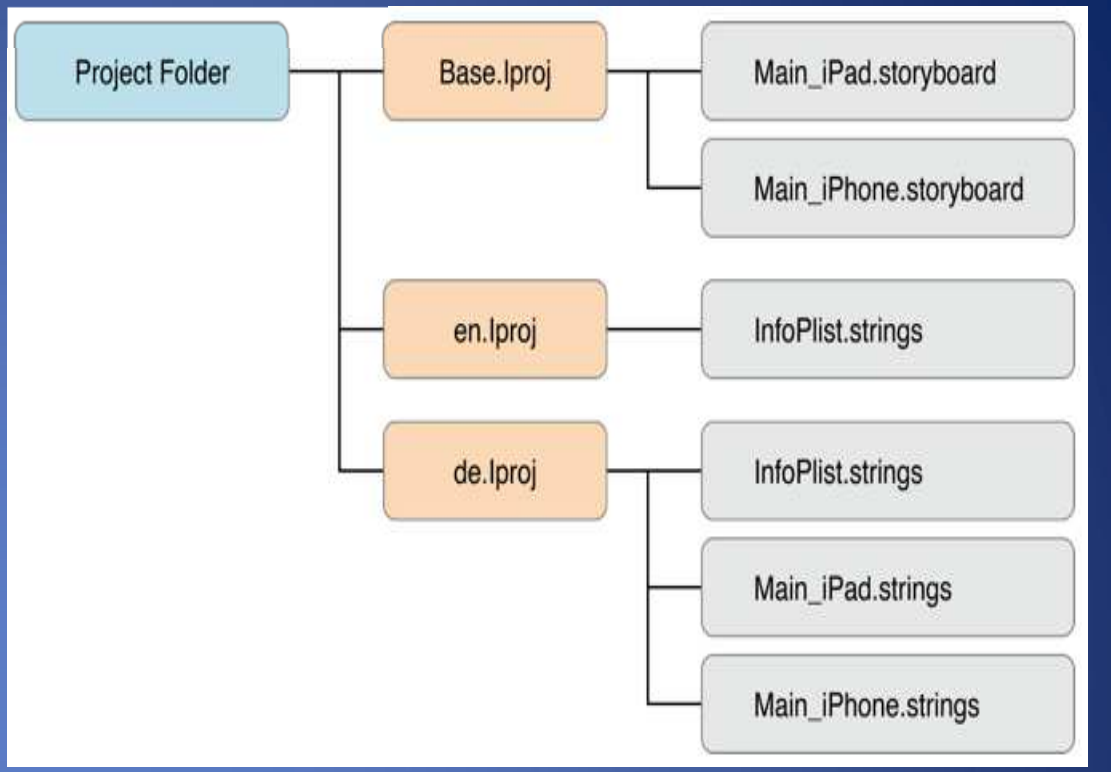
iOS Security: Platform Overview

IPA Container (details)

- **MyApp**: The **executable** file containing the compiled (unreadable) application source code.
- **Application**: Application **icons**.
- **Info.plist**: **Configuration** information, such as bundle ID, version number, and application display name.
- **Launch images**: The system uses one of the provided launch images as a **temporary background** until the application is fully loaded.
- **MainWindow.nib**: **Default interface** objects that are loaded when the application is launched.
- **Settings.bundle**: Application-specific **preferences** to be displayed in the Settings app.
- **Custom resource files**: Non-localized resources are placed in the top-level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources include **nib** files, **images**, **sound** files, **configuration** files, **strings** files, …

iOS Security: Platform Overview

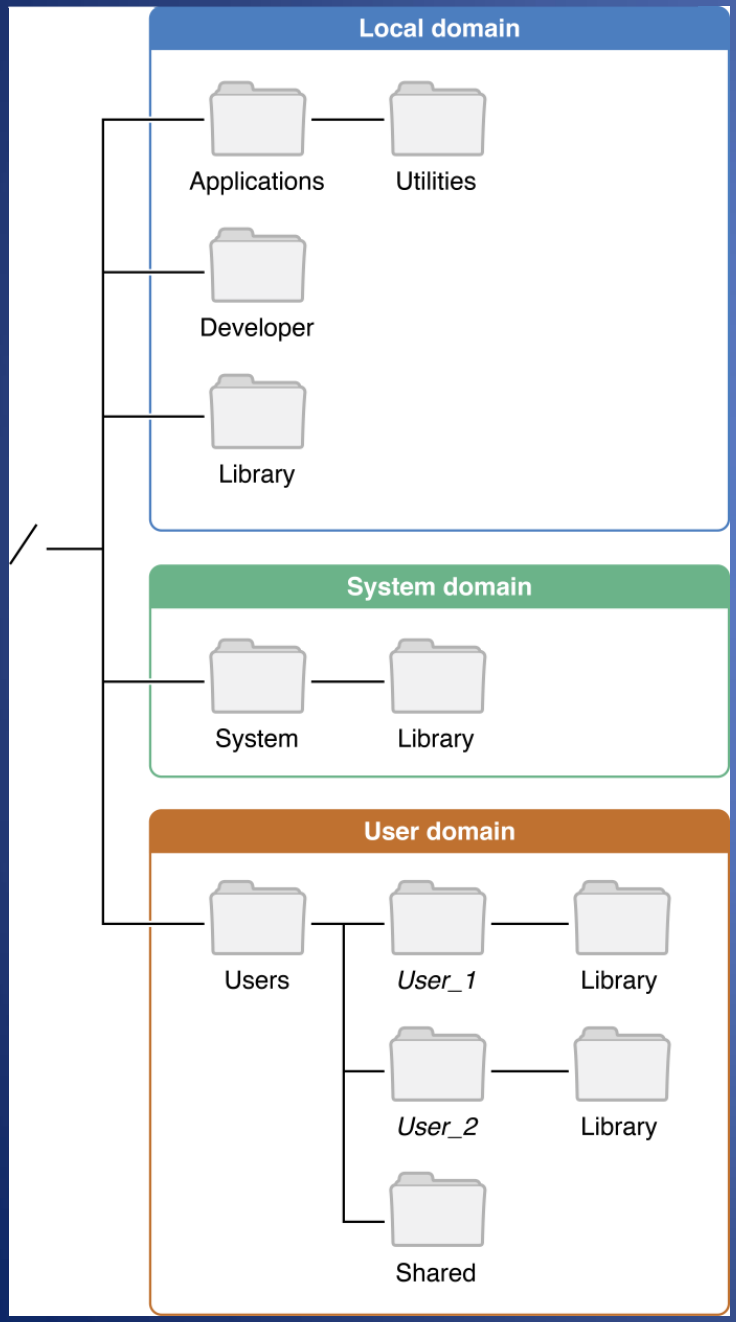
IPA Container (details)

