



# Seven Ways to Hang Yourself with Google Android

Yekaterina Tsipenyuk O'Neil

Principal Security Researcher

Erika Chin

Ph.D. Student at UC Berkeley



# Yekaterina Tsipenyuk O'Neil

- Founding Member of the Security Research Group at Fortify (now an HP Company)
- Code audits, identifying insecure coding patterns, and providing security content for Fortify's software security products
- B.S. and M.S. in CS from UC San Diego

# Erika Chin

- Ph.D. student in Computer Science at UC Berkeley (Security research group)
- B.S. from University of Virginia
- Research interest in improving mobile phone security
- Recently presented at MobiSys 2011 on vulnerabilities stemming from inter-application communication in Android

# Overview

- Introduction to Google Android
- Seven Ways to Hang Yourself
- Results of Empirical Analysis
- Conclusion

# Introduction to GOOGLE ANDROID



# Introduction to Google Android

- Android architecture
- Security model
- Application breakdown
  - Android manifest
  - Components
  - Inter-component communication

# Android Architecture

- Applications
- Application framework (SDK)
- Dalvik virtual machine
  - Customized bytecode (.dex files)
- Native libraries
  - Graphics, database management, WebKit, etc.
  - Accessed through Java interfaces
- Linux kernel
  - Device drivers, memory management, etc.

Higher



Lower

# Security Model

- Applications have unique UIDs
  - Run as separate processes on separate VMs
  - Typically cannot read each other's data and code
- Linux-style file permissions
- Android permissions protect
  - Access to sensitive APIs
  - Access to content providers
  - Inter- and intra-application communication



# Application Breakdown

- Applications are divided into *components*
- 4 types of components
  - Activities
  - Services
  - Broadcast Receivers
  - Content Providers

# Android Manifest

Each application contains a manifest

```
<manifest ...>
  <application>
    <activity android:name=".MyActivity">...</activity>
    <receiver android:name=".MyReceiver">...</receiver>
  </application>

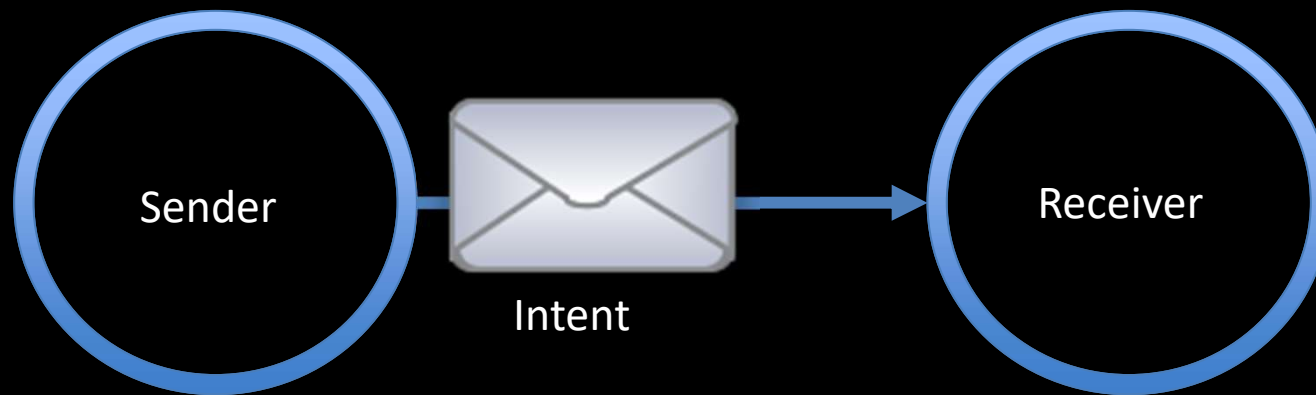
  <uses-sdk android:minSdkVersion="8" />
  <uses-feature android:name="android.hardware.CAMERA"/>

  <uses-permission
    android:name="android.permission.INTERNET" />
  <uses-permission
    android:name="android.permission.CAMERA" />

  <permission android:name="com.emc.NewPermission" ...>
</manifest>
```

