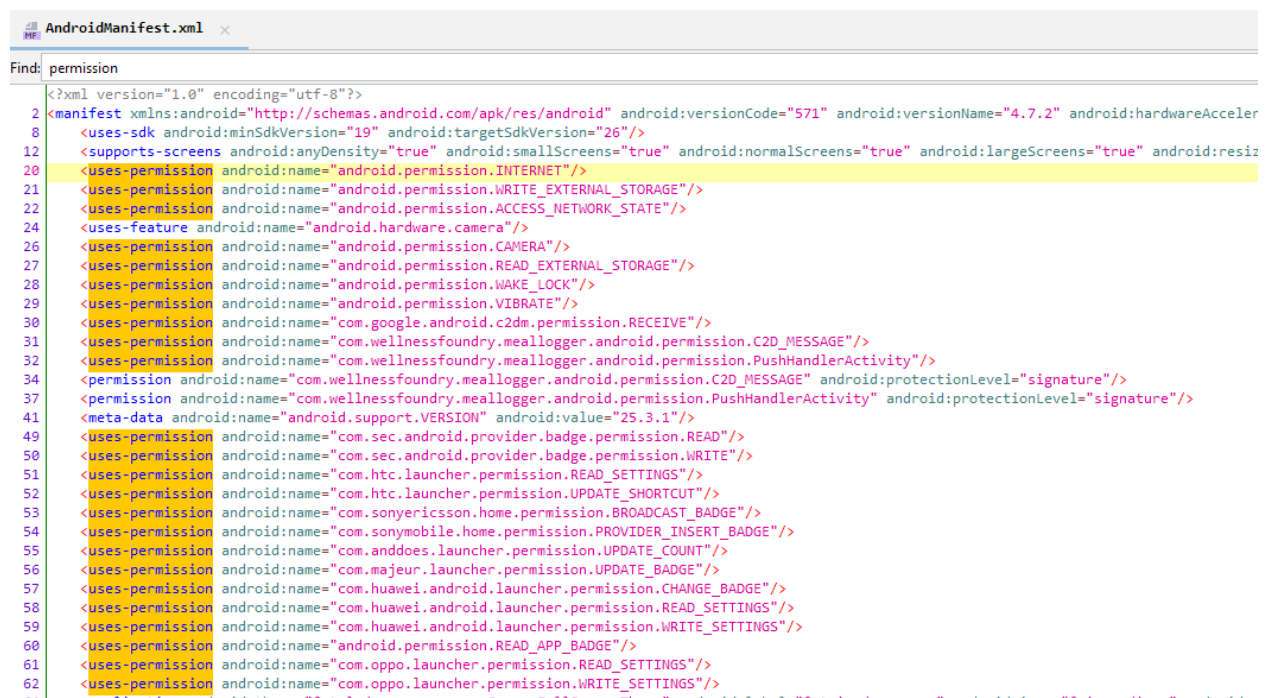


## Module 10 Assignment

**Testing App Permissions (MSTG-PLATFORM-1)**

By opening the AndroidManifest.xml file (shown below) in jadx and searching for the permission keyword, we can see the various permissions allowed by the Meallogger application. By looking closely at the actual permissions included, it seems that the read/write permissions for both external storage and settings are clearly suspicious and perhaps unneeded.



