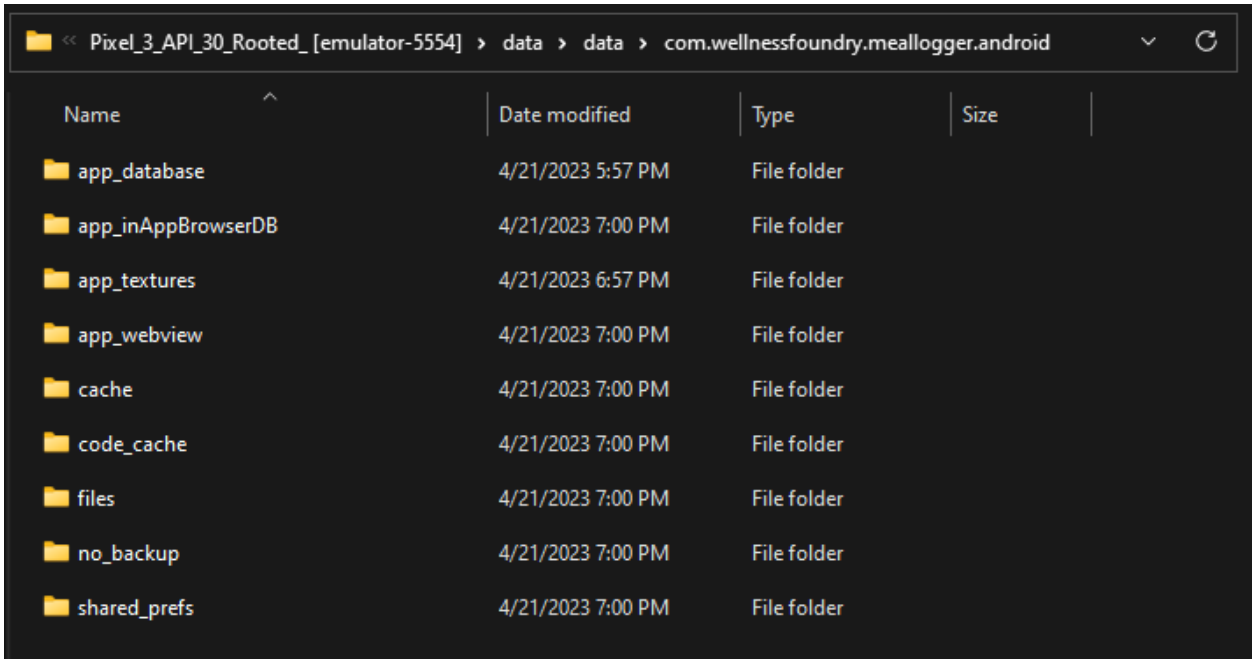
Chad Josim

CS 453

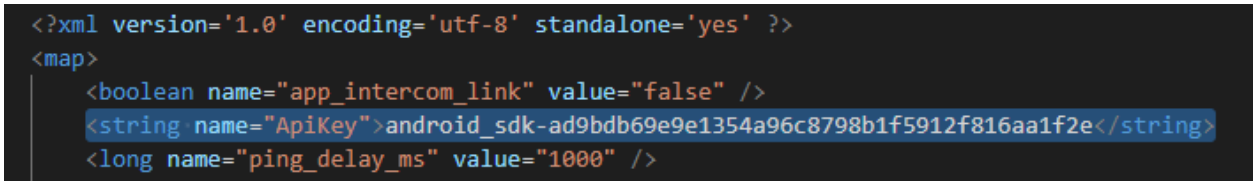
Module 12 Assignment

MSTG-STORAGE-1, MSTG-STORAGE-2

By performing dynamic app analysis and testing all functionalities, I was able to find and save all of the files generated from a session in the Meallogger application (shown below).