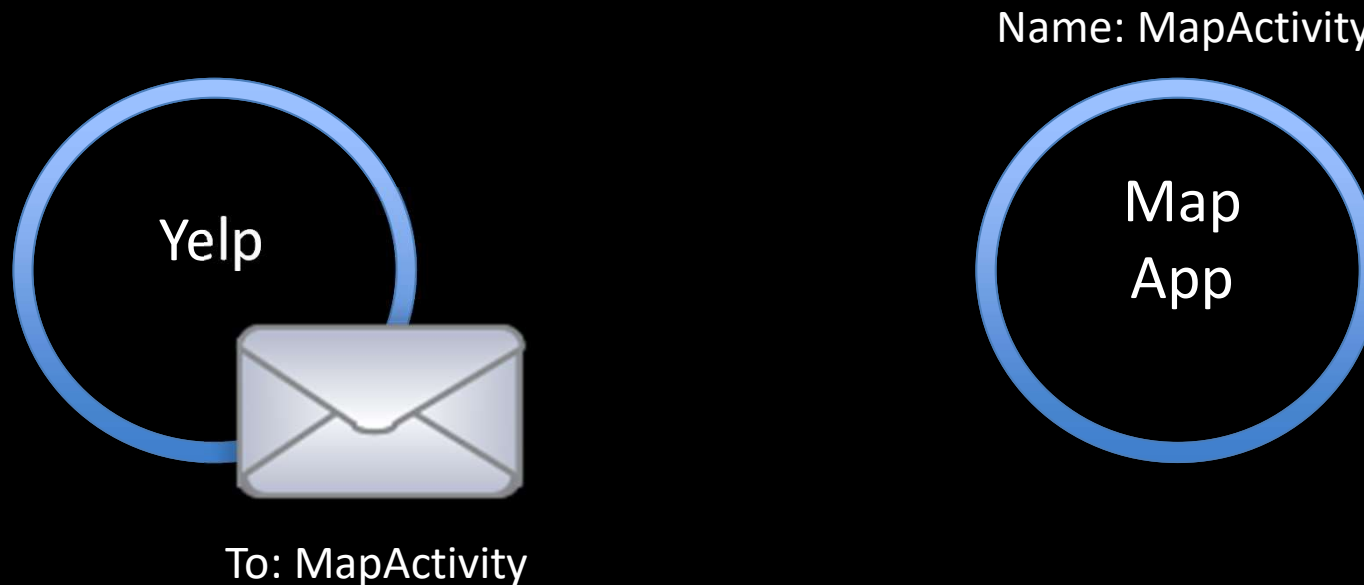
# Inter-Component Communication

- Uses Intents (messages)
- Intents can be sent between components
  - Used for both intra- and inter-application communication
  - Event notifications (including system events)