The screenshot shows the AndroidManifest.xml file with a search for 'permission'. The results are as follows:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="571" android:versionName="4.7.2" android:hardwareAccelerated="true">
3     <uses-sdk android:minSdkVersion="19" android:targetSdkVersion="26"/>
4     <supports-screens android:anyDensity="true" android:smallScreens="true" android:normalScreens="true" android:largeScreens="true" android:resizeable="true"/>
5     <uses-permission android:name="android.permission.INTERNET"/>
6     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
7     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
8     <uses-feature android:name="android.hardware.camera" android:required="false"/>
9     <uses-permission android:name="android.permission.CAMERA"/>
10    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
11    <uses-permission android:name="android.permission.WAKE_LOCK"/>
12    <uses-permission android:name="android.permission.VIBRATE"/>
13    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
14    <uses-permission android:name="com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE"/>
15    <uses-permission android:name="com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity"/>
16    <permission android:name="com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE" android:protectionLevel="signature"/>
17    <permission android:name="com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity" android:protectionLevel="signature"/>
18    <meta-data android:name="android.support.VERSION" android:value="25.3.1"/>
19    <uses-permission android:name="com.sec.android.provider.badge.permission.READ"/>
20    <uses-permission android:name="com.sec.android.provider.badge.permission.WRITE"/>
21    <uses-permission android:name="com.htc.launcher.permission.READ_SETTINGS"/>
22    <uses-permission android:name="com.htc.launcher.permission.UPDATE_SHORTCUT"/>
23    <uses-permission android:name="com.sonyericsson.home.permission.BROADCAST_BADGE"/>
24    <uses-permission android:name="com.sonymobile.home.permission.PROVIDER_INSERT_BADGE"/>
25    <uses-permission android:name="com.anddoes.launcher.permission.UPDATE_COUNT"/>
26    <uses-permission android:name="com.majeur.launcher.permission.UPDATE_BADGE"/>
27    <uses-permission android:name="com.huawei.android.launcher.permission.CHANGE_BADGE"/>
28    <uses-permission android:name="com.huawei.android.launcher.permission.READ_SETTINGS"/>
29    <uses-permission android:name="com.huawei.android.launcher.permission.WRITE_SETTINGS"/>
30    <uses-permission android:name="android.permission.READ_APP_BADGE"/>
31    <uses-permission android:name="com.oppo.launcher.permission.READ_SETTINGS"/>
32    <uses-permission android:name="com.oppo.launcher.permission.WRITE_SETTINGS"/>
33 </manifest>

```

Furthermore, by running the command:

```
adb shell dumpsys package com.wellnessfoundry.meallogger.android
```

I was able to find a similar list of permissions, but this output has been separated into declared permissions, requested/installed permissions, and runtime permissions (shown below). In the same nature as the AndroidManifest.xml file, we can see that the read/write permissions for external storage and settings are requested by the application. Looking at the custom (declared) permissions, the C2D\_MESSAGE allows the cloud to communicate with the device, while the PushHandlerActivity allows for custom notifications. Although not apparently malicious, it is important to note the existence of these custom permissions.

```
declared permissions:
  com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE: prot=signature, INSTALLED
  com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity: prot=signature, INSTALLED
requested permissions:
  android.permission.INTERNET
  android.permission.WRITE_EXTERNAL_STORAGE: restricted=true
  android.permission.ACCESS_NETWORK_STATE
  android.permission.CAMERA
  android.permission.READ_EXTERNAL_STORAGE: restricted=true
  android.permission.WAKE_LOCK
  android.permission.VIBRATE
  com.google.android.c2dm.permission.RECEIVE
  com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE
  com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity
  com.sec.android.provider.badge.permission.READ
  com.sec.android.provider.badge.permission.WRITE
  com.htc.launcher.permission.READ_SETTINGS
  com.htc.launcher.permission.UPDATE_SHORTCUT
  com.sonyericsson.home.permission.BROADCAST_BADGE
  com.sonymobile.home.permission.PROVIDER_INSERT_BADGE
  com.anddoes.launcher.permission.UPDATE_COUNT
  com.majeur.launcher.permission.UPDATE_BADGE
  com.huawei.android.launcher.permission.CHANGE_BADGE
  com.huawei.android.launcher.permission.READ_SETTINGS
  com.huawei.android.launcher.permission.WRITE_SETTINGS
  android.permission.READ_APP_BADGE
  com.oppo.launcher.permission.READ_SETTINGS
  com.oppo.launcher.permission.WRITE_SETTINGS
  android.permission.ACCESS_MEDIA_LOCATION
install permissions:
  com.google.android.c2dm.permission.RECEIVE: granted=true
  com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity: granted=true
  android.permission.INTERNET: granted=true
  android.permission.ACCESS_NETWORK_STATE: granted=true
  android.permission.VIBRATE: granted=true
  com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE: granted=true
  android.permission.WAKE_LOCK: granted=true
```

### **Testing for Injection Flaws and URLs in WebViews (MSTG-PLATFORM-2)**

By opening the AndroidManifest.xml file in jadx and searching for “intent-filter”, we are able to see potential custom URL schemes. By analyzing the results (shown below), we can see that there is a custom URL scheme, but it seems to be empty and unused. Otherwise, there exist activities that can be opened and viewed in a browser, which are most likely related to in-app assets.

