Building Android Sandcastles in Android's Sandbox

Nils

18th October 2010

MWR InfoSecurity

MWR InfoSecurity

Contents

MWR INFOSECURITY

1 Introduction

Previously researchers and attackers focussed on local kernel vulnerabilities in order to escalate privileges on Android mobile phones. This paper will take a look at the implementation of the sandbox and potential security vulnerabilities in the system and third party applications.

Section 2 will briefly look at previous research on Android phones. Section 3 will cover the foundation of the Android sandbox, packages, applications and potential attack vectors. Specific classes of weaknesses that may be encountered on phones will be discussed in Section 4, followed by examples found on an actual Phone in Section 5. Finally the paper will be concluded in Section 6.

1.1 Goal

This paper should lay the foundation for a security review of an actual Android phone from a user mode perspective. Using the information contained in this paper a security auditor should be able to review a preinstalled Android system and third party application with respect to their impact on the security of the Android phone.

2 Previous Research

Jesse Burns [8] presented on his exploratory findings about Android's security model in 2009. As part of his presentation he released tools for Intent fuzzing, Intent sniffing, Manifest file exploring and basic testing of packages.

Previous exploits, which were used to escalate privileges on an Android phone where local Linux kernel vulnerabilities that were either ported to Android (e.g. [6] or specific to the Android platform (e.g. [7]).

3 Android Basics

This section contains information on the specifics of the Android operating system and the Android application development process, which are both of importance when auditing Android applications and systems. Most of the following sections can be followed up in much more detail in the Android Developer Guide [1]. However the condensed information presented here should be sufficient to conduct a basic security review of an Android application.

3.1 The Sandbox

Android uses the Linux user and group model in order to perform sandboxing of applications. In general every application will create a user at installation time and every process of the application will run in the context of the user. However there are exceptions to this which are introduced by shared user IDs (see section 3.3.1).

Communication out of the sandboxed context is done using several IPC mechanisms which will be discussed later on in this paper. These IPC interfaces are an important attack vector for privilege escalation on a phone.

The Java virtual machine is not used for sandboxing, in fact any application is able to spawn other processes or load native code.

3.2 Application Development

Google distributes a Software Development Kit (SDK) for Java application development. Java is the preferred language for developing Android applications. In addition to the SDK Google also provides a Native Development Kit (NDK) which allows a developer to create native applications or libraries for Android phones.

Most of the following examples will focus on Applications developed in Java, as this is the case for the majority of Android applications that are currently available.

3.2.1 AndroidManifest.xml

Every Android application has an associated AndroidManifest.xml file in its root directory. This file is used by the Android system to determine important properties of the application. These properties include the names, classes and settings of components of the application. For a security audit it is important to know which of these components are exported and accessible with which permissions. The manifest XML configuration file defines permissions which are used by the application itself. A user is asked to grant these permissions to the application on installation if the application is installed using the Application Manager. Furthermore the application can define additional new permissions using this file, which may be enforced when accessing the application's components. The permissions required to access the application can either be set for the specific application or for its components.

Below is an example of an AndroidManifest.xml taken from the Android source code for the ContactsProviders content provider:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.providers.contacts"
        android:sharedUserId="android.uid.shared">

    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS" />
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="android.permission.READ_SYNC_STATS" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.USE_CREDENTIALS" />
    <uses-permission
android:name="com.google.android.googleapps.permission.GOOGLE_AUTH" />
    <uses-permission
android:name="com.google.android.googleapps.permission.GOOGLE_AUTH.cp" />
    <uses-permission android:name="android.permission.SUBSCRIBED_FEEDS_READ" />
    <uses-permission android:name="android.permission.SUBSCRIBED_FEEDS_WRITE" />

    <application android:process="android.process.acore"
        android:label="@string/app_label"
```

```
        android:icon="@drawable/app_icon">

        <provider android:name="ContactsProvider2"
            android:authorities="contacts;com.android.contacts"
            android:label="@string/provider_label"
            android:multiprocess="false"
            android:readPermission="android.permission.READ_CONTACTS"
            android:writePermission="android.permission.WRITE_CONTACTS">
            <path-permission
                    android:path="/contacts/search suggest query"
                    android:readPermission="android.permission.GLOBAL_SEARCH" />
        </provider>

        <provider android:name="CallLogProvider"
            android:authorities="call log"
            android:syncable="false" android:multiprocess="false"
            android:readPermission="android.permission.READ_CONTACTS"
            android:writePermission="android.permission.WRITE CONTACTS">
        </provider>