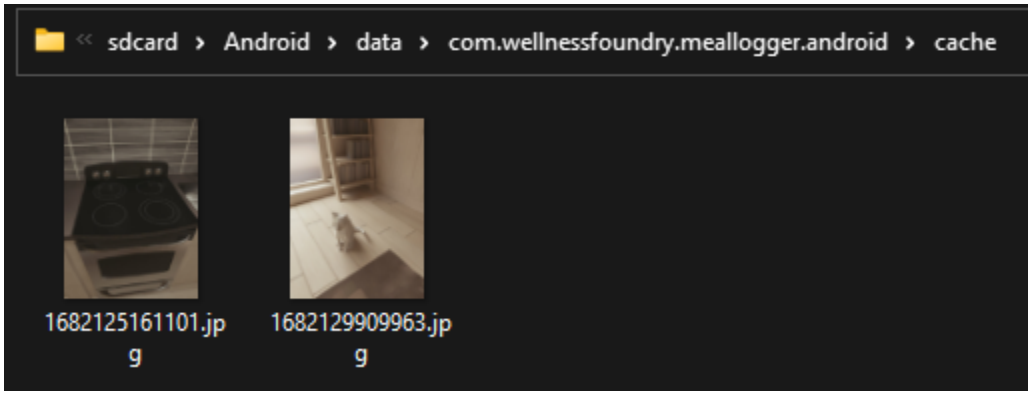


Interestingly, the app_database, app_inAppBrowserDB, app_textures, code_cache, and files directories were all empty. In this case, it seems that a large section of the app's database and important files were not unnecessarily stored. However, there were several xml files in shared_prefs, some of which exposed sensitive information, like an exposed ApiKey and user ID/email (shown below).



```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="intercomsdk-session-INTERCOM_SDK_INTERCOM_ID"></string>
    <string name="intercomsdk-session-INTERCOM_SDK_USER_ID">169533</string>
    <string name="intercomsdk-session-INTERCOM_SDK_EMAIL_ID">mpyra328@gmail.com</string>
</map>
```

Looking at the external storage file location (sdcard), all of the images that I took within the app (shown below) were actually saved, which is very significant.



MSTG-STORAGE-3

By opening the apk file in jadx and performing static analysis, I was able to search the source code for significant keywords, specifically those relating to logging and system print statements. Firstly, I was able to find a LogWriter class (shown below), which imports android.util.Log. Clearly, this is related to writing logs from the application, although there is nothing immediately suspicious within the class itself.

LogWriter

```
package android.support.p000v4.util;

import android.support.annotation.RestrictTo;
import android.util.Log;
import java.io.Writer;
```

By searching for other instances of "log._", I was able to locate several instances of logging. One such example (shown below) is related to Google Play utilities, which logs whether or not such services are up to date. Although not immediately malicious, it is still important to note.

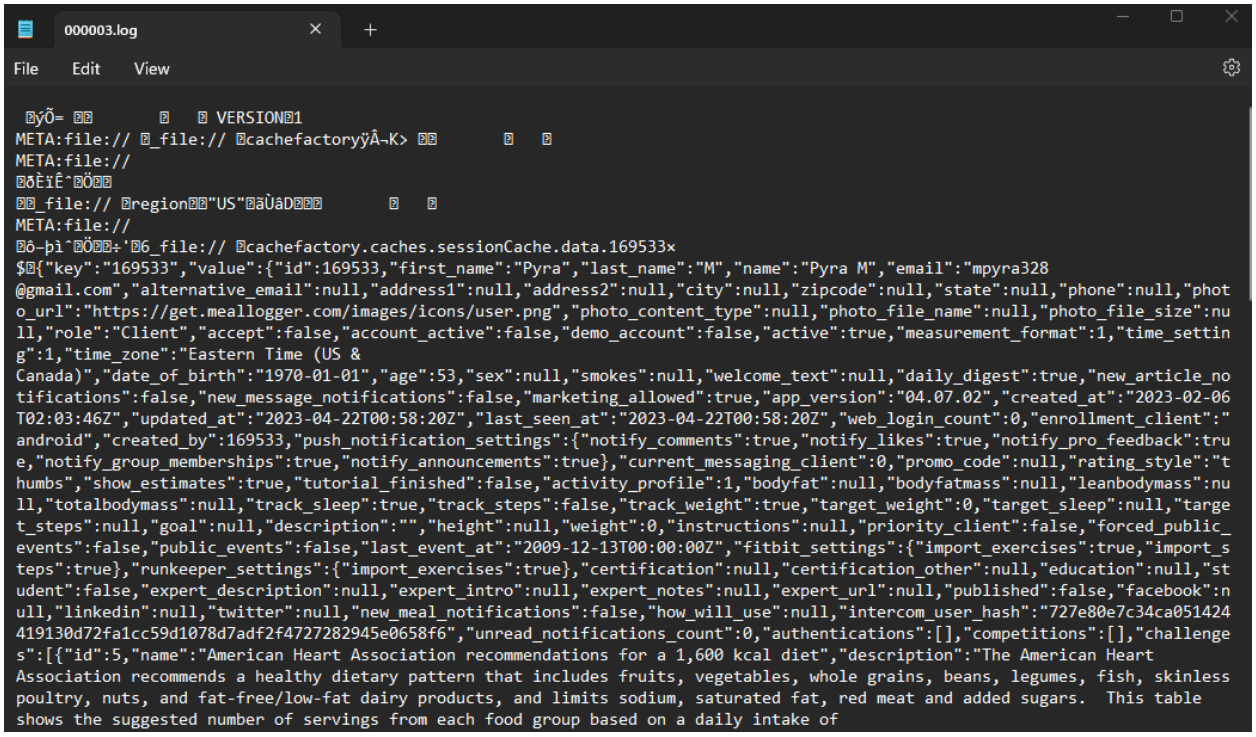
```
@VisibleForTesting
private static int zza(Context context, boolean z, int i) {
    Preconditions.checkArgument(i >= 0);
    PackageManager packageManager = context.getPackageManager();
    PackageInfo packageInfo = null;
    if (z) {
        try {
            packageInfo = packageManager.getPackageInfo("com.android.vending", 8256);
        } catch (PackageManager.NameNotFoundException e) {
            Log.w("GooglePlayServicesUtil", "Google Play Store is missing.");
            return 9;
        }
    }
    try {
        PackageInfo packageInfo2 = packageManager.getPackageInfo("com.google.android.gms", 64);
        GoogleSignatureVerifier googleSignatureVerifier = GoogleSignatureVerifier.getInstance(context);
        if (!googleSignatureVerifier.isGooglePublicSignedPackage(packageInfo2, true)) {
            Log.w("GooglePlayServicesUtil", "Google Play services signature invalid.");
            return 9;
        } else if (z && (!googleSignatureVerifier.isGooglePublicSignedPackage(packageInfo, true) || !packageInfo.signatures[
            Log.w("GooglePlayServicesUtil", "Google Play Store signature invalid.");
            return 9;
        } else if (GmsVersionParser.parseBuildVersion(packageInfo2.versionCode) < GmsVersionParser.parseBuildVersion(i)) {
            Log.w("GooglePlayServicesUtil", new StringBuilder(77).append("Google Play services out of date.  Requires ").app
            return 2;
        } else {
            ApplicationInfo applicationInfo = packageInfo2.applicationInfo;
            if (applicationInfo == null) {
                try {
                    applicationInfo = packageManager.getApplicationInfo("com.google.android.gms", 0);
                } catch (PackageManager.NameNotFoundException e2) {
                    Log.wtf("GooglePlayServicesUtil", "Google Play services missing when getting application info.", e2);
                    return 1;
                }
            }
            return !applicationInfo.enabled ? 3 : 0;
        }
    } catch (PackageManager.NameNotFoundException e3) {
        Log.w("GooglePlayServicesUtil", "Google Play services is missing.");
        return 1;
```

