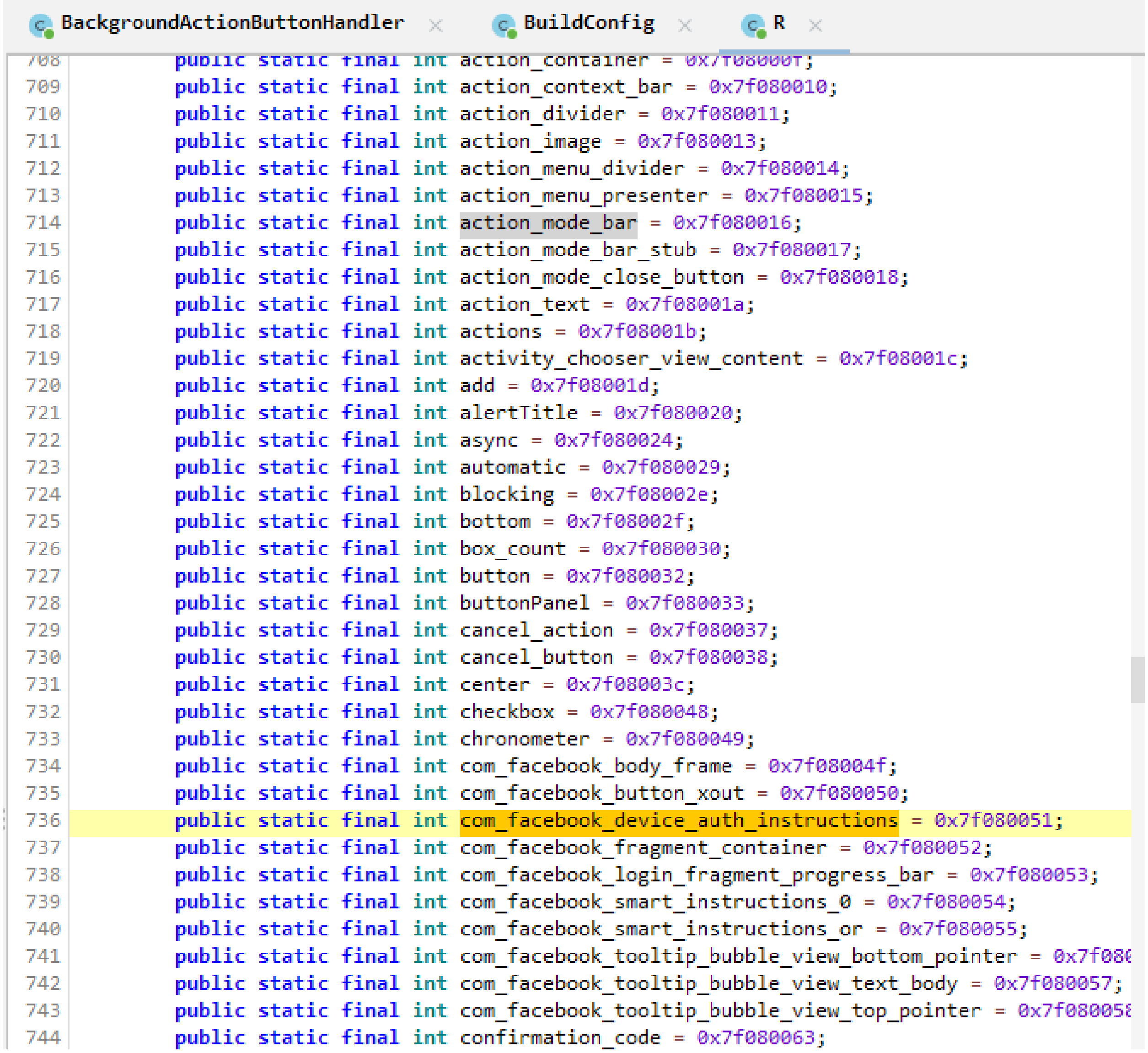
Chad Josim

CS 453

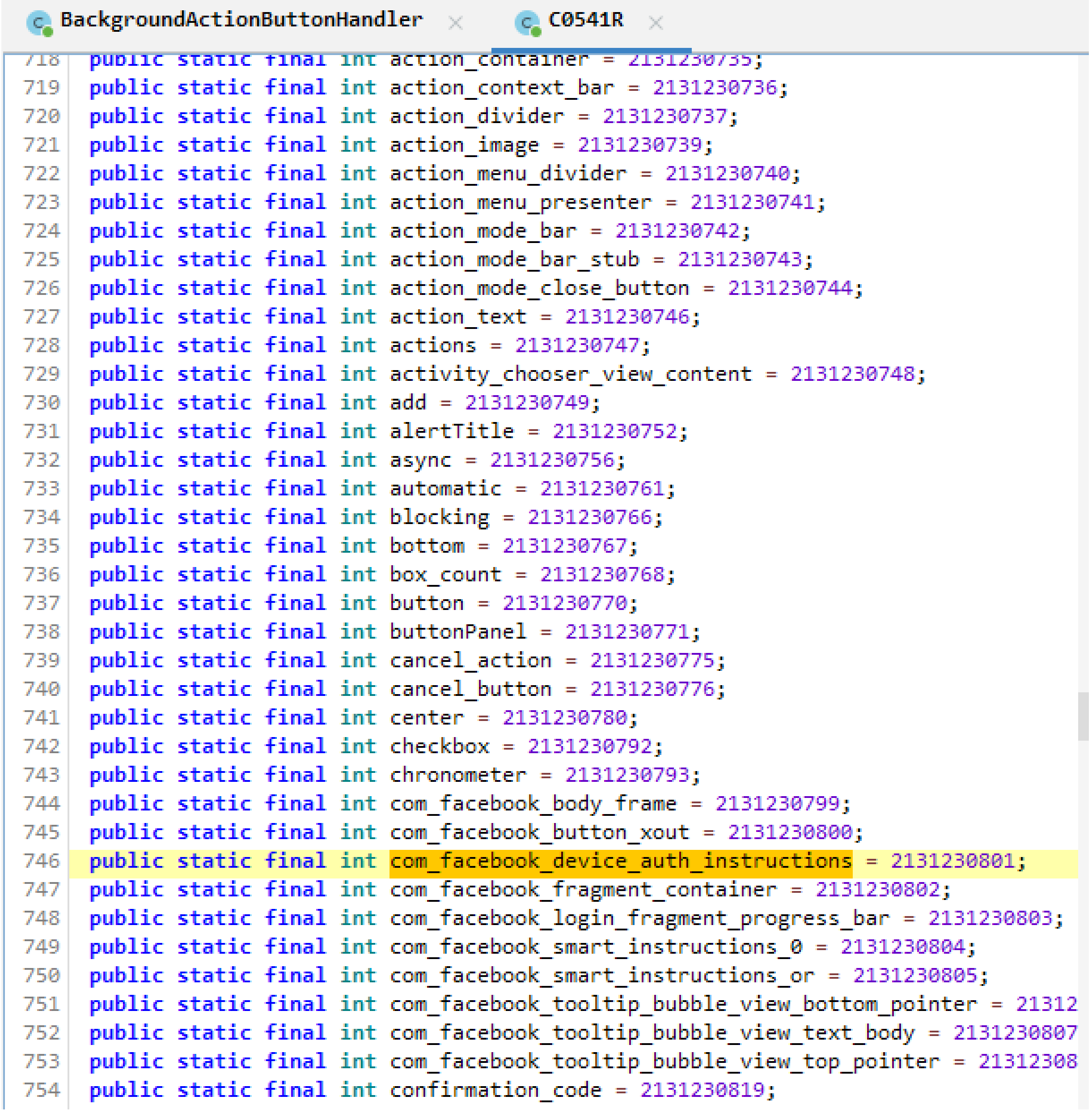
Module 9 Assignment

Obfuscation

The Meallogger Android application that I am investigating had several instances of code obfuscation. By using jadx, I was able to identify these occurrences by comparing the original and de-obfuscated versions of the source code. Recorded below are important/significant occurrences of such obfuscation:



```
        public static final int action_container = 0x7f08000f;
        public static final int action_context_bar = 0x7f080010;
        public static final int action_divider = 0x7f080011;
        public static final int action_image = 0x7f080013;
        public static final int action_menu_divider = 0x7f080014;
        public static final int action_menu_presenter = 0x7f080015;
        public static final int action_mode_bar = 0x7f080016;
        public static final int action_mode_bar_stub = 0x7f080017;
        public static final int action_mode_close_button = 0x7f080018;
        public static final int action_text = 0x7f08001a;
        public static final int actions = 0x7f08001b;
        public static final int activity_chooser_view_content = 0x7f08001c;
        public static final int add = 0x7f08001d;
        public static final int alertTitle = 0x7f080020;
        public static final int async = 0x7f080024;
        public static final int automatic = 0x7f080029;
        public static final int blocking = 0x7f08002e;
        public static final int bottom = 0x7f08002f;
        public static final int box_count = 0x7f080030;
        public static final int button = 0x7f080032;
        public static final int buttonPanel = 0x7f080033;
        public static final int cancel_action = 0x7f080037;
        public static final int cancel_button = 0x7f080038;
        public static final int center = 0x7f08003c;
        public static final int checkbox = 0x7f080048;
        public static final int chronometer = 0x7f080049;
        public static final int com_facebook_body_frame = 0x7f08004f;
        public static final int com_facebook_button_xout = 0x7f080050;
        public static final int com_facebook_device_auth_instructions = 0x7f080051;
        public static final int com_facebook_fragment_container = 0x7f080052;
        public static final int com_facebook_login_fragment_progress_bar = 0x7f080053;
        public static final int com_facebook_smart_instructions_0 = 0x7f080054;
        public static final int com_facebook_smart_instructions_or = 0x7f080055;
        public static final int com_facebook_tooltip_bubble_view_bottom_pointer = 0x7f080
        public static final int com_facebook_tooltip_bubble_view_text_body = 0x7f080057;
        public static final int com_facebook_tooltip_bubble_view_top_pointer = 0x7f08005