        <!-- TODO: create permissions for social data -->
        <provider android:name="SocialProvider"
            android:authorities="com.android.social"
            android:syncable="false"
            android:multiprocess="false"
            android:readPermission="android.permission.READ CONTACTS"
            android:writePermission="android.permission.WRITE CONTACTS" />
    </application>
</manifest>
```

In this example the application exports three IPC endpoints, all of them content providers. This is done using the <provider> tags. Access to these content providers is restricted using readPermission and writePermission attributes (see Section 3.4.3 for details). The application itself requests a set of permissions using the <uses-permission> tags.

Notably in this specific case a shared user id is used, as can been seen from the defined sharedUserId attributes. As a result the ContactsProvider application won't create its own user, but will share a User Id with other applications that use the same User Id instead. See Section 3.3.1 for more details on User Id sharing.

3.2.1.1 Manifests in Binary Packages

When auditing a phone the scenario will often arise where access to source files is not available, and thus no access to the original AndroidManifest.xml file will be available. However, in these cases there will be access to the binary APK packages. APK packages are very similar to JAR files in Java and can be extracted using ZIP decompression. For efficiency reasons the manifest file is stored in a binary format inside this ZIP file as well. In order to access the content of this file in a user readable format it is possible to use the aapt tool which is distributed with the Android SDK, as can be seen here:

```
$ aapt d xmltree com.htc.WeatherWidget.apk AndroidManifest.xml
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:sharedUserId(0x0101000b)="com.htc.rosie.uid.shared" (Raw:
"com.htc.rosie.uid.shared")
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.00" (Raw: "1.00")
```

