

# **A TECHNICAL DESCRIPTION OF AN ANDROID MALWARE ANALYSIS**

## **MALWARE TYPE:**

<https://github.com/ashishb/android>

**SKECHY SKYPE APP THAT LIVE IN THE POPULAR ANDROID DEVICE**

[https://github.com/ashishb/android-malware/tree/master/rouge\\_skype](https://github.com/ashishb/android-malware/tree/master/rouge_skype)

**DEPARTMENT OF COMPUTER SCIENCE**

**NOVEMBER 2019**

# **ABSTRACT**

This paper is a technical description of an android malware detection and analysis with the help of Santoku Linux, which is a Linux base tool. This paper will describe the main principles of the detection and analysis of a sample malware and a brief overview of the Android operating system malware, considering that it is the widely used operating system in tablets and smartphones. This paper also shows an example malware ,which is the popular sketchy skype app that is taken from the android malware repository on GitHub and goes in debt to do a static analysis of the malware. There are two different ways to do a malware analysis; static and dynamic analysis, but in this paper, we will be doing just the static analysis of this sample malware type.

# TABLE OF CONTENT

## Table of Contents

Abstract .....	2
Table of Content .....	3
Introduction .....	4
Creating an Emulator .....	5
Static Analysis .....	6-30
Conclusion .....	31
References .....	32
Acknowledgement .....	33

# INTRODUCTION

This paper examines the methods to analyze an Android malware sample. The malware when running on an Android device will request very extensive permissions such as external storage, preventing the phone from sleeping etc.... and the paper will also describe some of the various suspicious method that is gotten from the source code of the malware. Some of these methods include; “chmod” which is to enable additional applications to be downloaded once the malware is running on an Android device. This paper will also examine other methods such as the “telephony manager” which allow someone to manage how an android phone is being used to do things like sending SMS messages and some other sketchy stuff that a malware will try to do on an android device. We will analyze the full functionality of the malware by using both static and dynamic analysis techniques with the help of the various tools listed below:

**List of VMs used:** This lab exercise makes use of Santoku Linux VM.

**Tools:** AVD Manager, ADB, dex2jar, apktool

**Files used in this lab:** skype.apk, apktool,

## **MALWARE TYPE**

<https://github.com/ashishb/android>

SKECHY SKYPE APP THAT LIVE IN THE POPULAR ANDROID DEVICE

[https://github.com/ashishb/android-malware/tree/master/rouge\\_skype](https://github.com/ashishb/android-malware/tree/master/rouge_skype)

# CREATING AN EMULATOR

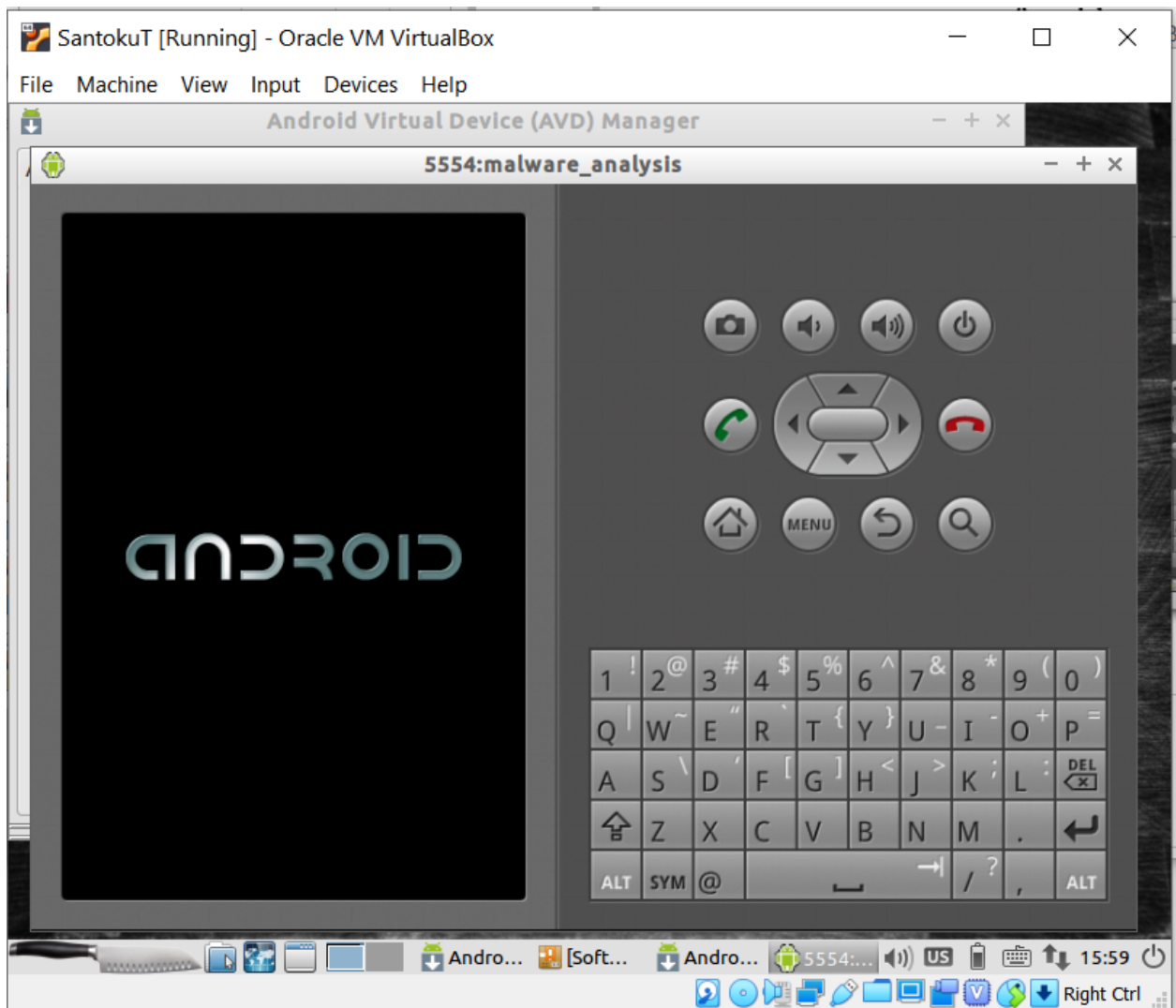
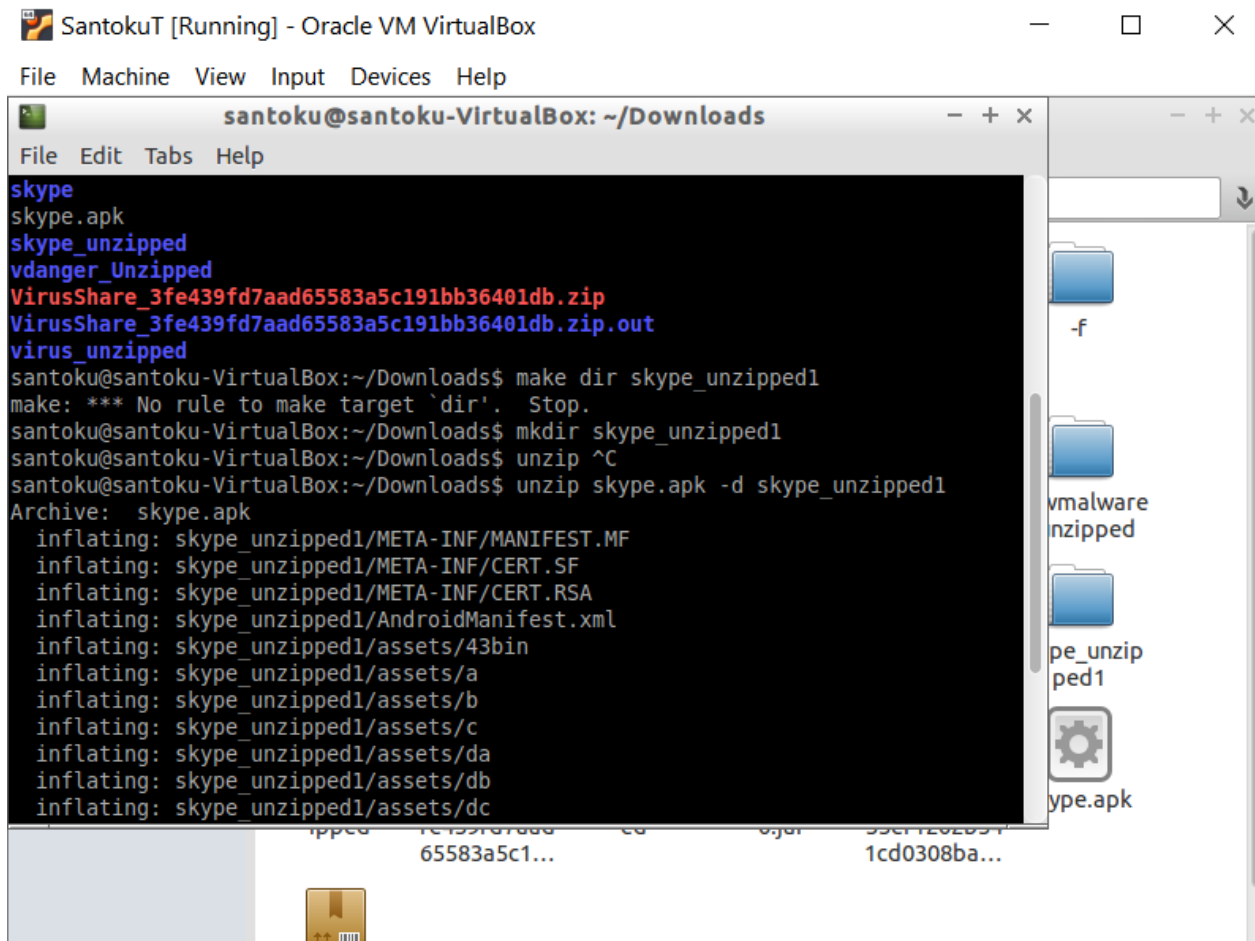


Figure 1.0

We start the malware analysis process by creating an Android virtual device also known as an emulator as shown in figure 1.0 above and this will help us in loading the malware onto the device using the linux command of “adb install” which will help us in doing a dynamic analysis of the malware to better understand the intent of the developer of the malware and the impact of the malware on an Android device.

# STATIC ANALYSIS

In this part of the paper, we will be decompiling the malware file “skype.apk” and examining the source code, so as to understand the behavior of the malware. The first step will be to create a directory on santoku linux through which we can store the unzipped malware file and this is indicated in the figure below:



```
SantokuT [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

santoku@santoku-VirtualBox: ~/Downloads
File Edit Tabs Help

skype
skype.apk
skype_unzipped
vdanger_Unzipped
VirusShare_3fe439fd7aad65583a5c191bb36401db.zip
VirusShare_3fe439fd7aad65583a5c191bb36401db.zip.out
virus_unzipped
santoku@santoku-VirtualBox:~/Downloads$ make dir skype_unzipped1
make: *** No rule to make target `dir'. Stop.
santoku@santoku-VirtualBox:~/Downloads$ mkdir skype_unzipped1
santoku@santoku-VirtualBox:~/Downloads$ unzip ^C
santoku@santoku-VirtualBox:~/Downloads$ unzip skype.apk -d skype_unzipped1
Archive:  skype.apk
  inflating: skype_unzipped1/META-INF/MANIFEST.MF
  inflating: skype_unzipped1/META-INF/CERT.SF
  inflating: skype_unzipped1/META-INF/CERT.RSA
  inflating: skype_unzipped1/AndroidManifest.xml
  inflating: skype_unzipped1/assets/43bin
  inflating: skype_unzipped1/assets/a
  inflating: skype_unzipped1/assets/b
  inflating: skype_unzipped1/assets/c
  inflating: skype_unzipped1/assets/da
  inflating: skype_unzipped1/assets/db
  inflating: skype_unzipped1/assets/dc
```

The file manager on the right shows the following structure:

- f
- vmalware
- unzipped
- pe\_unzip
- ped1
- skype.apk

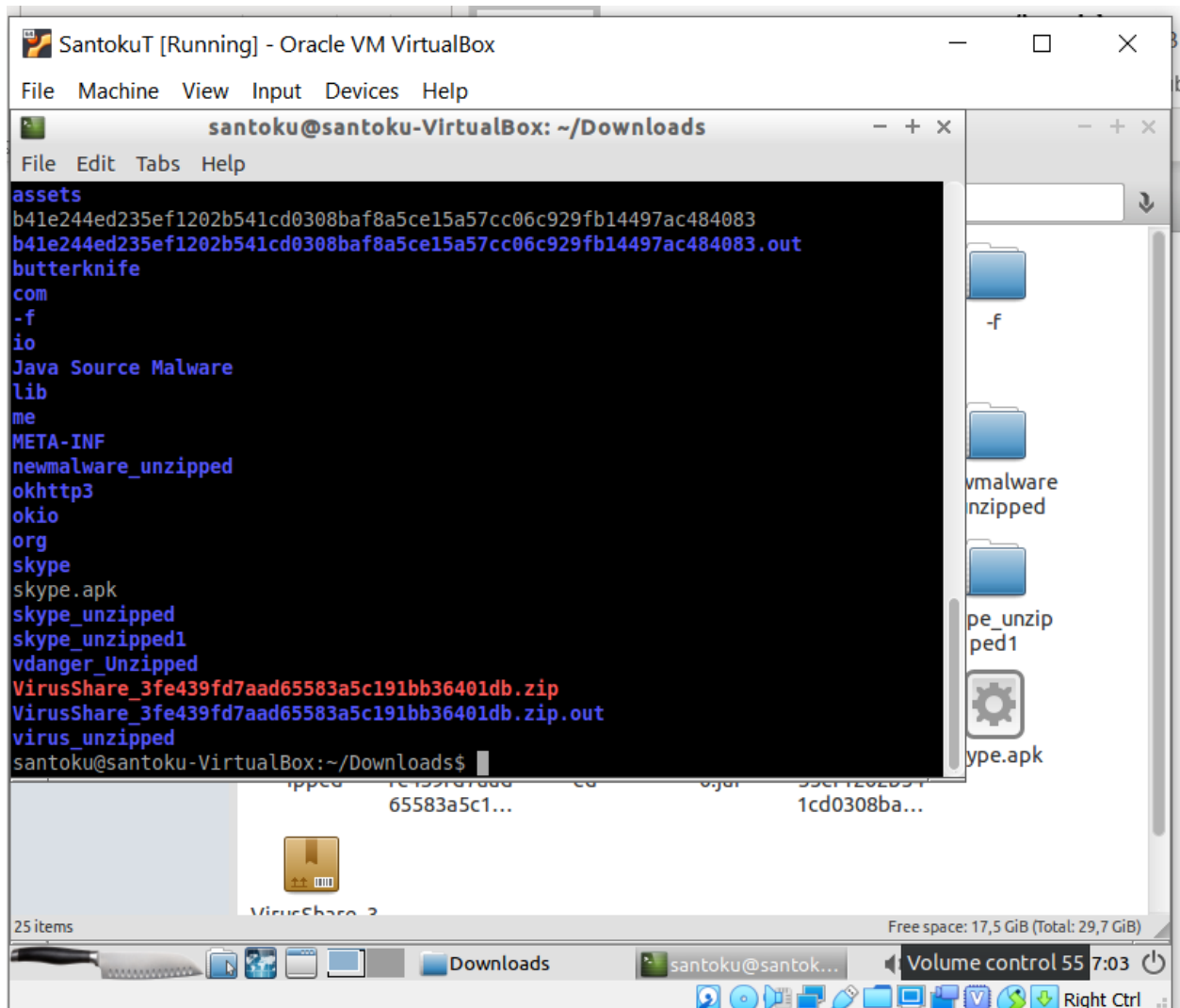


Figure 1.1

Figure 1.1 above shows two linux commands that is use; the first one is the “mkdir” which is to create a new directory and the directory is named “skype\_unzipped1”. The next linux command shown in figure 1.1 above is the “unzip” and “-d” which is to unzipped the malware Android application package (apk) file and then store the unzipped file into the newly created directory “skype\_unzipped1”. We can also see in figure 1.1 above that when we list the directory, we can now see the newly created unzipped malware file “skype\_unzipped1”. Now we will go to the newly created unzipped directory to see the files inside that directory:

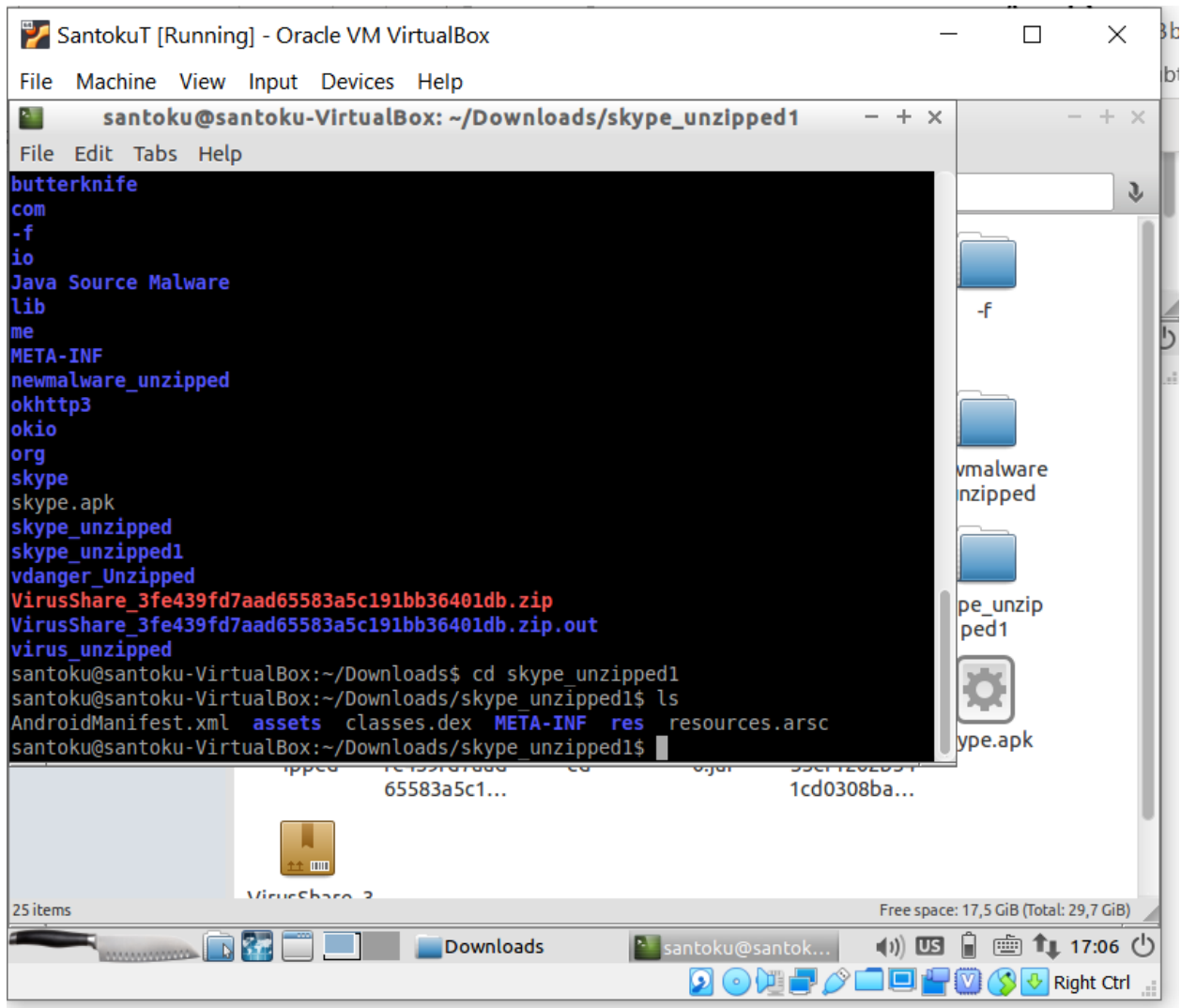


Figure 1.2

Figure 1.2 above shows the new created unzipped directory and all the files listed inside that directory and we could see that we have the AndroidManifest.xml file, which is of interest to us and we will try to open it using the “vim” command.



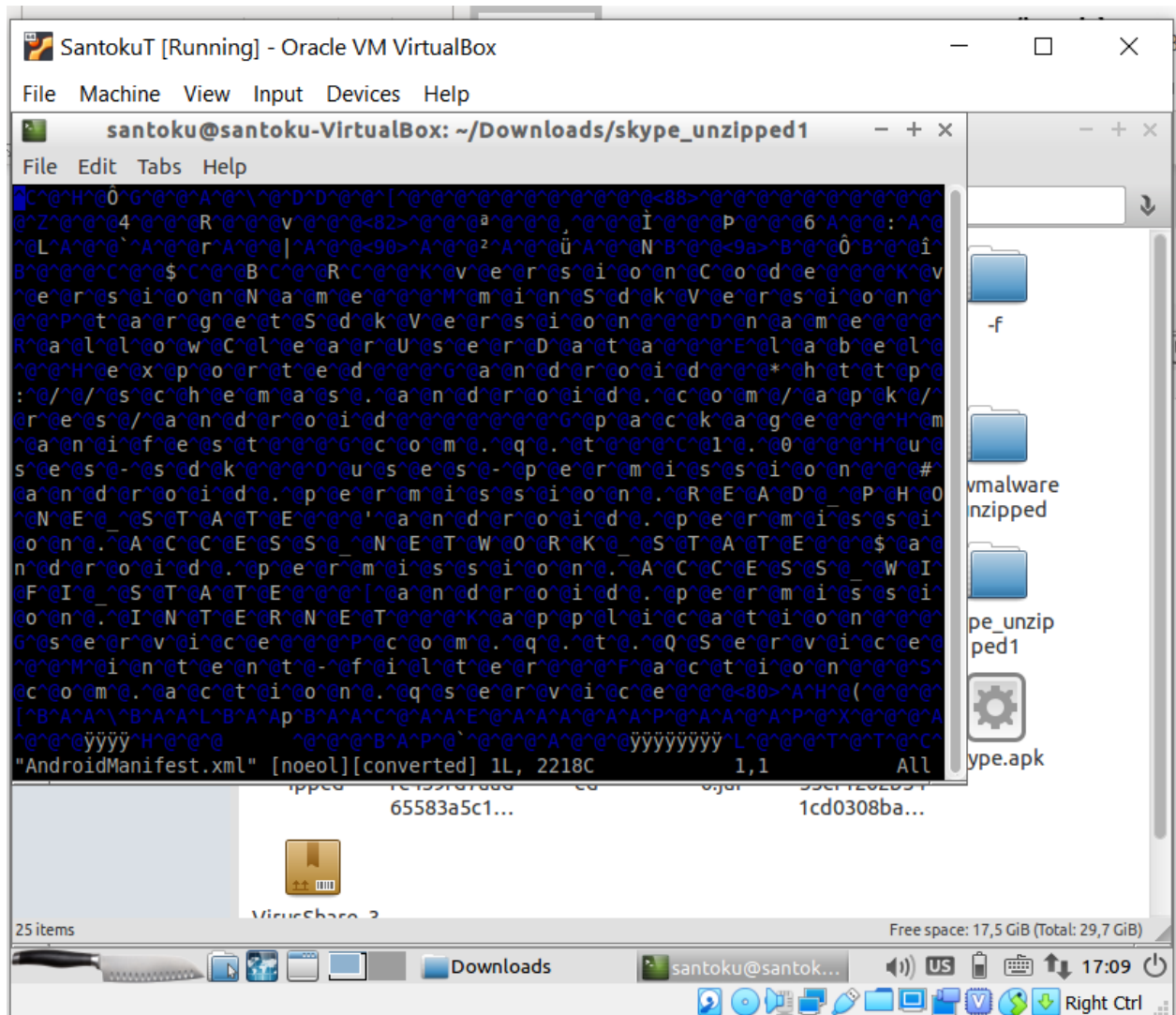


Figure 1.3

Figure 1.3 above shows the AndroidManifest.xml file of the unzipped malware file “skype\_unzipped1” but it is encrypted. So we will need to decompile the file before we will be able to see the decrypted content of the AndroidManifest.xml file and this is done by using the “apktool” to decrypt the file. It should be noted that we have two unzipped files of the malware; “skype\_unzipped1” and “skype\_unzipped “ as indicated in figure 1.1 above, so we will be using the “skype\_unzipped” from this point onward in this paper, but it doesn’t matter which unzipped file you use because they are all the same.