# Explicit Intents

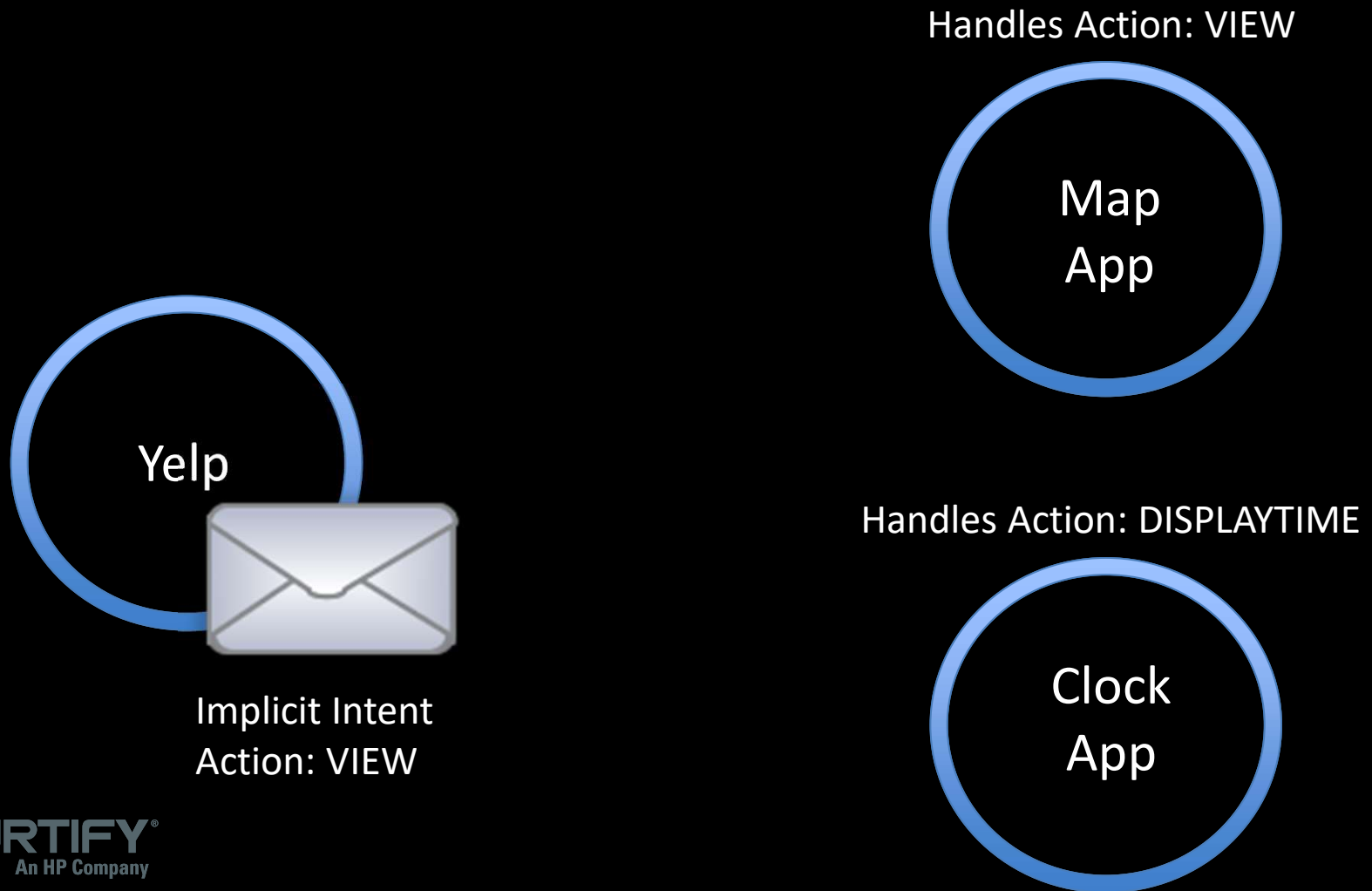
- Exact recipient is specified



Only the specified destination receives this message

# Implicit Intents

- Left up to the platform to decide where it should be delivered



# Implicit Intents

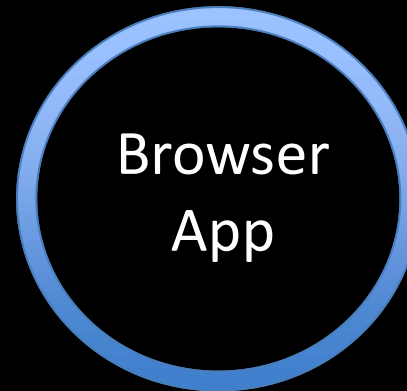


Implicit Intent  
Action: VIEW

Handles Action: VIEW



Handles Action: VIEW



# Explicit vs. Implicit Intents

## Explicit Intent:

```
Intent i = new Intent();  
i.setClassName("some.pkg.name",  
    "some.pkg.name.TestDestination");
```

## Implicit Intent:

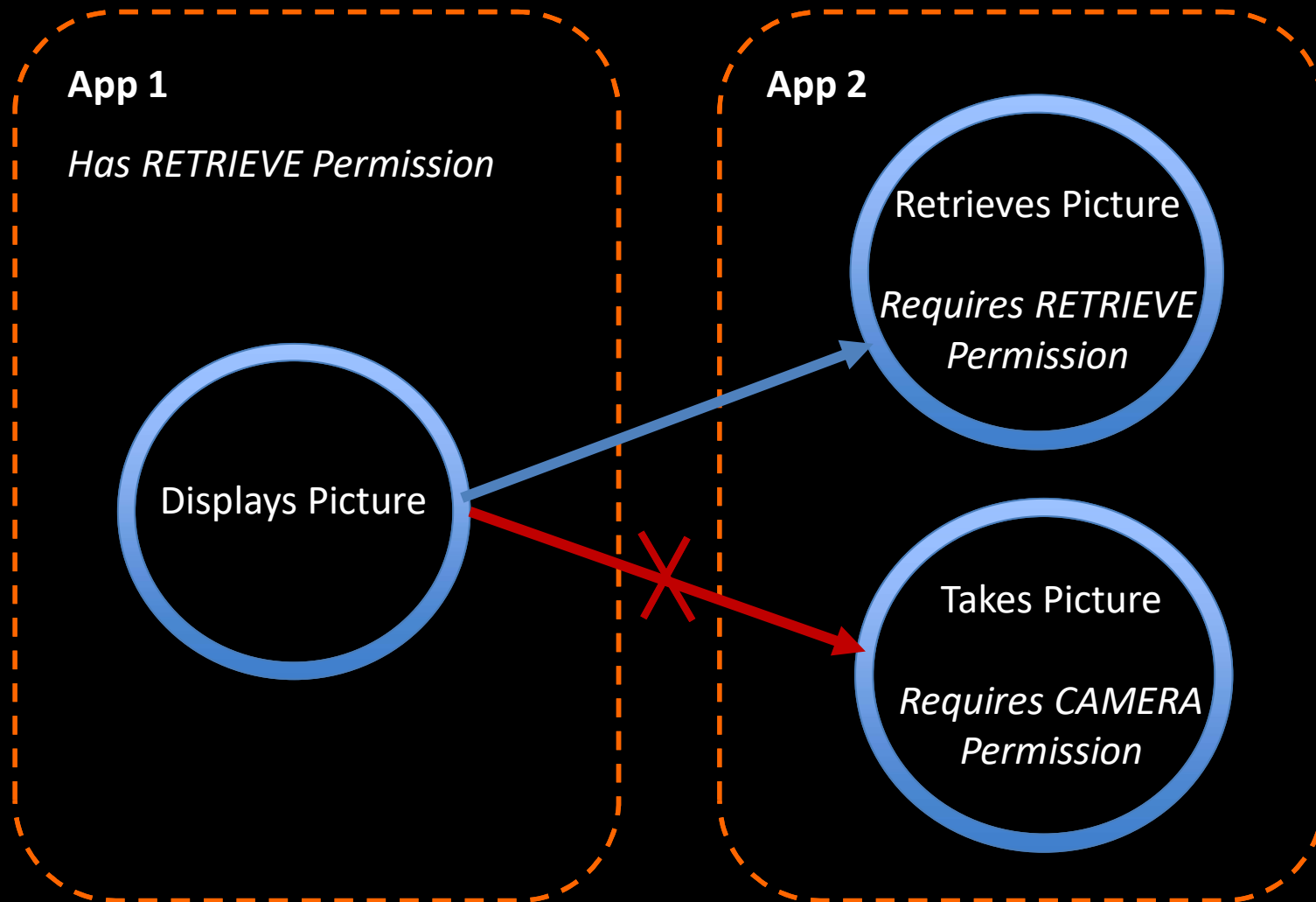
```
Intent i = new Intent();  
i.setAction("my.special.action");
```

