Mobile Device and Platform Security – Part II

John Mitchell
Two lectures on mobile security

› Introduction: platforms and trends
› Threat categories
  - Physical, platform malware, malicious apps
› Defense against physical theft
› Malware threats
› System architecture and defenses
  - Apple iOS security features and app security model
  - Android security features and app security model
› Security app development
  - WebView – secure app and web interface dev
  - Device fragmentation
ANDROID

History and early decisions
Android history

- Android, Inc founded by Andy Rubin around 2005
  - Worked with HTC-built device with a physical keyboard
  - Scrapped Blackberry-like phone when iPhone came out
  - First Android phone HTC Dream, Oct 2008 (T-Mobile G1): touchscreen and keyboard
- Open-source software project
- Backed and acquired by Google
HTC Dream

First phone had
- Android 1.6 (Donut)
- 3.15 megapixel rear camera with auto-focus
- 3.2 inch touchscreen
- Gmail, Google Maps, Search, Google Talk, YouTube, calendar, contacts, alarm
Android ecosystem

- Open-source software distributed by Google
  - Business goal: increase number of users and devices linked to core Google products
- Multiple hardware vendors
  - Each customize software for their products
- Open marketplace for apps
  - Aim for scale

Aside: open-source OS successful pre-mobile
App market

› Self-signed apps

› App permissions
  ▪ granted on user installation

› Open market
  ▪ Bad apps may show up on
  ▪ Shifts focus from remote exploit to privilege escalation
ANDROID PLATFORM

Theft and loss protection
Predictive security
- Look for malicious code in apps

Privacy advisor
- See if app can access private information

Locate lost phone
- Map location and make a sound

Lock and wipe
- Web interface to remotely remove data
- Data backup
- Store and retrieve from cloud

https://www.lookout.com/android
Device lock and unlock

› Similar PIN and fingerprint
› Fingerprint API lets users
  ▪ Unlock device
  ▪ Securely sign in to apps
  ▪ Use Android Pay
  ▪ Purchase on Play Store
ANDROID PLATFORM

Managed code
Managed code overview

› Java programming language
› Bytecode execution environment
   ▪ Verifier
   ▪ Run-time checks
   ▪ Memory safety
› Permission checking
   ▪ Stack inspection
Java language overview

Classes and Inheritance
- Object features
- Encapsulation
- Inheritance

Types and Subtyping
- Primitive and ref types
- Interfaces; arrays
- Exception hierarchy

Generics
- Subtype polymorphism.
  generic programming

Virtual machine
- Loader and initialization
- Linker and verifier
- Bytecode interpreter

Security
- Java “sandbox”
- Type safety
- Stack inspection
Managed code overview

Java programming language
 Bytecode execution environment
  - Verifier
  - Run-time checks
  - Memory safety
Permission checking
  - Stack inspection
Java Implementation

› Compiler and Virtual Machine
  ▪ Compiler produces bytecode
  ▪ Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode

› Why this design?
  ▪ Bytecode interpreter “manages” code execution safely
  ▪ Minimize machine-dependent part of implementation
Java Virtual Machine Architecture

Compile source code

Java Virtual Machine

- Loader
- Verifier
- Linker
- Bytecode Interpreter
JVM Linker and Verifier

› Linker
  ▪ Adds compiled class or interface to runtime system
  ▪ Creates static fields and initializes them
  ▪ Resolves names
    › Checks symbolic names and replaces with direct references

› Verifier
  ▪ Check bytecode of a class or interface before loaded
  ▪ Throw exception if error occurs
Verifier

› Bytecode may not come from standard compiler
  ▪ Evil hacker may write dangerous bytecode

› Verifier checks correctness of bytecode
  ▪ Every instruction must have a valid operation code
  ▪ Every branch instruction must branch to the start of some other instruction, not middle of instruction
  ▪ Every method must have a structurally correct signature
  ▪ Every instruction obeys the Java type discipline
    › Last condition is fairly complicated.
Bytecode interpreter / JIT