```
    A: package="com.htc.WeatherWidget" (Raw: "com.htc.WeatherWidget")
    E: uses-sdk (line=0)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x7
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x7
    E: uses-permission (line=8)
      A: android:name(0x01010003)="android.permission.GET_ACCOUNTS" (Raw:
"android.permission.GET_ACCOUNTS")
    E: uses-permission (line=9)
      A: android:name(0x01010003)="android.permission.READ SYNC SETTINGS" (Raw:
"android.permission.READ SYNC SETTINGS")
    E: application (line=11)
      A: android:label(0x01010001)=@0x7f060000
      A: android:icon(0x01010002)=@0x7f020005
      A: android:description(0x01010020)=@0x7f06001f
      E: activity (line=12)
        A: android:theme(0x01010000)=@0x7f07000b
        A: android:name(0x01010003)=".OptionActivity" (Raw: ".OptionActivity")
        A: android:screenOrientation(0x0101001e)=(type 0x10)0x5
        E: intent-filter (line=15)
          E: action (line=16)
            A: android:name(0x01010003)="CityOption" (Raw: "CityOption")
          E: category (line=17)
            A: android:name(0x01010003)="android.intent.category.DEFAULT" (Raw:
"android.intent.category.DEFAULT")
          E: data (line=18)
            A: android:mimeType(0x01010026)="com.htc.WeatherWidget/city_option"
(Raw: "com.htc.WeatherWidget/city_option")
```

3.3 Permissions

Android uses a permission model in order to prevent applications from interfering with the system in unexpected or insecure ways. When an application is installed a user is asked to grant permissions to the application. Once installed these permissions cannot be extended and only revoked by removing the application.

An application defines required permissions in its manifest files using one or multiple <uses-permission> tags. Many permissions are defined on any Android system by default and additional permissions can be added by any application. As mentioned before, the sandbox is implemented using the Linux user model and communication between applications and the system takes place using inter-process communication (see Section 3.4). Consequently Android uses the Linux group model for enforcing some permissions where appropriate. Where that is not possible or in the case of third party permissions, these are checked on the interface of the inter-process communication. An application can either decide to check permissions itself by checking messages passed to it or it can use the AndroidManifest.xml file to define required permissions for its exported communication endpoints.

3.3.1 Implicit Permission sharing

Android allows applications to share User Ids. This can be done to improve the efficiency of an application and allows applications to share processes, effectively removing the needed for inter-process communication. Applications can only share the same user id if they are either signed by the same developer or installed as a default system application.

The sharing of User Ids has a great impact on the security of the overall system. By sharing the same User Ids across multiple applications they will also effectively share

the same permissions. As now each of these applications - sharing the same user id - will also share the same permissions the sum of their attack surfaces will be open to an attacker to gain any of their permissions.

This is most commonly a problem with applications preinstalled on the phone, as other applications which share User Ids will be from the same developer and thus should be trusted to the same extent by a user.

An application can request a specific user id by defining the "sharedUserId" attribute as an attribute of the application tag in an AndroidManifest.xml file. If Android allows an application to use this specific user id, it can also decide to share processes with the other applications by defining the process attribute. The process attribute can either be defined for the application tag or for any of the tags defining inter-process communication endpoints (see section 3.4).

3.4 Inter-Process Communication

Inter-process communication (IPC) is a very heavily used feature on the Android platform. This is required as applications running across multiple processes with different permissions need a secure method of communication. In order to minimise the performance impact of IPC on the system a "Binder" supports the communication in the kernel. The Binder is responsible for accepting and routing of any incoming messages. Messages sent through the binder are serialised in parcel objects.

In this paper any component that actively accepts messages through IPC will be referred to as an IPC endpoint. In Android applications four different types of IPC endpoints will regularly be used: Services, content-providers, broadcast receivers and activities. Instrumentations are a fifth type of IPC endpoints, however these should not play a role from a security perspective, as they are removed from any application as soon as they are published. Instrumentations are only enabled during the development and debugging phase of an application. If an instrumentation is still enabled on a phone, this might immediately become a security risk. However, this is a very unlikely scenario, as these instrumentation should never make its way to a phone or an exported application.

3.4.1 Intents

Intents are used for communication with activities, services and broadcast receivers. They implement a special parcel object. An intent object is a data structure holding information that is either passed to an IPC endpoint or returned from it.

An intent object is defined by several properties:-