# Component Protection

- Components can be made accessible to other applications (exported) or be made private
  - Default is private 😊
  - Converted to public when component is registered to receive implicit Intents 😞
- Components can be protected by permissions



# Component Permissions



# Seven Ways to Hang Yourself with GOOGLE ANDROID

# Google Android Vulnerabilities

1. Unauthorized Intent Receipt
2. Intent Spoofing
3. Persistent Messages: Sticky Broadcasts
4. Insecure Storage
5. Insecure Network Communication
6. SQL Injection
7. Overprivileged Applications

# 1. Unauthorized Intent Receipt

- Attack: Malicious app intercepts an Intent
- Arises when Intents are implicit (public) and do not require receiving components to have strong permissions
- Can leak sensitive program data and/or change control flow

```
Intent i = new Intent();  
i.setAction("my.special.action");  
[startActivity|sendBroadcast|startService](i);
```

# 1. Example

IMDb App

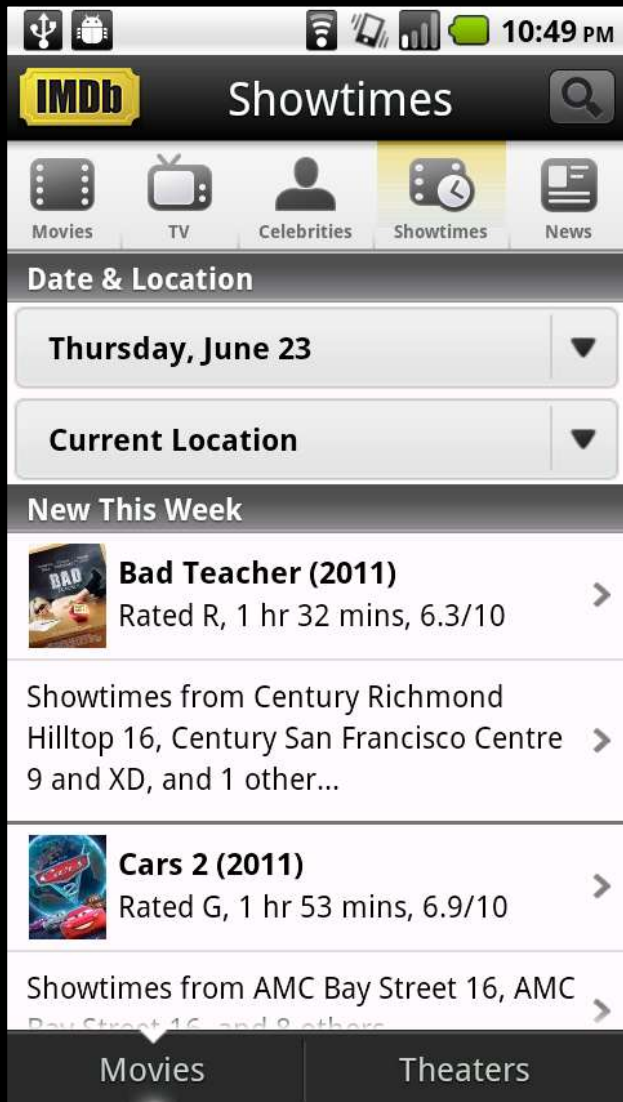
Handles Actions:  
*willUpdateShowtimes,*  
*showtimesNoLocationError*

Showtime  
Search



Implicit Intent  
Action:  
*willUpdateShowtimes*

Results UI



# 1. Example

IMDb App

Handles Actions:  
*willUpdateShowtimes,*  
*showtimesNoLocationError*

Showtime  
Search



Implicit Intent  
Action:  
*willUpdateShowtimes*

Results UI

# 1. Example

IMDb App

Showtime  
Search



Implicit Intent  
Action:  
*willUpdateShowtimes*

Eavesdropping App



Handles Action:  
*willUpdateShowtimes,*  
*showtimesNoLocationError*

Malicious  
Receiver

Sending Implicit Intents makes communication public



# 1. Recommended Fix

```
Intent i = new Intent();  
i.setClassName("some.pkg.name",  
    "some.pkg.name.TestDestination");
```

or

```
Intent i = new Intent();  
i.setAction("my.special.action");  
sendBroadcast(i, "my.special.permission");
```