- Standard Java virtual machine interprets instructions
  - Perform run-time checks such as array bounds
  - Possible to compile bytecode class file to native code
- Java programs can call native methods
  - Typically functions written in C

- Just-in-time compiler (JIT)
  - Translate set of bytecodes into native code, including checks
- Ahead-of-time (AOT)
  - Similar principles but prior to loading into runtime system
Type Safety of Java

› Run-time type checking
  - All casts are checked to make sure type safe
  - All array references are checked to make sure the array index is within the array bounds
  - References are tested to make sure they are not null before they are dereferenced.

› Additional features
  - Automatic garbage collection
  - No pointer arithmetic

If program accesses memory, that memory is allocated to the program and declared with correct type
Managed code overview

Java programming language
Bytecode execution environment
  - Verifier
  - Run-time checks
  - Memory safety
Permission checking
  - Stack inspection
Managed code overview

› Java programming language
› Bytecode execution environment
  ▪ Verifier
  ▪ Run-time checks
  ▪ Memory safety
› Permission checking
  ▪ Stack inspection
ANDROID PLATFORM

Platform security model
Android platform model

Architecture components
- Operating system, runtime environment
- Application sandbox
- Exploit prevention

Permission system
- Granted at install time
- Checked at run time

Inter-app communication
- Intent system
- Permission redelegation (intent input checking)
Android platform summary

- Linux kernel, browser, SQL-lite database
- Software for secure network communication
  - Open SSL, Bouncy Castle crypto API and Java library
- C language infrastructure
- Java platform for running applications
  - Dalvik bytecode, virtual machine / Android runtime (ART)
Managed code runs in app sandbox

Application development process: source code to bytecode
Security Features

› Isolation
  † Multi-user Linux operating system
  † Each application normally runs as a different user

› Communication between applications
  † May share same Linux user ID
    › Access files from each other
    › May share same Linux process and Dalvik VM
  † Communicate through application framework
    › “Intents,” based on Binder, discussed in a few slides
Application sandbox

- Each application runs with its UID in its own runtime environment
  - Provides CPU protection, memory protection
  - Only ping, zygote (spawn another process) run as root

- Applications announce permission requirement
  - Create a whitelist model – user grants access at install time

- Communication between applications
  - May share same Linux user ID
    - Access files from each other
    - May share same Linux process and runtime environment
  - Or communicate through application framework
    - “Intents,” reference monitor checks permissions
Android platform model

Architecture components
- Operating system, runtime environment
- Application sandbox
- Exploit prevention

Permission system
- Granted at install time
- Checked at run time

Inter-app communication
- Intent system
- Permission redelegation (intent input checking)
Exploit prevention

› Open source: public review, no obscurity

› Goals
  ▪ Prevent remote attacks, privilege escalation
  ▪ Secure drivers, media codecs, new and custom features

› Overflow prevention
  ▪ ProPolice stack protection
    › First on the ARM architecture
  ▪ Some heap overflow protections
    › Chunk consolidation in DL malloc (from OpenBSD)

› ASLR
  ▪ Avoided in initial release
    › Many pre-linked images for performance
  ▪ Later developed and contributed by Bojinov, Boneh
dlmalloc (Doug Lea)

› Stores meta data in band
› Heap consolidation attack
   ▪ Heap overflow can overwrite pointers to previous and next unconsolidated chunks
   ▪ Overwriting these pointers allows remote code execution
› Change to improve security
   ▪ Check integrity of forward and backward pointers
     › Simply check that back-forward-back = back, f-b-f=f
   ▪ Increases the difficulty of heap overflow
Android platform model

Architecture components
- Operating system, runtime environment
- Application sandbox
- Exploit prevention

Permission system
- Granted at install time
- Checked at run time

Inter-app communication
- Intent system
- Permission redelegation (intent input checking)
Android market