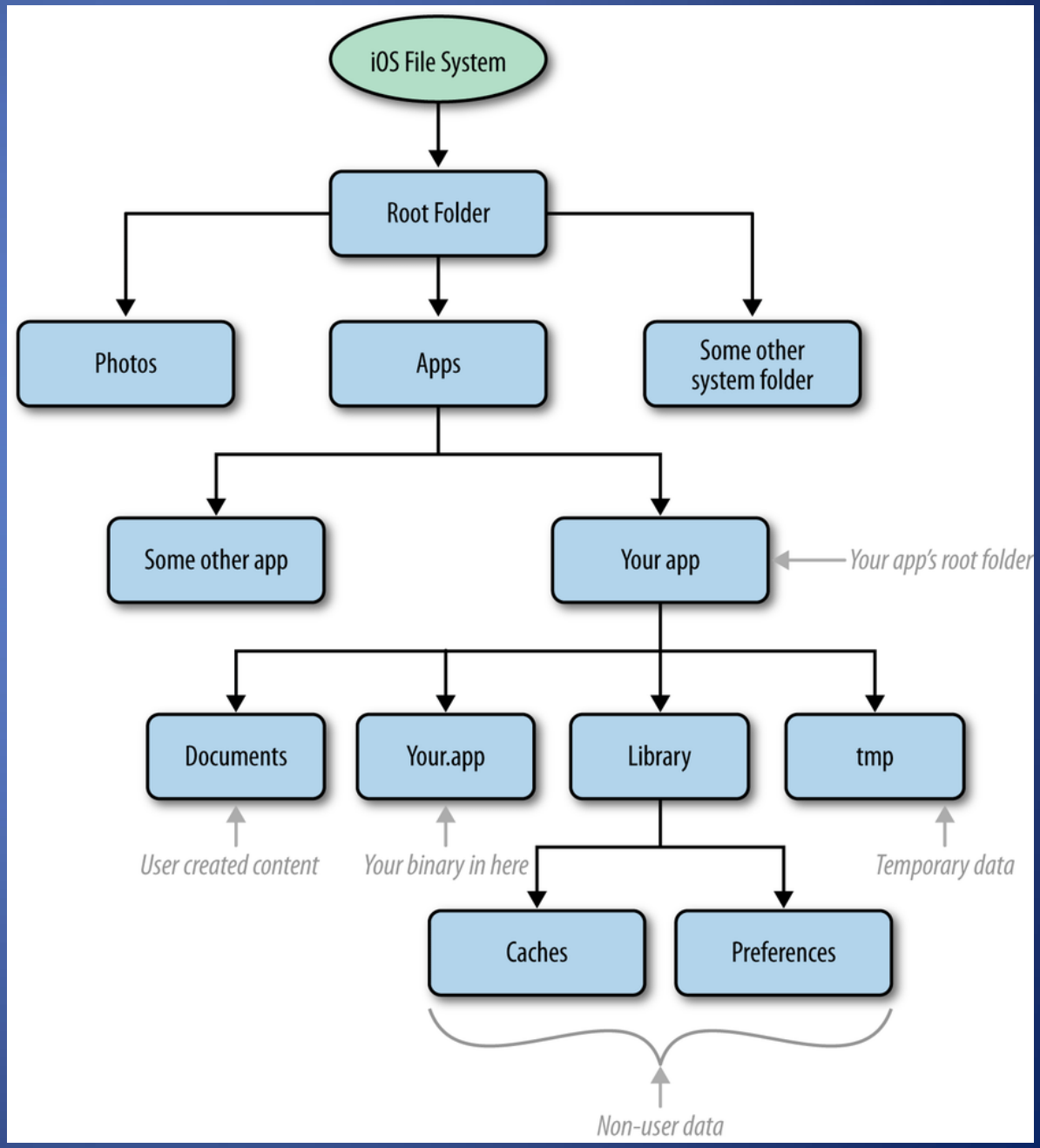


- **language.lproj** folder exists for each language that the application supports. It contains a storyboard and strings file.
- **storyboard** is a visual representation of the iOS application's user interface. It shows screens and the connections between those screens.
- **strings** file format consists of one or more key-value pairs and optional comments.

iOS Security: Platform Overview

iOS files and folders





iOS Security: Platform Overview

iOS folders

- Starting with iOS 8, applications are identified by **UUID** (Universal Unique Identifier), a 128-bit number.
- This number is the **name of the folder** in which the application itself was stored.

/var/mobile/Containers/Bundle/Application/[UUID]/Application.app contains the application.app data, and it stores the static content as well as the application's ARM-compiled binary. The contents of this folder is used to validate the code signature.
/var/mobile/Containers/Data/Application/[UUID]/Documents contains all the user-generated data. The application end user initiates the creation of this data.
/var/mobile/Containers/Data/Application/[UUID]/Library contains all files that aren't user-specific, such as caches, preferences, cookies, and property list (plist) configuration files.
/var/mobile/Containers/Data/Application/[UUID]/tmp contains temporary files which aren't needed between application launches.

- Since iOS 9.3.x, the Bundle path has changed again to **/var/containers/Bundle/Application/**

iOS Security: Platform Overview

iOS files 1/2

AppName.app

- This app's bundle contains the app and all its resources.
- This directory is visible to users, but users can't write to it.
- Content in this directory is not backed up.

Documents/

- Use this directory to store user-generated content.
- Visible to users and users can write to it.
- Content in this directory is backed up.
- The app can disable paths by setting **NSURLIsExcludedFromBackupKey** .

Library/

- This is the top-level directory for all files that aren't user data files.
- iOS apps usually use the Application Support and Caches subdirectories, but you can create custom subdirectories.

Library/Caches/

- Contains semi-persistent cached files.
- Invisible to users and users can't write to it.
- Content in this directory is not backed up.
- The OS may delete this directory's files automatically when the app is not running and storage space is running low.

iOS Security: Platform Overview

iOS files 2/2

Library/Application Support/

- Contains persistent files necessary for running the app.
- Invisible to users and users can't write to it.
- Content in this directory is backed up.
- The app can disable paths by setting NSURLIsExcludedFromBackupKey

Library/Preferences/

- Used for storing properties, objects that can persist even after an application is restarted.
- Information is saved, unencrypted, inside the application sandbox in a plist file called [BUNDLE_ID].plist.
- All the key/value pairs stored using NSUserDefaults can be found in this file.

tmp/

- Use this directory to write temporary files that need not persist between app launches.
- Contains non-persistent cached files. Invisible to users.
- Content in this directory is not backed up.
- The OS may delete this directory's files automatically when the app is not running and storage space is running low

iOS Security: Platform Overview

Installation process

Via the command line with **ipainstaller**
On a jailbroken device, you can recover the IPA for an installed iOS app with IPA Installer.
During mobile security assessments, developers often give you the IPA directly. They can send you the actual file or provide access to the development-specific distribution platform they use, e.g., HockeyApp or Testflight.

Via **Cydia Impactor**,
Created to jailbreak iPhones, but has been rewritten to sign and install IPA packages to iOS devices via sideloading.

The actual installation process is then handled by the **installd** daemon, which will unpack and install the application.
All applications must be signed with a certificate issued by Apple.
On a jailbroken phone, however, you can circumvent the signature verification with **AppSync**, a package available in the Cydia store.

Cydia is an alternative app store or software distribution system.
It contains numerous useful applications that leverage jailbreak-provided root privileges to execute advanced functionality.

iOS Security: Platform Overview

App Permissions

- iOS apps don't have pre-assigned permissions.
- Instead, the user is asked to grant permission during run time, when the app attempts to use a sensitive API for the first time.
- Apps that have been granted permissions are listed in the **Settings > Privacy** menu, allowing the user to modify the app-specific setting.
- Apple calls this permission concept privacy controls.

For example, when accessing a user's contacts, any call blocks the app while the user is being asked to grant or deny access.

The **following APIs require user permission**:

Contacts	Microphone
Calendars	Camera
Reminders	HomeKit
Photos	Health
Motion activity and fitness	Speech recognition
Location Services	Bluetooth sharing
Media Library	Social media accounts

iOS Security: Platform Overview

Keychain 1/3

- The iOS Keychain can be used to **securely store** short, sensitive bits of data, such as encryption keys and session tokens.
- It is implemented as an **SQLite database** that can be accessed through the Keychain APIs only.
- **Only one** Keychain is available to all apps.
- Access to the items can be **shared between apps signed by the same developer**
- Access to the Keychain is managed by the **securityd** daemon

The Keychain API includes the following main operations:

- SecItemAdd
- SecItemUpdate
- SecItemCopyMatching
- SecItemDelete

iOS Security: Platform Overview

Keychain 2/3

Items added to the Keychain are encoded as a **binary plist** and encrypted with a 128-bit AES **per-item key**

Note that larger blobs of data aren't meant to be saved directly in the Keychain-that's what the **Data Protection API** is for.

The following configurable accessibility values for **kSecAttrAccessible** are the **Keychain Data Protection** classes:

- **kSecAttrAccessibleAlwaysThisDeviceOnly** : The data in the Keychain item can always be accessed, regardless of whether the device is locked.
- **kSecAttrAccessibleWhenUnlocked** : The data in the Keychain item can be accessed only while the device is unlocked by the user
- …

iOS Security: Platform Overview

Keychain 3/3

AccessControlFlags define the mechanisms with which users can authenticate the key (SecAccessControlCreateFlags):

- **kSecAccessControlTouch IDCurrentSet** : Access the item via one of the fingerprints registered to Touch ID.
- **kSecAccessControlDevicePasscode** : Access the item via a **passcode**
- …

Please note that keys secured by Touch ID (via **kSecAccessControlTouch IDCurrentSet** are protected by the **Secure Enclave**: The Keychain holds a **token only**, not the actual key. The **key resides in the Secure Enclave**

Starting with iOS 9, you can do **ECC**-based signing operations in the Secure Enclave. The *private key* and the **cryptographic operations** reside within the Secure Enclave.

iOS Security: Platform Overview

Keychain Data Persistence

When an application is **uninstalled**,

- The Keychain data used by the application is **retained** by the device
- The data stored by the application sandbox which is **wiped**.

In the event that a user sells their device **without performing a factory reset**, the buyer of the device may be able to **gain access to the previous user's application accounts and data** by reinstalling the same applications used by the previous user.