        public static final int confirmation_code = 0x7f080063;
```



```
public static final int action_container = 2131230735;
public static final int action_context_bar = 2131230736;
public static final int action_divider = 2131230737;
public static final int action_image = 2131230739;
public static final int action_menu_divider = 2131230740;
public static final int action_menu_presenter = 2131230741;
public static final int action_mode_bar = 2131230742;
public static final int action_mode_bar_stub = 2131230743;
public static final int action_mode_close_button = 2131230744;
public static final int action_text = 2131230746;
public static final int actions = 2131230747;
public static final int activity_chooser_view_content = 2131230748;
public static final int add = 2131230749;
public static final int alertTitle = 2131230752;
public static final int async = 2131230756;
public static final int automatic = 2131230761;
public static final int blocking = 2131230766;
public static final int bottom = 2131230767;
public static final int box_count = 2131230768;
public static final int button = 2131230770;
public static final int buttonPanel = 2131230771;
public static final int cancel_action = 2131230775;
public static final int cancel_button = 2131230776;
public static final int center = 2131230780;
public static final int checkbox = 2131230792;
public static final int chronometer = 2131230793;
public static final int com_facebook_body_frame = 2131230799;
public static final int com_facebook_button_xout = 2131230800;
public static final int com_facebook_device_auth_instructions = 2131230801;
public static final int com_facebook_fragment_container = 2131230802;
public static final int com_facebook_login_fragment_progress_bar = 2131230803;
public static final int com_facebook_smart_instructions_0 = 2131230804;
public static final int com_facebook_smart_instructions_or = 2131230805;
public static final int com_facebook_tooltip_bubble_view_bottom_pointer = 21312
public static final int com_facebook_tooltip_bubble_view_text_body = 2131230807
public static final int com_facebook_tooltip_bubble_view_top_pointer = 21312308
public static final int confirmation_code = 2131230819;
```

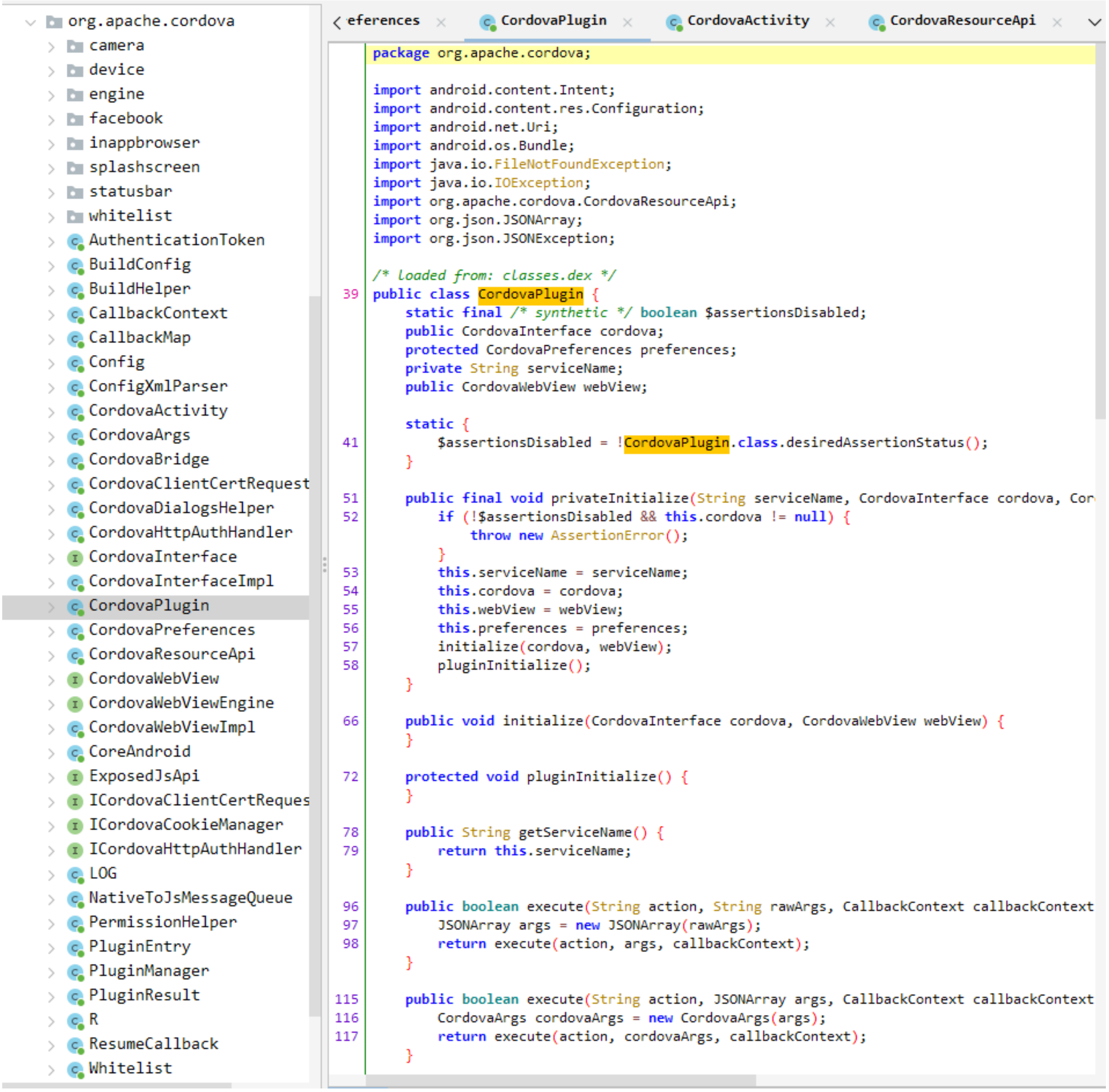
We can see that in the first screenshot, all of the values for each class variable are written in hexadecimal, indicating some amount of encryption. In the second screenshot, after de-obfuscating in jadx, we can see that the previously encrypted values are now shown with their actual values. There are several instances of this found among different "R" files containing the resource IDs for various Android assets. However, outside of this, there are no other obvious instances of obfuscation, such as string renaming or unreadable class names.

