Saarland University
Information Security & Cryptography Group
Prof. Dr. Michael Backes

Android Security Lab WS 2014/15

Lab 3: Android Middleware Extension

M.Sc. Sven Bugiel

Version 1.1 (October 7, 2014)

Version History

Version 1.1	Fixed `init` function of `SmartseclabPermissionService` Fixed `checkPermission` function in service skeleton listing Made the listings more “copy-and-past-able”

Login on Lab Machines

Username: smartseclab
Password: android

Introduction

In this Lab you will implement a new, simple system service which enforces permissions in dependence of the device's location. For instance, this could be used to enforce that applications on a device that is physically located at your workplace cannot take any pictures or record any audio, thus building a line of defense against espionage. Because the emulator does not support any camera or audio functionality, we demonstrate this location-aware enforcement based on the `READ_CONTACTS` permission.

Building a custom ROM for the Emulator

Your lab machines have been preconfigured with the necessary tools and libraries to build Android and the source code for Android v4.3_r3.1 was already downloaded.[1] The code is located in the *android-4.3_r3.1* directory in your home folder. To execute build commands, you first have to set up some environment variables in your shell (e.g., the location of prebuilt toolchains for cross-compilation to ARM bytecode). Thus, in every shell you want to execute `make` commands for building Android, execute first:

```
~/android-4.3_r3.1$ source build/envsetup.sh
```

This sets up general environment variables for Android. Next, you must select the build target, i.e., whether you are building for a specific device or the emulator. Execute the following command to set your build target to the default ARM-based emulator:

```
~/android-4.3_r3.1$ lunch aosp_arm-eng
```

Now, you can build the system by executing:

```
~/android-4.3_r3.1$ make -j3
```

Since the lab machines already ran through the build process (approx. 2 hours in the current hardware configuration), every new build will take significantly less time!

To test your custom build of Android on the emulator, you can run:

```
~/android-4.3_r3.1$ emulator -wipe-data
```

Note: If your system does not boot and you see in the log `DexOpt` errors and mismatching signature errors, that means there exist inconsistencies between files from the last build and your current build. To fix this error, execute:

```
~/android-4.3_r3.1$ make installclean
~/android-4.3_r3.1$ make -j3
```

Browsing the Android source code

Eclipse on the lab machines has been preconfigured as IDE for development of the Android OS middleware and system applications.[2] Thus, you already have a project called *android-4.3_r3.1*. The Android source code is managed in more than 300 distinct source code repositories, where each project implements a subsystem of Android. In the *android-4.3_r3.1* project in Eclipse you can see all of the Java-based projects and most of the important native code based projects, organized in source folders named after their repository and path in the Android source tree (cf. Figure 1). Because this comprises several hundreds of thousands lines of code, it is helpful to know some basic shortcuts in Eclipse for efficiently browsing the code:

[1] Check this link on how to initialize your own Android build environment: `http://source.android.com/source/building.html`

[2] `http://source.android.com/source/developing.html`

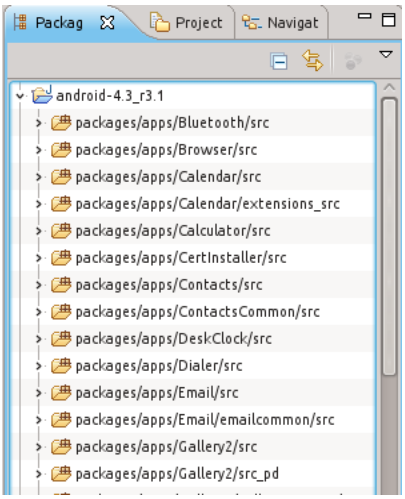


Figure 1: Ovreview of Android source code in Eclipse.

- Ctrl+Shift+R: Open a resource such a Java file
- Ctrl+O: Search and goto a specific function within a Java file
- Ctrl+Alt+H: Open the call hierarchy of a selected variable or function
- F3: Goto the declaration of a variable or function

Question 1:
Goto the function `checkComponentPermission` of the `android.app.ActivityManagerService` class. Which hardcoded permissions do the root and system server UID have?
Which UID does the system server have?

1 Location-aware Permission Enforcement

In this exercise you will extend the Android OS with a new, simple system service for location-aware permission enforcement (see Introduction). This task consists of 1) defining the interface of the new service, 2) implementing the service, and 3) integrating it into the system server so that it will started during boot and into the ActivityManagerService so that permission requests are re-directed to the new service.