When developing **logout** functionality for an iOS application, make sure that the Keychain data is **wiped** as part of account logout.

This will allow users to **clear their accounts before uninstalling** an application.

iOS Security: Platform overview

iOS implements **address space layout randomization** (ASLR) and **eXecute Never** (XN) bit to mitigate code execution attacks:

- ASLR **randomizes the memory location of the program's executable file, data, heap, and stack** every time the program is executed
- The XN mechanism allows iOS **to mark selected memory segments of a process as non-executable**. On iOS, the process stack and heap of user-mode processes is marked non-executable.

iOS Security: Testing Environment

Tooling

The iOS SDK simulator offers a **higher-level** simulation of an iOS device.

Most importantly, emulator binaries are compiled to **x86** code instead of **ARM** code.

Apps compiled for a real device don't run, making the simulator useless for black box analysis and reverse engineering.

The following is the most basic iOS app testing setup:

- Laptop with admin rights
- Wi-Fi network that permits client-to-client traffic
- At least one jailbroken iOS device (of the desired iOS version)
- Burp Suite or other interception proxy tool

iOS Security: Testing Environment

Jailbreak

The concepts of "rooting" and "flashing" on Android:

- **Rooting**: This typically involves installing the **“su” binary** on the system or replacing the whole system with a rooted custom ROM. Exploits aren't required to obtain root access as long as the bootloader is accessible.
- **Flashing custom ROMs**: This allows you to replace the OS that's running on the device after you unlock the bootloader. The bootloader may require an exploit to unlock it.

On iOS devices, **flashing a custom ROM is impossible** because the iOS bootloader only allows Apple-signed images to be booted and flashed.

This is why even official iOS images can't be installed if they aren't signed by Apple

It makes iOS downgrades only possible for as long as the previous iOS version is still signed.

iOS Security: Testing Environment

Automated static and dynamic analysis

The free and open source tools **MobSF** and **Needle** have some static and dynamic analysis functionality.

needle

Don't shy away from using automated scanners for your analysis

iOS Security: Testing Environment

Preparation of security assessment

- SSH connection via USB (if no Wifi): you can use **usbmuxd** to connect to your device's SSH server via USB
- Copy application files
- Dump Keychain items (**Keychain dumper**: jailbroken)
- Method tracing with **Frida** (jailbroken)
- Monitoring console logs (with **Xcode**)
- Setting up a Web Proxy (**Burp Suite**)
- Bypassing Certificate Pinning (**SSL Kill Switch 2** or **Burp Suite** mobile assistant)
- Network Monitoring/Sniffing (**Wireshark**)

iOS Security: Data Storage

Looking for sensitive data

- Static analysis: files, SQLite databases
- Dynamic analysis (**iMazing**: non jailbroken)
  - **Triggering all app functionality before the data is analyzed** is important because the app may store sensitive data only after specific functionality has been triggered. You can then perform static analysis for the data dump according to generic keywords and app-specific data.
  - The Keychain contents can be dumped during dynamic analysis. On a jailbroken device, you can use **Keychain dumper**. The path to the Keychain file is **/private/var/Keychains/keychain-2.db**. . On a non-jailbroken device, you can use **objection** to dump the Keychain (transfer files from app data directory)
  - **objection** is a runtime mobile security assessment framework that does not require a jailbroken or rooted device for both iOS and Android (if **Frida** library is added to the app and repackaged)
- Searching for Binary Cookies (**Needle**)
  - iOS applications often store binary **cookie** files in the application sandbox. Cookies are binary files containing cookie data for application WebViews.
- Searching for Property List Files
  - iOS applications often store data in property list (plist) files

iOS Security: Data Storage

Looking for sensitive data

- Searching for Cache Databases (**Needle**)
  - iOS applications can store data in cache databases
- Searching for SQLite Databases (**Needle**)
  - iOS applications typically use SQLite databases to store data required by the application
- Checking logs
  - Determining Whether Sensitive Data Is Sent to Third Parties: these services provide can involve tracking services to monitor the user's behavior while using the app, selling banner advertisements, or improving the user experience.
- Keyboard cache for autocorrection and spell checking.
  - Most keyboard input is cached by default, in **/private/var/mobile/Library/Keyboard/dynamic-text.dat**
- Exposed via IPC Mechanisms (prefer XPC Services)

iOS Security: Data Storage

Looking for sensitive data

- Testing backups
  - Backups can be made through **iTunes** or the cloud (via the **iCloud** backup feature).
  - Backup includes nearly all data stored on the device except highly sensitive data such as **Apple Pay** information and **Touch ID** settings
  - Keychain data is backed up as well, but the secrets in the Keychain **remain encrypted**.
  - **Class keys** necessary to decrypt the Keychain data aren't included in the backup (restoring the Keychain data requires **restoring the backup to a device and unlocking the device with the users passcode**: kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly)

If sensitive data is handled as recommended i.e. stored in the Keychain or encrypted with a key that's locked inside the Keychain, backups aren't a security issue.

iOS Security: Data Storage

Looking for sensitive data

- Auto generated screenshots
  - This feature is saving a screenshot when the application goes into the background
  - Screenshots are written to local storage, where they can be recovered by a rogue application with a sandbox bypass exploit or someone who steals the device
  - Stored in **/var/mobile/Containers/Data/Application/$APP_ID/Library/Caches/Snapshots/**
- Memory
  - Get a memory dump and analyze it in real time (with a debugger): **objection** or **Freedump** (non jailbroken)

iOS Security: Cryptography APIs

- Find deprecated algos
- Test random number generation
- Test key management
  - **Not storing a key at all**
  - It will ensure that no keymaterial can be dumped.
  - This can be archieved by using a Password Key Derivation function, such as PKBDF-2.
  - **When you need to store the key**
  - It is recommended to use the Keychain
  - The KeyChain supports two type of storage mechanisms:
    - a key is either secured by an encryption key stored in the secure-enclave
    - or the key itself is within the secure enclave.

iOS Security: Local Authentication

- Leverage the **Keychain** for implementing local authentication:
  - The app stores either a secret authentication token or another piece of secret data identifying the user in the Keychain. In order to authenticate to a remote service, the user must unlock the Keychain using their passphrase or fingerprint to obtain the secret data
- The fingerprint ID sensor is operated by the **SecureEnclave** security coprocessor and does not expose fingerprint data to any other parts of the system
- Apple introduced **Face ID** which allows authentication based on facial recognition.
- Developers have two options for incorporating Touch ID/Face ID authentication:
  - LocalAuthentication.framework is a **high-level API** (always prefer this one)
  - Security.framework is a **lower level API** to access Keychain Services
- Please be aware that using either **the LocalAuthentication.framework** or the **Security.framework** , will be a control that can be bypassed by an attacker as it does only return a boolean and no data to proceed with.
  - David Lidner https://www.youtube.com/watch?v=XhXIHVGCFFM

  On a jailbroken device tools like **Swizzler2** and **Needle** can be used to bypass LocalAuthentication (both use **Frida**)

iOS Security: Network APIs

App Transport Security

- The following is a summarized list of App Transport Security Requirements:
  - No HTTP connections are allowed
  - The X.509 Certificate has a SHA256 fingerprint and must be signed with at least a 2048-bit RSA key or a 256-bit ECC key.
  - Transport Layer Security (TLS) version must be 1.2 or above and must support Perfect Forward Secrecy (PFS) through Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange and AES-128 or AES-256 symmetric ciphers
- ATS restrictions can be disabled by configuring exceptions in the Info.plist file under the NSAppTransportSecurity key. These exceptions can be applied to:
  - Allow insecure connections (HTTP),
  - Lower the minimum TLS version,
  - Disable PFS or allow connections to local domains.