Looking more closely at the related MSTG Resilience requirements:

- **MSTG-RESILIENCE-9:** While obfuscation is applied in some capacity to parts of the source code, it is by no means consistent or effective, shown by jadx's ability to easily de-obfuscate these instances.
- **MSTG-RESILIENCE-11:** Similarly, the application's source code is encrypted to an extent, but by no means to a satisfactory degree. While many of the resource IDs are encrypted, many instances of important/sensitive code are easily viewable and accessible by the user, which will be demonstrated later in the document.
- **MSTG-RESILIENCE-12:** Although an obfuscation scheme was present, manual and automated de-obfuscation methods were highly effective against the application's decompiled source code. Overall, the obfuscation can be seen as relatively ineffective.

Type of App

By searching for keywords (such as apache, kotlin, js, javascript, etc.) in jadx, I was able to find that the Meallogger application uses apache, specifically cordova. Besides this, however, there were no other clues of a hybrid or third party app. Below is the screenshot showing the code relating to the apache (cordova) third party plugin:

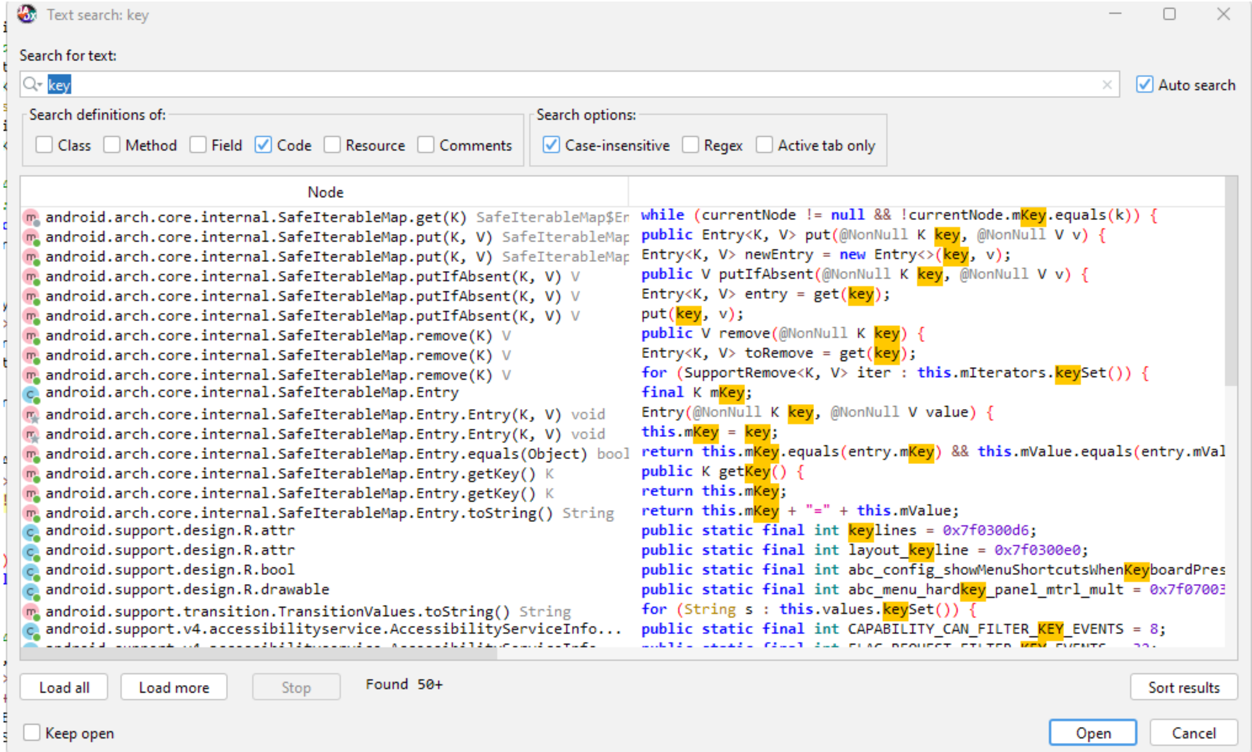


```
org.apache.cordova
  camera
  device
  engine
  facebook
  inappbrowser
  splashscreen
  statusbar
  whitelist
  AuthenticationToken
  BuildConfig
  BuildHelper
  CallbackContext
  CallbackMap
  Config
  ConfigXmlParser
  CordovaActivity
  CordovaArgs
  CordovaBridge
  CordovaClientCertRequest
  CordovaDialogsHelper
  CordovaHttpAuthHandler
  CordovaInterface
  CordovaInterfaceImpl
  CordovaPlugin
  CordovaPreferences
  CordovaResourceApi
  CordovaWebView
  CordovaWebViewEngine
  CordovaWebViewImpl
  CoreAndroid
  ExposedJsApi
  ICordovaClientCertReques
  ICordovaCookieManager
  ICordovaHttpAuthHandler
  LOG
  NativeToJsMessageQueue
  PermissionHelper
  PluginEntry
  PluginManager
  PluginResult
  R
  ResumeCallback
  Whitelist
```

```java
package org.apache.cordova;

import android.content.Intent;
import android.content.res.Configuration;
import android.net.Uri;
import android.os.Bundle;
import java.io.FileNotFoundException;
import java.io.IOException;
import org.apache.cordova.CordovaResourceApi;
import org.json.JSONArray;
import org.json.JSONException;

/* loaded from: classes.dex */
public class CordovaPlugin {
    static final /* synthetic */ boolean $assertionsDisabled;
    public CordovaInterface cordova;
    protected CordovaPreferences preferences;
    private String serviceName;
    public CordovaWebView webView;

    static {
        $assertionsDisabled = !CordovaPlugin.class.desiredAssertionStatus();
    }

    public final void privateInitialize(String serviceName, CordovaInterface cordova, Cor
        if (!$assertionsDisabled && this.cordova != null) {
            throw new AssertionError();
        }
        this.serviceName = serviceName;
        this.cordova = cordova;
        this.webView = webView;
        this.preferences = preferences;
        initialize(cordova, webView);
        pluginInitialize();
    }

    public void initialize(CordovaInterface cordova, CordovaWebView webView) {
    }

    protected void pluginInitialize() {
    }

    public String getServiceName() {
        return this.serviceName;
    }

    public boolean execute(String action, String rawArgs, CallbackContext callbackContext
        JSONArray args = new JSONArray(rawArgs);
        return execute(action, args, callbackContext);
    }

    public boolean execute(String action, JSONArray args, CallbackContext callbackContext
        CordovaArgs cordovaArgs = new CordovaArgs(args);
        return execute(action, cordovaArgs, callbackContext);
    }
```

Strings of Interest

By searching for a variety of keywords in jadx, I was able to find several instances of strings of interest, especially relating to permissions and secrets. Below are screenshots of the results with their corresponding search keyword:



Search for text:

key

☑ Auto search

Search definitions of: ☐ Class ☐ Method ☐ Field ☑ Code ☐ Resource ☐ Comments

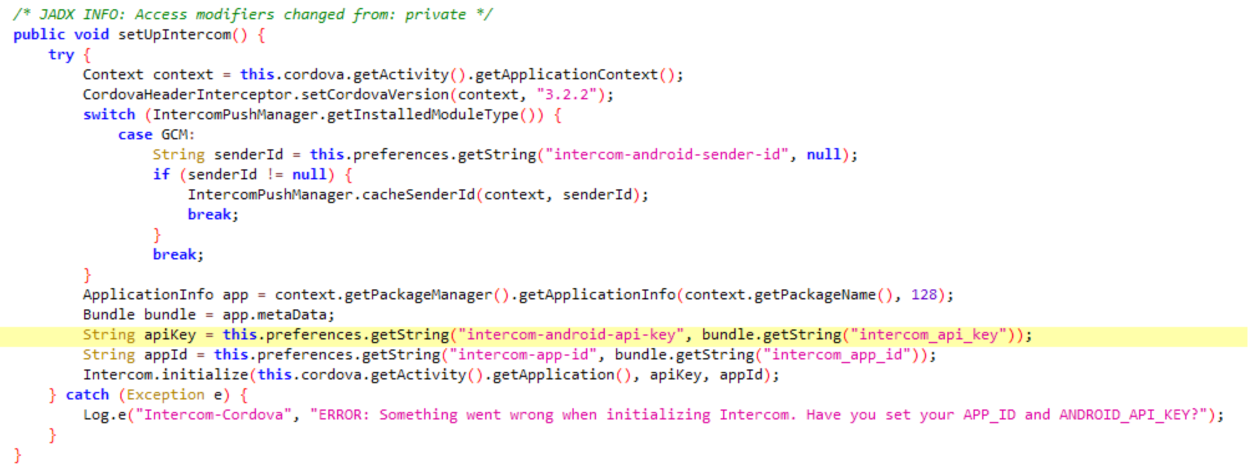
Search options: ☑ Case-insensitive ☐ Regex ☐ Active tab only