- Self-signed apps
- App permissions granted on user installation
- Open market
  - Bad applications may show up on market
  - Shifts focus from remote exploit to privilege escalation
Android permissions

› Example of permissions provided by Android

- “android.permission.INTERNET”
- “android.permission.READ_EXTERNAL_STORAGE”
- “android.permission.SEND_SMS”
- “android.permission.BLUETOOTH”

› Also possible to define custom permissions
Android permission model

```xml
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.NFC" />
<uses-permission android:name="android.permissionINTERNET" />
```

Android permission model

Android platform model

Architecture components
- Operating system, runtime environment
- Application sandbox
- Exploit prevention

Permission system
- Granted at install time
- Checked at run time

Inter-app communication
- Intent system
- Permission redelegation (intent input checking)
Application development concepts

› Activity – one-user task
  ▪ Example: scroll through your inbox
  ▪ Email client comprises many activities

› Intents – asynchronous messaging system
  ▪ Fire an intent to switch from one activity to another
  ▪ Example: email app has inbox, compose activity, viewer activity
    › User click on inbox entry fires an intent to the viewer activity, which then allows user to view that email
Android Intents

- Intent is a bundle of information, e.g.,
  - action to be taken
  - data to act on
  - category of component to handle the intent
  - instructions on how to launch a target activity

- Routing can be
  - Explicit: delivered only to a specific receiver
  - Implicit: all components that have registered to receive that action will get the message
Layers of security

- Each application executes as its own user identity
- Android middleware has reference monitor that mediates the establishment of inter-component communication (ICC)

Source: Penn State group Android security paper
MAC Policy Enforcement in Android. This is how applications access components of other applications via the reference monitor. Component A can access components B and C if permission labels of application 1 are equal or dominate labels of application 2.

Source: Penn State group, Android security tutorial
Security issues with intents

- Sender of an intent may
  - Verify that the recipient has a permission by specifying a permission with the method call
  - Use explicit intents to send the message to a single component

- Receivers must implement appropriate input checking to handle malicious intents
Attack: Permission redelegation

› Idea: an application without a permission gains additional privileges through another application

› Example of the “confused deputy” problem
Permission redelegation

Permission redelegation

 WifiControlApp granted AttackerApp permission without checking

How could this happen?

- App w/ permissions exposes a public interface
- Study in 2011
  - Examine 872 apps
  - 320 of these (37%) have permissions and at least one type of public component
  - Construct attacks using 15 vulnerabilities in 5 apps
- Reference
  - Permission Re-Delegation: Attacks and Defenses, Adrienne Felt, Helen Wang, Alexander Moshchuk, Steven Hanna, Erika Chin, Usenix 2011
Example: power control widget

- Default widgets provided by Android, present on all devices

- Can change Wi-fi, BT, GPS, Data Sync, Screen Brightness with only one click

- Uses Intent to communicate the event of switching settings

- A malicious app without permissions can send a fake Intent to the Power Control Widget, simulating click to switch settings
Principle of least privilege helps but is not solution by itself
Apps with permissions need to manage security

Android platform model

Architecture components
- Operating system, runtime environment
- Application sandbox
- Exploit prevention

Permission system
- Granted at install time
- Checked at run time

Inter-app communication
- Intent system
- Permission redelegation (intent input checking)
ANDROID PLATFORM

Mobile web apps
Mobile web apps
- Use WebView Java objects, implemented based on WebKit browser
- “JavaScript bridge” lets web content use Java objects exported by app

Security problems
- WebView does not isolate bridge access by frame or origin
- App environment may leak sensitive web information in URLs
- WebView does not provide security indicators
- ...
Mobile Web Apps

Mobile web app: embeds a fully functional web browser as a UI element
Obj foo = new Object();
addJavascriptInterface(foo, 'f');
JavaScript Bridge

Java

JavaScript

f.bar();
Security Concerns

Who can access the bridge?

