SECTRIO

MALWARE REPORT

Android: SpyAgent Malware

Date: 10 /05 /2022

K. Narahari

Overview:

Spy Agent is a malware capable of performing a number of malicious actions, such as keylogging, sending and receiving SMS messages, stealing contacts, making calls, requesting root access, etc. It also poses a significant threat to your privacy and device safety.
This malware performs the command executions on the android device shell. It gathers device hardware and software information which includes type of processor, memory information, storage, battery, IMEI, and network information.

File Hash: c9606e68f6bc314e268b8645e57c9716

Technical Analysis:

We downloaded and reverse-engineered the malware sample, performed static analysis on the .apk file and went through the manifest.xml, which is the primary file to begin the analysis.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="com.google.android.gms.permission.ACTIVITY_RECOGNITION" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.QUERY_ALL_PACKAGES" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE" />
```

Figure 1 - Permissions used in the Malicious Application

Initially, we started our analysis with the basic step of analysing the manifest file to check for the security permissions given to the application. We observed that the application uses more permissions than the actual functional requirements of the application. Few of the permissions given to the app seems to be malicious and those permissions are used to perform malicious activities on the victim device.

Also, this application has many permissions and services to access:

- contact details can be accessible
- phone details like dialled, received, and missed calls information
- Wi-Fi connectivity information
- hardware information like IMEI, processor, cpu, memory
- network information like gps, network provider
- Bluetooth and many other permissions

```
if (f6356a == null) {
    f6356a = new HashMap();
    String str2 = "/system/bin/sh";
    if (!new File(str2).exists() || !new File(str2).canExecute()) {
        str2 = "sh";
    }
```

Figure 2 - Shell Command Execution

The malicious application is executing the shell from the code "`/system/bin/shell`". Accessing shell is a critical vulnerability because using the shell, the attacker will have the complete access of the android device. With this application, the attacker gains complete control of the device and can perform various malicious operations.

```
private static final String CPU_LOCATION = "/sys/devices/system/cpu/";
private static final String CPU_NAME_REGEX = "cpu[0-9]+";
private static final String TAG = "GlideRuntimeCompat";

private RuntimeCompat() {
}

static int availableProcessors() {
    int availableProcessors = Runtime.getRuntime().availableProcessors();
    return Build.VERSION.SDK_INT < 17 ? Math.max(getCoreCountPre17(), availableProcessors) : availableProcessors;
}
```

Figure 3 - Gathering CPU Information

Using the command execution, the app is gathering the cpu information and the number of processors used by the victim device.

```
public static boolean S() {
    String[] strArr = {"/system/xbin/", "/system/bin/", "/system/sbin/", "/sbin/", "/vendor/bin/", "/su/bin/"};
    for (int i = 0; i < 6; i++) {
        try {
            String str = strArr[i] + "su";
            if (new File(str).exists()) {
                String d2 = d(new String[]{"ls", "-l", str});
                a.a("isRooted=%s", d2);
                if (TextUtils.isEmpty(d2) || d2.indexOf("root") == d2.lastIndexOf("root")) {
                    return false;
                }
```

Figure 4 - Root Checking

Root checking has been performed by the app to know the device privileges for further post exploitation and the different system file paths also have been declared in an array.

```
public static JSONObject h() {
    JSONObject jSONObject = new JSONObject();
    try {
        jSONObject.put("brand", Build.BRAND);
        jSONObject.put("ABI", Build.CPU_ABI);
        jSONObject.put("CPU", Build.HARDWARE);
        jSONObject.put("display", Build.DISPLAY);
        jSONObject.put("manufacturer", Build.MANUFACTURER);
        jSONObject.put("model", Build.MODEL);
        jSONObject.put("androidVersion", Build.VERSION.SDK_INT);
    } catch (JSONException unused) {
    }
```

Figure 5 - Gathering Device Information -1

```
this.valueOf.put("brand", str);
this.valueOf.put("model", str2);
this.valueOf.put("platform", "Android");
this.valueOf.put("platform_version", str3);
if (str4 != null && str4.length() > 0) {
    this.valueOf.put("advertiserId", str4);
}
if (str5 != null && str5.length() > 0) {
    this.valueOf.put("imei", str5);
}
if (str6 != null && str6.length() > 0) {
    this.valueOf.put("android_id", str6);
}
```

Figure 6 - Gathering Device Information -2

From the Figure 5 and 6, the app is gathering the following information :

- Device information about the type of the device
- Device vendor
- Model of the victim device
- Platform and its version,
- IMEI number
- Hardware components information like cpu, memory and other information.