Node	
android.arch.core.internal.SafeIterableMap.get(K) SafeIterableMap$Er	while (currentNode != null && !currentNode.mKey.equals(k)) {
android.arch.core.internal.SafeIterableMap.put(K, V) SafeIterableMap	public Entry<K, V> put(@NonNull K key, @NonNull V v) {
android.arch.core.internal.SafeIterableMap.put(K, V) SafeIterableMap	Entry<K, V> newEntry = new Entry<>(key, v);
android.arch.core.internal.SafeIterableMap.putIfAbsent(K, V) V	public V putIfAbsent(@NonNull K key, @NonNull V v) {
android.arch.core.internal.SafeIterableMap.putIfAbsent(K, V) V	Entry<K, V> entry = get(key);
android.arch.core.internal.SafeIterableMap.putIfAbsent(K, V) V	put(key, v);
android.arch.core.internal.SafeIterableMap.remove(K) V	public V remove(@NonNull K key) {
android.arch.core.internal.SafeIterableMap.remove(K) V	Entry<K, V> toRemove = get(key);
android.arch.core.internal.SafeIterableMap.remove(K) V	for (SupportRemove<K, V> iter : this.mIterators.keySet()) {
android.arch.core.internal.SafeIterableMap.Entry	final K mKey;
android.arch.core.internal.SafeIterableMap.Entry.Entry(K, V) void	Entry(@NonNull K key, @NonNull V value) {
android.arch.core.internal.SafeIterableMap.Entry.Entry(K, V) void	this.mKey = key;
android.arch.core.internal.SafeIterableMap.Entry.equals(Object) bool	return this.mKey.equals(entry.mKey) && this.mValue.equals(entry.mVal
android.arch.core.internal.SafeIterableMap.Entry.getKey() K	public K getKey() {
android.arch.core.internal.SafeIterableMap.Entry.getKey() K	return this.mKey;
android.arch.core.internal.SafeIterableMap.Entry.toString() String	return this.mKey + "=" + this.mValue;
android.support.design.R.attr	public static final int keylines = 0x7f0300d6;
android.support.design.R.attr	public static final int layout_keyline = 0x7f0300e0;
android.support.design.R.bool	public static final int abc_config_showMenuShortcutsWhenKeyboardPres
android.support.design.R.drawable	public static final int abc_menu_hardkey_panel_mtrl_mult = 0x7f07003
android.support.transition.TransitionValues.toString() String	for (String s : this.values.keySet()) {
android.support.v4.accessibilityservice.AccessibilityServiceInfo...	public static final int CAPABILITY_CAN_FILTER_KEY_EVENTS = 8;

Found 50+

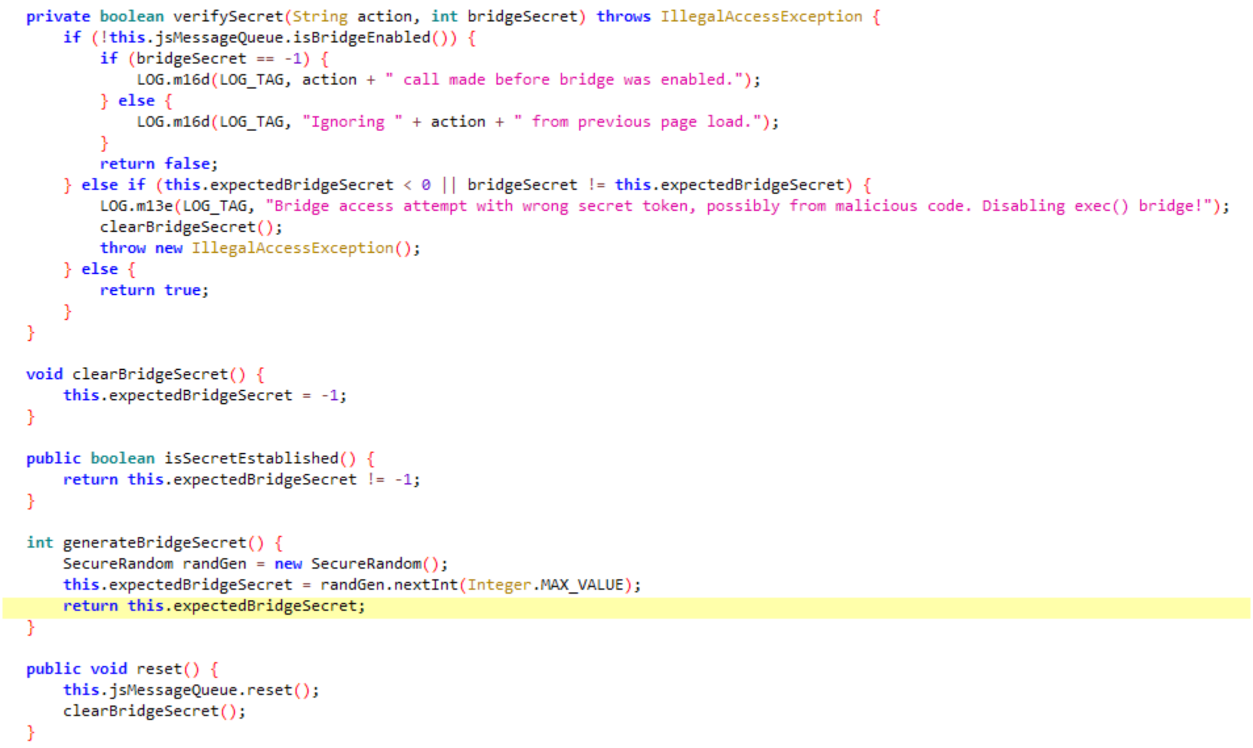
☐ Keep open

"key": There were no significant results for this keyword, as most references related to inner function calls as opposed to the hardcoding of any specific key. However, there were significant results for searching "api key," which is shown below.



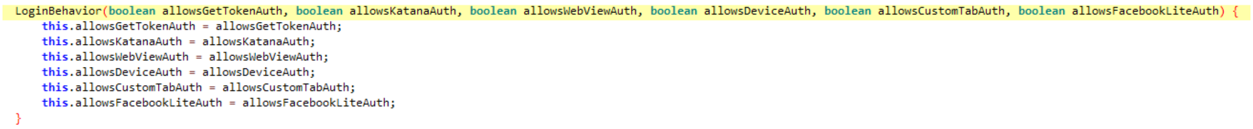
```
/* JADX INFO: Access modifiers changed from: private */
public void setUpIntercom() {
    try {
        Context context = this.cordova.getActivity().getApplicationContext();
        CordovaHeaderInterceptor.setCordovaVersion(context, "3.2.2");
        switch (IntercomPushManager.getInstalledModuleType()) {
            case GCM:
                String senderId = this.preferences.getString("intercom-android-sender-id", null);
                if (senderId != null) {
                    IntercomPushManager.cacheSenderId(context, senderId);
                    break;
                }
                break;
        }
        ApplicationInfo app = context.getPackageManager().getApplicationInfo(context.getPackageName(), 128);
        Bundle bundle = app.metaData;
        String apiKey = this.preferences.getString("intercom-android-api-key", bundle.getString("intercom_api_key"));
        String appId = this.preferences.getString("intercom-app-id", bundle.getString("intercom_app_id"));
        Intercom.initialize(this.cordova.getActivity().getApplication(), apiKey, appId);
    } catch (Exception e) {
        Log.e("Intercom-Cordova", "ERROR: Something went wrong when initializing Intercom. Have you set your APP_ID and ANDROID_API_KEY?");
    }
}
```

“api key”: Here, we can see that there is a hardcoded reference to an intercom api key. Although the key itself is not present, it is generally bad practice to have the api key be easily accessible within a public class.