- Testing Custom Certificate Stores and Certificate Pinning

iOS Security: Platform APIs

Custom URL Schemes

- In contrast to Android's IPC capability, iOS offers few options for communication between apps. In fact, there's no way for apps to communicate directly. Instead, Apple offers two types of indirect communication: file transfer through **AirDrop** and **custom URL schemes**.
- App declare support for the scheme and handle incoming URLs that use the scheme. Once the URL scheme is registered, other apps can open the app that registered the scheme, and pass parameters by creating appropriately formatted URLs and opening them with the **openURL** method.
- The Skype app registered the **skype://** protocol handler, which allowed other apps to trigger calls to other Skype users and phone numbers. Unfortunately, Skype didn't ask users for permission before placing the calls, **so any app could call arbitrary numbers without the user's knowledge** (https://www.dhanjani.com/blog/2010/11/insecure-handling-of-url-schemes-in-apples-ios.html).
- Attackers exploited this vulnerability by putting an invisible **<iframe src="skype://xxx?call"></iframe>** (where xxx was replaced by a premium number), so any Skype user who inadvertently visited a **malicious website** called the premium number.
- **FuzzDB** offers fuzzing dictionaries

iOS Security: Platform APIs

Webview

- WebViews are **in-app browser** components for displaying interactive web content
  - iOS WebViews support **JavaScript** execution by default, so script injection and cross-site scripting attacks can affect them.
  - As a best practice, **disable JavaScript** in a WKWebView unless it is explicitly required
- Testing WebView Protocol Handlers
  - The following schemas can be used within a WebView on iOS:
    - **http(s)://**
    - **file://**
    - **tel://**
- Determining Whether **Native Methods** Are Exposed Through WebViews
- Local File Inclusion
  - WebViews can **load content remotely** and **locally from the app data directory**. If the content is loaded locally, users should not be able to change the filename or path from which the file is loaded, and they shouldn't be able to edit the loaded file.

iOS Security: Platform APIs

Object Persistence

- Object Encoding
  - iOS comes with two protocols for object encoding and decoding for **Objective-C** or NSObjects: **NSCoding** and **NSSecureCoding**
  - The **issue with NSCoding** is that the object is often already constructed and inserted before you can evaluate the class-type. This allows an attacker to easily inject all sorts of data
- Codable (**Swift 4**)
  - By adding Codable to the inheritance list for the GivenClass, the methods **init(from:)** and **encode(to:)** are autmatically supported
- JSON
  - **JSON** itself can be stored anywhere, e.g., a (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g., encrypted/HMACed).
- Property Lists
  - You can persist objects to **PropertyLists** (also called Plists in previous sections).
- XML encoding
  - Therefore, it is key to disable external entity parsing if possible. See **XXE** in the Apple iOS Office viewer as an example (https://nvd.nist.gov/vuln/detail/CVE-2015-3784).

iOS Security: Code Quality

- Signed
- Debuggable app
  - Debugging iOS applications can be done using Xcode, which embeds a powerful debugger called lldb
  - Lldb is the default debugger since Xcode5 where it replaced GNU tools like gdb
- Finding Debugging Code and Verbose Error Logging (**Xcode**)
- Testing exception handling
- Make Sure That Free Security Features Are Activated
  - ARC - Automatic Reference Counting - memory management feature adds retain and release messages when required
  - Stack Canary - helps prevent buffer overflow attacks by means of having a small integer right before the return pointer.
  - PIE - Position Independent Executable - enables full ASLR for binary
- Checking for Weaknesses in Third Party Libraries

iOS Security: Tampering and Reverse Engineering

x

- **Objective-C** allows method invocations to be changed at run time. This makes hooking into other app functions easy (used by **Cycript** and other reverse engineering tools).
- This "**method swizzling**" is not implemented the same way in **Swift**, and the difference makes the technique harder to execute with Swift
- Xcode (IDE) and iOS SDK
- **Radare2** is a complete framework for reverse engineering and analyzing
- **IDA Pro ($)** can deal with iOS binaries. It has a built-in iOS debugger.
- **Hopper** offers similar static analysis features.

iOS Security: Tampering and Reverse Engineering

x

- Getting the IPA File
- Dumping Decrypted Executables
  - apps distributed via the App Store are also protected by Apple's FairPlay DRM system
  - As of now, the only way to obtain the decrypted code from a FairPlay-decrypted app is to dump it from memory while the app is running
  - Check with otool
- Getting Basic Information with Class-dump and Hopper Disassembler
- Debugging
- Cycript
  - Cycript is a scripting language developed by Jay Freeman (aka saurik).
  - It injects a JavaScriptCore VM into the running process.
  - Via the Cycript interactive console, users can then manipulate the process with a hybrid Objective-C++ and JavaScript syntax.
  - Accessing and instantiating Objective-C classes inside a running process is also possible.

iOS Security: Tampering and Reverse Engineering

x

- Installing Frida
  - Frida is a runtime instrumentation framework that lets you inject JavaScript snippets or portions of your own library into native Android and iOS apps
- Automated Repackaging with Objection
  - Objection is a mobile runtime exploration toolkit based on Frida.
  - One of the biggest advantages about Objection is that it enables testing with non-jailbroken devices.
  - It does this by automating the process of app repackaging with the FridaGadget.dylib library.
- Manual Repackaging
- Patching, Repackaging, and Re-Signing
- Installing and Running an App

iOS Security: Tampering and Reverse Engineering

Getting a Developer Provisioning Profile and Certificate

The **provisioning profile** is a **plist file** signed by Apple.

- It whitelists your code-signing certificate on one or more devices.
- In other words, this represents Apple explicitly allowing your app to run for certain reasons, such as debugging on selected devices (development profile).
- The provisioning profile also includes the entitlements granted to your app.
- The **certificate contains the private key!!** you'll use to sign.

Depending on whether you're registered as an iOS developer, you can obtain a certificate and provisioning profile in one of the following ways:

  - With an iOS developer account
  - With a Regular iTunes Account

iOS Security: Tampering and Reverse Engineering

Troubleshooting

Method Tracing with Frida

- Intercepting Objective-C methods is a useful iOS security testing technique (for data storage operations and network requests, for example).

As an example:

  - we can write a simple tracer for logging HTTP(S) requests made via standard iOS HTTP APIs.
  - We can also inject the tracer into the Safari web browser.

LAB 0001

Revision

Ressources

The **MUST** read:

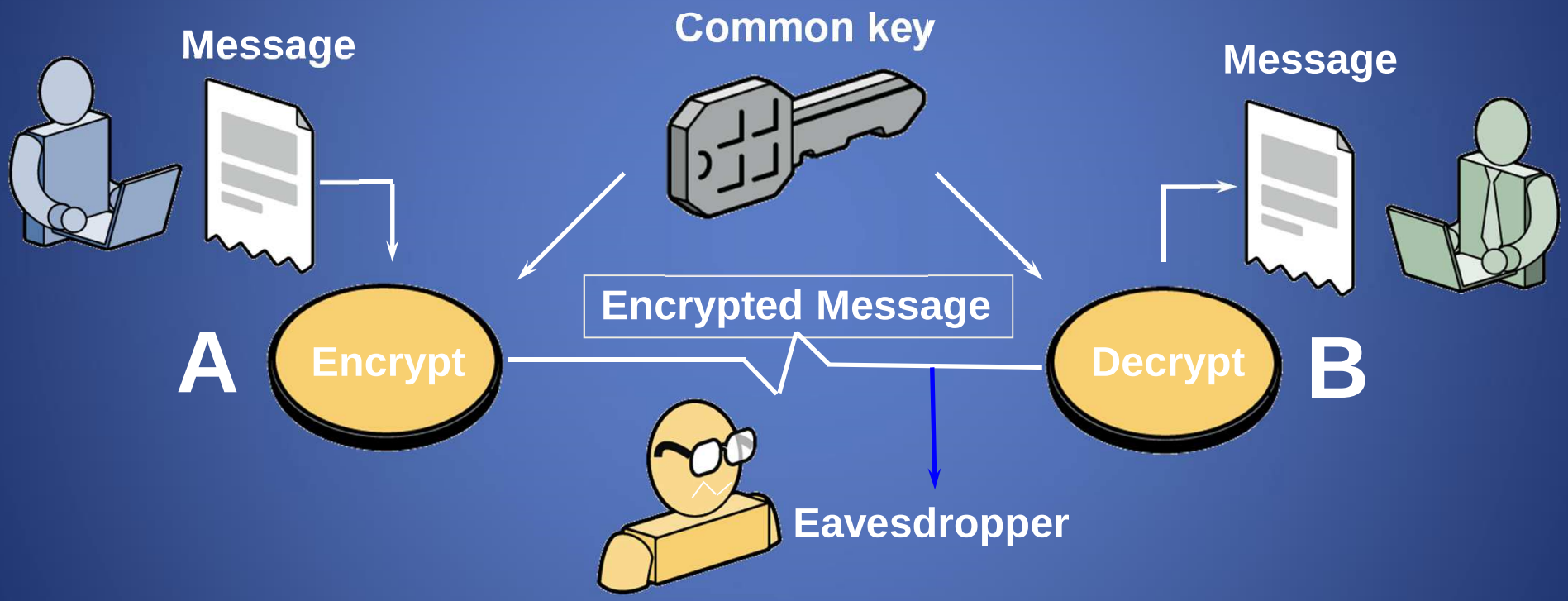
https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide

Standards references:

https://developer.android.com
e.g. https://developer.android.com/training/articles/security-tips
https://source.android.com
e.g. https://source.android.com/security/enhancements/enhancements41
https://www.xda-developers.com/
https://www.osboxes.org/android-x86/

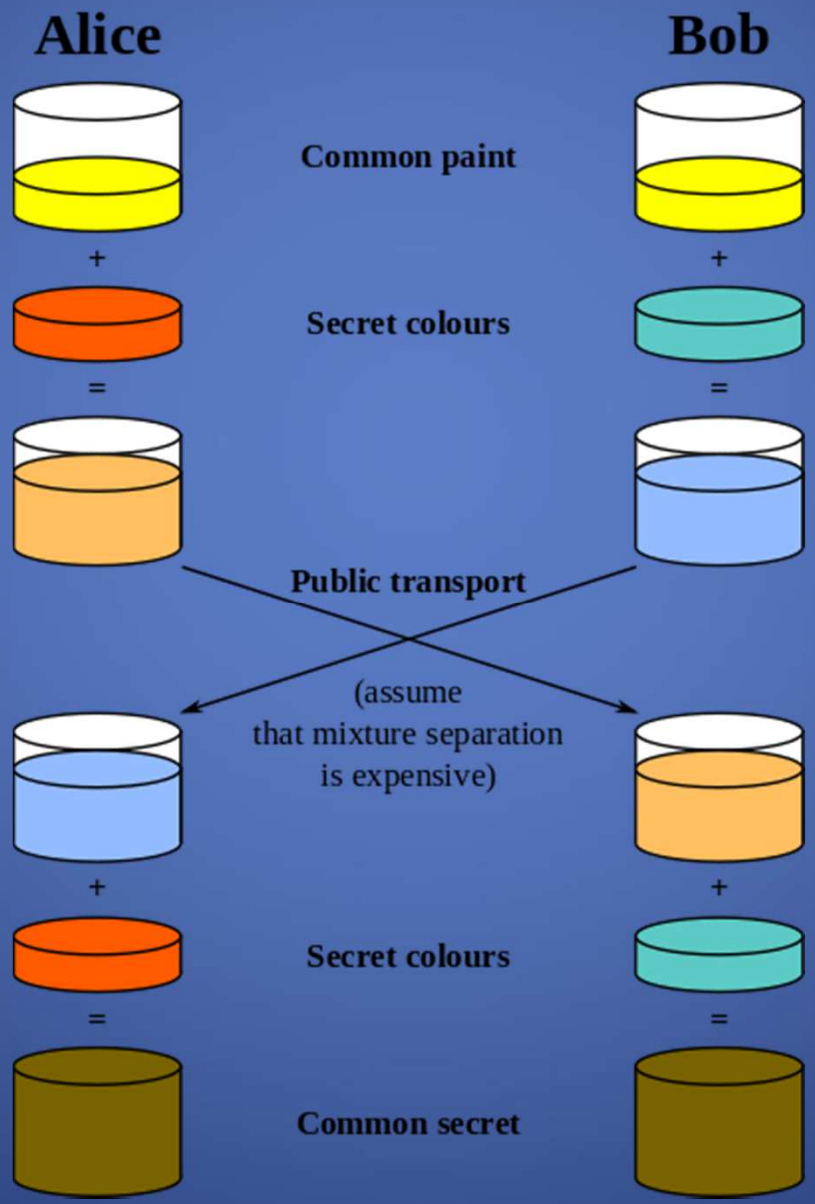
SPARE

Symmetric Key Cryptography Encryption