## 2. Intent Spoofing

- Attack: Malicious app sends an Intent, resulting in data injection/state change
- Arises when components are public and do not require senders to have strong permissions

```
<receiver android:name="my.special.receiver">  
  <intent-filter>  
    <action android:name="my.intent.action" />  
  </intent-filter>  
</receiver>
```

# 2. Example



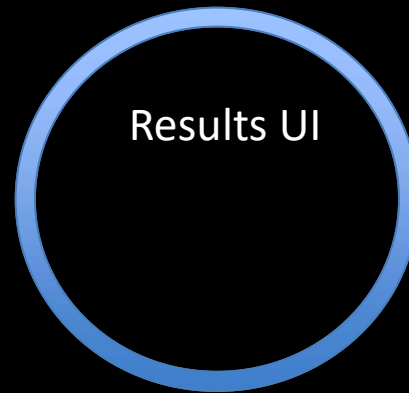
**Malicious Injection App**



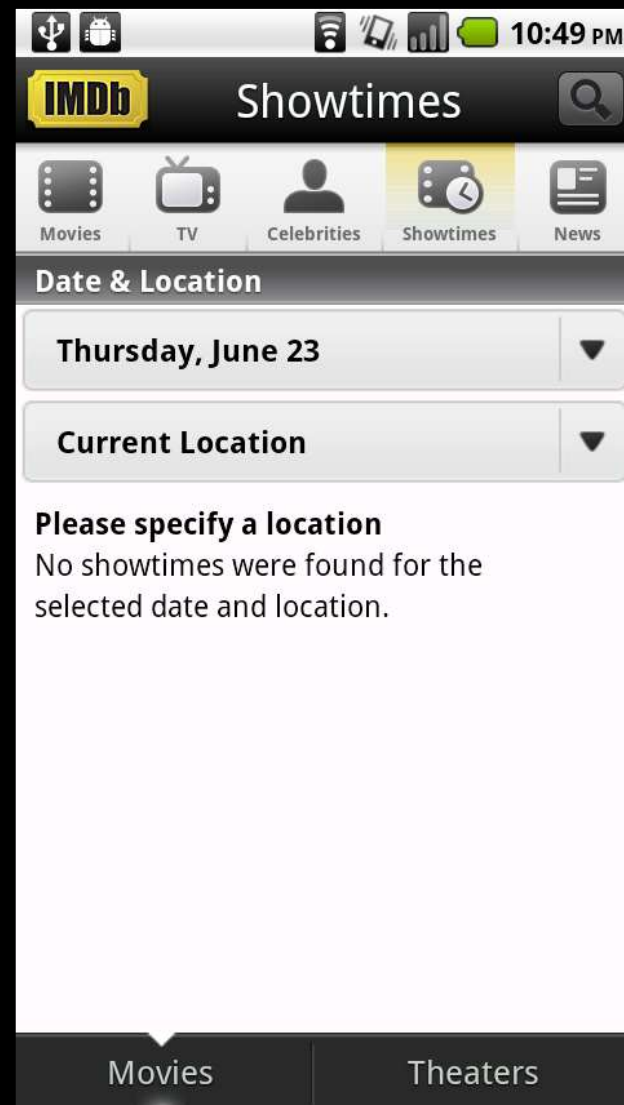
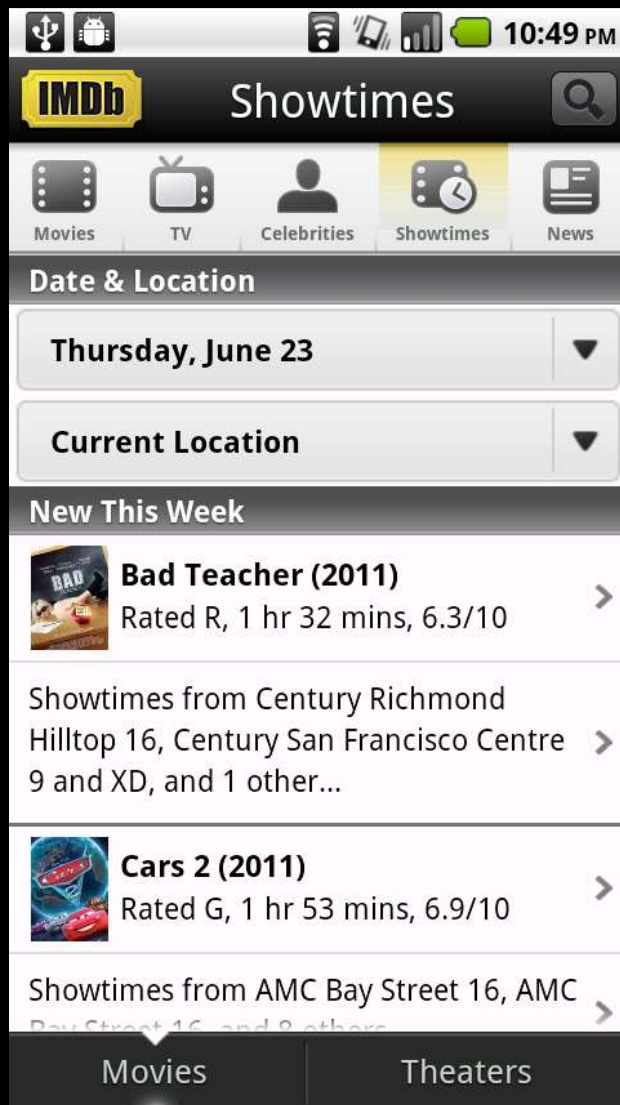
Action:  
*showtimesNoLocationError*