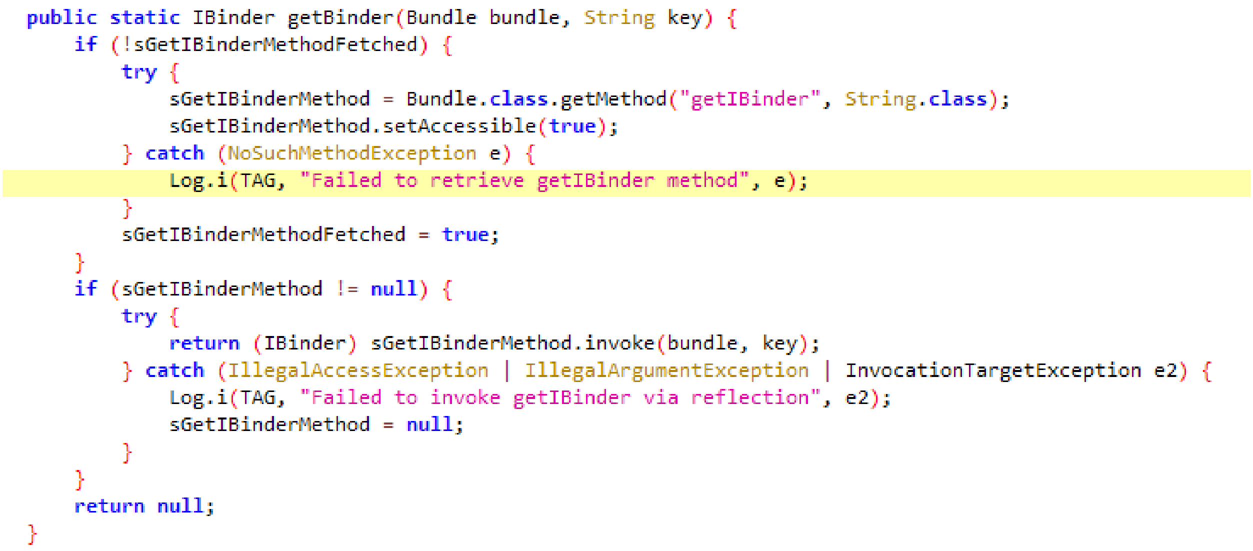
```
private boolean verifySecret(String action, int bridgeSecret) throws IllegalAccessException {
    if (!this.jsMessageQueue.isBridgeEnabled()) {
        if (bridgeSecret == -1) {
            LOG.m16d(LOG_TAG, action + " call made before bridge was enabled.");
        } else {
            LOG.m16d(LOG_TAG, "Ignoring " + action + " from previous page load.");
        }
        return false;
    } else if (this.expectedBridgeSecret < 0 || bridgeSecret != this.expectedBridgeSecret) {
        LOG.m13e(LOG_TAG, "Bridge access attempt with wrong secret token, possibly from malicious code. Disabling exec() bridge!");
        clearBridgeSecret();
        throw new IllegalAccessException();
    } else {
        return true;
    }
}

void clearBridgeSecret() {
    this.expectedBridgeSecret = -1;
}

public boolean isSecretEstablished() {
    return this.expectedBridgeSecret != -1;
}

int generateBridgeSecret() {
    SecureRandom randGen = new SecureRandom();
    this.expectedBridgeSecret = randGen.nextInt(Integer.MAX_VALUE);
    return this.expectedBridgeSecret;
}

public void reset() {
    this.jsMessageQueue.reset();
    clearBridgeSecret();
}
```

“secret”: There was a very significant result for this keyword, as we can see that a secret is generated using a random function. This is generally considered bad practice and highly unsafe; furthermore, this aligns with MobSF’s findings for the app.



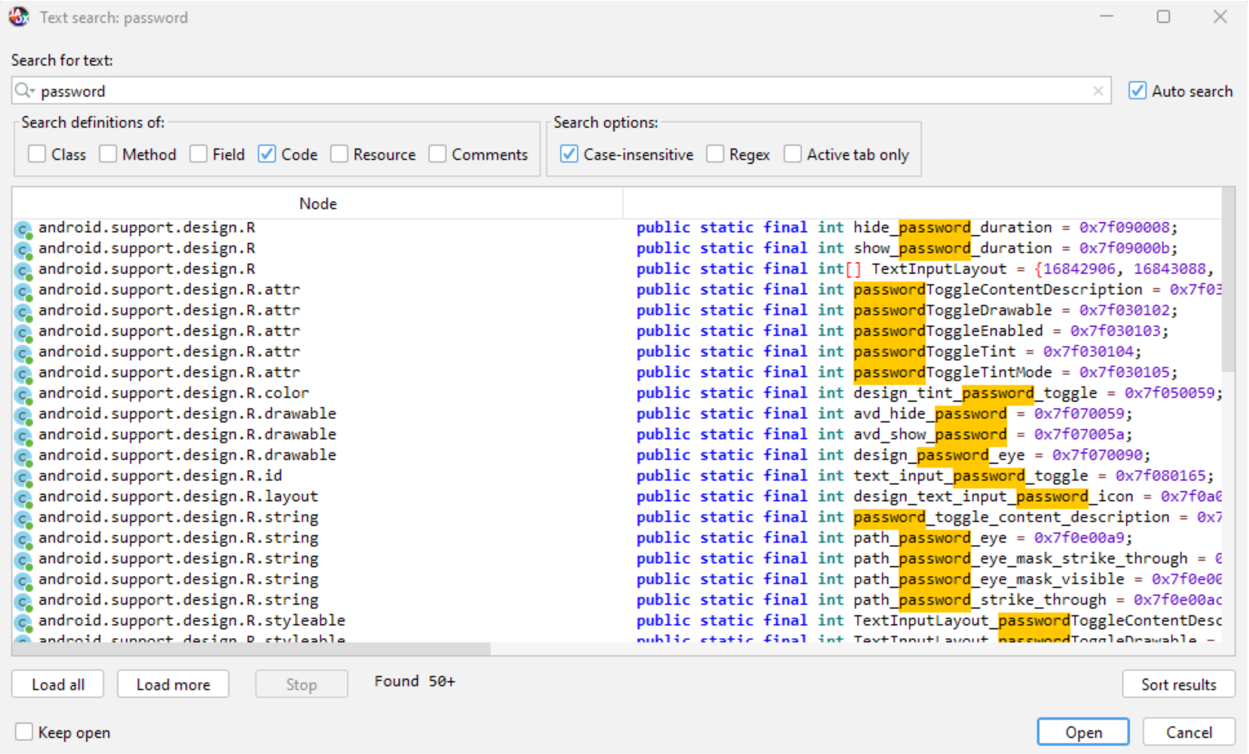
```
LoginBehavior(boolean allowsGetTokenAuth, boolean allowsKatanaAuth, boolean allowsWebViewAuth, boolean allowsDeviceAuth, boolean allowsCustomTabAuth, boolean allowsFacebookLiteAuth) {
    this.allowsGetTokenAuth = allowsGetTokenAuth;
    this.allowsKatanaAuth = allowsKatanaAuth;
    this.allowsWebViewAuth = allowsWebViewAuth;
    this.allowsDeviceAuth = allowsDeviceAuth;
    this.allowsCustomTabAuth = allowsCustomTabAuth;
    this.allowsFacebookLiteAuth = allowsFacebookLiteAuth;
}
```