Component name defines the name of the component that should handle the intent. This usually consists of the package name combined with the name of the class handling the intent. This information is then used by Android to identify the application that handles the message. This is an optional property and Android in some cases will identify the target application by other means.

Action is a string that describes the action that should be performed or in the case of broadcast receivers an event that has occurred. This is an optional property.

Data is the URI on which an action should be performed. In many cases these are content URI's which are resolved by the endpoint using content resolvers, if the appropriate permissions allow the endpoint to do so. This is an optional property.

Category is a string which generally holds additional information about the component that will handle the intent. An arbitrary number of these categories can be set for any intent.

Extras is a set of key-value pairs that are passed to the handling component. Extras can be any of the Java primitive data types, arrays of them, strings and serialisable objects. Extras are optional.

Flags is an integer value that contains flags which are handled differently by the different types of IPC endpoints. Flags are usually defined for activity and broadcasts intents.

An application can choose to restrict the set of messages by defining intent filters. This is done as part of the definition of the endpoint in the AndroidManifest.xml file. This is not a security mechanism and only used to filter for specific messages.

3.4.2 Services

Services are non visual components of an application. They run in the background and often fulfill time consuming tasks. A caller can bind to a service and afterwards use methods exposed by the service.

A service is simply implemented by extending the Service class in Java. This will automatically lead to the generation of the IPC interface and a stub class which is used to interface with the service. Any public method of the implemented service is available on this remote interface.

Services are defined in the Manifest file using a <service> tag which is contained in the application tag. From a security perspective the following attributes are of interest:

exported – defines whether the service is explicitly exported and thus can be accessed by other applications. The default value is false, however there are ways of implicitly exporting a service (see Section 3.4.6).

permission – a string defining the name of the permission which is needed to access the service. If another application is not granted this permission, no intent will be forwarded to the service by the system. If no permission is set the permission set for the <application> tag will be used instead. If no permission attribute is set for the application, no restrictions are imposed on accessing this service if it is exported.

process – the name of the process the application should run in. From a security point of view this is only of interest if set to a global process and thus shares the same process across multiple applications and potentially multiple trust boundaries (see Section 3.3.1).

3.4.2.1 System Serivces

Preinstalled system services are a special kind of service. System services usually run in highly privileged processes.

A user can list all system services by using the service command on an android device. For example the result on a HTC Legend phone will be:

```
# service list
Found 54 services:
0     phone: [com.android.internal.telephony.ITelephony]
1     iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2     simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3     isms: [com.android.internal.telephony.ISms]
4     htc checkin: [android.os.ICheckinService]
5     checkin: [android.os.ICheckinService]
6     appwidget: [com.android.internal.appwidget.IAppWidgetService]
7     backup: [android.backup.IBackupManager]
8     audio: [android.media.IAudioService]
9     wallpaper: [android.app.IWallpaperManager]
10    search: [android.app.ISearchManager]
11    location: [android.location.ILocationManager]
12    devicestoragemonitor: []
13    mount: [android.os.IMountService]
14    notification: [android.app.INotificationManager]
15    accessibility: [android.view.accessibility.IAccessibilityManager]
16    connectivity: [android.net.IConnectivityManager]
17    wifi: [android.net.wifi.IWifiManager]
18    netstat: [android.os.INetStatService]
19    input_method: [com.android.internal.view.IInputMethodManager]
20    clipboard: [android.text.IClipboard]
21    statusbar: [android.app.IStatusBar]
22    bluetooth a2dp: [android.bluetooth.IBluetoothA2dp]
23    bluetooth pbap: [android.bluetooth.IBluetoothPbap]
24    bluetooth spp: [android.bluetooth.IBluetoothSpp]
25    bluetooth_avrcp: [android.bluetooth.IBluetoothAvrcp]
26    bluetooth: [android.bluetooth.IBluetooth]
27    mcp monitor: [android.bluetooth.IMCPMonitor]
28    window: [android.view.IWindowManager]
29    sensor: [android.hardware.ISensorService]
30    alarm: [android.app.IAlarmManager]
31    hardware: [android.os.IHardwareService]
32    battery: []
33    content: [android.content.IContentService]
34    account: [android.accounts.IAccountManager]
35    permission: [android.os.IPermissionController]
36    activity.providers: []
37    activity.senders: []
38    activity.services: []
39    activity.broadcasts: []
40    cpuinfo: []
41    meminfo: []
42    activity: [android.app.IActivityManager]
43    package: [android.content.pm.IPackageManager]
44    telephony.registry: [com.android.internal.telephony.ITelephonyRegistry]
45    usagestats: [com.android.internal.app.IUsageStats]
46    batteryinfo: [com.android.internal.app.IBatteryStats]
47    power: [android.os.IPowerManager]
48    entropy: []
```