By searching for the system print statements, I was able to find a small number of instances within the source code (shown below). However, similar to the log findings, nothing here is inherently suspicious.

```
System.out.println("DiskLruCache " + directory + " is corrupt: " + journalIsCorrupt.getMessage() + ", removing");
System.out.println("DiskLruCache " + directory + " is corrupt: " + journalIsCorrupt.getMessage() + ", removing");
System.out.println("Error adding plugin " + className + ".");
```

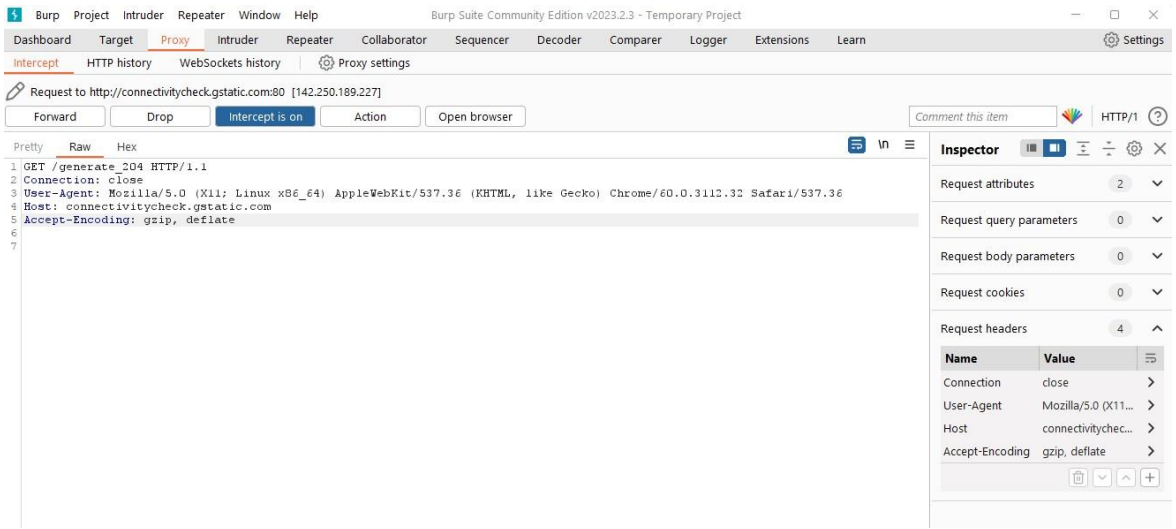
```
System.err.println("Failed to retrieve value from android.os.Build$VERSION.SDK_INT due to the following exception.");
System.err.println("Failed to retrieve value from android.os.Build$VERSION.SDK_INT due to the following exception.");
System.err.println("Failed to set 'rx.indexed-ring-buffer.size' with value " + sizeFromProperty + " => " + e.getMessage());
System.err.println("RxJavaErrorHandler threw an Exception. It shouldn't. => " + pluginException.getMessage());
System.err.println("Failed to set 'rx.buffer.size' with value " + sizeFromProperty + " => " + e.getMessage());
```