```
private static String d() throws Exception {
    FileInputStream fileInputStream = new FileInputStream(new File("/sys/class/net/wlan0/address"));
    String c2 = c(fileInputStream);
    fileInputStream.close();
    return c2;
}
```

Figure 7 - Gathering MAC Address of the Network Interface

From the Figure 7, the app is gathering the MAC address of the wireless interface. In general, out of 6 octets the first 3 octets give the information about the vendor/manufacturer of the hardware.

```
private void a(Context context) {
    SensorManager sensorManager = (SensorManager) context.getSystemService("sensor");
    this.f46a = sensorManager;
    Sensor defaultSensor = sensorManager.getDefaultSensor(1);
```

Figure 8 - Gathering the sensors information used by the device

The malicious app is gathering information about the different sensors used by the victim mobile device.

```
public static String y(Context context) {
    TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService("phone");
    if (telephonyManager != null) {
        return telephonyManager.getNetworkOperatorName();
    }
    return null;
}
```

Figure 9 - Gathering Network Operator Name

```
private static boolean R(Context context) {
    TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService("phone");
    if (telephonyManager == null || telephonyManager.getPhoneType() == 0) {
        return false;
    }
```

Figure 10 - Gathering the phone type

```
if (a.b(context, "android.permission.ACCESS_NETWORK_STATE")) {
    try {
        NetworkInfo activeNetworkInfo = ((ConnectivityManager) context.getSystemService("connectivity")).getActiveNetworkInfo();
        if (activeNetworkInfo == null) {
            return "no_network";
        }
```

Figure 11 - Network Connectivity Information

From Figure-9, 10, and 11 we can observe that using the telephony manager and invoking the phone system service the app is able to gather the network information and the name of the network provider. Also, from figure-10 we can see that it is gathering the type of the device.

```
WifiManager wifiManager;
WifiInfo connectionInfo;
if (!(context == null || !a.b(context, "android.permission.ACCESS_WIFI_STATE") || (wifiManager = (WifiManager) context.getApplicationContext().getSystemService("wifi"))
    String macAddress = connectionInfo.getMacAddress();
```

Figure 12 - Wi-Fi connection information -1

```
if (!h.b(this.f4528f) || (wifiManager = (WifiManager) this.f4528f.getSystemService("wifi")) == null) {
    str = null;
} else {
    WifiInfo connectionInfo = wifiManager.getConnectionInfo();
    str2 = connectionInfo.getBSSID();
    str = connectionInfo.getSSID();
}
```

Figure 13 - Wi-Fi connection Information -2

Figure 12 and 13 clearly shows that the malicious app is gathering the Wi-Fi device information. The gathered information includes the name (SSID) of the router, MAC address of the router.

```
private Location b() {
    Location location = null;
    Location c2 = b.b(this.f333b, "android.permission.ACCESS_COARSE_LOCATION") == 0 ? c("network") : null;
    if (b.b(this.f333b, "android.permission.ACCESS_FINE_LOCATION") == 0) {
        location = c("gps");
    }
```

Figure 14 - Gathering Location information with GPS