**IMDb App**

Handles Action:  
*willUpdateShowtimes,*  
*showtimesNoLocationError*



Receiving Implicit Intents makes the component public



Typical case

## 2. Recommended Fix

```
<receiver android:name="my.special.receiver"  
    android:exported=false>  
    ...  
</receiver>
```

or

```
<receiver android:name="my.special.receiver"  
    android:exported=true  
    android:permission="my.own.permission">  
    ...  
</receiver>
```

# 3. Persistent Messages: Sticky Broadcasts

- Broadcast Intent
  - One-to-many message
  - Delivered to all components registered to receive them
- “Sticky” Broadcast Intents are broadcasts that persist
  - Remain accessible after they are delivered
  - Re-broadcast to future Receivers

## 3. Problems with Persistent Messages

- Can leak sensitive program data
- Cannot be restricted to a certain set of receivers (cannot require a receiver to have a permission)
- Stays around after it has been sent
  - But anyone with BROADCAST\_STICKY permission can remove a sticky Intent you create

# 3. Example

Sticky broadcasts:



Sticky broadcast 1



Sticky broadcast 2



Sticky broadcast 3



**Malicious App**

Requests BROADCAST\_STICKY  
Permission

Victim app

Receiver  
(expects sticky  
broadcast 2)



Newly connected receiver will be unaware of the change



### 3. Recommended Fix

- Use regular broadcasts protected by the receiver permission instead, if possible
- Don't put sensitive data in sticky broadcast messages

## 4. Insecure Storage

- Can compromise sensitive program data
  - Passwords, Location, Contacts, etc.
- SD card
  - Files on the SD Card are world-readable
  - Files stay even after the application that wrote them is uninstalled

## 4. Example: Kindle App

- Saves e-books (.mbp and .prc) in a folder on the SD card
  - Some can be read by other applications (depends on the DRM)
- Saves covers of books
  - Privacy violation
- Folder is retained after uninstallation of Kindle
  - Next mobile owner can see all books

## 4. Recommended Fix

- Write to the application's SQLite database
- Write to the device's internal storage and make the file private  
(Context.MODE\_PRIVATE)
- If it must be on SD card, encrypt the data  
(AND don't store the key on the SD card!)

## 5. Insecure Network Communication

- Be careful of leaking sensitive data through HTTP connections

# 5. Examples

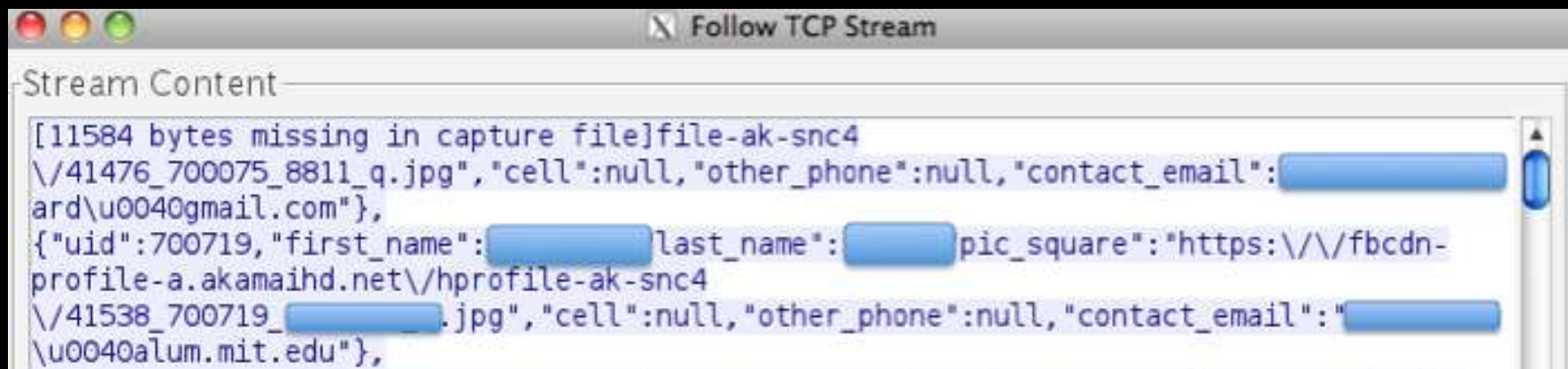
- Twitter: Tweets are sent in the clear



```
Stream Content
POST /1/statuses/update.json?status=Somehow%20I%27m%20thirstier%20after%20juice%20social%
20hour.&lat=37.87547546&long=-122.25871363000002 HTTP/1.1
Accept-Encoding: gzip
Content-Length: 0
Host: api.twitter.com
Connection: Keep-Alive
HTTP/1.1 200 OK
```