```

<application android:theme="@style/squarecamera__CameraFullScreenTheme" android:
  <activity android:theme="@android:style/Theme.DeviceDefault.NoActionBar" and
    <intent-filter android:label="@string/launcher_name">
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <intent-filter>
      <action android:name="android.intent.action.VIEW"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <category android:name="android.intent.category.BROWSABLE"/>
      <data android:scheme="meallogger"/>
    </intent-filter>
    <intent-filter>
      <action android:name="android.intent.action.VIEW"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <category android:name="android.intent.category.BROWSABLE"/>
      <data android:scheme=" " android:host=" " android:pathPrefix="/" />
    </intent-filter>
  </activity>

```

By searching the manifest file for “provider”, we can see that there are a number of content providers used by the application (shown below). The first instance shows a plugin for a file provider, which is likely used to sync local and remote data. Moreover, the second instance shows a similar plugin provided by Apache’s Cordova software, specifically related to the camera. Although neither of these are inherently malicious, it is important to note their presence.

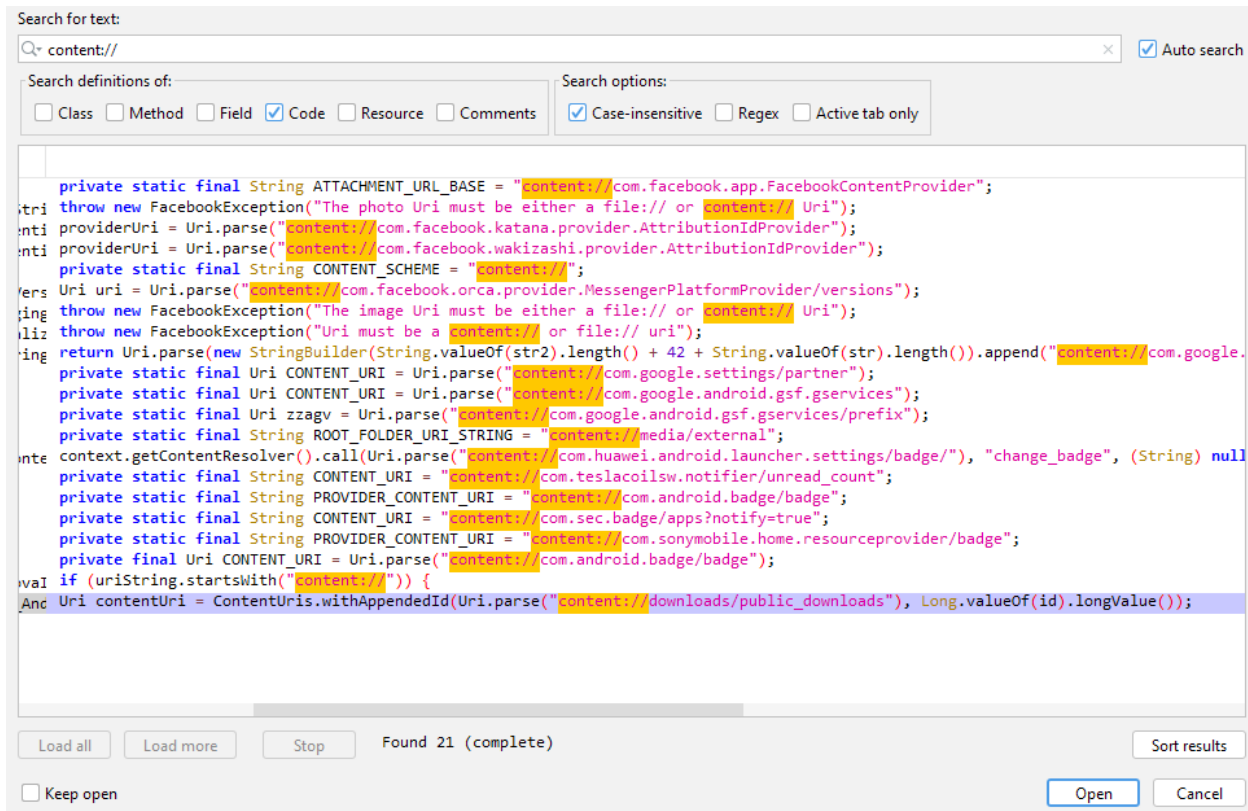
```