```
public static final Parcelable.Creator<zzbe> CREATOR = new zzbf();
@SafeParcelable.Field(getter = "getRequestId", id = 1)
private final String zza;
@SafeParcelable.Field(getter = "getExpirationTime", id = 2)
private final long zzb;
@SafeParcelable.Field(getter = "getType", id = 3)
private final short zzc;
@SafeParcelable.Field(getter = "getLatitude", id = 4)
private final double zzd;
@SafeParcelable.Field(getter = "getLongitude", id = 5)
private final double zze;
@SafeParcelable.Field(getter = "getRadius", id = 6)
private final float zzf;
@SafeParcelable.Field(getter = "getTransitionTypes", id = 7)
private final int zzg;
@SafeParcelable.Field(defaultValue = "0", getter = "getNotificationResponsiveness", id = 8)
private final int zzh;
@SafeParcelable.Field(defaultValue = "-1", getter = "getLoiteringDelay", id = 9)
private final int zzi;
```

Figure 15 - Gathering exact location using latitude and longitude values

SpyAgent accesses the exact location of the victim device by the location permissions declared and the exact latitude and longitude of the device is fetched and tracked by the application.

“Access coarse location” sources, such as the mobile network database, to determine an approximate phone location, where available. Malicious applications can use this to determine approximately where you are. “Access fine location” sources, such as the Global Positioning System on the phone, where available. Malicious applications can use this to determine where you are and may consume additional battery power.

```
o.g(UserBasicParam.class.getSimpleName(), this.k);
if (!com.permissionx.guolindev.b.b(this.h, android.permission.READ_CONTACTS')) {
    c.c.a.a.a.j().m(new c.c.a.a.d.b(PageEmum.PERMISSON, EventEmum.P_CONTACT));
}
```

Figure 16 - Gathering Contacts -1

```
public void IO(int i2) {
    Intent intent = new Intent();
    intent.setAction("android.intent.action.PICK");
    intent.setData(Uri.parse("content://com.android.contacts/contacts"));
    if (intent.resolveActivity(this.f5594e.getPackageManager()) != null) {
        startActivityForResult(intent, i2);
    }
}
```

Figure 17 - Gathering Contacts -2

From the above figure 16 and 17, allows the application to read all of the contact (address) data stored on your phone and this can be used to send your contacts data to other people.

IOC URL's:

http://phl-app.hcloudai.com
http://phl-attach.hcloudai.com
http://phl-app.hcloudai.com/hc/app/noAuth/param/submitBuryingPoint
http://phl-app.hcloudai.com/preview/philippines/philippines_privacy_policy.html
http://localhost/
https://%sonelink.%s/shortlink.sdk/v1
https://astat.bugly.cros.wr.pvp.net/:8180/rqd/async
https://%sgcdsdk.%s/install_data/v4.0/
https://%monitorsdk.%s/remote-debug?app_id=
https://%slaunches.%s/api/v
https://%sconversions.%s/api/v
https://%sgcdsdk.%s/install_data/v4.0/
https://%sadrevenue.%s/api/v

Mitre Techniques:

Deliver Malicious App via Authorized Store (T1475)
Broadcast receivers (T1402)
Application Discovery (T1418)
Masquerade as Legitimate Application(T1444)
Device Administrator Permissions (T1626)
Data Encrypted for Impact (T1471)
Access Sensitive Data in Device Logs(T1413
Unix Shell (T1623)

Key Logging (T1457)
Contact List (T1636)
System Network Configuration Discovery (T1422)
Location tracking (T1430)
System Information Discovery (T1082)
Audio Capture T(1429)

Sectrio Protection:

Sectrio detects the android sample as “SS_Gen_Android_Spyware_A”.

Our Honeypot Network:

This report has been prepared from the threat intelligence gathered by our honeypot network. This honeypot network is today operational in over 75 cities across the world. These cities have at least one of the following attributes:

- Are landing centers for submarine cables
- Are internet traffic hotspots
- House multiple IoT projects with a high number of connected endpoints
- House multiple connected critical infrastructure projects
- Have academic and research centers focusing on IoT
- Have the potential to host multiple IoT projects across domains in the future

Over 18 million attacks a day is being registered across this network of individual honeypots. These attacks are studied, analyzed, categorized, and marked according to a threat rank index, a priority assessment framework that we have developed within Sectrio. The honeypot network includes over 4000+ physical and virtual devices covering over 400+ device architectures and varied connectivity mediums globally. These devices are grouped based on the sectors they belong to for purposes of understanding sectoral attacks. Thus, a layered flow of threat intelligence is made possible.