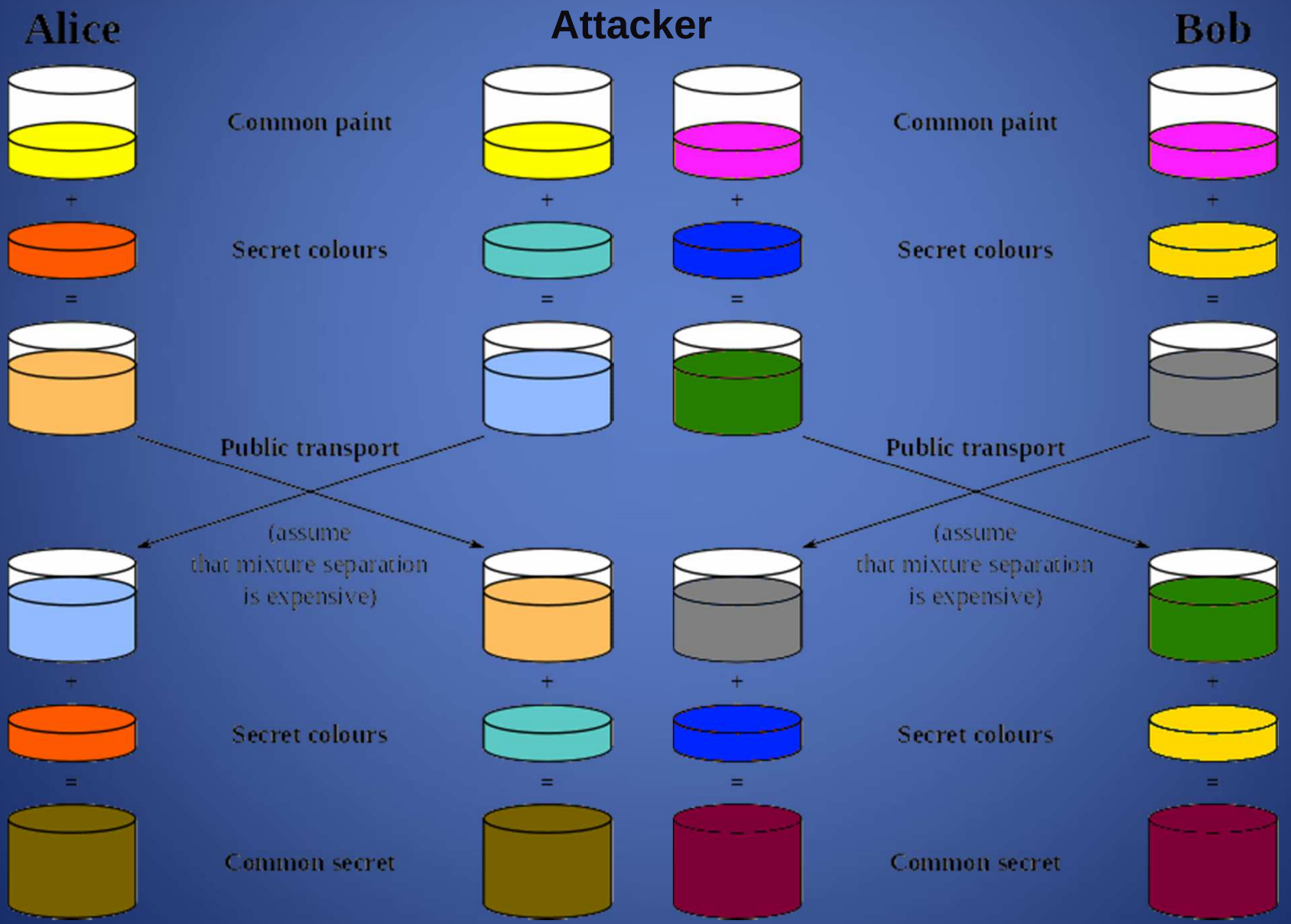


- DES, AES, RC2, RC5

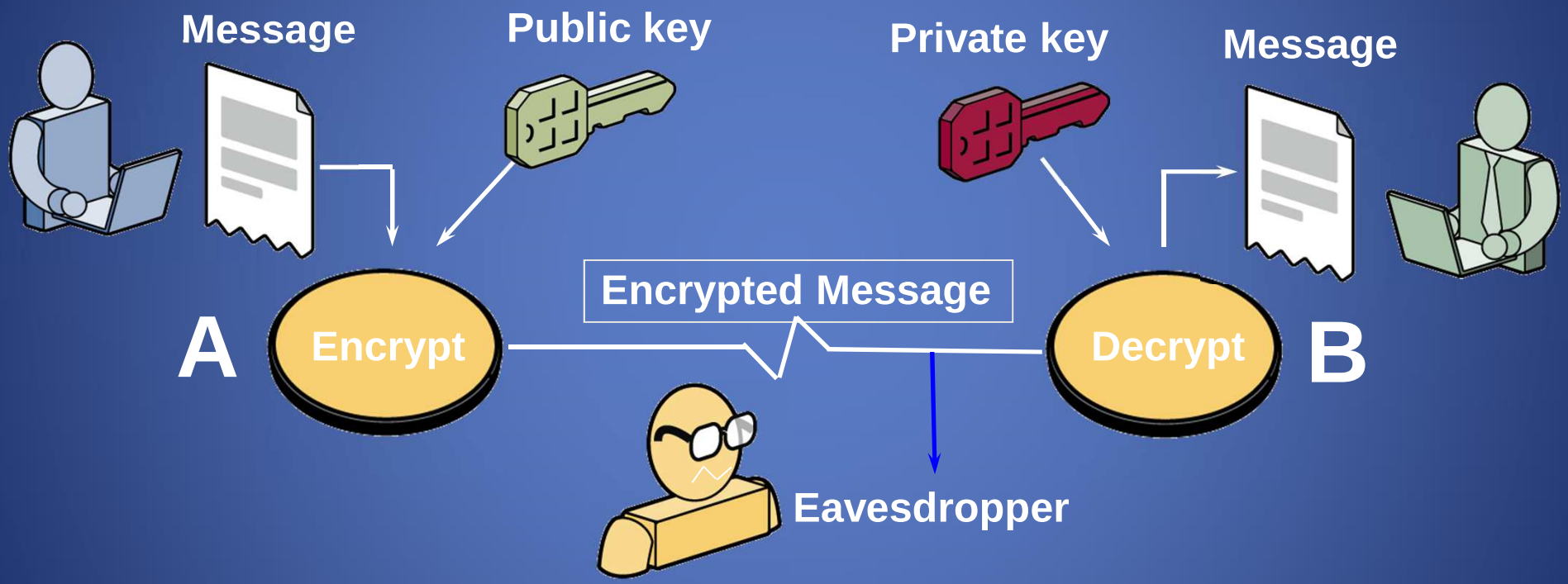
Diffie-Hellman Key Exchange



Diffie-Hellman Security



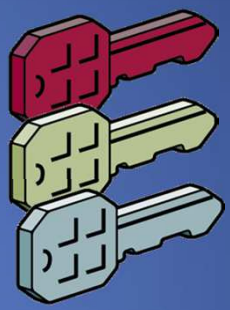
Asymmetric Key Cryptography Encryption



- RSA, ECC, IDEA

Public Key Encryption

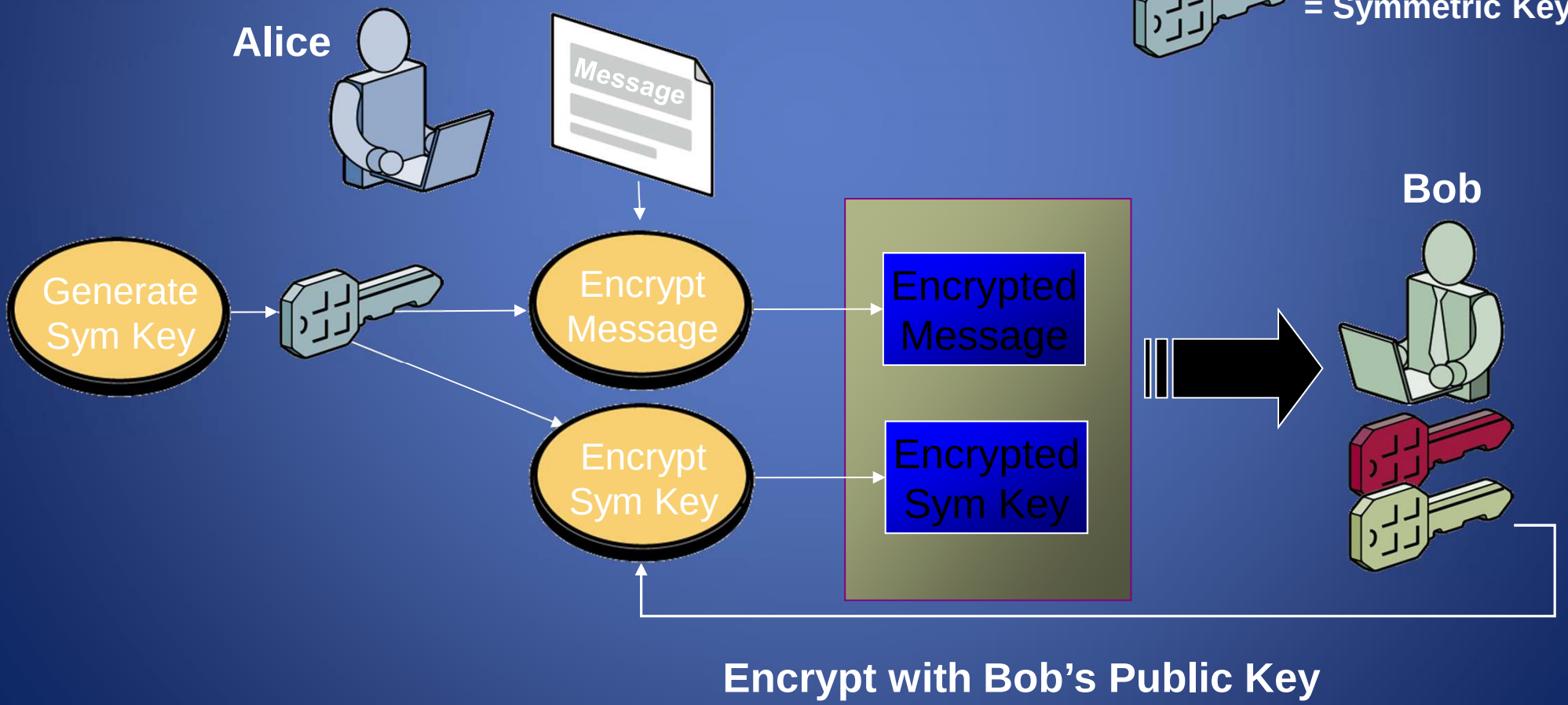
Symmetric keys encrypt data
Public keys encrypt symmetric keys