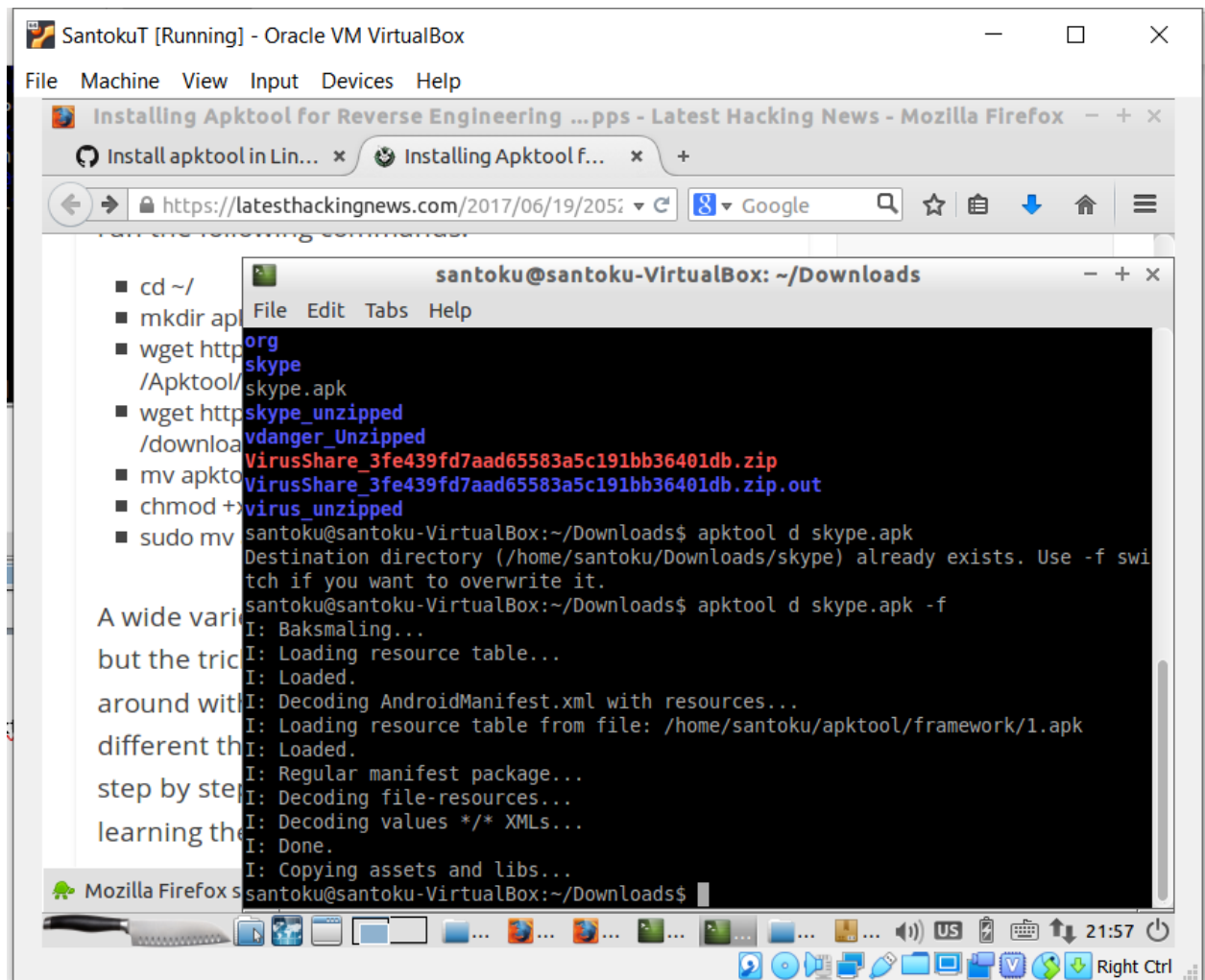


Figure 1.4

Figure 1.4 above shows the use of the apktool with the “d” command to decompile the original “skype.apk” file and now we will try to open the AndroidManifest.xml again this time and we notice that it is no longer encrypted, and this is shown below:

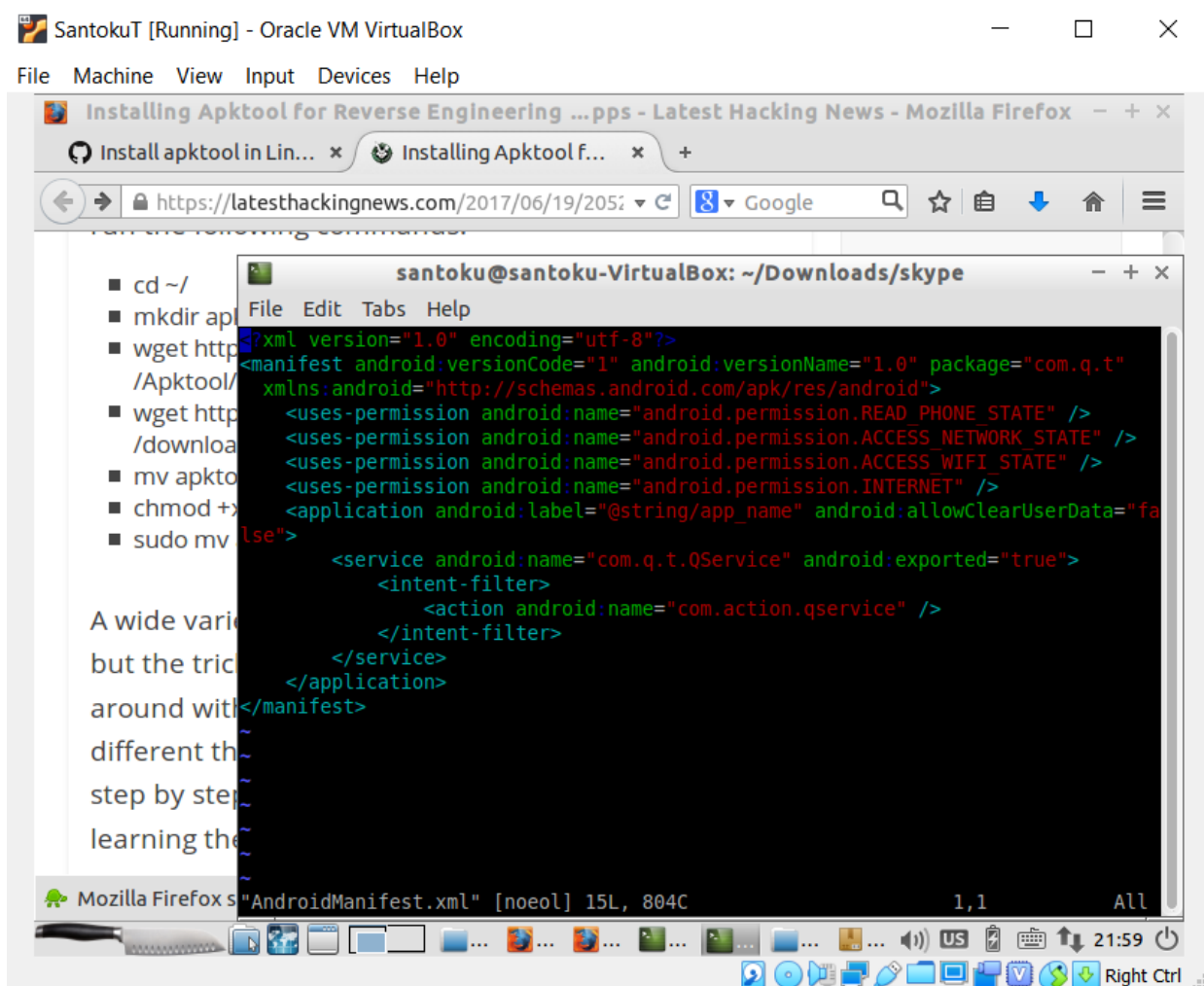


Figure 1.5

Figure 1.5 above shows the decrypted AndroidManifest.xml file which we opened using the “vim” command. The AndroidManifest.xml always shows the permission that an apk file will request when installed on an android device and we can see from figure 1.5 above the list of all the permissions that the application tends to requests and the services it wants to use. We can see that the malware is requesting the phone state, access to the network, access to the wifi and internet, but it should be noted that any standard android application will request these permissions listed above and so we can really tell from this AndroidManifest.xml file if this is

actually a malware or not, so we will need to dive deep into the application by looking into the “smali” file as shown below:

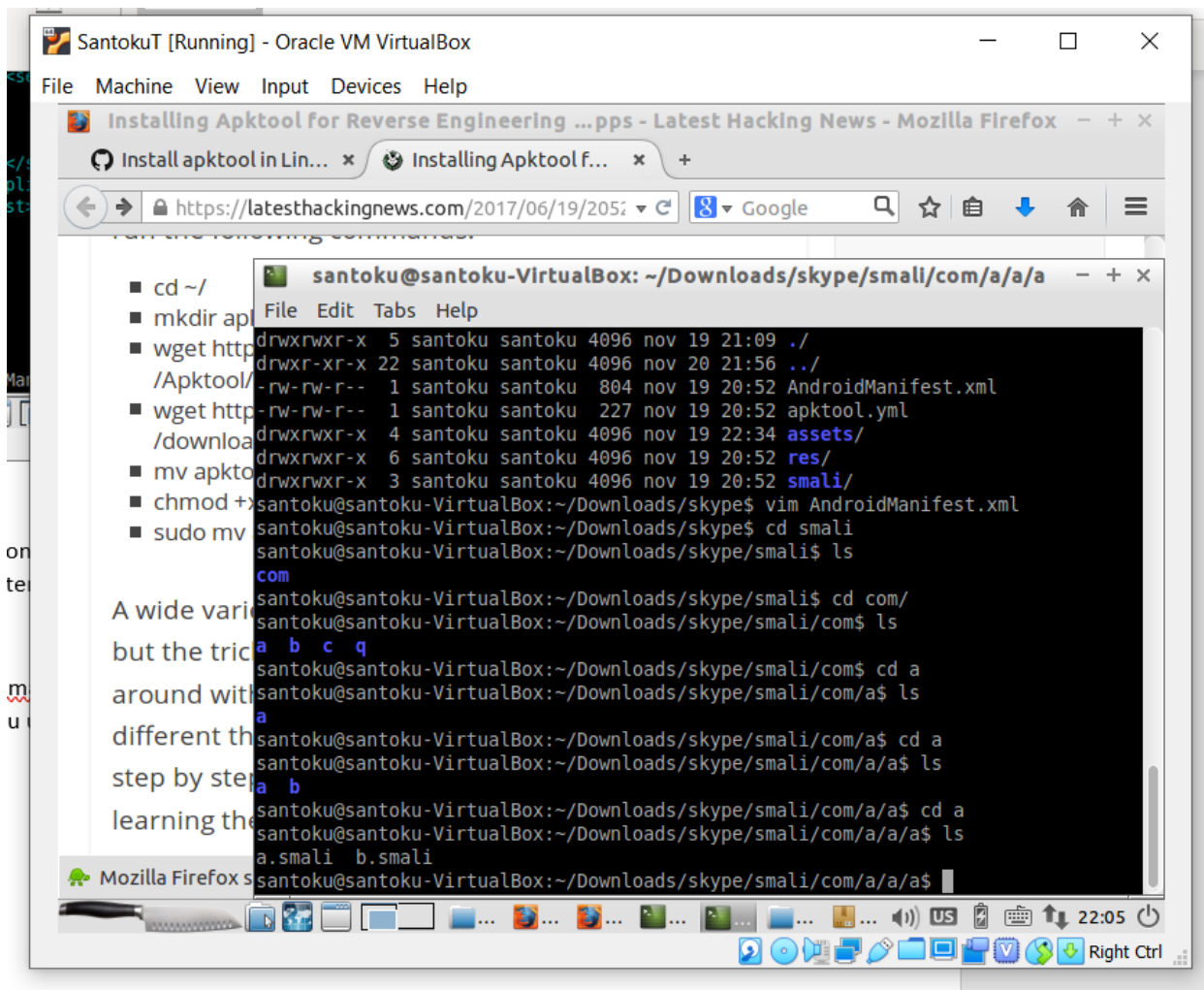


Figure 1.6

Figure 1.6 above shows us the path to get to the “smali” file which is an assemble style language that the “classes.dex” file is converted into whenever we decompiled the original malware file using the “apktool”. The “classes.dex” file is located in the same directory as the AndroidManifest.xml file as indicated in figure 1.2. we will now open the “s.smali” file that is shown in figure 1.6 above using the “vim” command and it is shown below as follow:

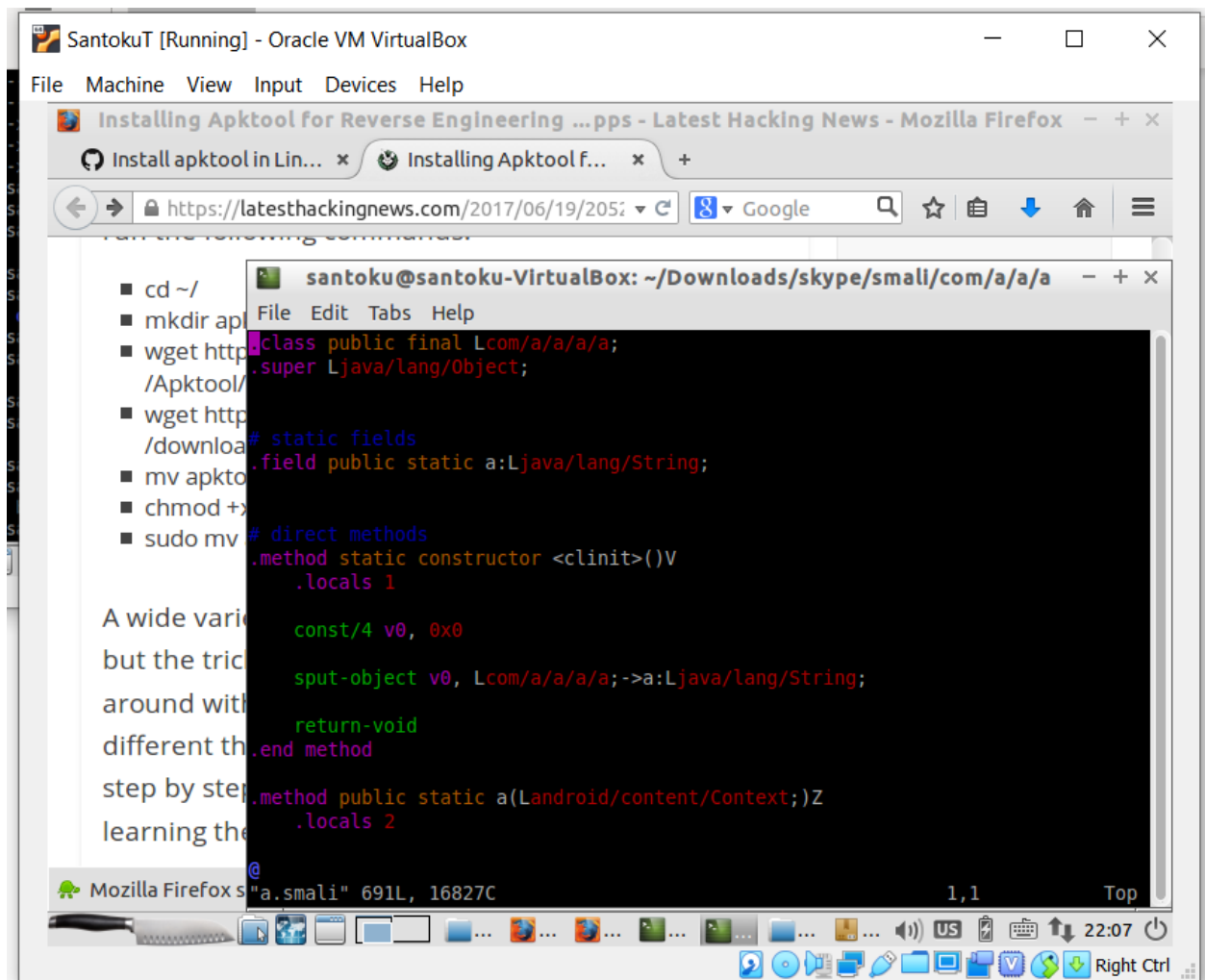


Figure 1.7

Figure 1.7 above shows the “smali” file and it is decrypted and the code is written in a java style assembly language, but later in this paper we will convert the format to a java style language so as to better understand the code and behavior of the malware. We will now go back into the decompiled malware file and take a look the resources folder which has the “strings” that will be use in the application and this will be one of the best ways to understand what the application is trying to do:

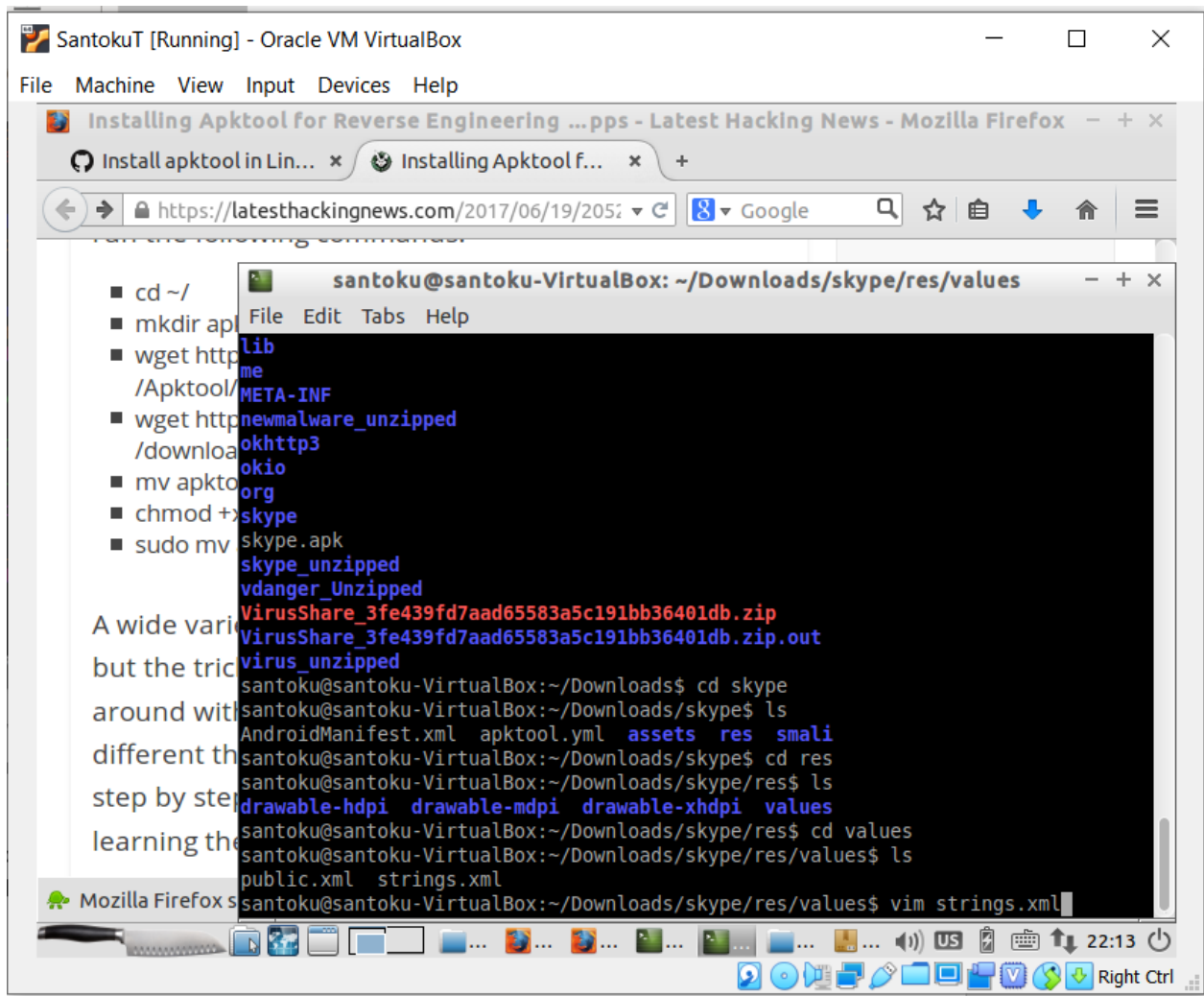


Figure 1.8

Figure 1.8 above shows the use of the “vim” command to open the “strings.xml” file and this is another way to find out what this malware application is trying to do and also look for any sketchy or suspicious language that has been used by the developers of the malware.

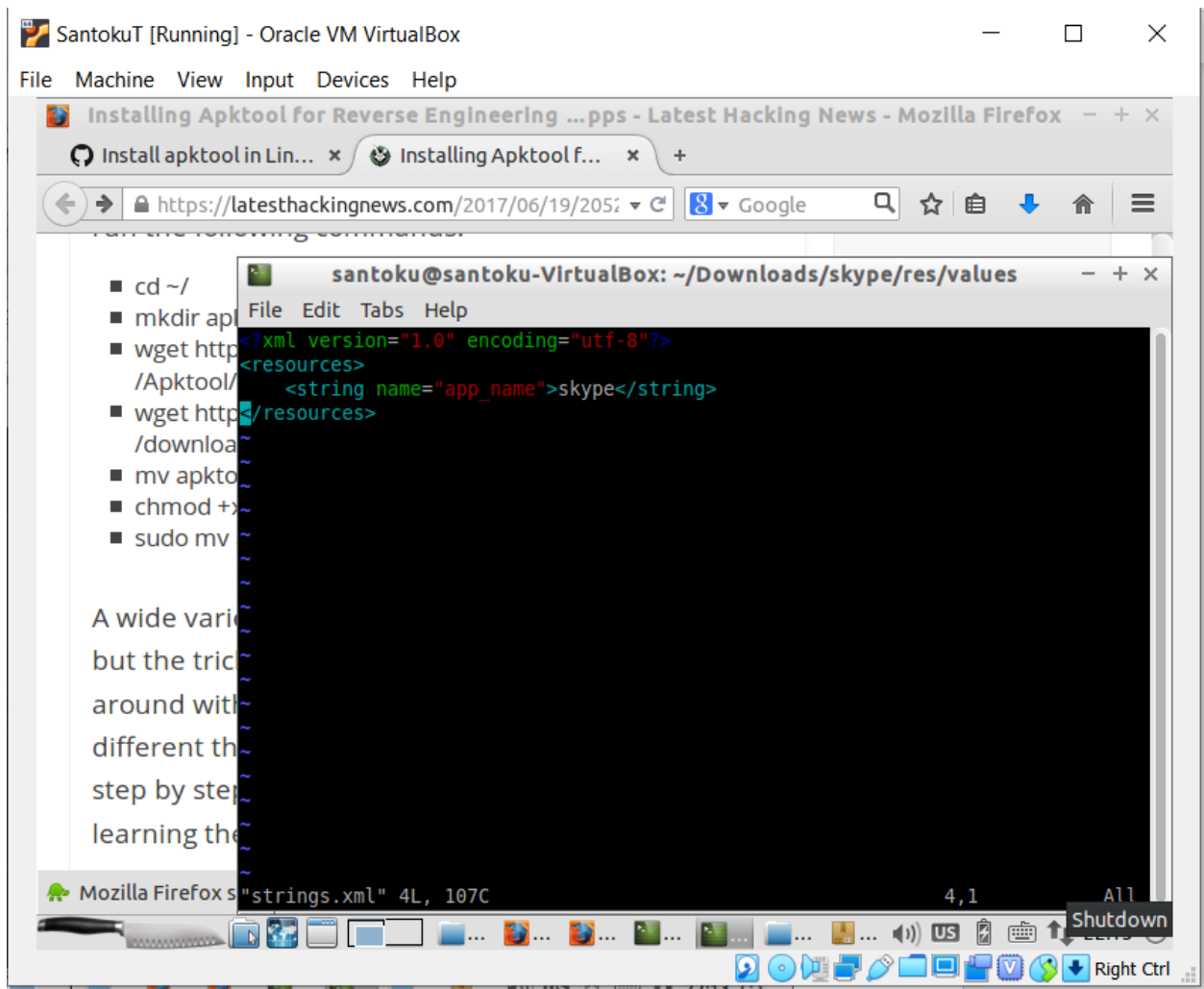


Figure 1.9

Figure above shows the strings file and in order to access the source code of this file, we will need to go into the original unzipped file of the malware “skype\_unzipped”. And convert the “classes.dex” file into a “jar” file type format so that we can be able to open and read the code using JD-GUI. We will first use the “dex2jar” command to convert the file to the “jar” file type format as shown below:







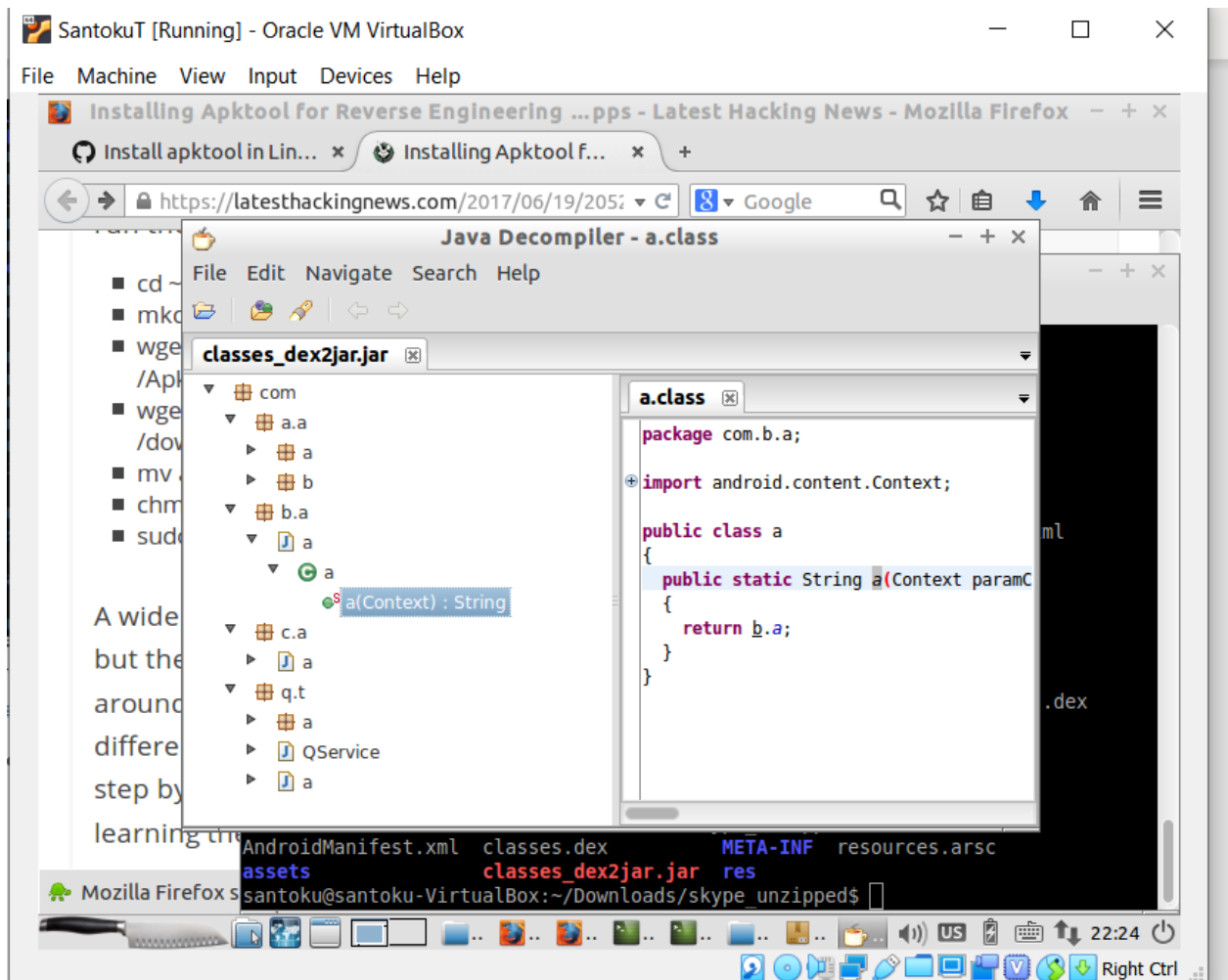


Figure 2.1

Figure 2.1 shows the “classes.dex2jar.jar” file that has been opened using JD-GUI and we can see that it has a bunch of obfuscated and disassembled classes from the “jar” file that we created. There are also a lot of different random service classes that have been created and downloaded and this is a big indication that this can really be a malware. We will continue to dive deep into the file, and we see:

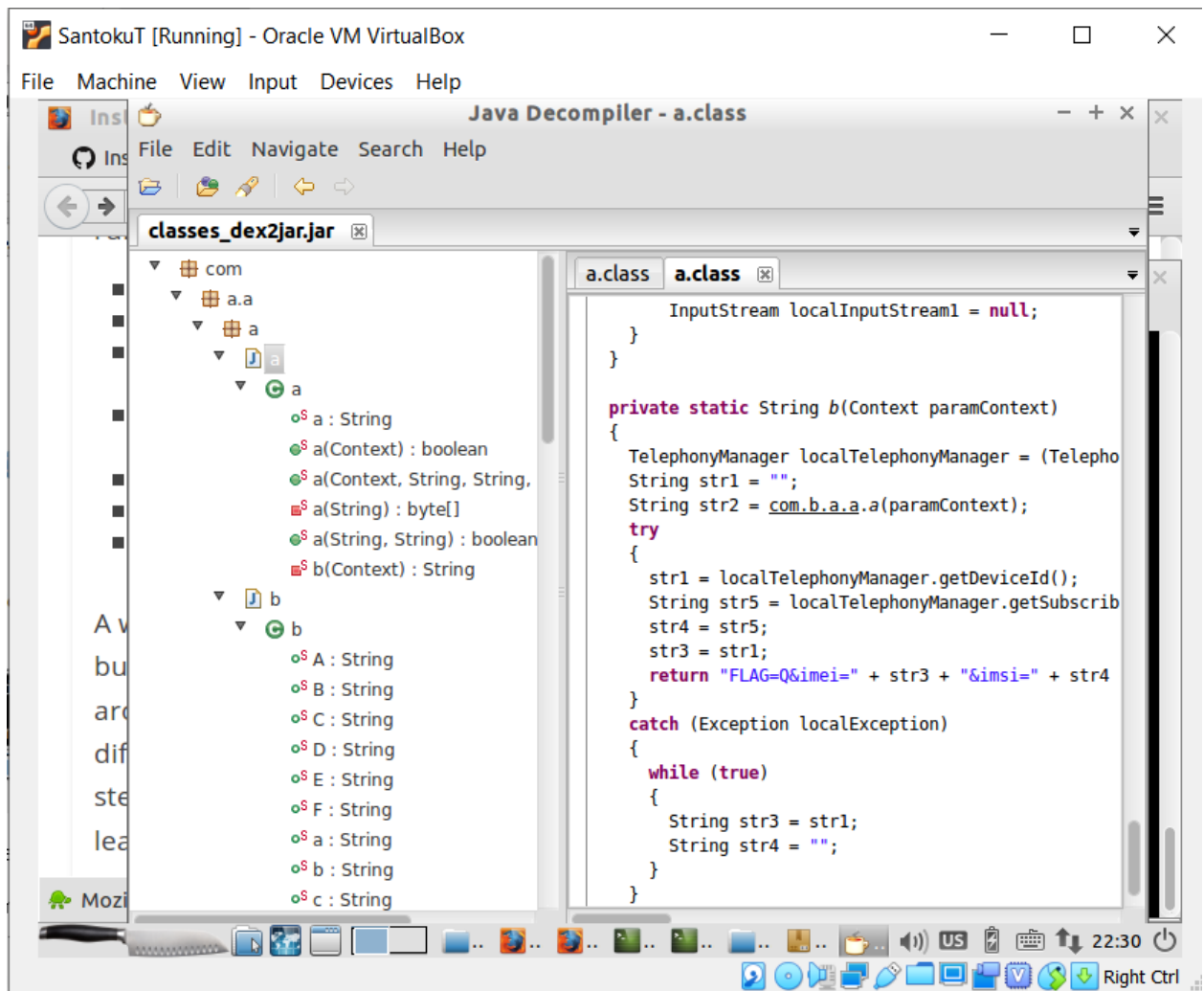
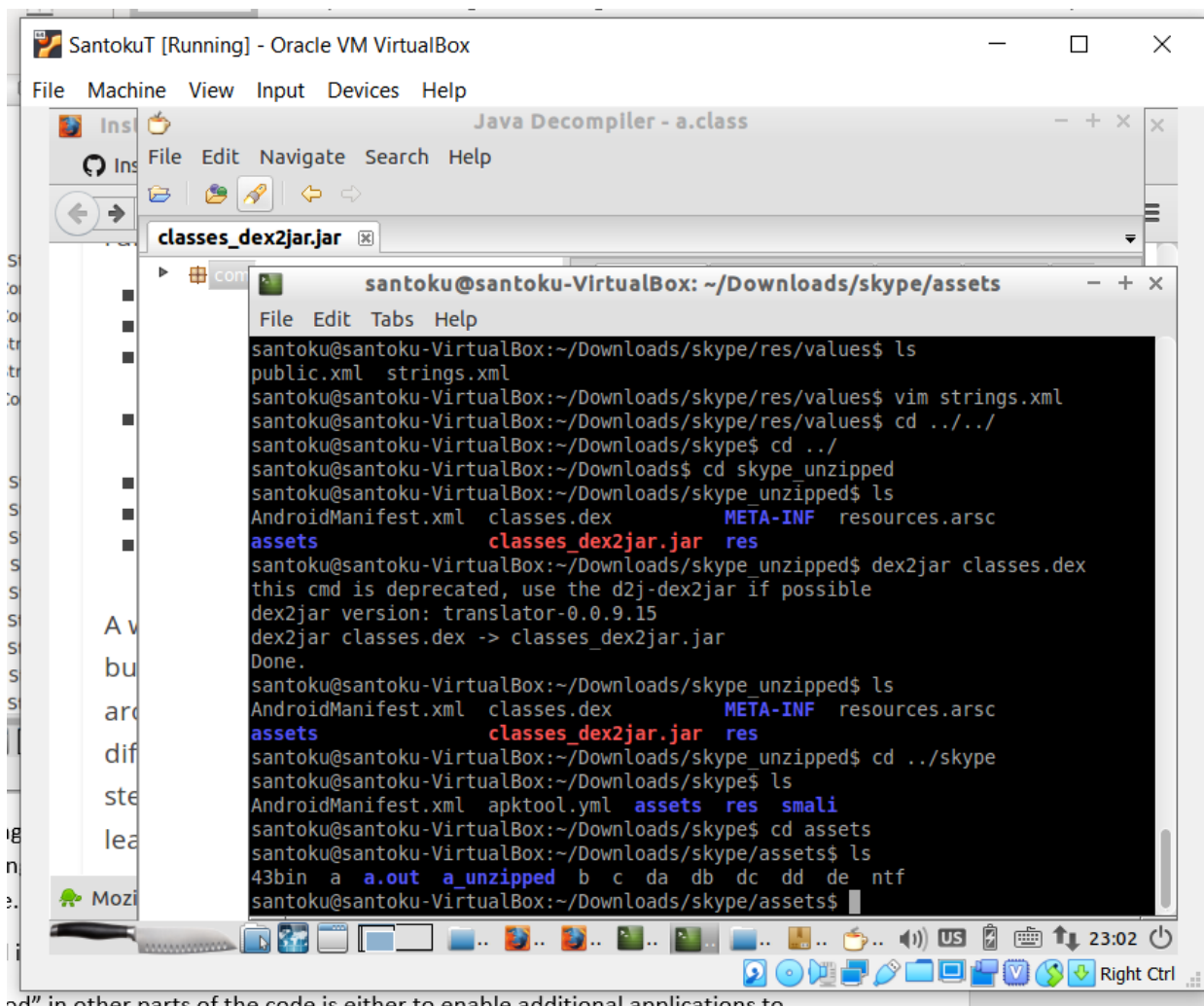


Figure 2.2

From figure 2.2 above we can start to see some suspicious methods that the developers of this malware have used, such as the “telephony manager” which is a method that allows someone to manage how an android phone is being used to do things like sending SMS messages and some other sketchy and dangerous things that a malware will try to do on an android device. You can see that the “telephony manager” method is being accessed in the above code in figure 2.2 above to get system service for the targeted android phone by the malware application. Also, when we scroll down we see the use of another suspicious method “chmod” which is a method

that is used to enable additional applications to be downloaded onto an android device. The use of these methods clearly indicates that this is a malware, so we will now continue our research by looking at the asset directory of the malware application as shown below:



The screenshot displays a VirtualBox environment with a window titled 'SantokuT [Running] - Oracle VM VirtualBox'. Inside the VM, a terminal window shows the following commands and output:

```
santoku@santoku-VirtualBox: ~/Downloads/skype/assets
File Edit Tabs Help
santoku@santoku-VirtualBox:~/Downloads/skype/res/values$ ls
public.xml  strings.xml
santoku@santoku-VirtualBox:~/Downloads/skype/res/values$ vim strings.xml
santoku@santoku-VirtualBox:~/Downloads/skype/res/values$ cd ../../
santoku@santoku-VirtualBox:~/Downloads/skype$ cd ../
santoku@santoku-VirtualBox:~/Downloads$ cd skype_unzipped
santoku@santoku-VirtualBox:~/Downloads/skype_unzipped$ ls
AndroidManifest.xml  classes.dex  META-INF  resources.arsc
assets               classes_dex2jar.jar  res
santoku@santoku-VirtualBox:~/Downloads/skype_unzipped$ dex2jar classes.dex
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.15
dex2jar classes.dex -> classes_dex2jar.jar
Done.
santoku@santoku-VirtualBox:~/Downloads/skype_unzipped$ ls
AndroidManifest.xml  classes.dex  META-INF  resources.arsc
assets               classes_dex2jar.jar  res
santoku@santoku-VirtualBox:~/Downloads/skype_unzipped$ cd ../skype
santoku@santoku-VirtualBox:~/Downloads/skype$ ls
AndroidManifest.xml  apktool.yml  assets  res  smali
santoku@santoku-VirtualBox:~/Downloads/skype$ cd assets
santoku@santoku-VirtualBox:~/Downloads/skype/assets$ ls
43bin  a  a.out  a_unzipped  b  c  da  db  dc  dd  de  ntf
santoku@santoku-VirtualBox:~/Downloads/skype/assets$
```

Overlaid on the terminal is a 'Java Decompiler - a.class' window showing a file explorer view of 'classes\_dex2jar.jar'. The system tray at the bottom indicates the time is 23:02.