- Everyone
No origin distinction in WebView calls

```javascript
f.bar();
```
Analysis of Public Apps

› How many mobile web apps?
› How many use JavaScript Bridge?
› How many vulnerable?
Experimental Results

› 737,828 free apps from Google Play (Oct ’13)
› 563,109 apps embed a browser
› 219,404 use the JavaScript Bridge
› 107,974 have at least one security violation
Most significant vulnerabilities

1. Loading untrusted web content

2. Leaking URLs to foreign apps

3. Exposing state changing navigation to foreign apps
Loading untrusted web content

“You should restrict the web-pages that can load inside your WebView with a whitelist.”

- Facebook

“…only loading content from trusted sources into WebView will help protect users.”

- Adrian Ludwig, Google
Forms of navigation

// In app code
myWebView.loadUrl("foo.com");

<!-- In HTML -->
<a href="foo.com">click!</a>

<!-- More HTML -->
<iframe src="foo.com"/>

// In JavaScript
window.location = "foo.com";
public boolean shouldOverrideUrlLoading(WebView view, String url) {

    // False -> Load URL in WebView
    // True   -> Prevent the URL load
}

public boolean shouldOverrideUrlLoading(
    WebView view, String url) {

    String host = new URL(url).getHost();
    if (host.equals("stanford.edu"))
        return false;
    log("Overrode URL: " + url);
    return true;
}
Reach Untrusted Content?

› 40,084 apps with full URLs and use JavaScript Bridge
› 13,683 apps (34%) can reach untrusted content
Exposing sensitive information in URLs

- Android apps communicate using intents
  - An implicit intent is delivered to any app whose filter matches
  - An intent filter can declare zero or more `<data>` elements, such as
    - `mimeType` - e.g., `android:mimeType="video/mpeg"`
    - `scheme` - e.g., `android:scheme="http"

- When a WebView loads a page, an intent is sent to the app
  - Another app can register a filter that might match this intent
  - If the URL contains sensitive information, this information can be stolen
Example

OAuth protocol for browser-based web authentication
- Used by Google, Facebook, LinkedIn and other identity providers
- In some configurations, may return a session token as part of a URL

Mobile app developers may try to use OAuth through WebView
- A form of session token is returned as part of a URL
- Delivered through an implicit intent
- May reach any app with filter that specifies protocol scheme my_oauth

Malicious app may steal a session token from a vulnerable app
- Malicious app registers an implicit intent with scheme my_oauth
- Waits for a URL containing the form of session token returned by OAuth.
Handling SSL Errors

onReceivedSslError

1. handler.proceed()
2. handler.cancel()
3. view.loadUrl(...
Mishandling SSL Errors

117,974 apps implement `onReceivedSslError`
29,652 apps (25%) **must** ignore errors
## Primary results

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>% Relevant</th>
<th>% Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsafe Nav</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>HTTP</td>
<td>40</td>
<td>56</td>
</tr>
<tr>
<td>Unsafe HTTPS</td>
<td>27</td>
<td>29</td>
</tr>
</tbody>
</table>
Outdated Apps

The diagram shows the normalized vulnerability rate for outdated and updated apps across different categories:

- **Unsafe Nav**: Outdated Apps have a higher normalized vulnerability rate compared to Updated Apps.
- **HTTP**: The normalized vulnerability rate is similar for both outdated and updated apps.
- **Unsafe HTTPS**: Outdated Apps have a higher normalized vulnerability rate compared to Updated Apps.

The y-axis represents the normalized vulnerability rate, while the x-axis shows the categories of apps.
Libraries

- 29% unsafe nav
- 51% HTTP
- 53% unsafe HTTPS
Additional security issues

Based on 998,286 free web apps from June 2014
Summary

› Mobile web apps
  - Use WebView Java objects, implemented based on WebKit browser
  - “JavaScript bridge” lets web content use Java objects exported by app

› Security problems
  - WebView does not isolate bridge access by frame or origin
  - App environment may leak sensitive web information in URLs
  - WebView does not provide security indicators
  - ...
  - Many browser security mechanism are not automatically provided by WebView
ANDROID PLATFORM

Target fragmentation
Summary

› Android apps can run using outdated OS behavior
  - The large majority of Android apps do this
  - Including popular and well maintained apps

› Outdated security code invisibly permeates the app ecosystem
  - “Patched” security vulnerabilities still exist in the wild
  - “Risky by default” behavior is widespread
“If the device is running Android 6.0 or higher... [the app] must request each dangerous permission that it needs while the app is running.