1.1 Defining the new permission service

1. In the source folder `frameworks/base/core/java/` create a new package `android.os.smartseclab`
2. In the package `android.os.smartseclab`, create a file `ISmartseclabPermissionService.aidl` with the following content. This defines the interfaces of our service.

```
package android.os.smartseclab;

interface ISmartseclabPermissionService
{
  int checkSmartseclabPermission(int uid, String permission);
  void init();
  void close();
}
```

3. Open the makefile `Android.mk` in the `frameworks/base` folder

4. In this makefile, add the last line to the 4th `LOCAL_SRC_FILES` as shown in the following listing. This tells the build system that it should create the *Stub* of our service (don't forget the backslash at the end of the second last line):

```
LOCAL_SRC_FILES += \
  core/java/android/accessibilityservice/IAccessibilityServiceConnection.aidl \
  core/java/android/accessibilityservice/IEventListener.aidl \
  core/java/android/accounts/IAccountManager.aidl \
  core/java/android/accounts/IAccountManagerResponse.aidl \

  ...

  wifi/java/android/net/wifi/p2p/IWifiP2pManager.aidl \
  core/java/android/os/smartseclab/ISmartseclabPermissionService.aidl
```

5. Before we implement the new service, we have to actually generate the *Stub* of our new service. Because we extended the API of our Android with a new service, we also have to update the API. Both can be accomplished, by executing:

```
~/android-4.3_r3.1$ make update-api
```

You should see output similar to:

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-55-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JLS36G
OUT_DIR=out
============================================
Aidl: framework <= frameworks/base/core/java/android/os/smartseclab/
    ISmartseclabPermissionService.aidl

...

DroidDoc took 11 sec. to write docs to out/target/common/docs/api-stubs
Copying current.txt
```

The first line shows, that the build system generated the *Stub* using the AIDL compiler. *Current.txt* contains a list of the functions in the (new) Android API in our build.

6. To inform Eclipse about the new *Stub* classes, refresh the project by selecting the android-4.3_r3.1 project in Eclipse and pressing F5. Refreshing can take between 1-2 minutes.

1.2 Implementing the new permission service

Now that the interface of our service is defined and its *Stub* generated, we can implement the service by extending the *Stub*. This is very similar to implementing the service of App1 in the first Lab session.

1. In the source folder `frameworks/base/services/java` create a new package `com.android.smartseclab`. Packages in this folder will be part of the system server process.
2. In this package create a new class `SmartseclabPermissionService.java`, which will implement the service.
3. A minimal implementation of the new service looks as follows:

```
package com.android.smartseclab;

import android.os.smartseclab.ISmartseclabPermissionService;

public class SmartseclabPermissionService extends ISmartseclabPermissionService.Stub
    {

  // Single entry point to perform a dynamic permission check
  public int checkSmartseclabPermission(int uid, String permission)
  {
    return PackageManager.PERMISSION\_GRANTED;;
  }

  //Initialize this service
  public void init() {}

  // Trigger shutdown of this service
  public void close() {}
}
```

Copy this code into your `SmartseclabPermissionService.java` and we will extend this minimal implementation in the following:

4. First, we need a number of new attributes:

```
private static final String TAG = "SmartseclabPermissionService";
// Singleton
private static SmartseclabPermissionService mInstance;
// Context to use to interact with the system
private Context mContext;
// Indicates whether initialized and ready to handle permission checks
private boolean init = false;


// PackageManager reference to query information about packages
private PackageManager mPackageManager;
// Intent object to be sent back to us upon location changes
private PendingIntent pi;
// Custom BroadcastReceiver for the Intent
private FenceReceiver mFenceRecv;
// Currently active policy to allow / deny the read contacts permission
private int allowReadContacts = PackageManager.PERMISSION_GRANTED;
// Latitude the work
private static final double LATITUDE = 49.257340;
// Longitude of the work location
private static final double LONGITUDE = 7.045550;
private Location workLocation;
// Radius around work location
private static final float RADIUS = 100.0f;
// Action string for our Broadcast Intent upon location updates
private static final String FENCE_INTENT_ACTION = "smartseclabpermissionservice.
    action.FENCE";

// Fixed ID for th e Broadcast Intent => Easier to cancel on shutdown
private static final int FIXED_PI_ID = 42;
```

5. Next, we implement a singleton pattern, because our service should exist only once in the system:

```
// Private constructor
private SmartseclabPermissionService(Context ctx)
{
  mContext = ctx;
  // Set fixed location to E1.1 2.0 6
  workLocation = new Location(LocationManager.GPS_PROVIDER);
  workLocation.setLatitude(LATITUDE);
  workLocation.setLongitude(LONGITUDE);
}

// Singletonpattern
public static SmartseclabPermissionService getInstance(Context ctx)
{
  if(mInstance == null)
  {
    mInstance = new SmartseclabPermissionService(ctx);
  }
  return mInstance;
}
```

6. Now, we implement our permission check, which will return the current value of `allowReadContacts` when the permission check is triggered for the UID of a 3rd party application and the `READ_CONTACTS` permission:

```
public int checkSmartseclabPermission(int uid, String permission)
{
  if(!init)
  {
    Log.i(TAG, "checkSmartseclabPermission called, but not yet ready");
    return PackageManager.PERMISSION_GRANTED;
  }

  if(permission == null)
  {
    Log.w(TAG, "checkSmartseclabPermission called with illegal arguments!");
    return PackageManager.PERMISSION_GRANTED;
  }

  // We only enforce for receive sms permission
  if(!permission.equals("android.permission.READ_CONTACTS"))
  {
    return PackageManager.PERMISSION_GRANTED;
  }