Figure 2.3a

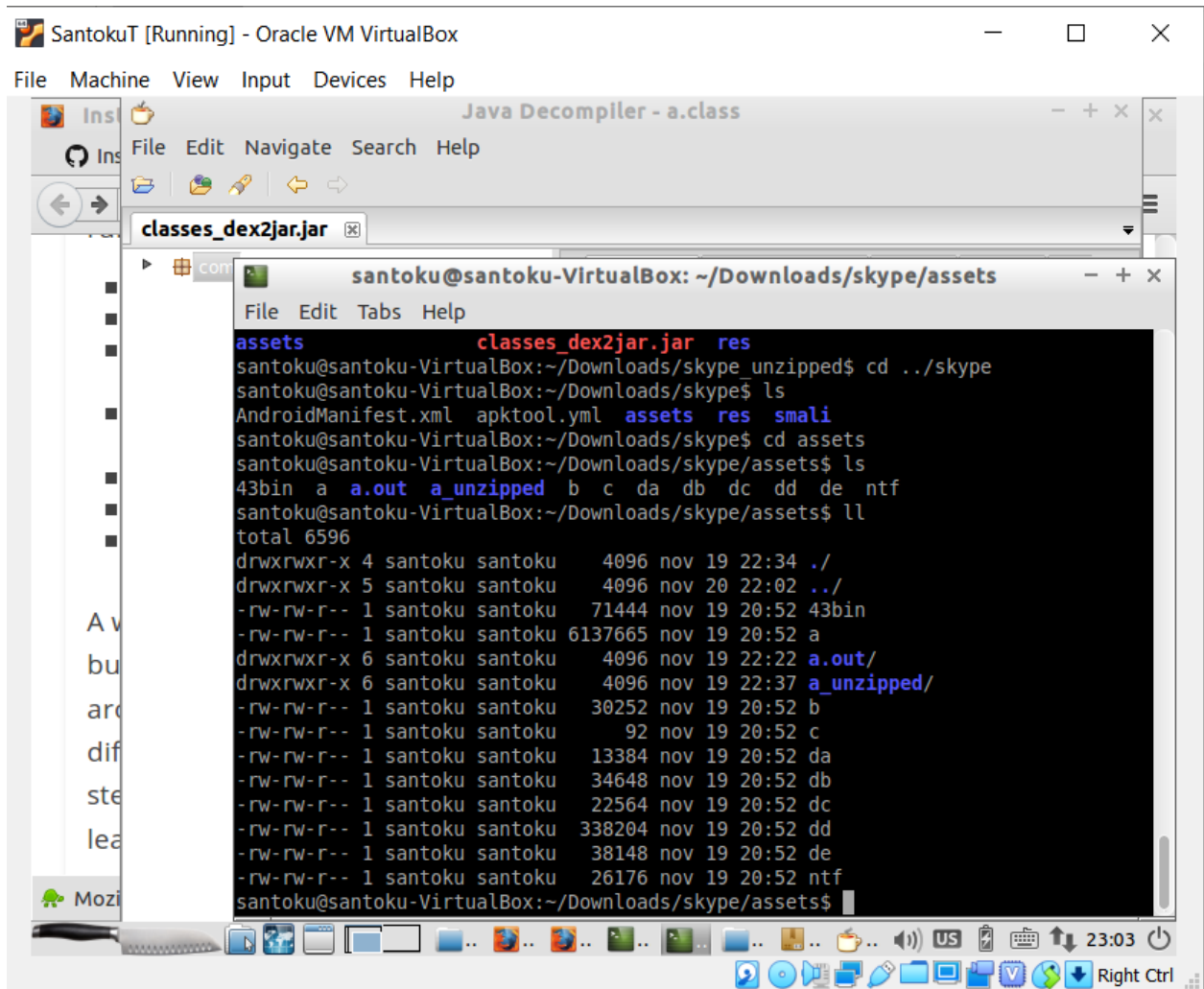


Figure 2.3b

Figure 2.3a and 2.3 still shows a bunch of random obfuscated file names and there are mostly binary files present inside this resources folder, so we will need to open a couple of this files to find the “jar” file because it is our main interest.

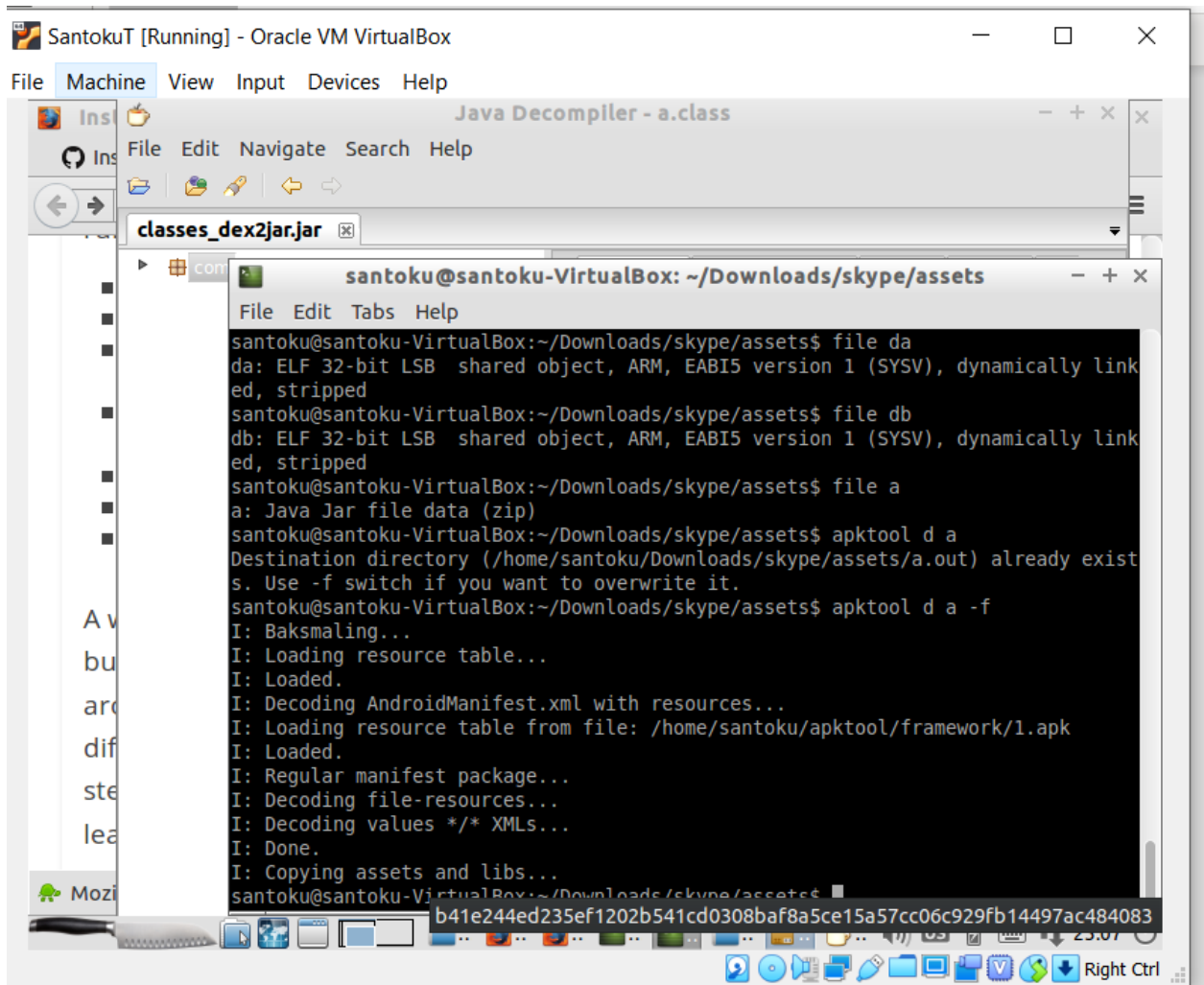


Figure 2.4

We now see that the “jar” file is located inside the “a” file as indicated above in figure 2.4 and it also shows how we have decompiled the “a” file and after decompiling the “a” file we another file know as the “a.out” file as shown below:

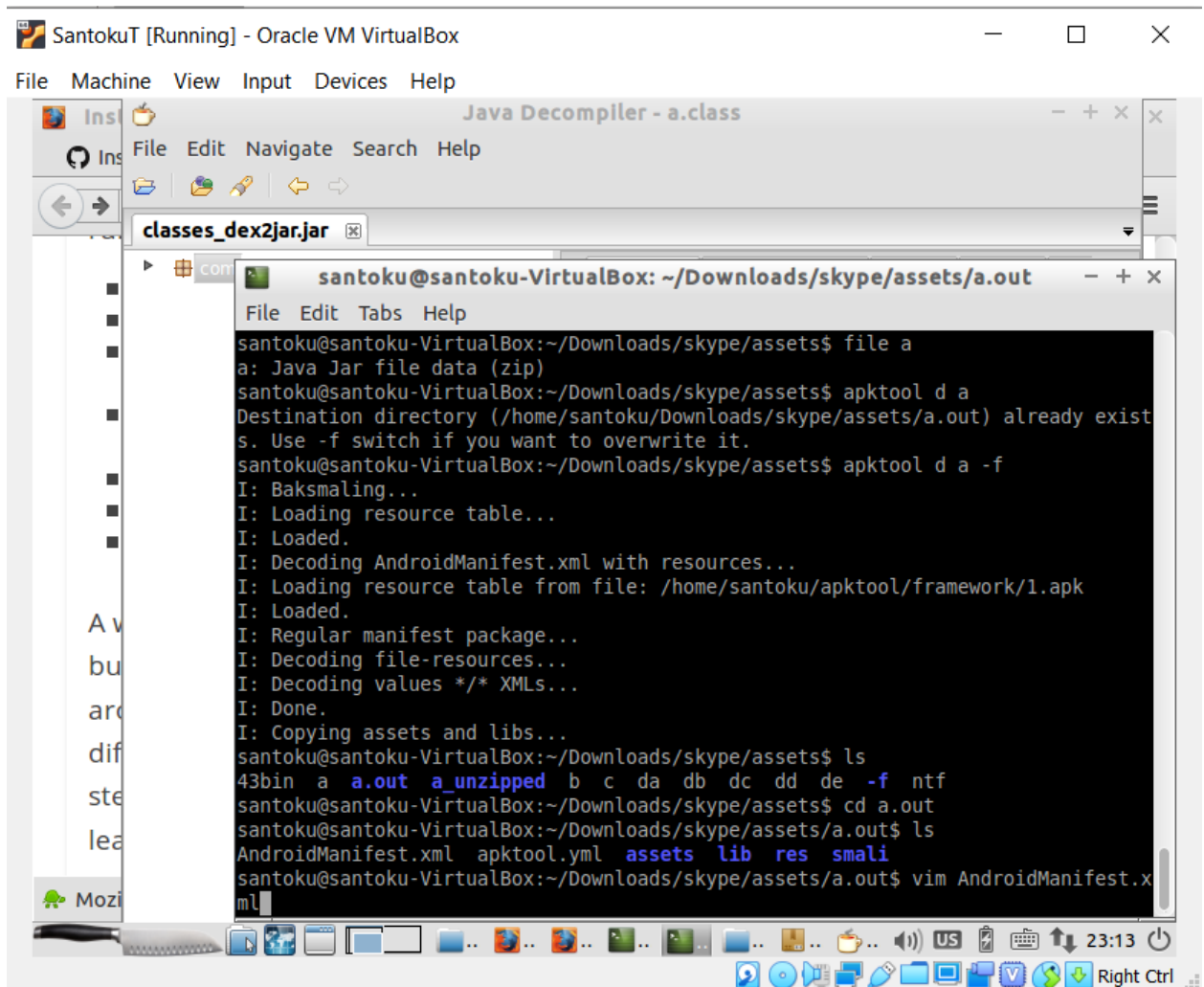


Figure 2.5

Figure 2.5 above shows the various files that are present inside the “a.out” file and we now see we have another AndroidManifest.xml file which should be of big interest to us because the initial AndroidManifest.xml file we saw before didn’t really tell us if this was a malware or not. We will now open the AndroidManifest.xml file using the “vim” command as shown below:

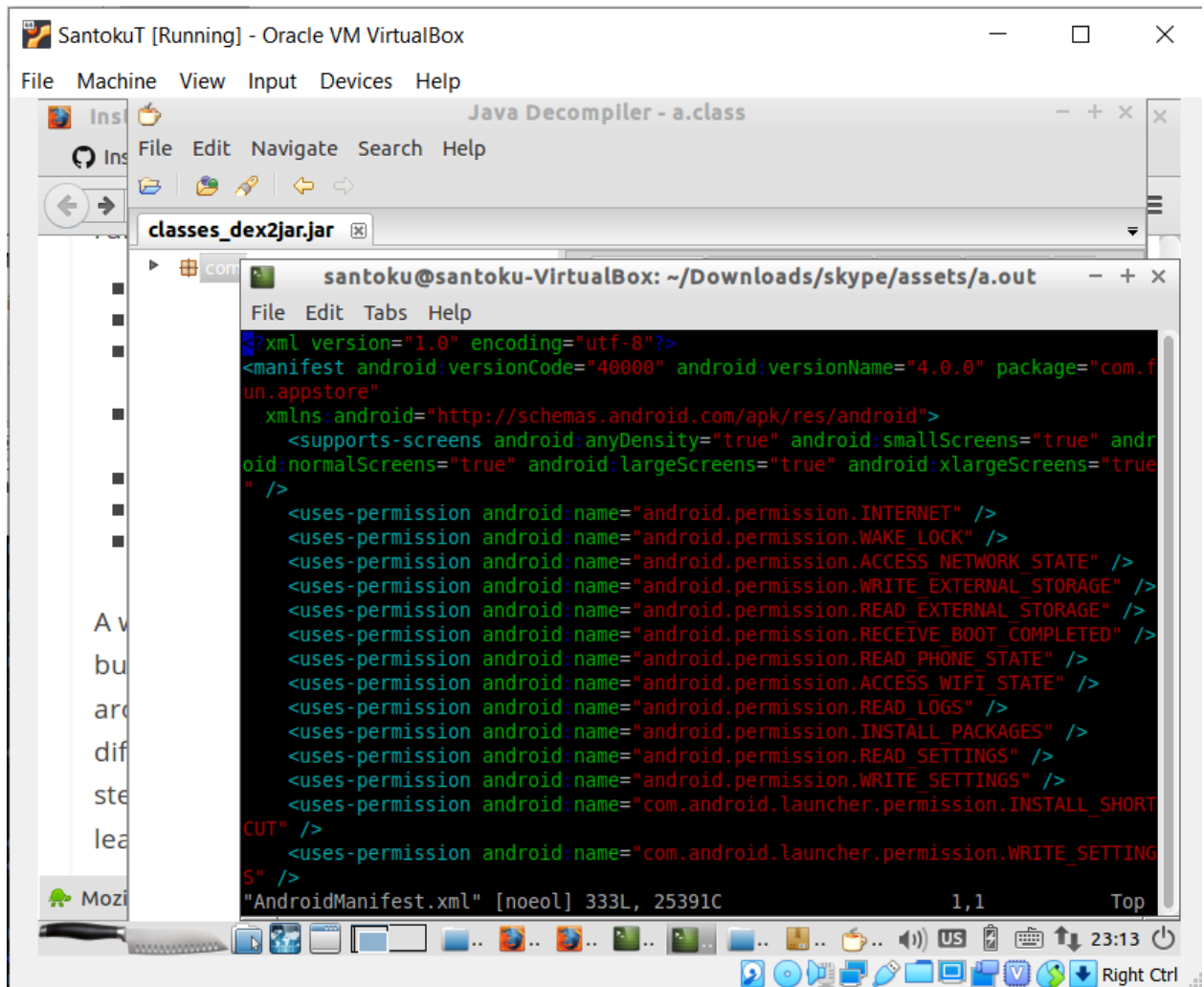


Figure 2.6a



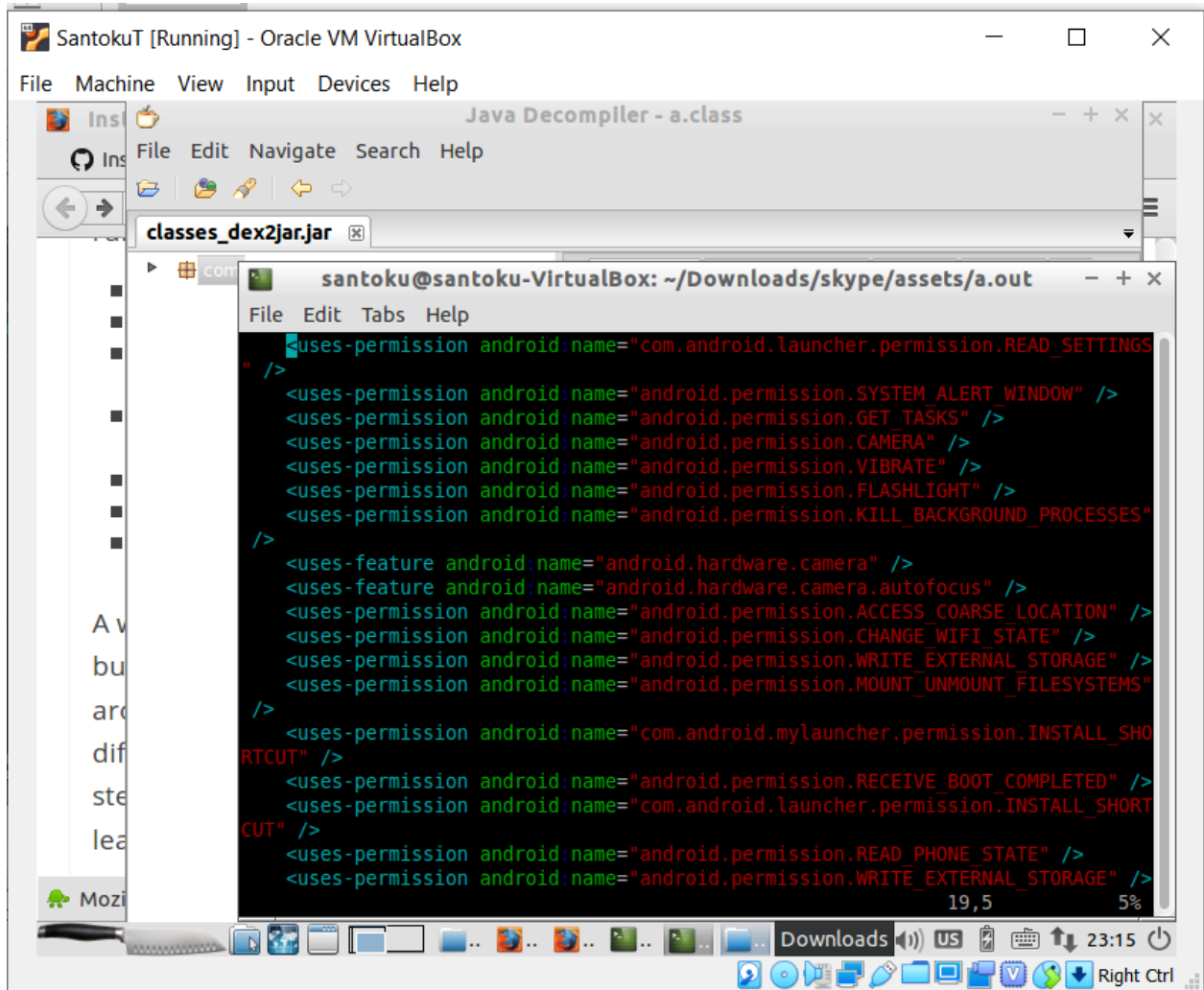


Figure 2.6b

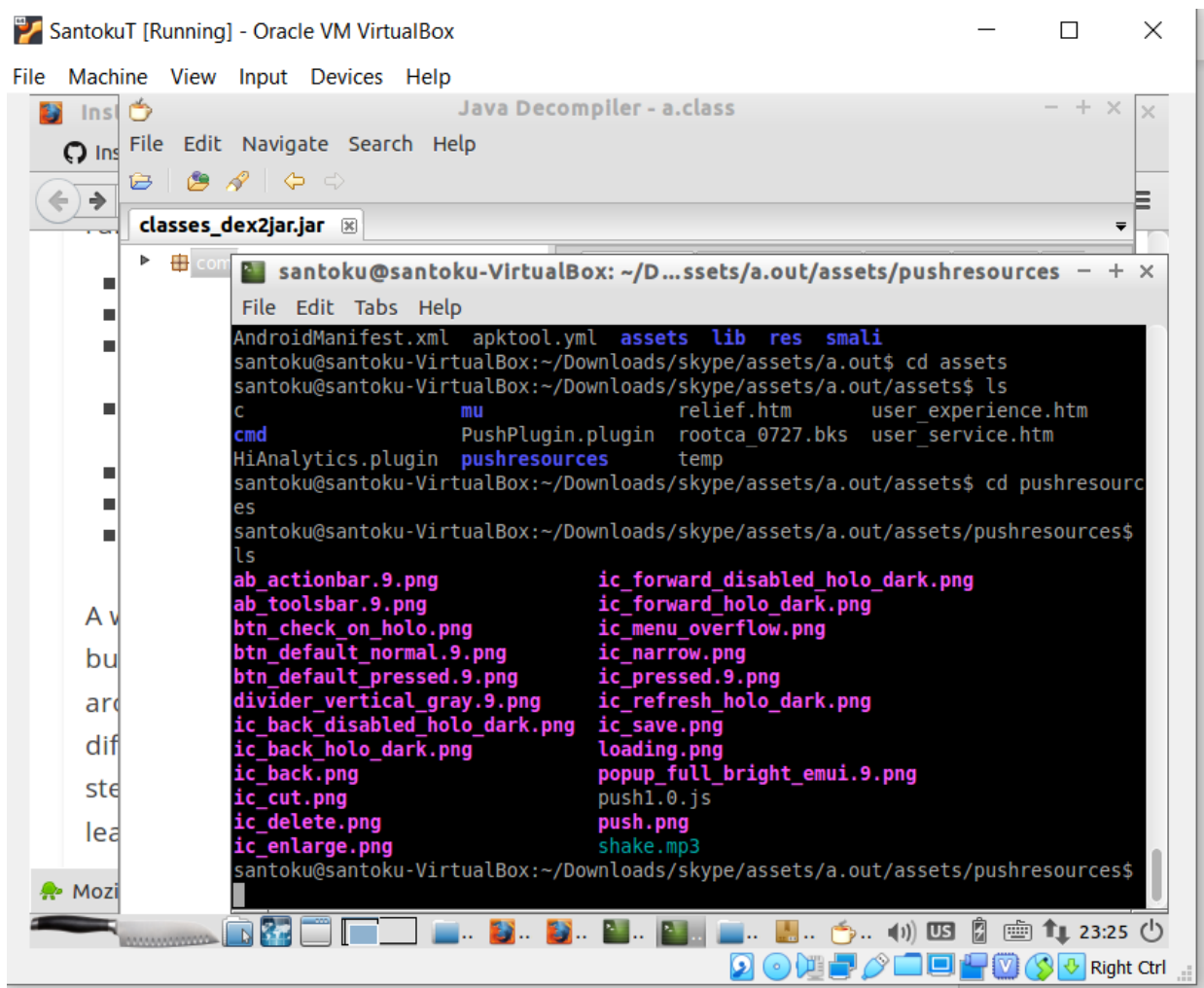
When we open the AndroidManifest.xml file using the “vim” command as shown in figure 2.6a and 2.6b above, we noticed that this file has way more permission request as compared to the first AndroidManifest.xml file we had opened in figure 1.5 earlier. This therefore implies this is another application on its own because there are activities everywhere and a whole bunch of services being requested and called.

We can therefore imply that the original application which we had examined the AndroidManifest.xml file in figure 1.5 earlier was just a dummy application to fool or deceive



malware analysts like myself and any antivirus software or malware checkers because this AndroidManifest.xml file in figure 2.6a and 2.6b above seems to be the real deal because we can see that the permission being requested are very extensive such as installing packages, reading logs, reading the phone state, writing to external storage, preventing the phone from going to sleep and all of these permission request are out of the norms of the normal permissions a standard android application will request when installed on an any android device.