<receiver
  <provider android:name="nl.xservices.plugins.FileProvider" android:exported="false" android:authorities="com
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/sharing_paths"/>
  </provider>
  . . . . .

<provider android:name="org.apache.cordova.camera.FileProvider" android:exported="false" android:authorities="com.w
  <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/camera_provider_paths"/>
</provider>
. . . . .

```

By opening the apk file in jadx and searching for “content://”, we are able to see that there are a number of URLs used in conjunction with the Meallogger application. Overall, most of them seems to be related to either a Facebook asset or a built-in android badge. Looking closer into the files where each instance is located, such assumptions are confirmed. Following this trend, using the adb content query yielded no significant results in relation to the URLs. However, this might be helpful to look into in the future.



Finally, looking into the AndroidManifest.xml file, there was no sign of *android.webkit.WebView.EnableSafeBrowsing*. Thus, this means that safe browsing is enabled by default in the application.

### Testing Deep Links and Custom URL Schemes (MSTG-PLATFORM-3)

By looking for the data keyword in the intent-filter blocks present within the AndroidManifest.xml file, I was able to find two instances (shown below). As mentioned in the previous section, there seems to be an android scheme labeling the app itself, as well as an empty URL scheme. This could be significant for testing in the future.

```
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="meallogger"/>
</intent-filter>
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme=" " android:host=" " android:pathPrefix="/">
</intent-filter>
```

## Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)

By opening the AndroidManifest.xml file in jadx and searching for “intent-filter” under another component element, we are able to see a number of exported activities from the Meallogger application (shown below). In the first screenshot, it is clear that the intent-filter contents are related to activity that can be viewed and opened within a browser. Although not inherently malicious, it is important to note. In the second and third screenshots, we can see more signs of exported receivers and services. While many of these lines are non-impactful, the “CampaignTrackingReceiver” for Google Analytics and “IDListenerService” seem suspicious and perhaps unneeded by the application. Future investigation could be made on these specific activities. Other than those instances, there are no activities present in the xml file with an “exported=true” flag.

```
<activity android:theme="@android:style/Theme.DeviceDefault.NoActionBar" android:exported="true">
    <intent-filter android:label="@string/launcher_name">
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="meallogger"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme=" " android:host=" " android:pathPrefix="/" />
    </intent-filter>
</activity>