Finally, by performing dynamic analysis and searching the data folder for logs, I was able to find one significant log file (shown below). Although some parts are unreadable, the majority of the file shows relevant information pertaining to a session within the app. At a glance, the log file contains very sensitive information, including email, DOB, age, weight, and much more. Further investigation could reveal more instances of such logging.



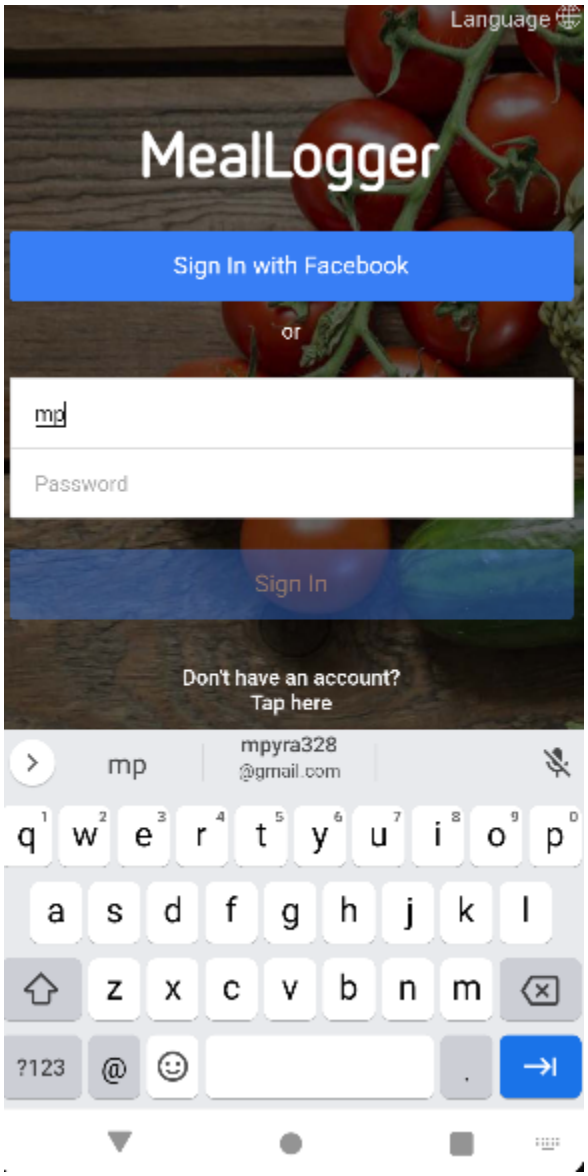
MSTG-STORAGE-4

To perform dynamic app analysis for this standard, I used a Burp Suite proxy to intercept HTTPs traffic from the application. From this, I found several instances of intercepted traffic, with one such example shown below. Although some of the information seems to be unencrypted, it is not clear whether or not sensitive information was sent to third parties. Future investigation could prove useful to further examine such activity.



MSTG-STORAGE-5

By performing dynamic app analysis, I was able to test the keyboard cache for the Meallogger application. After trying all of the places where sensitive data could be inputted, I found that the login page, specifically where it asks for your email (not password), pulls up suggested strings (shown below). Therefore, the keyboard cache has not been disabled for this field.