“token”: For this keyword, we can see that there is a function that accesses tokens for several different web modes, including Katana, webview, and even Facebook. Although this is not necessarily an issue, it seems interesting that the login authentication tokens are easily accessible and locatable.



```java
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.



Search for text:

password

☑ Auto search

Search definitions of:

☐ Class ☐ Method ☐ Field ☑ Code ☐ Resource ☐ Comments

Search options:

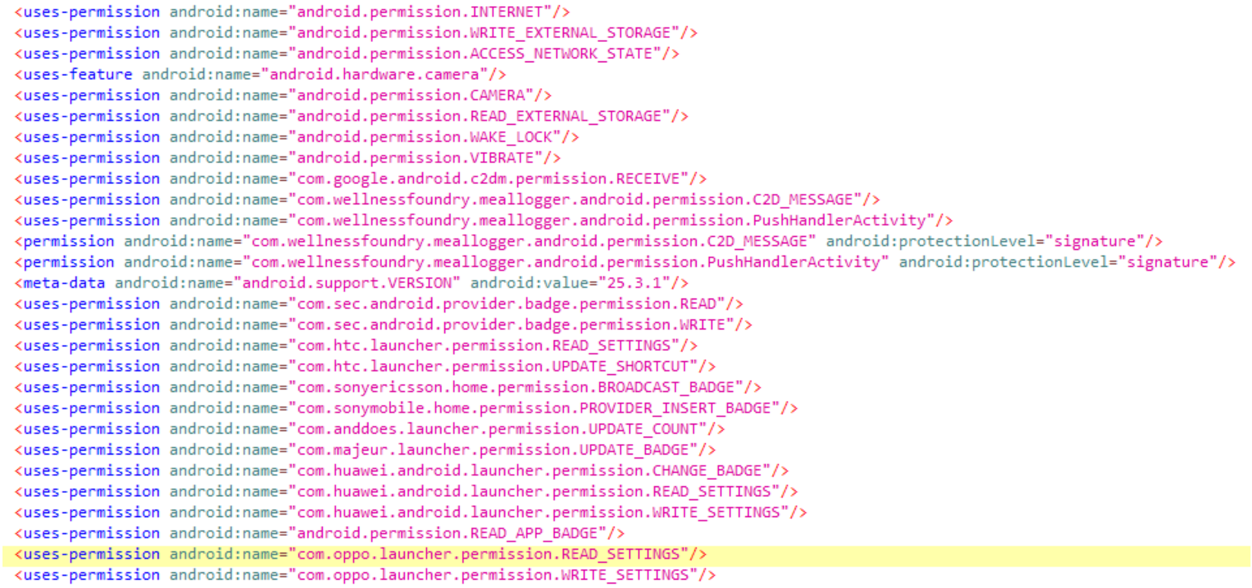
☑ Case-insensitive ☐ Regex ☐ Active tab only

Node	
android.support.design.R	public static final int hide_password_duration = 0x7f090008;
android.support.design.R	public static final int show_password_duration = 0x7f09000b;
android.support.design.R	public static final int[] TextInputLayout = {16842906, 16843088,
android.support.design.R.attr	public static final int passwordToggleContentDescription = 0x7f03
android.support.design.R.attr	public static final int passwordToggleDrawable = 0x7f030102;
android.support.design.R.attr	public static final int passwordToggleEnabled = 0x7f030103;
android.support.design.R.attr	public static final int passwordToggleTint = 0x7f030104;
android.support.design.R.attr	public static final int passwordToggleTintMode = 0x7f030105;
android.support.design.R.color	public static final int design_tint_password_toggle = 0x7f050059;
android.support.design.R.drawable	public static final int avd_hide_password = 0x7f070059;
android.support.design.R.drawable	public static final int avd_show_password = 0x7f07005a;
android.support.design.R.drawable	public static final int design_password_eye = 0x7f070090;
android.support.design.R.id	public static final int text_input_password_toggle = 0x7f080165;
android.support.design.R.layout	public static final int design_text_input_password_icon = 0x7f0a0
android.support.design.R.string	public static final int password_toggle_content_description = 0x7
android.support.design.R.string	public static final int path_password_eye = 0x7f0e00a9;
android.support.design.R.string	public static final int path_password_eye_mask_strike_through = 0
android.support.design.R.string	public static final int path_password_eye_mask_visible = 0x7f0e00
android.support.design.R.string	public static final int path_password_strike_through = 0x7f0e00ac
android.support.design.R.styleable	public static final int TextInputLayout_passwordToggleContentDesc
android.support.design.R.styleable	public static final int TextInputLayout_passwordToggleDrawable =

Found 50+

☐ Keep open

“password”: While there seems to be a number of results at first, most of the outputs for this keyword refer to the actual design and layout of the login page, rather than any hardcoded passwords.

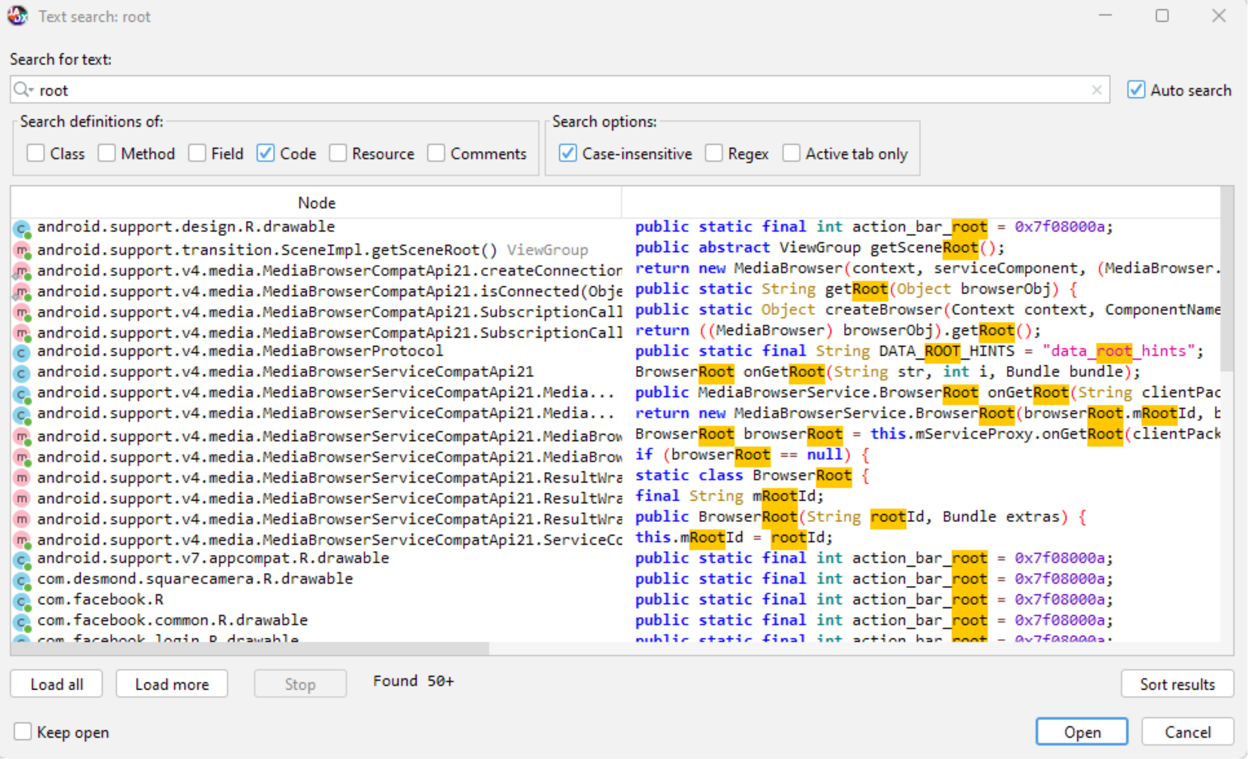