<receiver android:name="nl.xservices.plugins.ShareChooserPendingIntent" android:enabled="true">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
    </intent-filter>
</receiver>
<provider android:name="nl.xservices.plugins.FileProvider" android:exported="false" android:authorities="com.wellnessfoundry.meallo"
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/sharing_paths"/>
</provider>
<service android:name="io.intercom.android.sdk.IntercomIntentService" android:exported="false">
    <intent-filter android:priority="999">
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
    </intent-filter>
</service>
<receiver android:name="com.google.android.gms.analytics.AnalyticsReceiver" android:enabled="true">
    <intent-filter>
        <action android:name="com.google.android.gms.analytics.ANALYTICS_DISPATCH"/>
    </intent-filter>
</receiver>
<service android:name="com.google.android.gms.analytics.AnalyticsService" android:enabled="true" android:exported="false"/>
<receiver android:name="com.google.android.gms.analytics.CampaignTrackingReceiver" android:enabled="true" android:exported="true">
    <intent-filter>
        <action android:name="com.android.vending.INSTALL_REFERRER"/>
    </intent-filter>
</receiver>
```

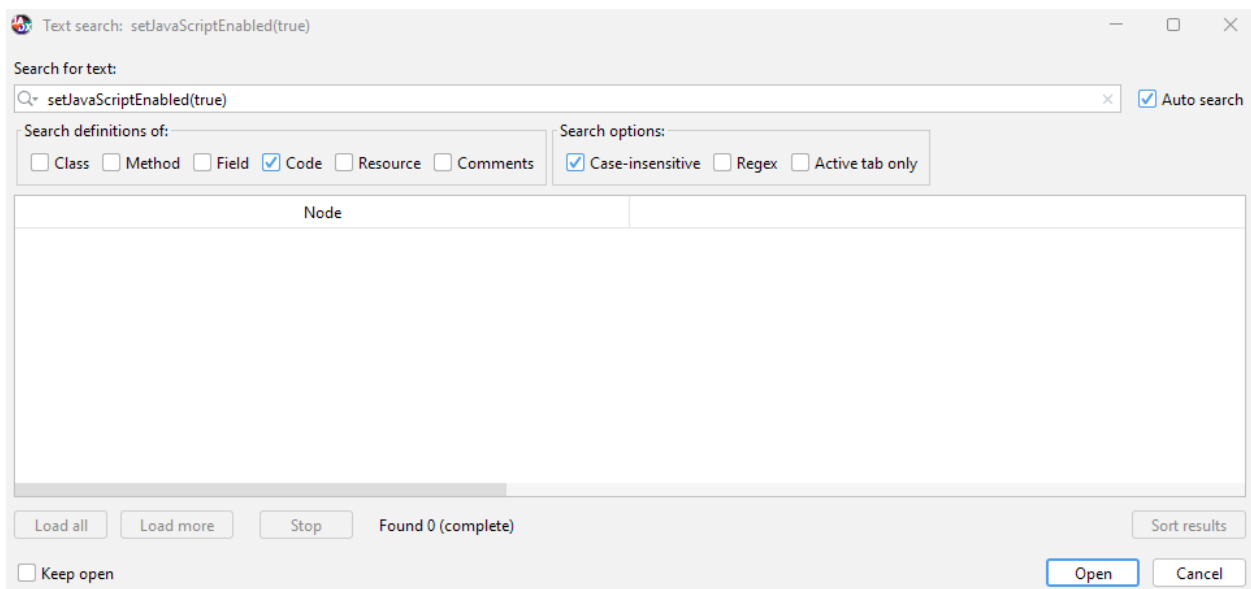
```

<receiver android:name="com.google.android.gms.gcm.GcmReceiver" android:permission="com.google.android.c2dm.permission.SEND" android:exported="true">
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
    <category android:name="com.wellnessfoundry.meallogger.android"/>
  </intent-filter>
</receiver>
<service android:name="com.adobe.phonegap.push.GCMIntentService" android:exported="false">
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
  </intent-filter>
</service>
<service android:name="com.adobe.phonegap.push.PushInstanceIDListenerService" android:exported="false">
  <intent-filter>
    <action android:name="com.google.android.gms.iid.InstanceID"/>
  </intent-filter>
</service>

```

## **Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5)**

Searching for “setJavaScriptEnabled(true)” in jadx yielded no results, as shown below.



## **Testing enforced updating (MSTG-ARCH-9)**

Although I was not able to download an outdated version of the application, I was still able to find signs of forced updates. By opening the apk file in jadx and searching for the “update” keyword, I was able to find that there is a flag labeled

*SERVICE\_VERSION\_UPDATE\_REQUIRED*

as well as a case statement that returns the value for that same flag. In this sense, there definitely seems to be a check for whether or not the app is fully updated within the current system.