MSTG-STORAGE-6

By searching the xml file in jadx, I was able to find significant results relating to content providers and exposed data. Firstly, there were a variety of "android:exported" flags, with some set to true and some set to false (shown below). Along this line of thinking, we can reason that some of these flags are not necessary and likely give unneeded read/write permissions. Similarly, although there were instances of "permission" flags, their protection levels were not properly set, leading to potential vulnerabilities.

```
</receiver>
<service android:name="com.google.android.gms.analytics.CampaignTrackingService" android:enabled="true" android:exported="false"/>
<activity android:name="com.desmond.squarecamera.CameraActivity"/>
<activity android:name="me.crosswall.photo.pick.PickPhotosActiviy"/>
<activity android:name="com.adobe.phonegap.push.PushHandlerActivity" android:permission="com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity" android:exported="true"/>
<receiver android:name="com.adobe.phonegap.push.BackgroundActionButtonHandler"/>
<receiver android:name="com.adobe.phonegap.push.PushDismissedHandler"/>
<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>
```

Other testing for providers and data exposure had been done in a previous assignment, so I will include that here:

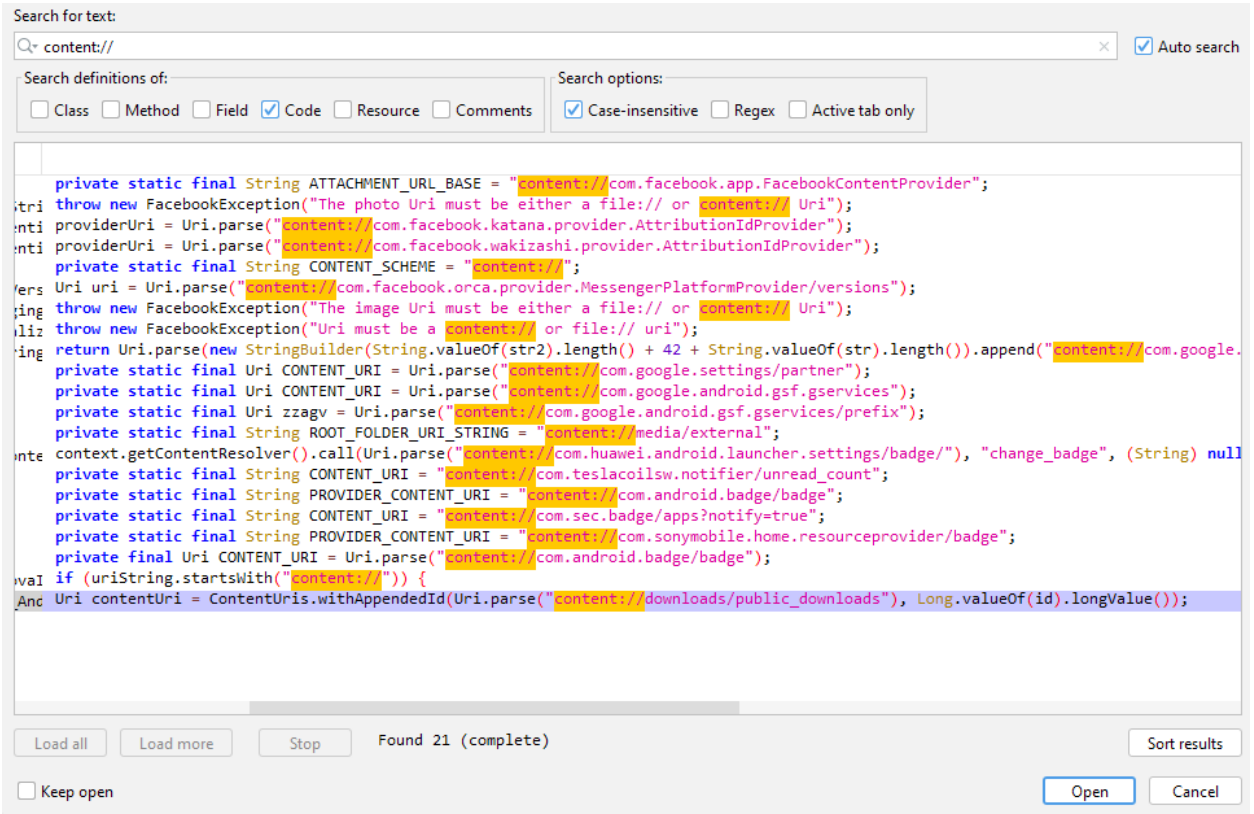
By opening the AndroidManifest.xml file in jadx and searching for "intent-filter", we are able to see potential custom URL schemes. By analyzing the results (shown below), we can see that there is a custom URL scheme, but it seems to be empty and unused. Otherwise, there exist activities that can be 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.