## 5. Examples

- Facebook: Despite having a fully encrypted traffic option on the web app, the mobile app sends everything in the clear



The screenshot shows a window titled "Follow TCP Stream" with a "Stream Content" pane. The content is a JSON array of objects, each representing a user profile. The data is unencrypted and clearly visible. The first object has a "uid" of 700719 and a "first\_name" field. The second object has a "uid" of 700719 and a "first\_name" field. The "pic\_square" field in the first object contains a URL starting with "https://".

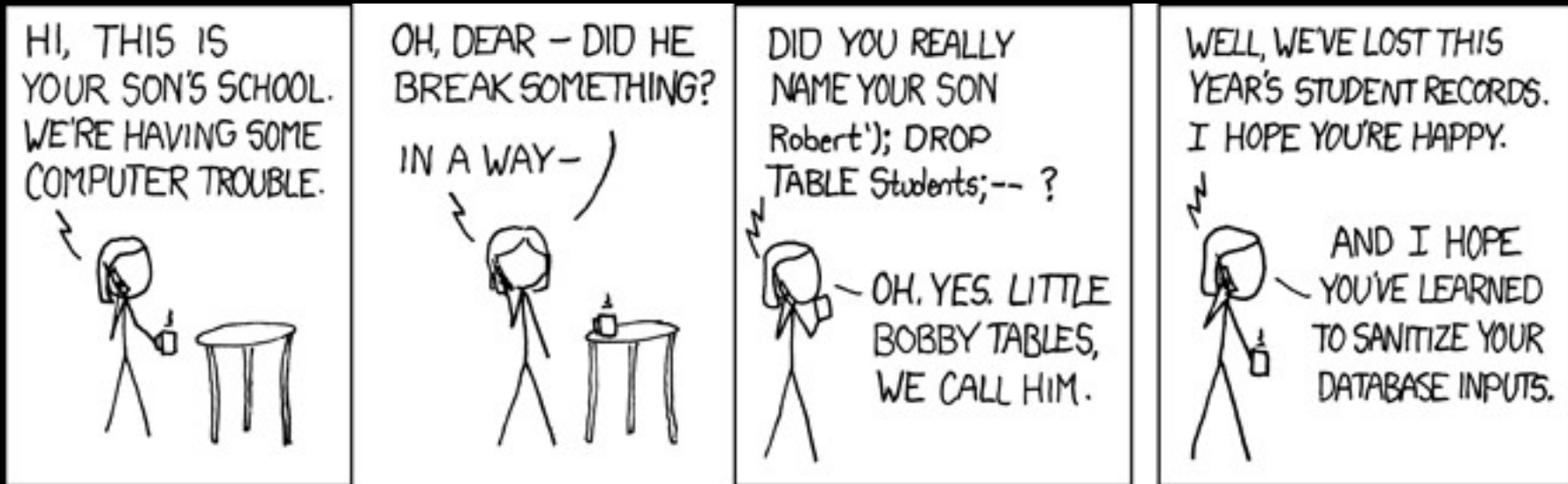
```
[11584 bytes missing in capture file]file-ak-snc4
\/41476_700075_8811_q.jpg", "cell":null, "other_phone":null, "contact_email": "
ard\u0040gmail.com"},
{"uid":700719, "first_name": "last_name": "pic_square": "https://fbcdn-
profile-a.akamaihd.net/hprofile-ak-snc4
\/41538_700719_ .jpg", "cell":null, "other_phone":null, "contact_email": "
\u0040alum.mit.edu"},
```

## 5. Recommended Fix

- When using WebViews, connect to HTTPS when possible
- Don't send passwords in the clear
- Treat your mobile app as you would a web app



# 6. SQL Injection



- SQLiteDatabase class methods susceptible to general SQL Injection:
  - delete
  - execSQL
  - rawQuery
  - update
  - updateWithNoConflict