```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-feature android:name="android.hardware.camera"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<uses-permission android:name="com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE"/>
<uses-permission android:name="com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity"/>
<permission android:name="com.wellnessfoundry.meallogger.android.permission.C2D_MESSAGE" android:protectionLevel="signature"/>
<permission android:name="com.wellnessfoundry.meallogger.android.permission.PushHandlerActivity" android:protectionLevel="signature"/>
<meta-data android:name="android.support.VERSION" android:value="25.3.1"/>
<uses-permission android:name="com.sec.android.provider.badge.permission.READ"/>
<uses-permission android:name="com.sec.android.provider.badge.permission.WRITE"/>
<uses-permission android:name="com.htc.launcher.permission.READ_SETTINGS"/>
<uses-permission android:name="com.htc.launcher.permission.UPDATE_SHORTCUT"/>
<uses-permission android:name="com.sonyericsson.home.permission.BROADCAST_BADGE"/>
<uses-permission android:name="com.sonymobile.home.permission.PROVIDER_INSERT_BADGE"/>
<uses-permission android:name="com.anddoes.launcher.permission.UPDATE_COUNT"/>
<uses-permission android:name="com.majeur.launcher.permission.UPDATE_BADGE"/>
<uses-permission android:name="com.huawei.android.launcher.permission.CHANGE_BADGE"/>
<uses-permission android:name="com.huawei.android.launcher.permission.READ_SETTINGS"/>
<uses-permission android:name="com.huawei.android.launcher.permission.WRITE_SETTINGS"/>
<uses-permission android:name="android.permission.READ_APP_BADGE"/>
<uses-permission android:name="com.oppo.launcher.permission.READ_SETTINGS"/>
<uses-permission android:name="com.oppo.launcher.permission.WRITE_SETTINGS"/>
```

Looking at the AndroidManifest.xml file below, we can see all of the permissions associated with the Meallogger application. Interestingly, the app seems to be able to write to external storage, as well as read settings (like contacts). Furthermore, it is able to detect camera and application activity from the device. For obvious reasons, this is very unsafe, and such suspicions are confirmed by the MobSF report.

Root Detection

After investigation, the Meallogger application does not seem to have any implementation of root detection. Searching jadx for common root keywords, like superuser or /bin/su, did not return any search results. Searching for “root”, as shown below, also does not return any meaningful results, as most instances refer to the instantiation of a browser rather than a root user. In the same sense, due to the lack of related code, there is no apparent obfuscation for root detection.



Text search: root

Search for text: root

☑ Auto search

Search definitions of: □ Class □ Method □ Field ☑ Code □ Resource □ Comments

Search options: ☑ Case-insensitive □ Regex □ Active tab only

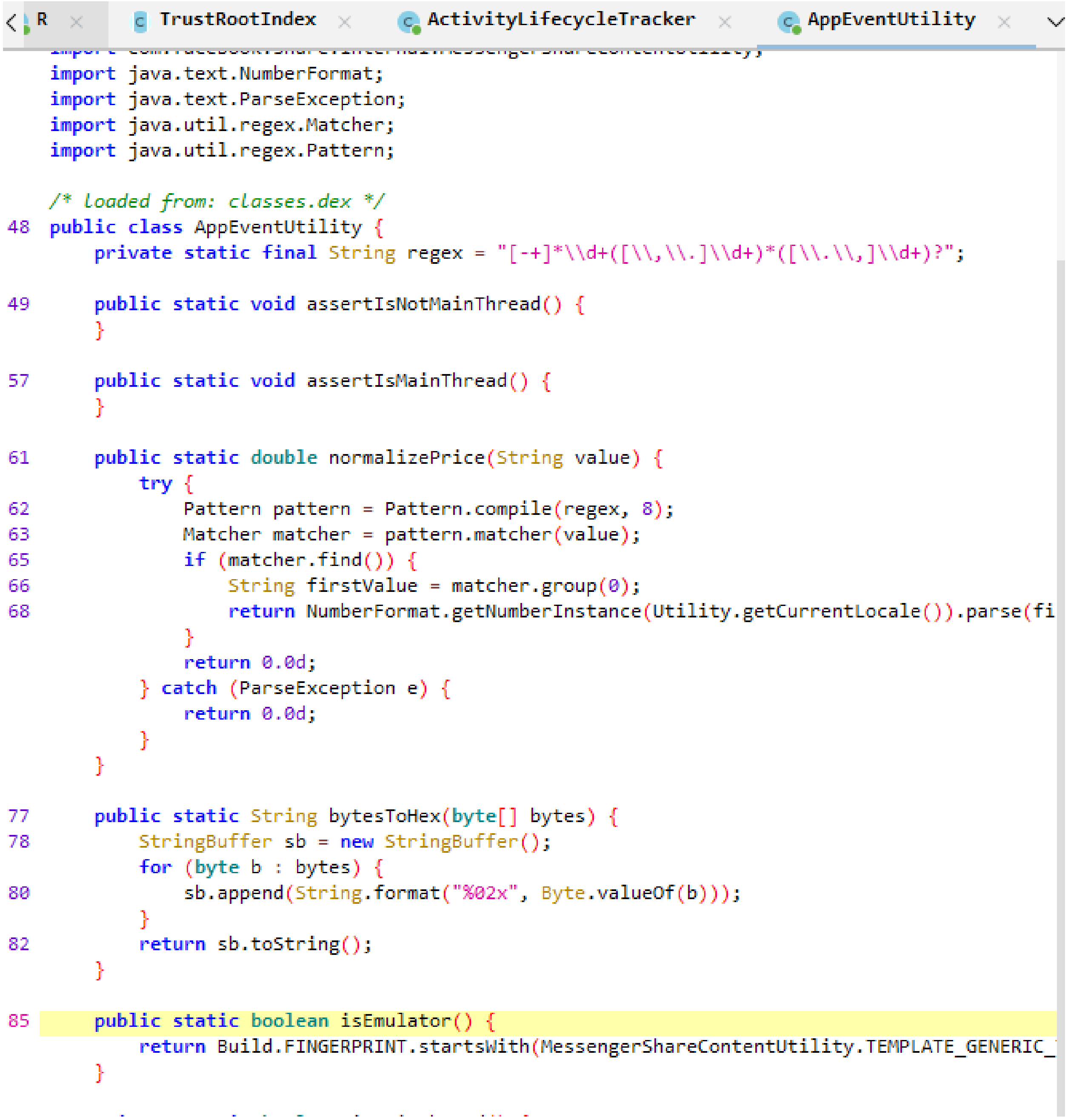
Node	
android.support.design.R.drawable	`public static final int action_bar_root = 0x7f08000a;`
android.support.transition.SceneImpl.getSceneRoot() ViewGroup	`public abstract ViewGroup getSceneRoot();`
android.support.v4.media.MediaBrowserCompatApi21.createConnection	`return new MediaBrowser(context, serviceComponent, (MediaBrowser.`
android.support.v4.media.MediaBrowserCompatApi21.isConnected(Obje	`public static String getRoot(Object browserObj) {`
android.support.v4.media.MediaBrowserCompatApi21.SubscriptionCall	`public static Object createBrowser(Context context, ComponentName`
android.support.v4.media.MediaBrowserCompatApi21.SubscriptionCall	`return ((MediaBrowser) browserObj).getRoot();`
android.support.v4.media.MediaBrowserProtocol	`public static final String DATA_ROOT_HINTS = "data_root_hints";`
android.support.v4.media.MediaBrowserServiceCompatApi21	`BrowserRoot onGetRoot(String str, int i, Bundle bundle);`
android.support.v4.media.MediaBrowserServiceCompatApi21.Media...	`public MediaBrowserService.BrowserRoot onGetRoot(String clientPac`
android.support.v4.media.MediaBrowserServiceCompatApi21.Media...	`return new MediaBrowserService.BrowserRoot(browserRoot.mRootId, b`
android.support.v4.media.MediaBrowserServiceCompatApi21.MediaBrow	`BrowserRoot browserRoot = this.mServiceProxy.onGetRoot(clientPack`
android.support.v4.media.MediaBrowserServiceCompatApi21.MediaBrow	`if (browserRoot == null) {`
android.support.v4.media.MediaBrowserServiceCompatApi21.ResultWra	`static class BrowserRoot {`
android.support.v4.media.MediaBrowserServiceCompatApi21.ResultWra	`final String mRootId;`
android.support.v4.media.MediaBrowserServiceCompatApi21.ResultWra	`public BrowserRoot(String rootId, Bundle extras) {`
android.support.v4.media.MediaBrowserServiceCompatApi21.ServiceCc	`this.mRootId = rootId;`
android.support.v7.appcompat.R.drawable	`public static final int action_bar_root = 0x7f08000a;`
com.desmond.squarecamera.R.drawable	`public static final int action_bar_root = 0x7f08000a;`
com.facebook.R	`public static final int action_bar_root = 0x7f08000a;`
com.facebook.common.R.drawable	`public static final int action_bar_root = 0x7f08000a;`
com.facebook.login.R.drawable	`public static final int action_bar_root = 0x7f08000a;`

Found 50+

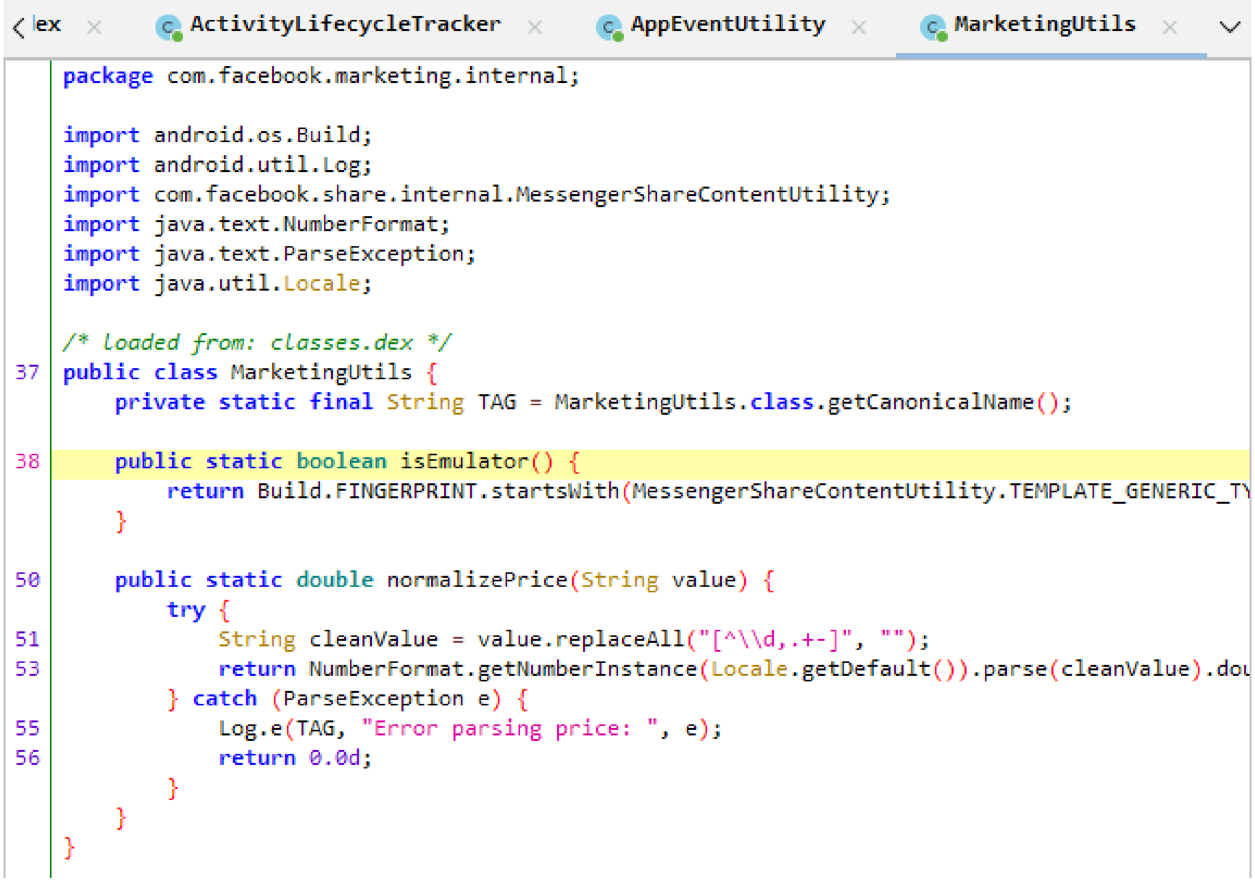
□ Keep open

Similarly, opening the Meallogger application in a rooted virtual device on Android Studio did not yield any significant results. The application works exactly as intended, and there were no popups or limitations in rooted mode.

Emulator/Fingerprinting