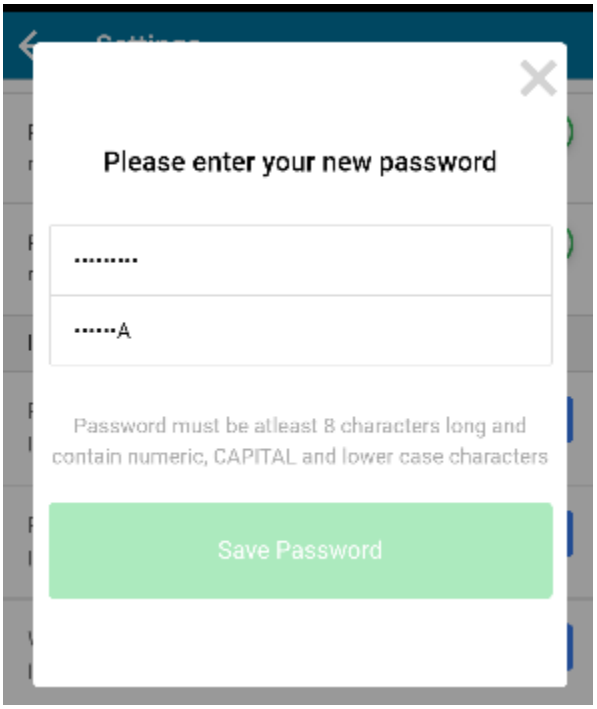
```
<provider android:name="nl.xservices.plugins.FileProvider" android:exported="false" android:authorities="com
    <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@xml/sharing_paths"/>
</provider>

<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.



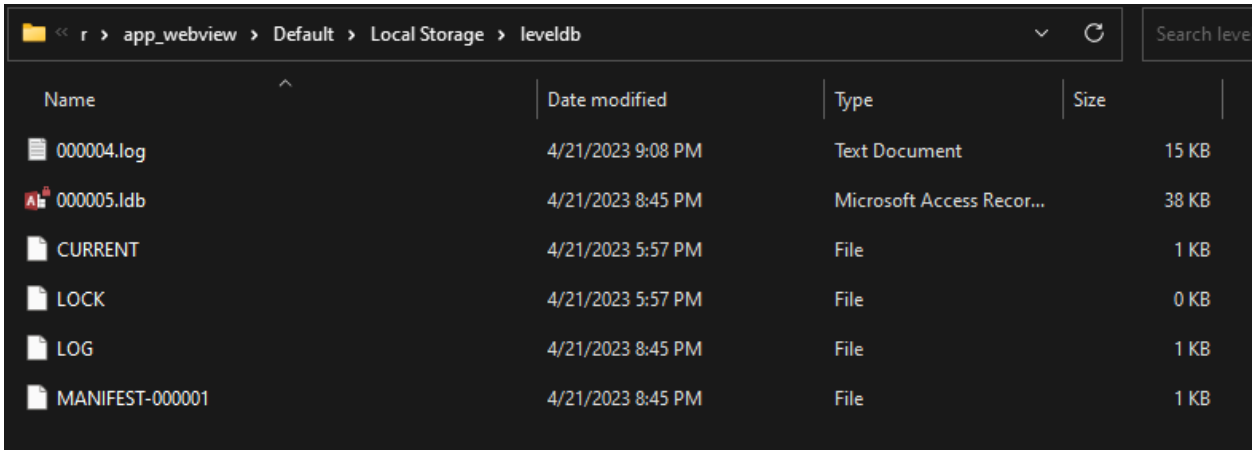
MSTG-STORAGE-7

By performing dynamic app analysis, I was able to test whether or not the application leaks any sensitive data to the user interface. Essentially, for anywhere that you login, the password field seems to be protected by changing the actual input characters into dots (as shown below). However, it is important to note that the actual plaintext characters do flash for an instant before turning into a dot.

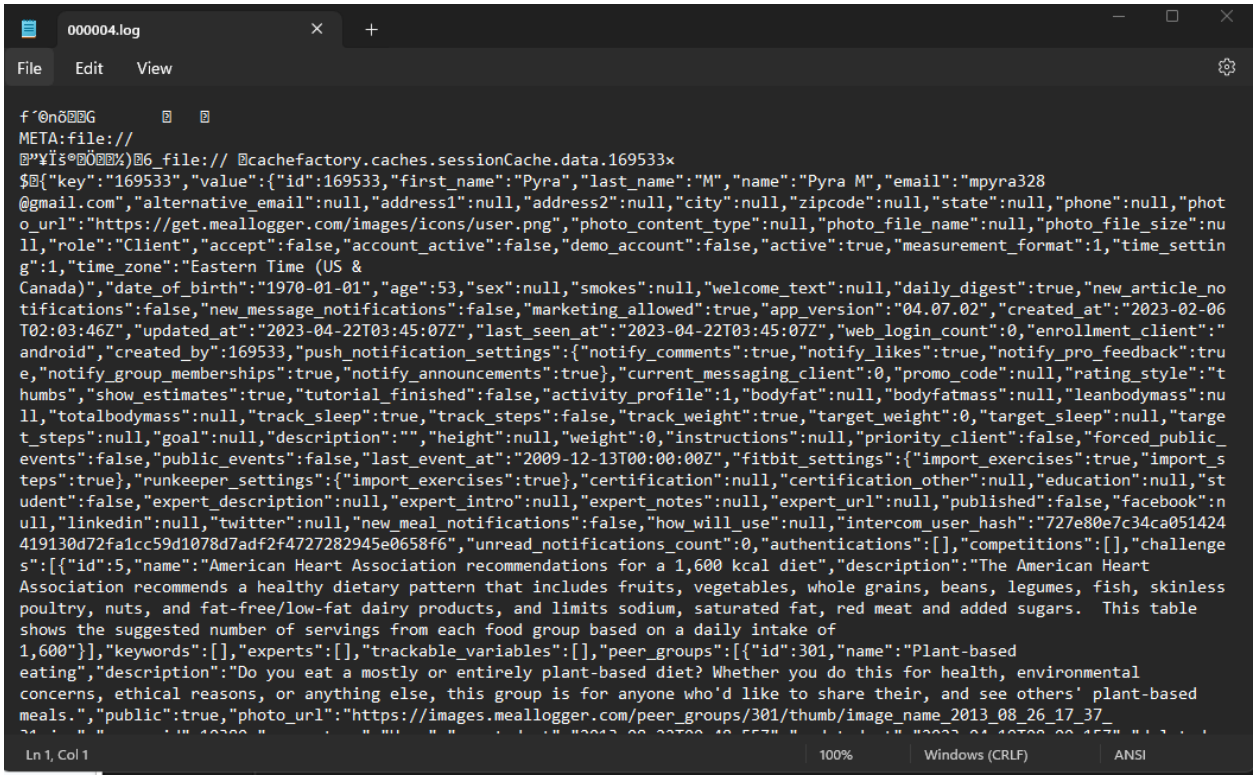