  Log.i(TAG, "Checking for uid="+uid);
  // Check if we handle a system UID or not
  String[] packages = mPackageManager.getPackagesForUid(uid);
  try {
    // Index 0 is sufficient, because in case of shared UIDs all of them are system
        or non
    ApplicationInfo ai = mPackageManager.getApplicationInfo(packages[0], 0);
    if((ai.flags & ApplicationInfo.FLAG_SYSTEM)==0)
    {
      Log.i(TAG, "uid="+uid+" is not a system package. Returning "+allowReadContacts
          );
      return allowReadContacts;
    }
  } catch (NameNotFoundException e) {
    Log.e(TAG, "Error retrieving application info for uid="+uid, e);
    return PackageManager.PERMISSION_GRANTED;
```

```
  }

  Log.i(TAG, "uid="+uid+" is a system package. Returning "+PackageManager.
      PERMISSION_GRANTED);
  return PackageManager.PERMISSION_GRANTED;
}
```

7. Next, we implement the initialization function, which will bring this service into a state in which it can enforce permissions (i.e., we have a valid reference to the PackageManager) and which will start monitoring the location of the device. The latter one is accomplished by registering an Intent object with the LocationManager and the LocationManager will send this Intent back to us when the location has changed:

```
// Initialize our service and make it ready to handle permission checks.
public void init()
{
  Log.i(TAG, "Initializing.");

  // Reference t o PackageManagerService
  mPackageManager = mContext.getPackageManager();

  mFenceRecv = new FenceReceiver();

  // Register our receiver to get the location update
  mContext.registerReceiver(mFenceRecv, new IntentFilter(FENCE_INTENT_ACTION));

  LocationManager lm = (LocationManager) mContext.getSystemService(Context.
      LOCATION_SERVICE);
  // Create Intent to be send when location changes
  Intent i = new Intent(FENCE_INTENT_ACTION);
  i.setFlags(Intent.FLAG_FROM_BACKGROUND);
  pi = PendingIntent.getBroadcast(mContext, FIXED_PI_ID, i, PendingIntent.
      FLAG_CANCEL_CURRENT);
  if(lm.getAllProviders().contains(LocationManager.GPS_PROVIDER))
    lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 30000, 10, pi);
  else
    lm.requestLocationUpdates(LocationManager.PASSIVE_PROVIDER, 30000, 10, pi);

  init = true;

  Log.i(TAG, "Finished initializing.");
}
```

8. Now, we implement the close function, which is called during device shutdown and which simply remove the location update request registered during initialization:

```
// Trigger shutdown of this service //
public void close()
{
  // Cancel our location update
  if(pi != null)
  {
    LocationManager lm = (LocationManager) mContext.getSystemService(Context.
        LOCATION_SERVICE);
    lm.removeUpdates(pi);
  }
}
```

9. Finally, we implement our custom broadcast receiver, which is called upon location changes. Add the following *nested* class:

```
// Private class to receive the Intent sent by the LocationManager
// when device location changes
private class FenceReceiver extends BroadcastReceiver
{
  private final String TAG = "SmartseclabPermissionService.FenceReceiver";

  @Override
  public void onReceive(Context context, Intent intent) {
    // Set the current permission policy based on our distance to the work location
    if(intent.hasExtra(LocationManager.KEY_LOCATION_CHANGED))
    {
      Location newLoc = (Location) intent.getExtras().get(LocationManager.
          KEY_LOCATION_CHANGED);
      float dist = workLocation.distanceTo(newLoc);
      Log.i(TAG, "New location = "+newLoc.getLatitude()+":"+newLoc.getLongitude()+"
          ; Distance to work location = "+dist);
      if(dist < RADIUS)
      {
        Log.i(TAG, "In security sensitive area!");
        allowReadContacts = PackageManager.PERMISSION_DENIED;
      } else {
        Log.i(TAG, "Outside security sensitive area");
        allowReadContacts = PackageManager.PERMISSION_GRANTED;
      }
    }
  }
}
```

1.3 Integrating the new permission service

To integrate our new permission service into the Android system, we have to modify and extend two crucial system services: `ActivityManagerService`, which among other things implements the API for permission checks, and `SystemServer`, which among other things spawns all crucial system services during boot up.

ActivityManagerService.java

We extend the `ActivityManagerService` to 1) implement a call to initialize our permission service, 2) shutdown our permission during system halt, and 3) call our permission service during permission checks. To this end, we modify `ActivityManagerService.java` at different places.

1. First, add a new private *ISmartseclabPermissionService* attribute, which allows the `ActivityManagerService` to make RPC to our service (*Proxy* object to our service's *Stub*):

```
public final class ActivityManagerService  extends ActivityManagerNative
  implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

  ISmartseclabPermissionService mSmartseclabPermissionService = null;

  private static final String USER_DATA_DIR = "/data/user/";
  static final String TAG = "ActivityManager";

  ...
```

2. Next, add a function to initialize our permission service and get the *Proxy* object for our service. Add this function, e.g., directly before the `checkComponentPermission` function:

```
...

// Will be called from SystemServer as soon as LocationManagerService is running
public void smartseclabPermissionServiceInit() {
  Log.i(TAG, "initializing SmartseclabPermissionService");
  mSmartseclabPermissionService = ISmartseclabPermissionService.Stub.asInterface(
      ServiceManager.getService("SmartseclabPermissionService"));
  try {
    mSmartseclabPermissionService.init();
  } catch (RemoteException e) {
    Log.e(TAG, "Error initializing SmartseclabPermissionService", e);
    mSmartseclabPermissionService = null;
  }
}

/**
 * This can be called with or without the global lock held.
 */
int checkComponentPermission(String permission, int pid, int uid,
  int owningUid, boolean exported) {

...
```

3. To correctly shutdown the system, we modify the function `shutdown` to call `close()` on our permission service:

```
public boolean shutdown(int timeout) {

  ...