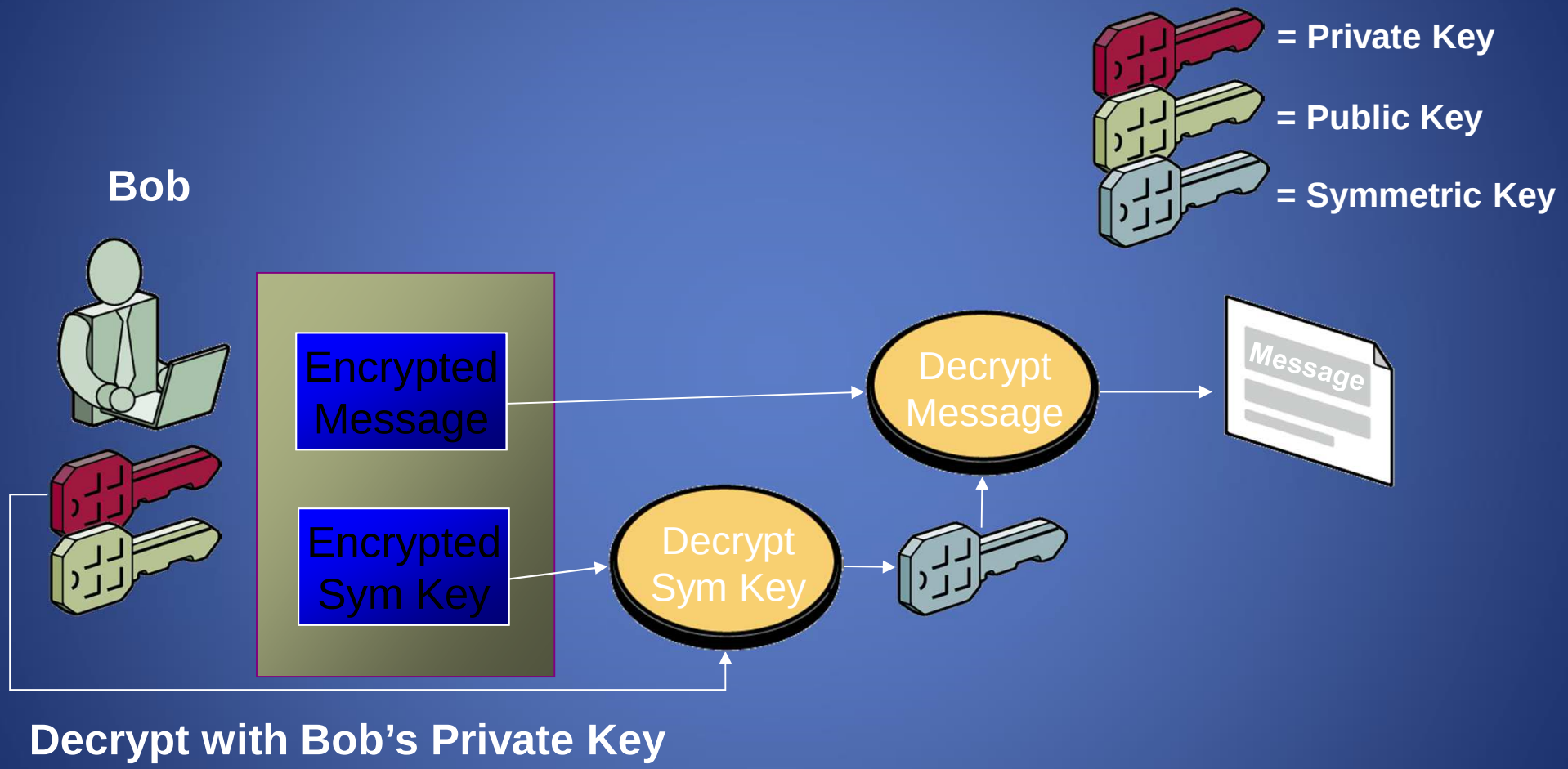
= Private Key

= Public Key

= Symmetric Key



Private Key Decryption



Public key and symmetric key cryptography are complementary technologies

Hash Functions

It was the best of times,
it was the worst of times

It was the best of thymes,
it was the worst of times

Small Difference

Large Difference

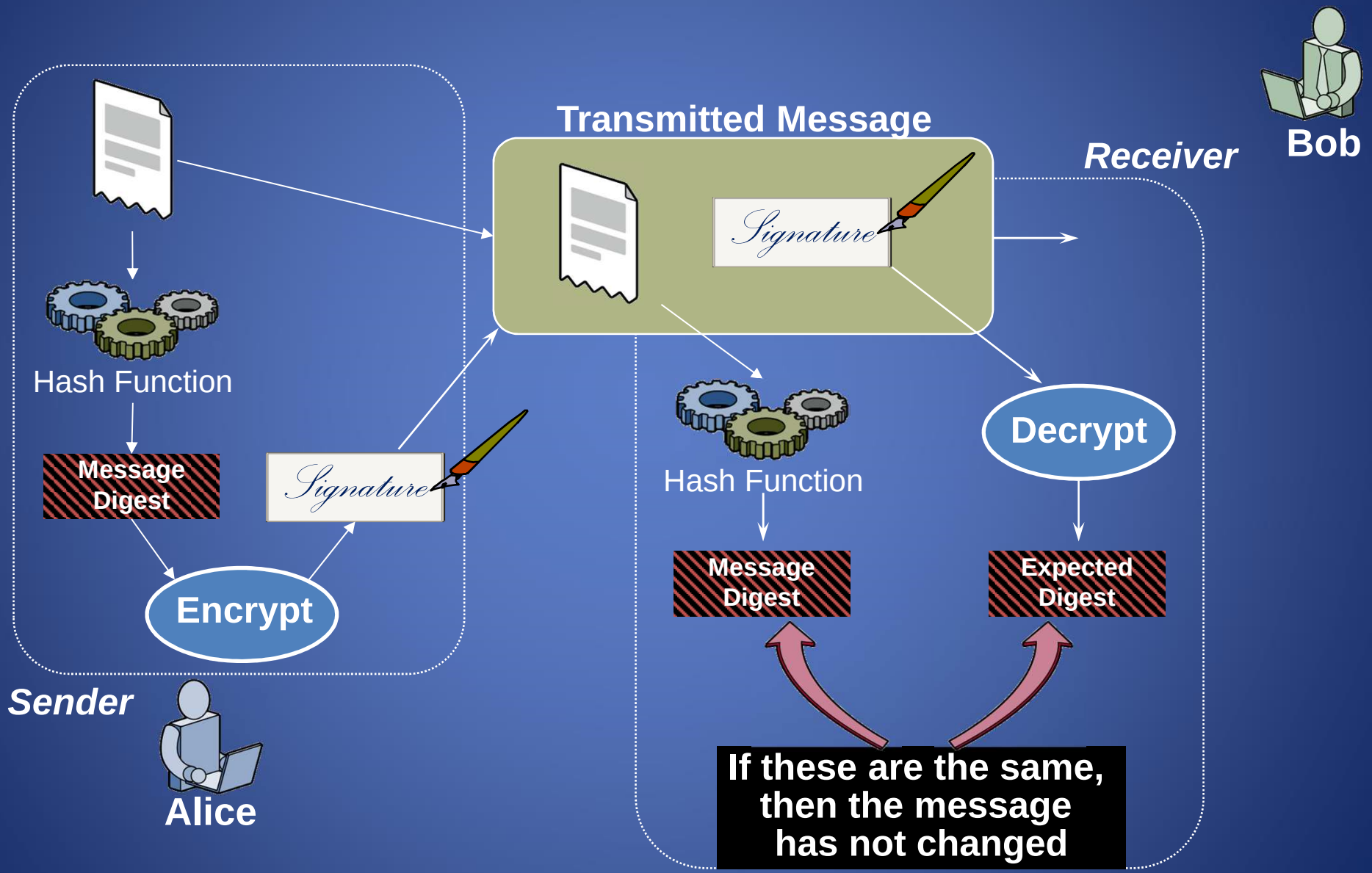
3au8 e43j jm8x g84w

b6hy 8dhy w72k 5pqd

Examples: MD5 (128 bit), SHA-1 (160 bit)