MSTG-STORAGE-8

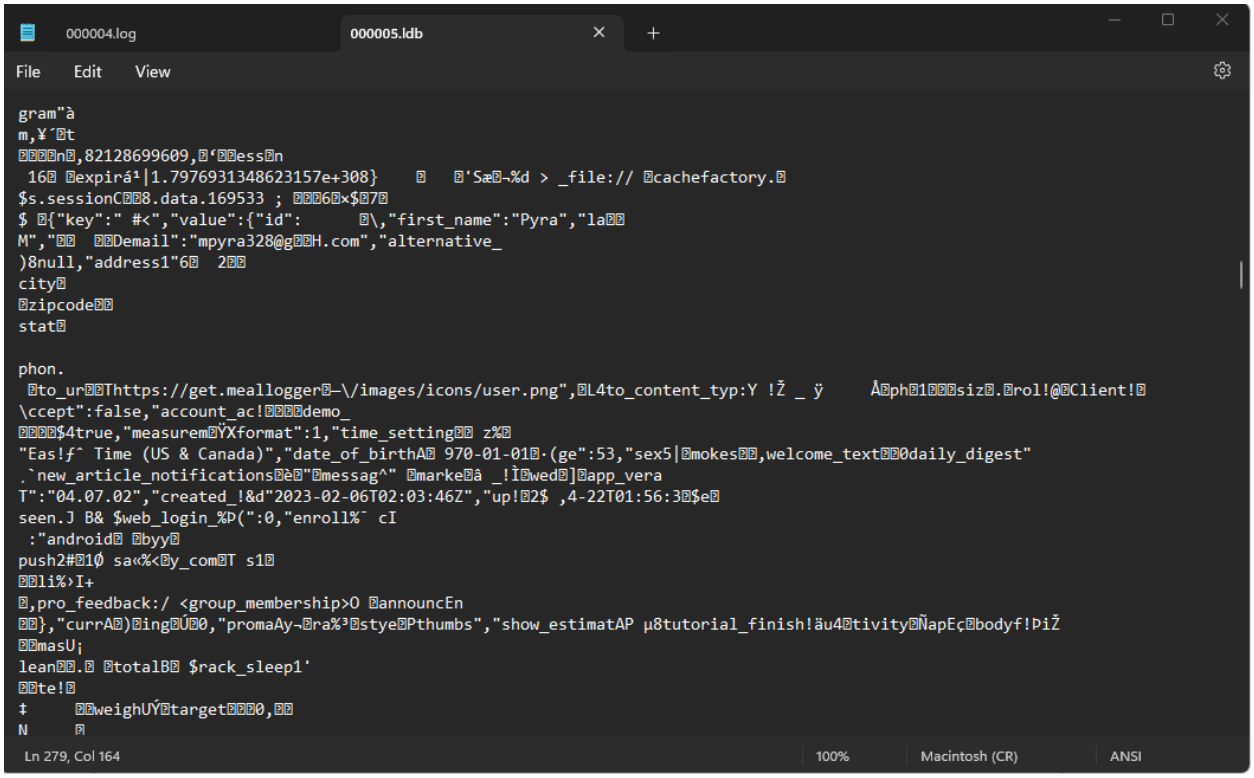
By performing dynamic app analysis, I was able to backup the data from my Android emulator. Essentially, I used the adb backup command, and unpacked the .ab file using the Android Backup Extractor (ABE) for Windows. From the unpacked files, I was able to find many similar results as found when investigating MSTG-STORAGE-1 and 2. The xml files were the same compared to the previous ones, but there were some new files under the Local Storage folder (shown below).



The “0000004.log” file (shown below), was quite similar to the log file found when investigating MSTG-STORAGE-3, containing sensitive information like name, email, DOB, address, and much more.

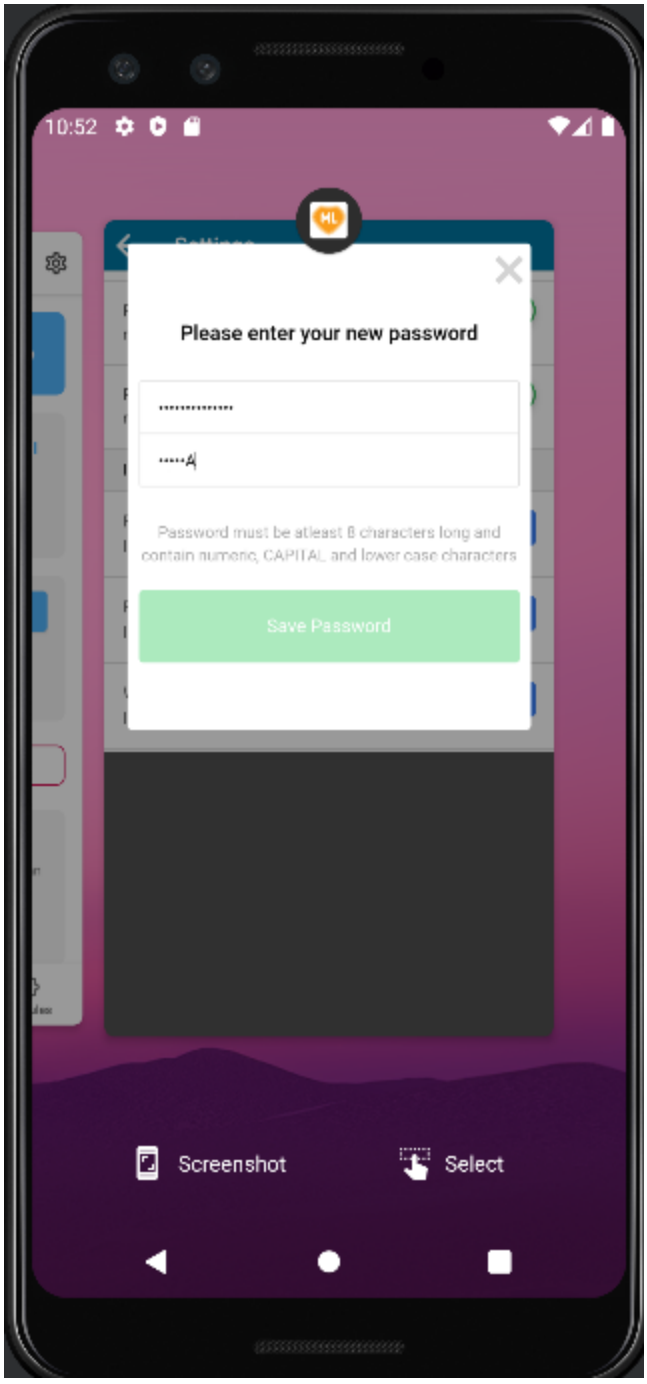


The other notable file was the “000005.ldb” file (shown below). Although most of it was unreadable, the parts that were readable showed similar sensitive information, like email and address. Future investigation could further demonstrate this exposure.



MSTG-STORAGE-9

For static testing purposes, searching for the method “applicationDidEnterBackground” within jadx did not yield any results. This is further supported by the findings of the dynamic testing I performed. By typing in sensitive information, such as by changing the password, we can see that the snapshot taken in the app-switcher screen still contains this information (shown below).


10:52
Please enter your new password
Password must be atleast 8 characters long and contain numeric, CAPITAL and lower case characters
Save Password
Screenshot
Select