  // Shutdown the SmartseclabPermissionService
  try {
    mSmartseclabPermissionService.close();
  } catch (RemoteException e) {
    Log.e(TAG, "Error shutting down SmartseclabPermissionService", e);
  }

  mAppOpsService.shutdown();
  mUsageStatsService.shutdown();
  mBatteryStatsService.shutdown();

  return timedout;
}
```

4. Finally, we modify the `checkComponentPermission` function to call our permission service in order to check whether a permission is really granted to a caller:

```
int checkComponentPermission(String permission, int pid, int uid,
  int owningUid, boolean exported) {
  // We might be performing an operation on behalf of an indirect binder
  // invocation, e.g. via {@link #openContentUri}.  Check and adjust the
  // client identity accordingly before proceeding.
  Identity tlsIdentity = sCallerIdentity.get();
  if (tlsIdentity != null) {
    Slog.d(TAG, "checkComponentPermission() adjusting {pid,uid} to {"
          + tlsIdentity.pid + "," + tlsIdentity.uid + "}");
    uid = tlsIdentity.uid;
    pid = tlsIdentity.pid;
  }
```

```
  if (pid == MY_PID) {
    return PackageManager.PERMISSION_GRANTED;
  }

  int res = ActivityManager.checkComponentPermission(permission, uid, owningUid,
      exported);

  if(res == PackageManager.PERMISSION_GRANTED)
  {
    if(mSmartseclabPermissionService == null || permission == null)
    {
      return PackageManager.PERMISSION_GRANTED;
    }
    try {
      return mSmartseclabPermissionService.checkSmartseclabPermission(uid,
          permission);
    } catch (RemoteException e) {
      Log.e(TAG, "Error calling checkSmartseclabPermission", e);
      return PackageManager.PERMISSION_GRANTED;
    }
  } else {
    return PackageManager.PERMISSION_DENIED;
  }
}
```

SystemServer.java Next, we extend the `SystemServer` to 1) spawn our permission service during boot up and 2) to trigger initialization of our permission service as soon as all necessary higher-level system services are running (e.g., Location service).

5. To extend `SystemServer.java`, we extend its function `run()` at three different places:

```
  ...

  InputManagerService inputManager = null;
  TelephonyRegistry telephonyRegistry = null;

  // 1) SmartseclabPermissionService
  SmartseclabPermissionService sps = null;

  // Create a shared handler thread for UI within the system server.
  // This thread is used by at least the following components:

  ...

  // Critical services...
  boolean onlyCore = false;
  try {
    // Wait for installd to finished starting up so that it has a chance to
    // create critical directories such as /data/user with the appropriate
    // permissions.  We need this to complete before we initialize other services.
    Slog.i(TAG, "Waiting for installd to be ready.");
    installer = new Installer();
    installer.ping();

    Slog.i(TAG, "Power Manager");
    power = new PowerManagerService();
    ServiceManager.addService(Context.POWER_SERVICE, power);

    Slog.i(TAG, "Activity Manager");
    context = ActivityManagerService.main(factoryTest);
```

```
    // 2) Add SmartseclabPermissionService with context of AM
    Slog.i(TAG, "Smartseclab Permission Service");
    sps = SmartseclabPermissionService.getInstance(context);
    ServiceManager.addService("SmartseclabPermissionService", sps);

    Slog.i(TAG, "Display Manager");
    display = new DisplayManagerService(context, wmHandler, uiHandler);
    ServiceManager.addService(Context.DISPLAY_SERVICE, display, true);

    ...

        try {
          if (locationF != null) locationF.systemReady();
        } catch (Throwable e) {
          reportWtf("making Location Service ready", e);
        }
        // 3) LocationManager is up, thus we can initialize our permission service
        ActivityManagerService.self().smartseclabPermissionServiceInit();
        try {
          if (countryDetectorF != null) countryDetectorF.systemReady();
        } catch (Throwable e) {
          reportWtf("making Country Detector Service ready", e);
        }

    ...
```

1.4 Building and Testing

After your integrated the new permission service into the system, we now build our new custom ROM and test the location-aware permission enforcement on the Android emulator using a provided test app.

1. Before building, you first have to remove all existing system and data images in order to avoid possible inconsistencies between cached data and our new build. In a shell in which you have set the environment variables and executed the *lunch* command, execute now:

```
~/android-4.3_r3.1$ make installclean
```

You should see an output similar to:

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-49-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JLS36G
OUT_DIR=out
============================================
Deleted emulator userdata images.
Deleted images and staging directories.
```

2. Now build the new custom ROM by executing:

```
~/android-4.3_r3.1$ make -j3
```

The build can take between 2-5 minutes. When the build was successful, you should see at the end of a lengthy output something similar to:

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-55-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JLS36G
OUT_DIR=out
============================================

...