```
49    SurfaceFlinger: [android.ui.ISurfaceComposer]
50    media.audio_policy: [android.media.IAudioPolicyService]
51    media.camera: [android.hardware.ICameraService]
52    media.player: [android.media.IMediaPlayerService]
53    media.audio_flinger: [android.media.IAudioFlinger]
```

All of these preinstalled services add to the overall attack surface of a device. Each of the system services might export an arbitrary number of methods.

As with any service in Android there is no standard way of enumerating exported methods during runtime.

3.4.3 Content-Providers

Content providers provide data and information to applications. Usually a content provider is used to offer data to other applications; however it might also be restricted to only provide information to components of its own application.

A component or application requesting data from the content-provider does not communicate with the content provider directly, but uses a so called content resolver instead. The data is returned in a table format, similar to a relational database. Cursors are used to access the data. The content resolver will also provide methods for inserting, updating and deleting of data.

The implementation of the content-provider can choose to store the information in any meaningful way or even generate it on-the-fly. The data could be stored on the file system or in volatile memory. However the most common way of implementing content providers is using SQLite Database, which brings its own security implications with it (see Section 4.1).

A content provider is described in the AndroidManifest.xml file using the <provider> tag. The following attributes are of interest for the security of the system:

exported – If set to false the content provider will not be exported. An unexported content provider will only be accessible to the components of the application defining the content provider. The default value is true, hence every content provider is accessible to other applications by default.

readPermission – the name of the permission that is required to access data stored in the content provider.

writePermission – the name of the permission that is required to change or insert data in the content provider.

permission – the name for the permission that is needed to generally access the provider. This is a shortcut, which can be used to set “readPermission” and “writePermission” at once. However, “readPermission” and “writePermission” will never take precedence over the permission attribute.

grantUriPermission – if grantUriPermission is set to true, access to the content provider can be temporarily granted to an application, overcoming any required permissions. The access will be limited to a specific URI. The default value is false. If this is not enabled temporary access can still be enabled using <grant-uri-permission> sub tags.

multiprocess – Usually any request to a content provider will be satisfied by creating a process in the context of the application defining the content provider, or reusing an existing process in this context. If the multiprocess attribute is set to true, any application requesting data will instantiate the content provider in its own process, this will remove the overhead of interprocess communication. However it is likely to impact the security of the provider, as the requesting application would have full access to the process of this content provider.

If <grant-uri-permission> sub tags are defined for a <provider> tags, these will define for which URIs temporary access can be granted to other applications. This is similar to the grantUriPermission attribute.

3.4.4 Broadcast Receivers

Broadcast receivers process and react to broadcast messages. Broadcast messages often originate from the system but may also be sent from other applications. Applications are not allowed to send system broadcasts. Some of the broadcast events can only be received by applications that have the permissions to do so.

A broadcast receiver is defined in the AndroidManifest.xml file using the <receiver> tag. The following XML attributes are of interest from a security perspective:

exported – defines whether the broadcast receiver is explicitly exported and thus can be accessed by other applications. The default value is false, however there are ways of implicitly exporting a broadcast receiver (see Section 3.4.6).

permission – a string defining the name of the permission which is needed to send broadcasts to the application. If another application is not granted this permission, no intent will be forwarded to the broadcast receiver by the system. If no permission is set the permission set for the application tag will be used. If no permission attribute is set for the application, no restrictions are imposed on accessing this broadcast receiver if it is exported.

process – the name of the process the application should run in. From a security point of view this is only interesting if set to a global process and therefore set to share the same process across multiple applications and potentially multiple trust boundaries (see Section 3.3.1).

3.4.5 Activities

Activities are the visual components of an application. Generally each dialog of the application is implemented as an activity. An activity can be started by the

application itself or if exported by other applications. Additional information, which could be seen as arguments, can be passed to the activity in the Intent object when it is started.

An activity is defined using an <activity> tag in the application's manifest file. The following properties will be of interest in a security audit:

exported – defines whether the activity is explicitly exported and thus can be accessed by other applications. The default value is false, however there are ways of implicitly exporting an activity (see Section 3.4.6).

permission – a string defining the name of the permission which is needed to start the activity. If another application is not granted this permission, no intent will be forwarded to the broadcast receiver by the system. If no permission is set the permission set for the application tag will be used. If no permission attribute is set for the application, no restrictions are imposed on starting this activity if it is exported.

process – the name of the process the application should run in. From a security point of view this is only interesting if set to a global process and therefore set to share the same process across multiple applications and potentially multiple trust boundaries (see Section 3.3.1).

3.4.5.1 Activity Alias

An alias for an activity can be defined in the manifest file using the <activity-alias> tag. An alias defines a target activity in the same application to which any intents targeted at the alias are forward. Notably the alias is able to overwrite attributes with a security impact on the target activity, such as the "exported" or the "permission" attribute. Furthermore none of the attributes of the target activity are inherited by the alias.

3.4.6 Default Export behaviour

Only exported IPC endpoints can be accessed by other applications and processes in Android. Content-providers are the only IPC endpoints that are exported by default.

An Android developer can choose to explicitly export a service, broadcast receiver or activity. This is done by setting the export attribute for the defining tag in the AndroidManifest.xml (see Section 3.2.1) to true.

Interestingly services, broadcast receivers and activities can also be exported implicitly by defining intent filters in the XML configuration. Any intent filter will lead to immediate exporting of the endpoint and thus will allow any other application to talk to this endpoint; unless it is protected by permission requirements (see section 3.3). In the case of an implicit export a developer can still choose not to export this endpoint by setting the export attribute to false.

This default export behaviour is not very intuitive and will in many cases lead to falsely exported endpoints. A developer may not expect that setting an intent filter, a

mechanism used to restrict the type of messages arriving at an endpoint will open the endpoint up to all the other applications and thus make it a target for exploitation attempts.

3.5 Reversing and Decompilation

When auditing a phone and third party applications, access to the source code is often not available. Even though Android's core is open source, not all of the standard Google applications commonly found on phones are available as source. In order to still being able to properly review applications several tools for reversing and decompilation are available.

3.5.1 APK files

Every package on the Android system is stored and distributed in APK files. APK files are zip compressed archives, similar to Java JAR files. Any tool capable of unzipping ZIP archives can be used to access the content. In third party applications a classes.dex file can be found, that contains the Java byte code. For applications that are distributed with the phone a classes.dex files is often not included in the APK, instead an odex file is stored in various locations of the file system. Odex files are Java classes optimised for the specific device.

3.5.2 DEX Files

For reversing dex files a convenient tool called dex2jar is available (see [4]). It does what the name says and creates standard JAR archives from dex files. These JAR archive can now be opened using a Java decompiler such as jdgui [5].

3.5.3 ODEX Files

In the case of optimised byte code, no such convenient way of obtaining the source code is available. However another tool called bakismali can be used to turn the odex files in a fairly readable byte code format [6].

4 Vulnerabilities

This section will describe some of the vulnerabilities that can be commonly found in Android applications.

4.1 SQL Injection

As discussed earlier in this paper SQLite databases are commonly used as the backend storage for content providers. As with many SQL database implementations, these are prone to SQL injection. A malicious Android application with restricted access to such a content provider could now use improperly sanitised input to a SQL statement to obtain other protected information from the same database.

An application can request data from a content provider using a content resolver. The "SettingProvider" content provider will be used as an example. Access to the settings content provider is not restricted and thus granted to any application by

default. In this example the application requests all system settings from the settings content provider as follows:

```
Cursor cur = this.getContentResolver().query(Settings.System.CONTENT_URI, null,
null, null, null);
Log("count: " + cur.getCount());
```

As a result the application will log the number of returned setting name and value pairs, in this example 47.

The query method of the ContentResolver object is implemented as follows:

```
final Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs, String sortOrder)
```

By passing null as parameter for "projection", "selection", "selectionArgs" and "sortOrder" these fields are omitted. And the resulting SQL statement will be:

```
select * from system;
```

This will change if a string for the selection parameter is added, for example if "_id=1" is passed. The resulting SQL statement will then be:

```
select * from system where (_id=1);
```

At this point the weakness becomes apparent and an attacker can use this to access data stored in the same SQLite database or in other databases which share the same SQLite database process. As SQLite 3 allows the use of so called scalar subqueries in the where part of a SELECT statement (see [2] for details) an attacker can use this to extract arbitrary data from the database using standard methods for SQL injection used regularly in web application exploits. One working example for this is:

```
Cursor cur = this.getContentResolver().query(Settings.System.CONTENT_URI, null,
"(select count(*) from secure where name='adb_enabled' and value='0')=0", null,
null);
log("count: " + cur.getCount());
```

As a result the count result will again be 47 when adb is disabled and 0 otherwise. This specific example is not a vulnerability, as the secure settings data is world readable anyway. However, the same database contains information about application bookmarks and previously connected Bluetooth devices, information which probably should not be leaked to every installed application.

Owing to how the constructed SELECT statements are passed to the SQLite database an attacker can't construct statement lists (e.g. using ";") and is limited to read operations in the SELECT statement. Also, all interesting functions, such as "load_extension", were found to be disabled in this SQLite implementation.

In the far less likely case where an attacker has write, but no read access to the content provider, the update method can be used to obtain read access to all databases and tables in a similar way. The update method is defined as follows:

```
final int update(Uri uri, ContentValues values, String where, String[]
selectionArgs)
```

Again the WHERE clause can be constructed similarly to the query method case. The returned integer will be the number of update rows and can be used to leak the information for every tested case to the attacker.

As the way in which data is accessed from SQLite based content providers is a fundamental design issue in Android, the only viable fix for this issue would be not to share common databases for information to which the access should be restricted using different permissions. With the current design, a content SQLite based content provider with write but no read access, cannot be implemented securely, other than denying WHERE strings completely.

It should be noted that SQL injection can potentially be used to exploit temporary access granted using the grantUriPermission attribute or the <grant-uri-permission> tags in order to obtain full access to the data source, if it is provided using SQLite storage.

4.1.1 Injection through Projections

Whilst researching different methods of SQL injection in content providers, a more efficient way using the projections arguments was identified. The projection argument is an array of strings containing the names of the columns which values should be returned. These names will be filled in immediately after the SELECT statement and aren't filtered prior to that. The following example shows how to retrieve the contents of the "bluetooth_devices" table in a very efficient manner:

```
String[] ar = {" * from secure; "};
Cursor cur = this.getContentResolver().query(Settings.System.CONTENT_URI, ar, null,
null);
text = "count: " + cur.getCount();
```

The attacker will still be restricted to SELECT statements and thus write operations are not feasible.

4.2 Native APIs

As most packages and applications on Android are implemented in Java, these are much less prone to memory corruption vulnerabilities then natively implemented applications. An attacker might want to focus on any natively implemented components as these are more likely to introduce critical vulnerabilities. When looking at the reverse engineered code of the application it can prove particularly useful to look out for the use of the System.loadLibrary() method, which is used to load native libraries. Also the "native" keyword in Java code is something an auditor

should pay attention to. This is particularly interesting if the keyword is used for public methods in an exported service.

4.3 Java Object Deserialisation

As discussed previously "intents" which are sent to IPC endpoints can contain primitive data types, arrays and serialisable Java objects. As with any deserialisation routine, these may potentially expose risks during the parsing of the data structures.

It was found that any application that expects a serialisable Java object can be forced into deserialising any Java object, as the type checking is only done during the cast after the object has been created.

Java objects may potentially execute unsafe operations during the deserialisation and initialisation phase. The attack surface for this class of bugs is very large and can be extended even further for an application by classes implemented by the application itself.

In Oracle's Java implementation deserialisation of arbitrary objects would also allow for deserialisation of arbitrary byte code if it happens in an unprotected context and thus would lead to arbitrary code execution. However this feature is not implemented in Android's Java virtual machine.

4.4 Exploit WebKit Vulnerabilities through WebViews

Applications in Android are able to use WebView objects in order to display web content, such as HTML or SVG. The WebView content is implemented using the WebKit rendering engine and is actually used by the browser application itself to render web pages. If an attacker is able to either control the source for the WebView object or inject HTML into the WebView, it will be possible to exploit any WebKit vulnerability in order to gain arbitrary code execution in the context of the application using the WebView. As it has been shown before WebKit vulnerabilities are reasonably common.

5 Case Study: HTC Legend

As a case study a partial audit was performed of an HTC Legend phone running Android version 2.1. The findings outlined in this section will also apply to some extent to any HTC phones with similar software versions.

Even though Google's standard implementation as it can be found in the emulator, for example, is fairly secure. Tests have showed that this is not necessarily the case for all phones on the market. In fact some of them are not well secured and vulnerabilities have been introduced by phone manufacturers.

5.1 Extracting Information from Installed Packages

In addition to the packages installed by default on the device, several third party packages have been installed from the Android market.

A list of all installed packages can be obtained using the following command:

```
# pm list packages
pm list packages
package:com.google.android.location
package:com.htc.FriendStreamWidget
package:com.android.launcher
package:com.htc.dcs.impl
package:com.google.android.providers.enhancedgooglesearch
package:com.htc.TwitterWidget
package:com.android.googlesearch
package:com.android.phone
package:com.htc.fm
package:com.android.calculator2
package:org.connectbot
package:com.android.htmlviewer
package:com.android.bluetooth
package:com.android.providers.calendar
package:com.google.android.providers.settings
package:com.htc.android.mail
package:com.htc.provider.CustomizationSettings
package:com.htc.htcmailwidgets
package:com.android.browser
[…]
```

In total 138 packages were found to be installed on the phone. The "pm path" command can be used to obtain the path of any of those packages. Using the following shell script all package files can be downloaded from the phone:

```
for i in `../adb shell pm list packages | sed -e "s/^.*://g" | sed -e "s/[^a-zA-Z0-
9]*$//g" | sed -e "s/ //g"`; do
      echo "package: " $i
      mkdir pkgs/$i
      apppath=`../adb shell pm path $i | sed -e "s/^.*://g" | sed -e "s/[^a-zA-Z0-
9]*$//g"`
      echo "path: " $apppath
      ../adb pull $apppath ./pkgs/$i/.
done
```

The script will enumerate all packages installed on the device and download all of the "apks" files for these applications.

In the next step all AndroidManifest.xml files from the packages will be extracted into a readable format using the "aapt" tool. Running the following script inside the "pkgs" directory will automatically store human-readable AMtree.xml files in each of the directories.

```
for i in `ls`; do
      cd $i
      aapt d xmltree *.apk AndroidManifest.xml > AMtree.xml
      cd ..
done
```

These AMtree.xml files will now contain all the data important for a review of all the IPC endpoints. This data includes all services, content-providers, broadcast receivers and activities. These files will also hold information regarding whether they are exported to other applications or protected by any permission settings.

5.2 Shared User-ids

The information that has been gathered can now be used to determine which applications share User Ids and thus share permissions. These applications are especially interesting to an attacker or auditor.

In total 8 shared user ids were found amongst the applications that were preinstalled on the phone. The following list shows an excerpt of the list that was automatically generated from a HTC Legend phone:

```
UID: 9999
Applications: com.htc.FriendStreamWidget, com.htc.TwitterWidget,
com.htc.htcmailwidgets, com.htc.NewsReaderWidget, com.htc.StockWidget,
com.htc.widget.clockwidget, com.htc.htccalendarwidgets, com.htc.footprints.widgets,
com.htc.htccontactwidgets, com.htc.htcmsgwidgets, com.htc.htcsyncwidget,
com.htc.launcher, com.htc.WeatherWidget, com.htc.htcsettingwidgets,
com.htc.photo.widgets, com.htc.htcbookmarkwidget, com.htc.MusicWidget,
com.htc.htcsearchwidgets
Permissions: android.permission.INTERNET, com.htc.htctwitter.permission.useprovider,
android.permission.ACCESS_FINE_LOCATION, android.permission.ACCESS_NETWORK_STATE,
android.permission.ACCESS_WIFI_STATE, android.permission.GET_ACCOUNTS,
android.permission.READ_SYNC_SETTINGS, android.permission.READ_CALENDAR,
android.permission.WRITE_CALENDAR,
com.google.android.googleapps.permission.GOOGLE_AUTH.mail,
android.permission.READ_CONTACTS, android.permission.CALL_PHONE,
android.permission.CALL_PRIVILEGED, android.permission.READ_SMS,
com.htc.socialnetwork.permission.useprovider,
android.permission.RECEIVE_BOOT_COMPLETED, android.permission.WRITE_CONTACTS,
android.permission.RECEIVE_SMS, android.permission.RECEIVE_MMS,
android.permission.SEND_SMS, android.permission.VIBRATE,
android.permission.WRITE_SMS, android.permission.CHANGE_NETWORK_STATE,
android.permission.READ_PHONE_STATE, android.permission.WAKE_LOCK,
android.permission.EXPAND_STATUS_BAR, android.permission.GET_TASKS,
android.permission.SET_WALLPAPER, android.permission.SET_WALLPAPER_HINTS,
android.permission.WRITE_SETTINGS, com.htc.launcher.permission.READ_SETTINGS,
com.htc.launcher.permission.WRITE_SETTINGS, android.permission.SET_TIME_ZONE,
android.permission.READ_SYNC_STATS, android.permission.WRITE_EXTERNAL_STORAGE,
android.permission.BROADCAST_STICKY, android.permission.WRITE_SECURE_SETTINGS,
android.permission.CHANGE_WIFI_STATE, android.permission.CLEAR_APP_USER_DATA,
android.permission.MODIFY_PHONE_STATE, android.permission.ACCESS_COARSE_LOCATION,
android.permission.WRITE_APN_SETTINGS, android.permission.ACCESS_CHECKIN_PROPERTIES,
android.permission.BLUETOOTH, android.permission.BLUETOOTH_ADMIN,
android.permission.ACCESS_WIMAX_STATE, android.permission.CHANGE_WIMAX_STATE,
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS,
android.permission.ACCESS_LOCATION, android.permission.ACCESS_ASSISTED_GPS,
android.permission.ACCESS_NETWORK_LOCATION, android.permission.ACCESS_GPS,
com.android.browser.permission.READ_HISTORY_BOOKMARKS,
com.android.browser.permission.WRITE_HISTORY_BOOKMARKS
```

In the generated output the user id (9999) can be seen, the set of applications sharing this user id and the sum of all permission shared across these applications. From this the problems introduced by shared User Ids can easily be seen. For example the WeatherWidget application, usually a candidate for very limited permissions has the permissions to make phone calls or to send and receive text messages.

5.3 Vulnerabilities

In this section some example weaknesses found in a default setup of the HTC Legend Android phone are discussed. Notably these weaknesses exist in HTC's configuration

of Android phones. Other Android mobile phones will not be vulnerable to the same issues.

5.3.1 Audio Recording Permission bypass

Recording audio from the microphone is a feature that should be protected, as it can be used to eavesdrop on conversations and phone calls of users. On a default Android phone, the *android.permission*.RECORD_AUDIO permission is required by an application in order to record audio from the microphone on a phone.

However HTC added its own recording service to the phone, which is not restricted in any way and can be used by any application to record audio. The "com.htc.soundrecorder" package defines the RecordingService service as follows:

```
E: service (line=50)
        A: android:name(0x01010003)="RecordingService" (Raw: "RecordingService")
        A: android:exported(0x01010010)=(type 0x12)0xffffffff
```

It is not protected by any permission and exported explicitly as the "exported" attribute is set. In tests it was possible to use this service to record audio using a low privileged application.

5.3.2 INSTALL_PACKAGES permission in Browser

Probably the most useful permission to an attacker is the "android.permission.INSTALL_PACKAGES" permissions. This permission can be used to install arbitrary applications with arbitrary permissions without any user interaction.

It was discovered that this permission is granted to the browser application on HTC phones. Notably this is not the case in default Android installation. Further investigation showed that this is done to install the FlashLitePlayer application from inside the browser. However this would allow any attacker with a WebKit exploit to install arbitrary applications and thus to gain arbitrary permissions on an HTC Android phone.

This issue can either be exploited by an attacker using vulnerabilities in the browser directly or by malicious applications that use browser vulnerabilities to escalate privileges on the device.

6 Conclusion

Even though previous research has focussed on Linux kernel vulnerabilities for privilege escalation, it has been shown that a large attack surface exists in the sandbox itself. Vulnerabilities in packages and applications can be used by malicious applications or by malware installed through successful exploitation attempts in order to gain further privileges on Android phones. In many cases full root access is

not required in order to gain access to the information that might be of interest to an attacker.

Furthermore it was found that the security standard of actual phones often derivates significantly from that of the standard build of Google's Android release.

7 References

- [1] Android Developer Guide, http://developer.android.com/guide/
- [2] SQLite Query Language, http://www.sqlite.org/lang_expr.html
- [3] dex2jar, http://code.google.com/p/dex2jar/
- [4] Java Decompiler GUI, http://java.decompiler.free.fr/?q=jdgui
- [5] smali/bakismali, http://code.google.com/p/smali/
- [6] Android sock_sendpage() exploit, http://packetstormsecurity.org/filedesc/android-root-20090816.tar-gz.html
- [7] Local Android exploit, Sebastian Krahmer, http://c-skills.blogspot.com/2010/07/android-trickery.html
- [8] Exploratory Android Surgery, Jesse Burns, iSEC Partners, https://www.isecpartners.com/files/iSEC_Android_Exploratory_Blackhat_2009.pdf

MWR InfoSecurity
St. Clement House
1-3 Alencon Link
Basingstoke, RG21 7SB
Tel: +44 (0)1256 300920
Fax: +44 (0)1256 844083
mwrinfosecurity.com