Public-Key – Signature & Verification

Hashing + Encryption = Signature Creation

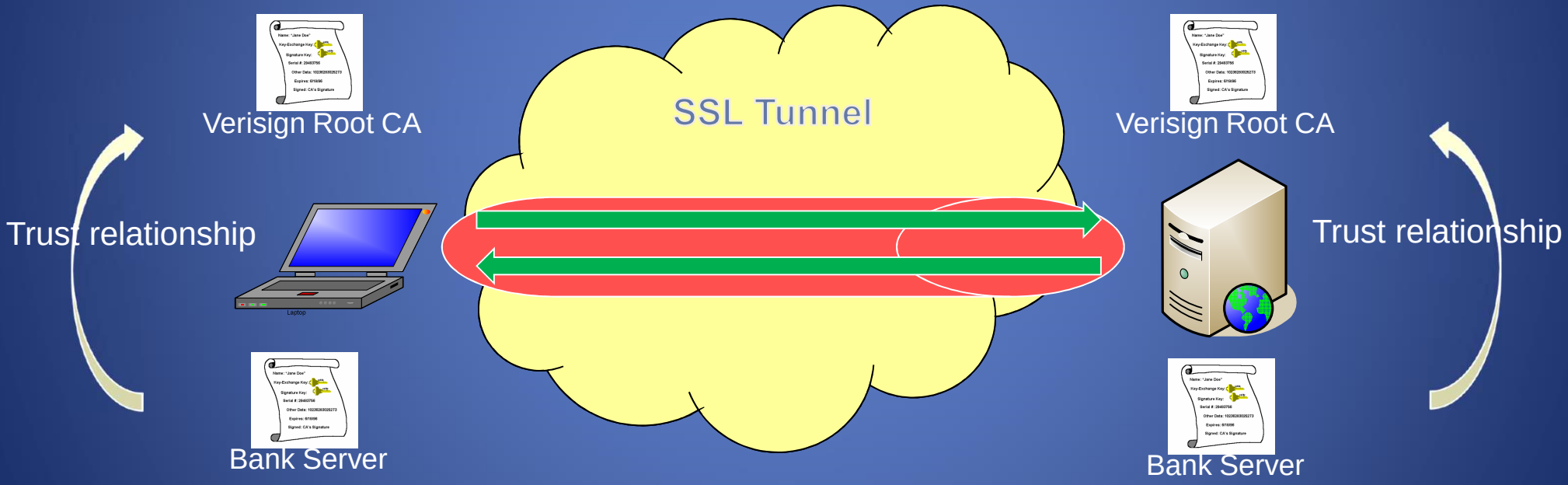


Hashing + Decryption = Signature Verification

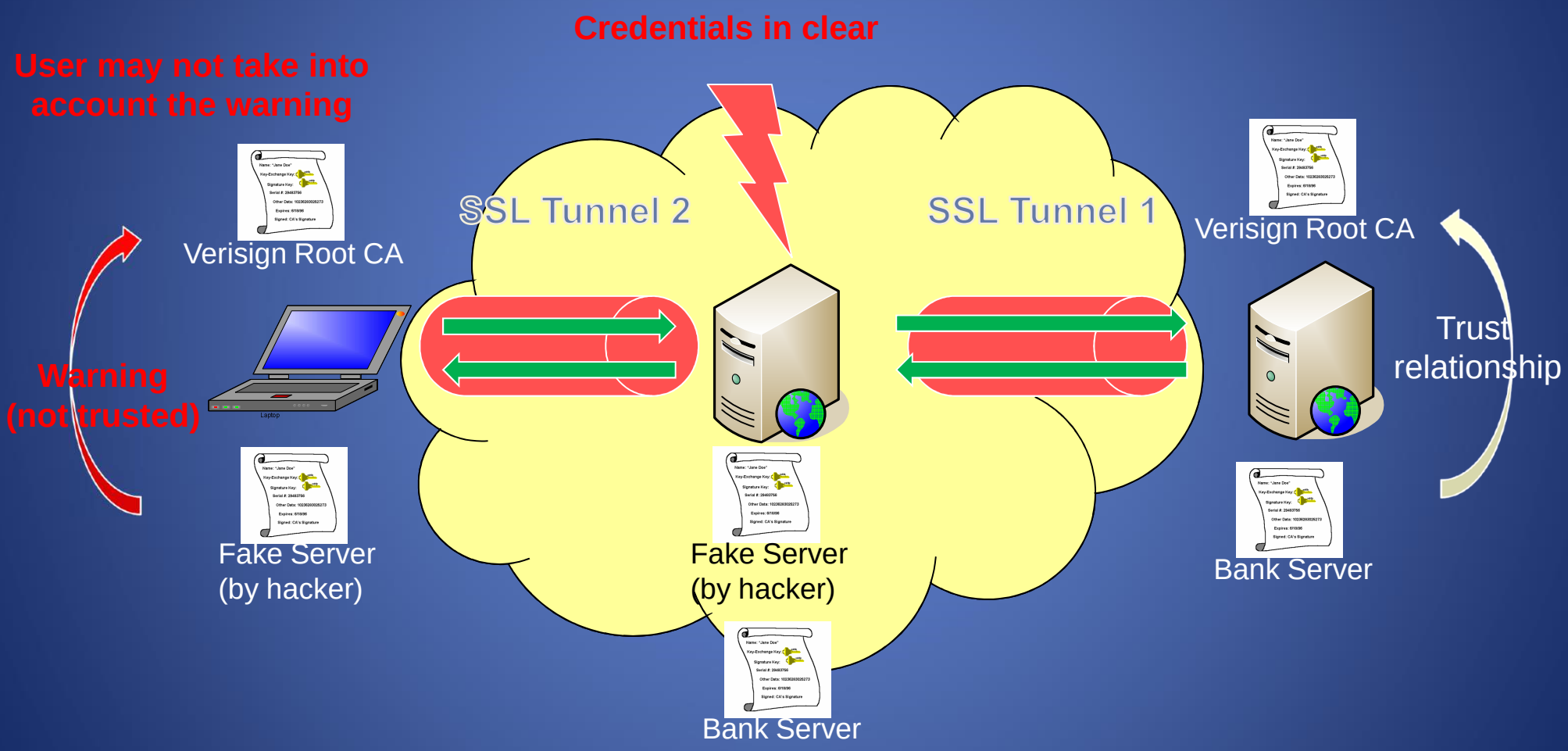
Signature vs MAC vs hash

Cryptographic primitive	Hash	MAC	Digital signature
Security Goal			
Integrity	Yes	Yes	Yes
Authentication	No	Yes	Yes
Non-repudiation	No	No	Yes
Kind of keys	none	symmetric keys	asymmetric keys

Nominal SSL connection: server authentication



Man In The Middle



An inexperienced user may not take into account the warning

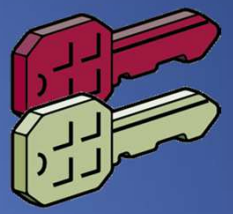
Certificate

A message which at
identifies its subscribe
period, and (5) is digita

Certificate Extensions		
Key Usage:	digitalSignature keyCertSign keyAgreement cRLSign keyEncipherment	dataEncipherment nonRepudiation encipherOnly decipherOny
Extended Key Usage:	serverAuth timeStamping emailProtection OCSPSigning	codeSigning clientAuth
Certificate Policies: URL of CPS and Policy notice text		
Subject Alternative Name: rfc822name, IP Address, DNS Name		
CRL Distribution Point: URL of the Certificate Revocation List		

Version: v3
Serial No:
Algorithms:
Subject DN
Validity per
Public key: e5 53 1b 7 …
Signature

Trust a certificate: signature creation



= Private Key

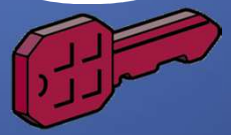
= Public Key

Serial Number
Issuer X.500 distinguished name
Validity Period
Subject X.500 distinguished name
Public key
Key/certificate usage
Extensions
CA Digital Signature

Hash Function

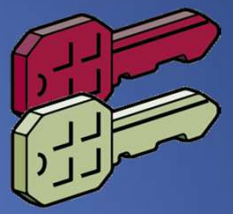
Message Digest

Encrypt



CA Private Key

Trust a certificate: hash comparison



= Private Key

= Public Key

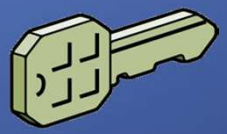
Serial Number
Issuer X.500 distinguished name
Validity Period
Subject X.500 distinguished name
Public key
Key/certificate usage
Extensions
CA Digital Signature

Hash Function

Message Digest

Decrypt

Expected Digest



CA Public Key

X509 certificate verification

- First verify the certificate's integrity
- The certificate is checked to ensure that it has not expired
- The certificate usage fields are checked to ensure that the certificate is being used for the purpose it was intended (via the keyUsage field)
- The certificate is checked to see that it has been issued by a trusted certificate authority
- See if the certificate has been revoked by the Certificate Authority (due to a possible compromise of the certificate or of the Certificate Authority itself).
  - Via certificate revocation lists (CRLs)
  - Or via an online check called Online Certificate Status Protocol (OCSP).

X509 certificate revocation

- Différentes méthodes de révocation:
  - CRLs (Certificate Revocation List)
  - CRL Distribution Points,
  - Delta CRLs,
  - OCSP (Online Certificate Status Protocol): vérification en temps réel
  - SCVP (Simple Certificate Validation Protocol): protocole déchargeant un client de la vérification complète d'un certificat
- Un certificat comporte dans une extension la ou les méthodes supportées
  - Adresse du serveur (ou des serveurs) CRL et nom du fichier contenant la liste des CRLs, …

X509 Attributes: Key Usage

- digitalSignature: verifying digital signatures used in an authentication service and/or an integrity service
- nonRepudiation -- renamed to contentCommitment : verify digital signatures used to provide a non-repudiation service
- keyEncipherment: used for enciphering private or secret key
- dataEncipherment: used for directly enciphering raw user data without the use of an intermediate symmetric cipher
- keyAgreement: used for key agreement (e.g. Diffie-Hellman)
- keyCertSign: used for verifying signatures on public key certificates
- cRLSign: used for verifying signatures on certificate revocation lists
- encipherOnly: used only for enciphering data while performing key agreement (with keyAgreement)
- decipherOnly: used only for deciphering data while performing key agreement (with keyAgreement)

X509 Attributes: Extended Key Usage

http://tools.ietf.org/html/rfc5280

- id-kp-serverAuth
  - TLS WWW server authentication
  - Key usage bits that may be consistent: digitalSignature, -- keyEncipherment or keyAgreement
- id-kp-clientAuth
  - TLS WWW client authentication
  - Key usage bits that may be consistent: digitalSignature -- and/or keyAgreement
- id-kp-codeSigning
  - Signing of downloadable executable code
  - Key usage bits that may be consistent: digitalSignature

X509 Attributes: Extended Key Usage

http://tools.ietf.org/html/rfc5280

- id-kp-emailProtection
  - Email protection
  - Key usage bits that may be consistent: digitalSignature, -- nonRepudiation, and/or (keyEncipherment or keyAgreement)
- id-kp-timeStamping
  - Binding the hash of an object to a time
  - Key usage bits that may be consistent: digitalSignature -- and/or nonRepudiation
- id-kp-OCSPSigning
  - Signing OCSP responses
  - Key usage bits that may be consistent: digitalSignature -- and/or nonRepudiation