Combining NOTICE files into HTML
Combining NOTICE files into text
Installed file list: out/target/product/generic/installed-files.txt
Target system fs image: out/target/product/generic/obj/PACKAGING/
    systemimage_intermediates/system.img
Running:  mkyaffs2image -f out/target/product/generic/system out/target/product/
    generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/
    generic/root/file_contexts system
Install system fs image: out/target/product/generic/system.img
```

3. Now you can start your custom build Android on the emulator. Simply execute the following command and observe during boot up in a second shell the log for the SmartseclabPermissionService entries to verify that it correctly works:

```
~/android-4.3_r3.1$ emulator -wipe-data
```

4. After the system has booted, test the correct enforcement of the `READ_CONTACTS` permission using the provided App1 app. This app tries to query the Contacts provider every time its MainActivity resumes. To install and run it on the emulator of your custom build follow these steps:
   (a) Right-click on the App1 project and chose `Run As→Android Application`
   (b) In the pop-up that no AVD exists, answer with **No** whether you to create one
   (c) In the following screen chose your emulator as device to install and run App1
5. By default, `SmartseclabPermissionService.checkPermission` returns `PackageManager.PERMISSION_GRANTED`. Thus, when App1 automatically starts after installation, nothing should happen (i.e., not crash of the app or similar).
6. Close the app on the virtual device.
7. Now change the location of the virtual device using the telnet server of the emulator:

```
geo fix 7.045 49.257
```

Question 2:
Verify in the log output, that the SmartseclabPermissionService has indeed received a broadcast about the location update and that the new location is within the security perimeter!

8. Open again App1 on the device.

Question 3:
Verify that the enforcement was correctly applied. What was the new result for the execution of App1?

Question 4:
In this exercise, you used the physical location of the device as context for dynamic permissions. Can you think of other parameters and scenarios for dynamic permission enforcement?

Question 5:
The enforcement in this exercise is not really *gracefully*, i.e., apps quickly receive an unexpected exception. Can you think of a more *graceful* permission enforcement in which apps do not gain access to security/privacy sensitive data but simultaneously won't receive unexpected exceptions?

Question 6:
Could you use the same instrumentation of `checkComponentPermission` that you implemented in this exercise in order to enforce the `Internet` or `Write_External_Storage` permissions? Why/Why not? If not, how could it be done?

2 Provenance Information for Intents (Optional/Homework)

One of the most common inter-application communication are *Intent* messages. For instance, starting an Activity, sending broadcast messages, starting a service, or contacting *IntentServices* are based on Intents. Because the sender of an Intent can specify implicit targets for receiving his Intent (e.g., using the action string for which other must register themselves as receivers) and because sending/receiving Intents might be protected by a permission, Intent messages are generally delivered through the *ActivityManagerService*. That means, the call-chain for Intents is generally

$$\textit{Sender component} \longrightarrow \textit{ActivityManagerService} \longrightarrow \textit{Receiver component}$$

That also means, that `Binder.getCallingUid` loses its effectiveness when receiving Intents messages, because it returns only the direct IPC sender, i.e., the *ActivityManagerService*.

In this task, you should extend the `Intent.java` class such that it saves the UID of its creator process and preserves this information while being forwarded through the system to the receiver. The receiver component then should be able to use this information to gain information on which other app has triggered its execution.

1. First, add a new private attribute to the `Intent` class to store the creator UID and extend the default constructor to save its creator UID:

```
...

private Intent mSelector;
private ClipData mClipData;

private int originUID = -1;

// ---------------------------------------------------------------------

/**
 * Create an empty intent.
 */
public Intent() {
  originUID = android.os.Process.myUid();
}

...
```

2. However, Intent objects are very frequently copied. Thus, it is important to extend all (copy) constructors similarly to the default constructor. This includes the following constructors:
   - `public Intent(Intent o)`
   - `private Intent(Intent o, boolean all)`
   - `public Intent(String action)`
   - `public Intent(String action, Uri uri)`
   - `public Intent(Context packageContext, Class<?> cls)`
   - `public Intent(String action, Uri uri, Context packageContext, Class<?> cls)`
   - `public int fillIn(Intent other, int flags)`
3. Additionally, since Intent objects can be sent via Binder, they must be parcelable (i.e., serializable) to be send as a flat byte stream. Thus, the functions which parcel and unparcel Intent objects must be extended to write/read the new `originUid` attribute:
   - `public void writeToParcel(Parcel out, int flags)`
   - `public void readFromParcel(Parcel in)`

4. Finally, of course a getter function for `originUid` must be added:

```
public int getOriginUid() {
  return originUID;
}
```

5. System applications, that are part of the system build, are already aware of this extended Intent API and you can instrument some Intent-receiving components within the system packages (i.e., components in `packages/apps` in the source tree; these can be, e.g., found through the `AndroidManifest.xml` files in `packages/apps`). Alternatively, extend the `Intent.public void toShortString` function to print the `originUid` and later watch the logcat output for Intent logs and whether they contain `originUid`.

6. Now you are ready to build your new system and check that your modified system apps can identify their Intent sender. Since you extended the interface of the Intent object, you first have to update the API:

```
~/android-4.3_r3.1$ make update-api
~/android-4.3_r3.1$ make -j3
```

7. In order to allow also 3rd party apps to use `Intent.getOriginUid()`, you have to provide app developers with an SDK for your extended Android API. To compile the SDK execute the following command (**Note:** This builds can take a long time!):

```
~/android-4.3_r3.1$ lunch sdk-eng
~/android-4.3_r3.1$ make sdk -j3
```

After the build was successful, you should see at the end of the log output something like:

```
Package SDK: out/host/linux-x86/sdk/android-sdk_eng.smartseclab_linux-x86.zip
```

This means, that the above mentioned zip file contains the SDK for Linux x86 host machines. The content of the zip can also be found at `out/host/linux-x86/sdk/android-sdk_eng.smartseclab_linux-x86`.

App developers can now use this SDK to build apps for your customized Android, on which Intents store their creator UID. To actually use the SDK they adjust, for instance, in Eclipse the *SDK Location* variable in the *Android* section of the general Eclipse *Preferences*. Alternatively, they can build their apps from command line[3] and adjust their app project settings to use your SDK for building the app.

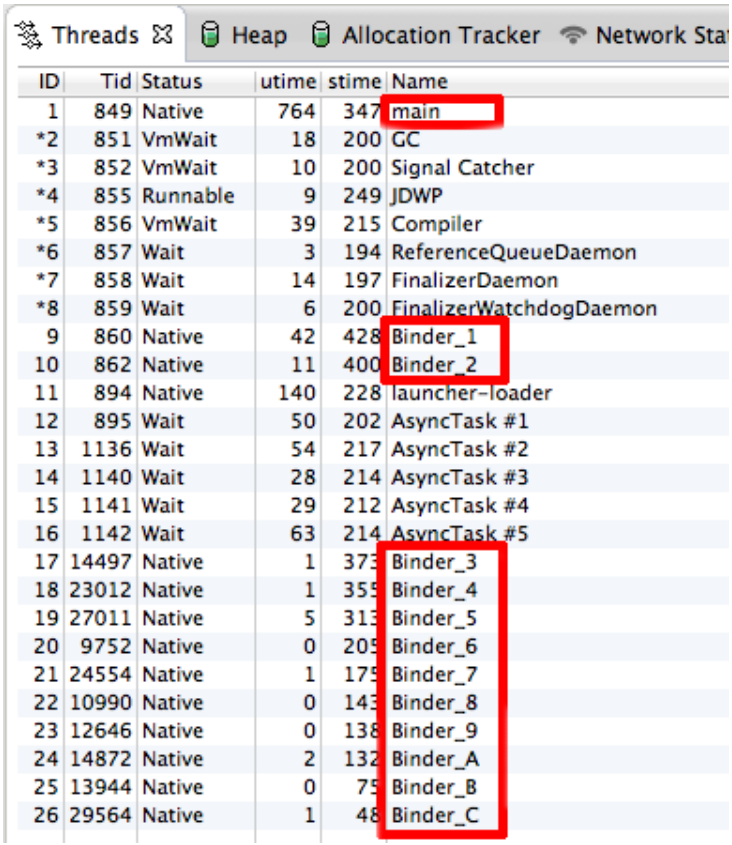
Question 7:
Intents are just messages between apps, which can also be forwarded and which apps can create as they like. Even with the security extension implemented in this task, what is a potential attack scenario in which an app can be fooled by a malicious app into trusting a fake `originUID` value? Which additional information must the receiver have and verify to trust the `originUID` value?

Question 8:
Android also knows so called *PendingIntent*[4], which is a description of an Intent and target action to perform with it and which can be given to another application that can extend and send it at later time. For instance, in Task 1, we gave a PendingIntent to the LocationManager, which send it back to our SmartseclabPermissionService every time a location update was available. However, by *"giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity)"*. Considering that the target of the PendingIntent can also be another application, what does this imply for the meaning of `originUid` for the receiver? How could you address this problem by further extending the ActivityManagerService, which is responsible for sending Intents?

[3] `http://developer.android.com/tools/building/building-cmdline.html`
`http://developer.android.com/tools/projects/projects-cmdline.html`

3 Providing Calling UID on MainThread (Optional/Homework)

In this task, you will implement a mechanism similar to the previous Task 2, however, addressing *intra*-application communication instead of *inter*-application communication. As you have learned in the homework of Lab 2, every Android application has several Binder threads, responsible for receiving incoming Binder IPC messages, and a main thread, by default responsible for executing certain application components (e.g., Activities or BroadcastReceivers). See also Figure 2. The information that Binder provides about IPC sender applications (i.e., PID and UID) are stored in the ThreadLocalStorage (TLS) of the receiving Binder thread. Thus, every application component executing in context of a Binder thread can successfully retrieve this information. However, components executing in context of the main thread do **not** by default have access to this information, thus preventing these components from implementing a more fine-grained access control (see Lab 2).



ID	Tid	Status	utime	stime	Name
1	849	Native	764	347	main
*2	851	VmWait	18	200	GC
*3	852	VmWait	10	200	Signal Catcher
*4	855	Runnable	9	249	JDWP
*5	856	VmWait	39	215	Compiler
*6	857	Wait	3	194	ReferenceQueueDaemon
*7	858	Wait	14	197	FinalizerDaemon
*8	859	Wait	6	200	FinalizerWatchdogDaemon
9	860	Native	42	428	Binder_1
10	862	Native	11	400	Binder_2
11	894	Native	140	228	launcher-loader
12	895	Wait	50	202	AsyncTask #1
13	1136	Wait	54	217	AsyncTask #2
14	1140	Wait	28	214	AsyncTask #3
15	1141	Wait	29	212	AsyncTask #4
16	1142	Wait	63	214	AsyncTask #5
17	14497	Native	1	373	Binder_3
18	23012	Native	1	355	Binder_4
19	27011	Native	5	313	Binder_5
20	9752	Native	0	205	Binder_6
21	24554	Native	1	175	Binder_7
22	10990	Native	0	143	Binder_8
23	12646	Native	0	138	Binder_9
24	14872	Native	2	132	Binder_A
25	13944	Native	0	75	Binder_B
26	29564	Native	1	48	Binder_C

Figure 2: Threads in Android applications.

The task is now to modify the most common multi-threading mechanism for Binder threads and their main thread such that the Binder PID/UID information are forwarded along direct control flows to the main thread. This channel consists of `android.os.Message` objects that are posted from the Binder thread to the message queue of the main thread, from where the messages are sequentially dispatched by the main thread's `Looper` to `Handlers` which process the message. Various kinds of handlers exist and in this task we are only interested in the most important for Android applications, `ActivityThread`, which handles messages for Activity life-cycle management, processing (broadcast) Intents, etc.

1. First, extend the `android.os.Message` class such that it can store the Binder information for calling UID and PID:

```
...

public Messenger replyTo;

// New attributes for Binder information
```

```
/* package */ int fromUid = -1;
/* package */ int fromPid = -1;

/** If set message is in use */
/*package*/ static final int FLAG_IN_USE = 1 << 0;

...
```

2. Of course, all copy constructors and the parceling functions of `Message.java` must be extended to correctly hold these new attributes:
   - `public static Message obtain(Message orig)`
   - `public void copyFrom(Message o)`
   - `/*package*/ void clearForRecycle()`
   - `public void writeToParcel(Parcel dest, int flags)`
   - `private void readFromParcel(Parcel source)`
3. Also add setters and getters functions for these new attributes:
   - `int getFromUid()`
   - `int getFromPid()`
   - `void setFromUid(int uid)`
   - `void setFromPid(int pid)`
4. Next, extend the handlers, `android.os.Handler`, to attach the available Binder information to new messages that it will add to a the message queue of a target thread:

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis)
{
  msg.target = this;
  if (mAsynchronous) {
    msg.setAsynchronous(true);
  }

  // Attach Binder information of current IPC context
  msg.setFromUid(Binder.getCallingUid());
  msg.setFromPid(Binder.getCallingPid());


  return queue.enqueueMessage(msg, uptimeMillis);
}
```

5. Now we extend the Binder user-space libraries and classes such, that we can store and retrieve these new information to and from the Thread Local Storage of threads. This is very similar to the way how Android stores the calling UID and PID that can be retrieved via `Binder.getCallingUid` and `Binder.getCallingPid`. These information are managed in the native `IPCThreadState` class, which we also extend with new attributes for *fromUid* and *fromPid* information and getters/setters for these attributes. Extend the `IPCThreadState.h` as follows:

```
// New functions
int getFromUid();
int getFromPid();
void setFromUid(uid_t uid);
void setFromPid(pid_t pid);
...

// New attributes
pid_t mFromPid;
uid_t mFromUid;
```

6. Next, implement the functions in `IPCThreadState.cpp` and extend the constructor to assign default values to `mFromUid` and `mFromPid`:

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(androidGetTid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0),
      mFromUid(-1),
      mFromPid(-1)
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}

int IPCThreadState::getCallingUid()
{
  return mCallingUid;
}

int IPCThreadState::getFromUid()
{
  return mFromUid;
}

int IPCThreadState::getFromPid()
{
  return mFromPid;
}

void IPCThreadState::setFromUid(uid_t uid)
{
  mFromUid = uid;
}

void IPCThreadState::setFromPid(pid_t pid)
{
  mFromPid = pid;
}
```

7. To access this native functions from `Binder.java`, we have to extend the Java Native Interface for the Binder libraries. Open `android_util_Binder.cpp` and implement as well as register the bridge functions between `Binder.java` and `IPCThreadState.cpp`:

```
static jint android_os_Binder_getFromUid(JNIEnv* env, jobject clazz)
{
  return IPCThreadState::self()->getFromUid();
}

static jint android_os_Binder_getFromPid(JNIEnv* env, jobject clazz)
{
  return IPCThreadState::self()->getFromPid();
}

static void android_os_Binder_setFromUid(JNIEnv* env, jobject clazz, jint uid)
{
  return IPCThreadState::self()->setFromUid(uid);
}

static void android_os_Binder_setFromPid(JNIEnv* env, jobject clazz, jint pid)
```

```
{
  return IPCThreadState::self()->setFromPid(pid);
}

...