## 6. SQL Injection: Query String Injection

- Unlike typical SQL injection, Query String Injection allows malicious users to view unauthorized records, but not to alter the database
- Query string injection occurs when:
  1. Data enters a program from an untrusted source
  2. The data is used to dynamically construct a part of a SQL query string

## 6. Example

```
c = invoicesDB.query(  
    Uri.parse(invoices),  
    columns,  
    "productCategory = '" +  
        productCategory + "' and  
    customerID = '" + customerID + "'",  
    null,  
    null,  
    null,  
    "'" + sortColumn + "' asc",  
    null  
);
```

## 6. Example

productCategory = "Fax Machines"

customerID = "12345678"

sortColumn = "price"

```
select * from invoices
```

```
  where productCategory = 'Fax Machines' and
```

```
  customerID = '12345678'
```

```
  order by 'price' asc
```

Returns invoice records for ONE customer

## 6. Example

```
productCategory = "Fax Machines" or productCategory = \ ""  
customerID = "12345678"  
sortColumn = "\ " order by 'price'
```

```
select * from invoices  
  where productCategory = 'Fax Machines' or  
         productCategory = "" and customerID =  
         '12345678' order by ""  
         order by 'price' asc
```

Returns invoice records for ALL customers

## 6. Recommended Fix

Use parameterized queries!!!

```
c = invoicesDB.query(  
    Uri.parse(invoices),  
    columns,  
    "productCategory = ? and customerID = ?",  
    {productCategory, customerID},  
    null,  
    null,  
    "" sortColumn + "'asc", null  
);
```

# 7. Overprivileged Applications

- Overprivileged applications – applications that request more permissions than the app actually requires

## 7. Why is this dangerous?

- Violates the principle of least privilege
- Any vulnerability may give the attacker that privilege
- Users may get accustomed to seeing and accepting unnecessary permission requests from third party applications



## 7. How can this occur?

- Common causes
  - Confusing permission names
  - Testing artifacts
  - Using deputies
  - Error propagation through message board advice
  - Related methods

# 7. Example: Using Deputies

## App 1

Does not need  
CAMERA permission

Wants Picture



Implicit Intent  
Action: *IMAGE\_CAPTURE*

## Camera App

Needs CAMERA  
permission

Takes  
Picture

Handles Action:  
*IMAGE\_CAPTURE*

# 7. Example: Bad Message Board Advice

Third hit on Google search

3 Answers

active

oldest

votes



It broadcasts whenever you connect or disconnect from Wifi, in other words, Wifi State.

8

You can do it using the following intent-filters:

- android.net.wifi.WIFI\_STATE\_CHANGED
- action android:name="android.net.wifi.STATE\_CHANGE
- android.net.wifi.suppliment CONNECTION\_CHANGE



Which needs the following permission:

- uses-permission android:name="android.permission.ACCESS\_WIFI\_STATE"

Not true for android.net.wifi.STATE\_CHANGE

## 7. Recommended Fix

- Have Google improve their documentation
- Use tools to identify overprivilege

# Empirical Results Analyzing Applications Built on GOOGLE ANDROID

# Summary of Results

Type	# of Vulnerable Apps
Unauthorized Intent Receipt	50%
Intent Spoofing	40%
Persistent Messages: Sticky Broadcasts	6%
Insecure Storage	28%
SQL Injection	17%
Overprivileged Applications	31%

# Challenges

- Coding conventions
  - Callbacks and reflection are a challenge for traditional static analysis techniques
- Documentation
  - Google provides little documentation, which is often incomplete or out-of-date

# Documentation Analysis

- Android 2.2 documents permission requirements for only 78 out of 1207 API calls found by the Berkeley team
  - 6 out of 78 are incorrect
  - 1 of the documented permissions does not exist



# Vulnerability Identification

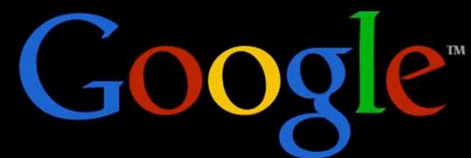
- Of the 7 vulnerabilities presented:
  - 5 vulnerability categories currently can be identified by Fortify's SCA tools
  - 4 vulnerability categories currently can be identified by UC Berkeley's tools
  - 6 categories will be integrated into the current tools

# Related Work

- Adrienne Porter Felt, David Wagner, UC Berkeley ['11] - Overprivilege
- Will Enck, Penn State ['09] – information leakage through Broadcast Intents
- Jesse Burns ['09] – other common developers' errors
- Dan Wallach – WiFi leaks

# Conclusion

- Android has its own set of security pitfalls
- Static analysis can help developers avoid these problems
- UC Berkeley and Fortify are working to incorporate state-of-the-art static analysis into Fortify's tools



# Seven Ways to Hang Yourself with Google Android

Yekaterina Tsipenyuk O'Neil

Principal Security Researcher

Erika Chin

Ph.D. Student at UC Berkeley