```
import com.facebook.share.internal.MessengerShareContentUtility;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/* loaded from: classes.dex */
public class AppEventUtility {
    private static final String regex = "[-+]*\\d+([\\,\\.]\\d+)*([\\.\\,]\\d+)?";

    public static void assertIsNotMainThread() {
    }

    public static void assertIsMainThread() {
    }

    public static double normalizePrice(String value) {
        try {
            Pattern pattern = Pattern.compile(regex, 8);
            Matcher matcher = pattern.matcher(value);
            if (matcher.find()) {
                String firstValue = matcher.group(0);
                return NumberFormat.getNumberInstance(Utility.getCurrentLocale()).parse(fi
            }
            return 0.0d;
        } catch (ParseException e) {
            return 0.0d;
        }
    }

    public static String bytesToHex(byte[] bytes) {
        StringBuffer sb = new StringBuffer();
        for (byte b : bytes) {
            sb.append(String.format("%02x", Byte.valueOf(b)));
        }
        return sb.toString();
    }

    public static boolean isEmulator() {
        return Build.FINGERPRINT.startsWith(MessengerShareContentUtility.TEMPLATE_GENERIC_
    }
```



```
package com.facebook.marketing.internal;

import android.os.Build;
import android.util.Log;
import com.facebook.share.internal.MessengerShareContentUtility;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

/* loaded from: classes.dex */
public class MarketingUtils {
    private static final String TAG = MarketingUtils.class.getCanonicalName();

    public static boolean isEmulator() {
        return Build.FINGERPRINT.startsWith(MessengerShareContentUtility.TEMPLATE_GENERIC_TY
    }

    public static double normalizePrice(String value) {
        try {
            String cleanValue = value.replaceAll("[^\\d,.+-]", "");
            return NumberFormat.getNumberInstance(Locale.getDefault()).parse(cleanValue).dou
        } catch (ParseException e) {
            Log.e(TAG, "Error parsing price: ", e);
            return 0.0d;
        }
    }
}
```

“return” statement:

return
Build.FINGERPRINT.startsWith(MessengerShareContentUtility.TEMPLATE_GENERIC_TYPE) || Build.FINGERPRINT.startsWith("unknown") || Build.MODEL.contains("google_sdk") || Build.MODEL.contains("Emulator") || Build.MODEL.contains("Android SDK built for x86") || Build.MANUFACTURER.contains("Genymotion") || (Build.BRAND.startsWith(MessengerShareContentUtility.TEMPLATE_GENERIC_TYPE) && Build.DEVICE.startsWith(MessengerShareContentUtility.TEMPLATE_GENERIC_TYPE)) || "google_sdk".equals(Build.PRODUCT);

Looking at the screenshots above, we can see that the application clearly has some form of emulator detection, and it also has the ability to fingerprint the device. Closely analyzing the return statement, it is apparent that the application recognizes all of the significant flags (fingerprint, model, manufacturer, brand, etc.) related to device fingerprinting. However, it is important to note that the application does not have any popups or differing functionality when run in an emulator, as opposed to a normal device.

Looking more closely at the related MSTG Resilience requirements:

- **MSTG-RESILIENCE-5:** While the application certainly detects being run in an emulator, it does not necessarily respond in any unique way apparent to the user. It is possible that discrete background processes are affected by the emulator, but such activity is not easily discernible.
- **MSTG-RESILIENCE-10:** The application follows this standard, implementing device fingerprinting derived from multiple unique properties. As seen by the screenshots above, the application notes each of the significant flags related to fingerprinting (model, manufacturer, brand, etc.)