static const JNINativeMethod gBinderMethods[] = {
     /* name, signature, funcPtr */
    { "getCallingPid", "()I", (void*)android_os_Binder_getCallingPid },
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid },
    { "clearCallingIdentity", "()J", (void*)android_os_Binder_clearCallingIdentity
        },
    { "restoreCallingIdentity", "(J)V", (void*)
        android_os_Binder_restoreCallingIdentity },
    { "setThreadStrictModePolicy", "(I)V", (void*)
        android_os_Binder_setThreadStrictModePolicy },
    { "getThreadStrictModePolicy", "()I", (void*)
        android_os_Binder_getThreadStrictModePolicy },
    { "flushPendingCommands", "()V", (void*)android_os_Binder_flushPendingCommands
        },
    { "init", "()V", (void*)android_os_Binder_init },
    { "destroy", "()V", (void*)android_os_Binder_destroy },

    /* Register new functions for storing/retrieving fromUid and fromPid in TLS */
    { "getFromUid", "()I", (void*)android_os_Binder_getFromUid },
    { "getFromPid", "()I", (void*)android_os_Binder_getFromPid },
    { "setFromUid", "(I)V", (void*)android_os_Binder_setFromUid },
    { "setFromPid", "(I)V", (void*)android_os_Binder_setFromPid },
};
```

8. Now extend `Binder.java` to provide these functions in the Android OS API:

```
public static final native int getFromUid();
public static final native int getFromPid();

public static final native void setFromUid(int uid);
public static final native void setFromPid(int pid);
```

9. To actually use the new Binder functions in other parts of the middleware, run `make update-api` as shown earlier in this exercise.

10. Finally, after the API has been updated, we extend `Looper.java`, which dispatches messages from the message queue to handlers. Every time the next queued message should be handled, the looper should adjust the `fromUid` and `fromPid` values of its thread to the values attached to the message. After dispatching and handling the message, it should restore the original values:

```
public static void loop() {
  final Looper me = myLooper();
  if (me == null) {
    throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this
        thread.");
  }
  final MessageQueue queue = me.mQueue;

  // Make sure the identity of this thread is that of the local process,
  // and keep track of what that identity token actually is.
  Binder.clearCallingIdentity();
  final long ident = Binder.clearCallingIdentity();

  for (;;) {
```

```
    Message msg = queue.next(); // might block
    if (msg == null) {
      // No message indicates that the message queue is quitting.
      return;
    }

    // This must be in a local variable, in case a UI event sets the logger
    Printer logging = me.mLogging;
    if (logging != null) {
      logging.println(">>>>> Dispatching to " + msg.target + " " +
      msg.callback + ": " + msg.what);
    }

    /* For the duration of handling this message, we update the Binder information
     * according to the information attached to the message.
     */
    int oldFromUid = Binder.getFromUid();
    int oldFromPid = Binder.getFromPid();
    if(msg.fromUid != -1)
    {
      if(logging != null) {
        logging.println("----- Updating Binder information from "+oldFromUid+":"+
            oldFromPid+" -> "+msg.getFromUid()+":"+msg.getFromPid());
        logging.println("----- Binder information are "+Binder.getCallingUid()+":"+
            Binder.getCallingPid());
      }
      Binder.setFromUid(msg.getFromUid());
      Binder.setFromPid(msg.getFromPid());
    } else {
      if(logging != null) {
        logging.println("----- Message without Binder information");
      }
    }

    // Calls the target handler
    msg.target.dispatchMessage(msg);

    // Message handling is finished. Restore original values.
    if(logging != null)
    {
      logging.println("----- Restoring Binder information from "+Binder.getFromUid()
          +":"+Binder.getFromPid()+" -> "+oldFromUid+":"+oldFromPid);
      Binder.setFromUid(oldFromUid);
      Binder.setFromPid(oldFromPid);
    }

    if (logging != null) {
      logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);
    }

    // Make sure that during the course of dispatching the
    // identity of the thread wasn't corrupted.
    final long newIdent = Binder.clearCallingIdentity();
    if (ident != newIdent) {
      Log.wtf(TAG, "Thread identity changed from 0x"
      + Long.toHexString(ident) + " to 0x"
      + Long.toHexString(newIdent) + " while dispatching to "
      + msg.target.getClass().getName() + " "
      + msg.callback + " what=" + msg.what);
    }

  msg.recycle();
  }
}
```

11. To activate the logging in the above code for loopers of the handler `ActvitiyThread.H`, change the `if` condition in the `main()` function of `ActivityThread.java`:

```
if (true) {
  Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG, "ActivityThread"));
}
```

12. Now build the system and test it on the emulator. If you want to enable developers of 3rd party applications to be able to use the new Binder functions to check who their caller is (e.g., in their Activities or IntentServices), you again have to create a custom SDK for them.

Question 9:
When you activate the logging in `ActivityThread.java`, observe the different values for `Binder.getCallingUid()` and `Message.getFromUid()` during execution of the system. Can you explain what it means when these values are equal and when they are different?

Question 10:
When using `Messengers`[5] in Android to communicate with a remote service, `Messages` are send via Binder IPC from the client to the service, where they are dispatched to `Handlers` by the service's `Looper`. In this scenario, what does this entail for the meaning of `fromUid` and `fromPid`, which are attached to the message, for the service that receives and processes the message?