```

public class CommonStatusCodes {
    public static final int API_NOT_CONNECTED = 17;
    public static final int CANCELED = 16;
    public static final int DEVELOPER_ERROR = 10;
    public static final int ERROR = 13;
    public static final int INTERNAL_ERROR = 8;
    public static final int INTERRUPTED = 14;
    public static final int INVALID_ACCOUNT = 5;
    public static final int NETWORK_ERROR = 7;
    public static final int RESOLUTION_REQUIRED = 6;
    @Deprecated
    public static final int SERVICE_DISABLED = 3;
    @Deprecated
    public static final int SERVICE_VERSION_UPDATE_REQUIRED = 2;
    public static final int SIGN_IN_REQUIRED = 4;
    public static final int SUCCESS = 0;
    public static final int SUCCESS_CACHE = -1;
    public static final int TIMEOUT = 15;

    @NonNull
    public static String getStatusCodeString(int i) {
        switch (i) {
            case -1:
                return "SUCCESS_CACHE";
            case 0:
                return "SUCCESS";
            case 1:
            case 9:
            case 11:
            case 12:
            default:
                return new StringBuilder(32).append("unknown status code: ").append(i).toString();
            case 2:
                return "SERVICE_VERSION_UPDATE_REQUIRED";
        }
    }
}

```

### **Making Sure That the App is Properly Signed (MSTG-CODE-1)**

By running the `apksigner verify` command in the terminal, we can see that the Meallogger application has both V1 and V2 signatures.

```

Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): false
Verified using v4 scheme (APK Signature Scheme v4): false
Verified for SourceStamp: false
Number of signers: 1

```

## Testing Whether the App is Debuggable (MSTG-CODE-2)

By opening the AndroidManifest.xml file in jadx and searching for the “android:debuggable” attribute, there were no instances of this attribute.

## Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4)

After checking the MobSF report, there were no signs of the specified strings. However, by searching for a variety of keywords in jadx, I was able to find various instances for “log”, “error”, and “StrictMode”. Below are screenshots and descriptions of the results:

```
public static IBinder getBinder(Bundle bundle, String key) {
    if (!sGetIBinderMethodFetched) {
        try {
            sGetIBinderMethod = Bundle.class.getMethod("getIBinder", String.class);
            sGetIBinderMethod.setAccessible(true);
        } catch (NoSuchMethodException e) {
            Log.i(TAG, "Failed to retrieve getIBinder method", e);
        }
        sGetIBinderMethodFetched = true;
    }
    if (sGetIBinderMethod != null) {
        try {
            return (IBinder) sGetIBinderMethod.invoke(bundle, key);
        } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e2) {
            Log.i(TAG, "Failed to invoke getIBinder via reflection", e2);
            sGetIBinderMethod = null;
        }
    }
    return null;
}
```

“log”: For this keyword, we can see that the source code does in fact log information in the system, using a public class. Although the effects of this are not easily apparent, logging sensitive information on the system’s memory is bad practice, as similarly noted by MobSF.

```
private final FutureTask<Result> mFuture = new FutureTask<Result>(this, mWorker) { // from class: android.support.v4.content.ModernAsyncTask.3
    @Override // java.util.concurrent.FutureTask
    protected void done() {
        try {
            Result result = get();
            ModernAsyncTask.this.postResultIfNotInvoked(result);
        } catch (InterruptedException e) {
            Log.w(ModernAsyncTask.LOG_TAG, e);
        } catch (CancellationException e2) {
            ModernAsyncTask.this.postResultIfNotInvoked(null);
        } catch (ExecutionException e3) {
            throw new RuntimeException("An error occurred while executing doInBackground()", e3.getCause());
        } catch (Throwable t) {
            throw new RuntimeException("An error occurred while executing doInBackground()", t);
        }
    }
};
```



**“error”:** For this keyword, we can see that there is in fact an instance of verbose error messaging. However, it is not entirely clear what the error is referring to, so this might not be a significant issue.

```
public final T get() {  
    boolean z;  
    HashSet<String> hashSet;  
    Context context;  
    T retrieve;  
    if (this.zzmz != null) {  
        return this.zzmz;  
    }  
    StrictMode.ThreadPolicy allowThreadDiskReads = StrictMode.allowThreadDiskReads();  
    synchronized (sLock) {  
        z = zzmw != null && zzd(zzmw);  
        hashSet = zzmy;  
        context = zzmw;  
    }  
}
```

**“StrictMode”:** For this keyword, we can see a very significant instance of StrictMode. In this case, the application is able to read from the system’s disk, which can lead to many potential vulnerabilities. Further investigation into this specific thread policy could yield more possible issues with this finding.