- Android Developer Reference
“If the device is running Android 6.0 or higher and your app's target SDK is 6.0 or higher [the app] must request each dangerous permission that it needs while the app is running.

- Android Developer Reference
App
Collecte
Outdatedness
App
Update
dNegligent
Outdatedness
Outdatedness
Android
5.0
Released
Android
5.1
Released
App
Update
d
Android
6.0
Released
App
Collecte
d
Fragment Injection

Malicious Intent

Extra.SHOW_FRAGMENT "Attacked Fragment"
Extra.SHOW_FRAG_ARG
Data
Other Extras

Vulnerable App

PreferenceActivity

Attacked Fragment

securityintelligence.com/new-vulnerability-android-framework-fragment-injection/
Fragment Injection

› Fixed in Android 4.4
› Developers implement isValidFragment to authorize fragments

// Put this in your app
protected boolean isValidFragment(String fName) {
    return MyFrag.class.getName().equals(fName);
}
Fragment Injection

Vulnerable if:
- Targets 4.3 or lower (31%)
- Some class inherits from PreferenceActivity (4.8%)
- That class is exported (1.1%)
- That class does not override isValidFragment (0.55%)

4.2% of apps vulnerable if no fix was ever implemented
Mixed Content in WebView

› Major web browsers block Mixed Content
› In Android 5.0, WebViews block Mixed Content by default
› Can override default with setMixedContentMode()
SOP for file:// URLs in WebView

› Android 4.1 separate file:// URLs are treated as unique origins
› Can override with setAllowFileAccessFromFileURLs()
Summary

› Android apps can run using outdated OS behavior
  - The large majority of Android apps do this
  - Including popular and well maintained apps

› Outdated security code invisibly permeates the app ecosystem
  - “Patched” security vulnerabilities still exist in the wild
  - “Risky by default” behavior is widespread
Two lectures on mobile security

› Introduction: platforms and trends
› Threat categories
   ▪ Physical, platform malware, malicious apps
› Defense against physical theft
› Malware threats
› System architecture and defenses
   ▪ Apple iOS security features and app security model
   ▪ Android security features and app security model
› Security app development
   ▪ WebView – secure app and web interface dev
   ▪ Device fragmentation

Tues

Thurs
Comparison: iOS vs Android

› App approval process
  ▪ Android apps from open app store
  ▪ iOS vendor-controlled store of vetted apps

› Application permissions
  ▪ Android permission based on install-time manifest
  ▪ All iOS apps have same set of “sandbox” privileges

› App programming language
  ▪ Android apps written in Java; no buffer overflow...
  ▪ iOS apps written in Objective-C
## Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>iOS</th>
<th>Android</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open market</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Closed market</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vendor signed</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Self-signed</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>User approval of permissions</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Managed code</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Native code</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>iOS</th>
<th>Android</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Open market</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Closed market</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Vendor signed</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Self-signed</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>User approval of permissions</td>
<td>x</td>
<td></td>
<td>7-&gt;8</td>
</tr>
<tr>
<td>Managed code</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Native code</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two lectures on mobile security

› Introduction: platforms and trends
› Threat categories
  ▪ Physical, platform malware, malicious apps
› Defense against physical theft
› Malware threats
› System architecture and defenses
  ▪ Apple iOS security features and app security model
  ▪ Android security features and app security model
› Security app development
  ▪ WebView – secure app and web interface dev
  ▪ Device fragmentation