We will now also take a look at the asset file of this “a.out” directory or file which is the inner application of the malware that has the real AndroidManifest.xml file and we will see that it has a bunch of strange assets like images and mp3 files as shown below:



The screenshot shows a Java Decompiler window titled "Java Decompiler - a.class" with a menu bar (File, Edit, Navigate, Search, Help) and a toolbar. The main pane displays the contents of a directory named "pushresources" in a terminal-like font. The directory listing includes various files and subdirectories, such as "AndroidManifest.xml", "apktool.yml", "assets", "lib", "res", and "smali". The terminal output shows the following commands and results:

```
santoku@santoku-VirtualBox: ~/Downloads/skype/assets/a.out$ cd assets
santoku@santoku-VirtualBox: ~/Downloads/skype/assets/a.out/assets$ ls
c                               mu                               relief.htm                    user_experience.htm
cmd                             PushPlugin.plugin             rootca_0727.bks               user_service.htm
HiAnalytics.plugin              pushresources                  temp
santoku@santoku-VirtualBox: ~/Downloads/skype/assets/a.out/assets$ cd pushresources
santoku@santoku-VirtualBox: ~/Downloads/skype/assets/a.out/assets/pushresources$ ls
ab_actionbar.9.png              ic_forward_disabled_holo_dark.png
ab_toolbar.9.png                ic_forward_holo_dark.png
btn_check_on_holo.png           ic_menu_overflow.png
btn_default_normal.9.png        ic_narrow.png
btn_default_pressed.9.png       ic_pressed.9.png
divider_vertical_gray.9.png     ic_refresh_holo_dark.png
ic_back_disabled_holo_dark.png  ic_save.png
ic_back_holo_dark.png           loading.png
ic_back.png                     popup_full_bright_emui.9.png
ic_cut.png                      push1.0.js
ic_delete.png                   push.png
ic_enlarge.png                  shake.mp3
santoku@santoku-VirtualBox: ~/Downloads/skype/assets/a.out/assets/pushresources$
```

Figure 2.7a

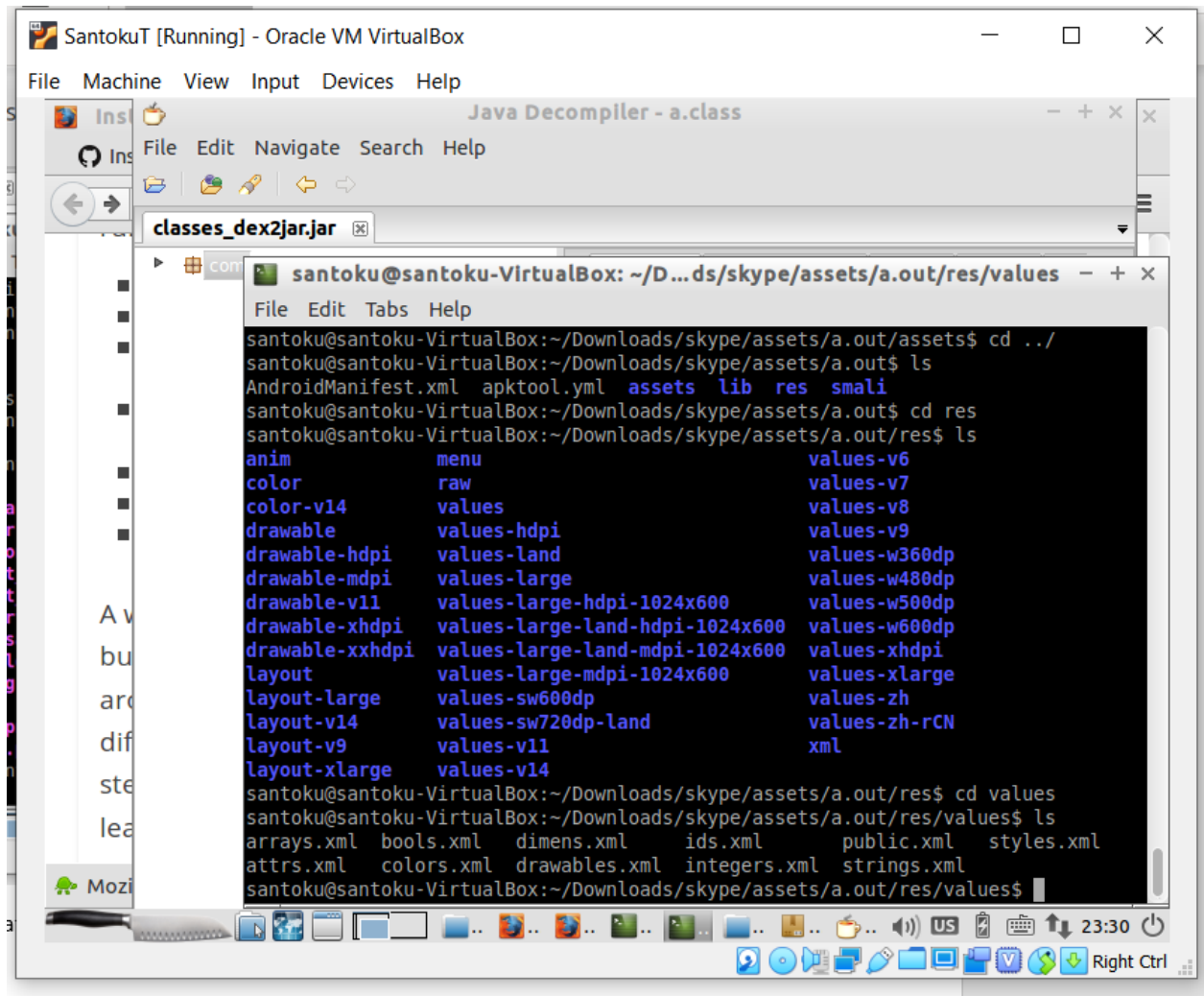


Figure 2.7b

Figure 2.7a and 2.7b above shows the path to open the string file of the inner application which is the application that really executes the malware. We will now open the string file again using the “vim” command as shown below:

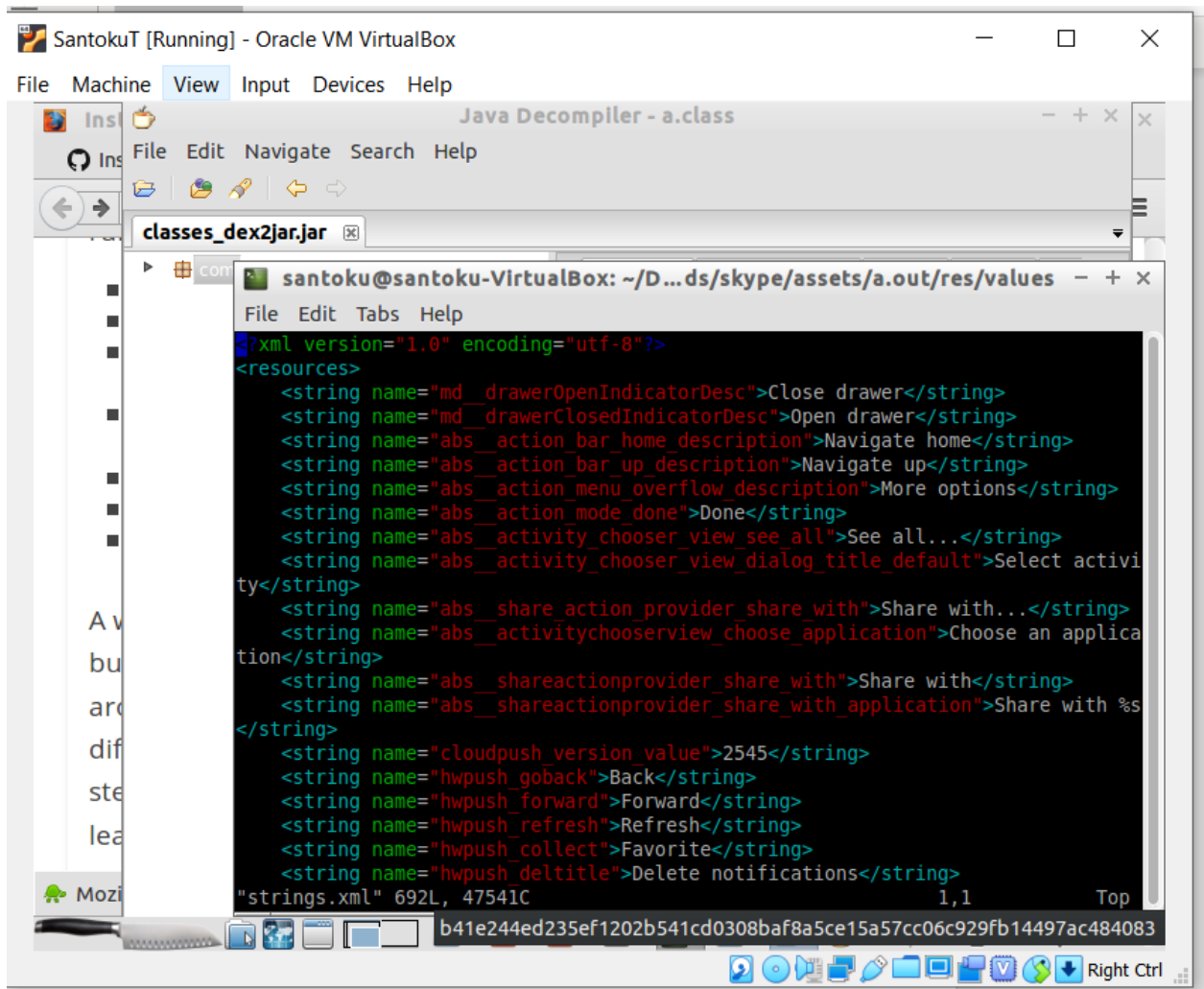


Figure 2.8a

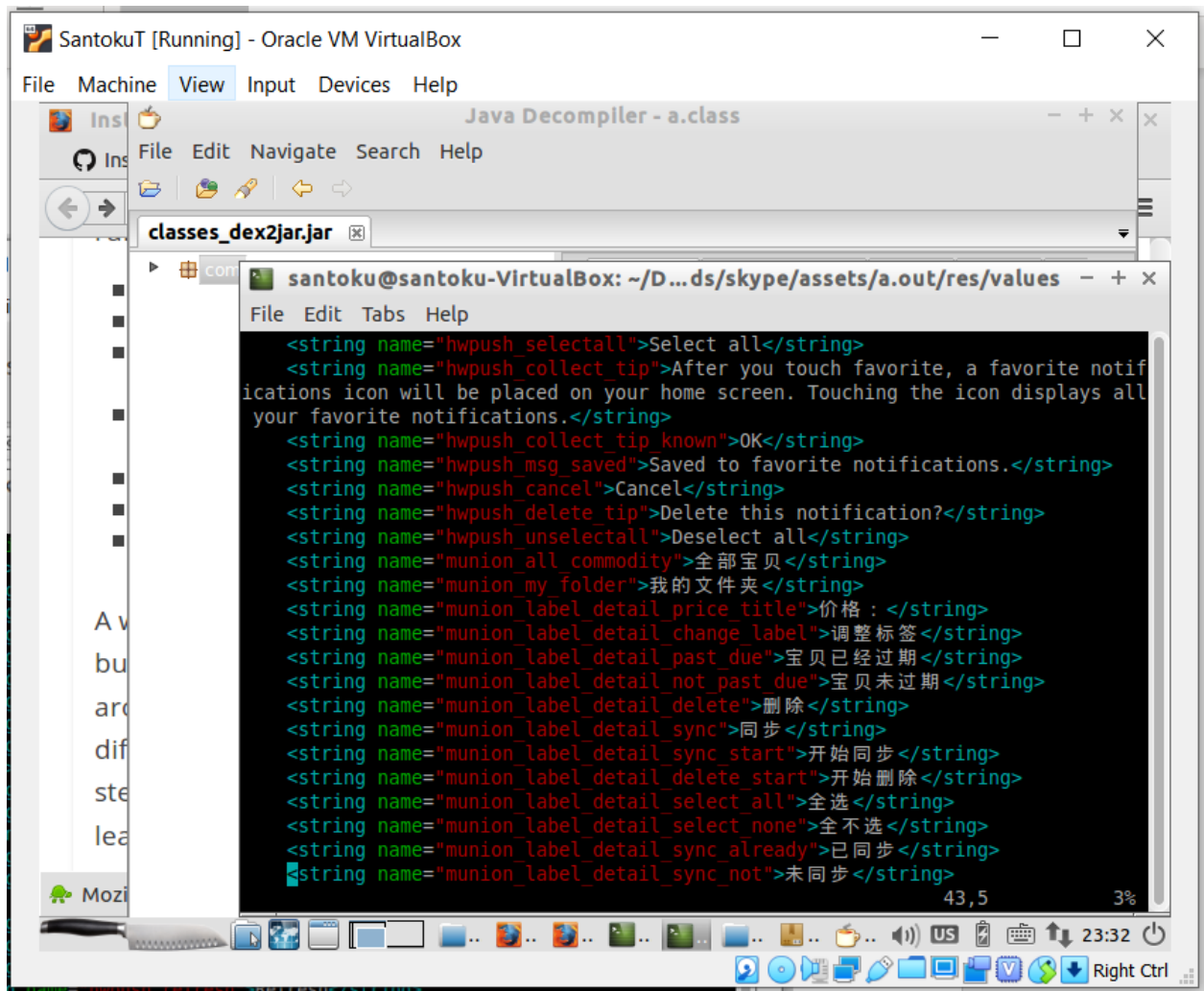


Figure 2.8b

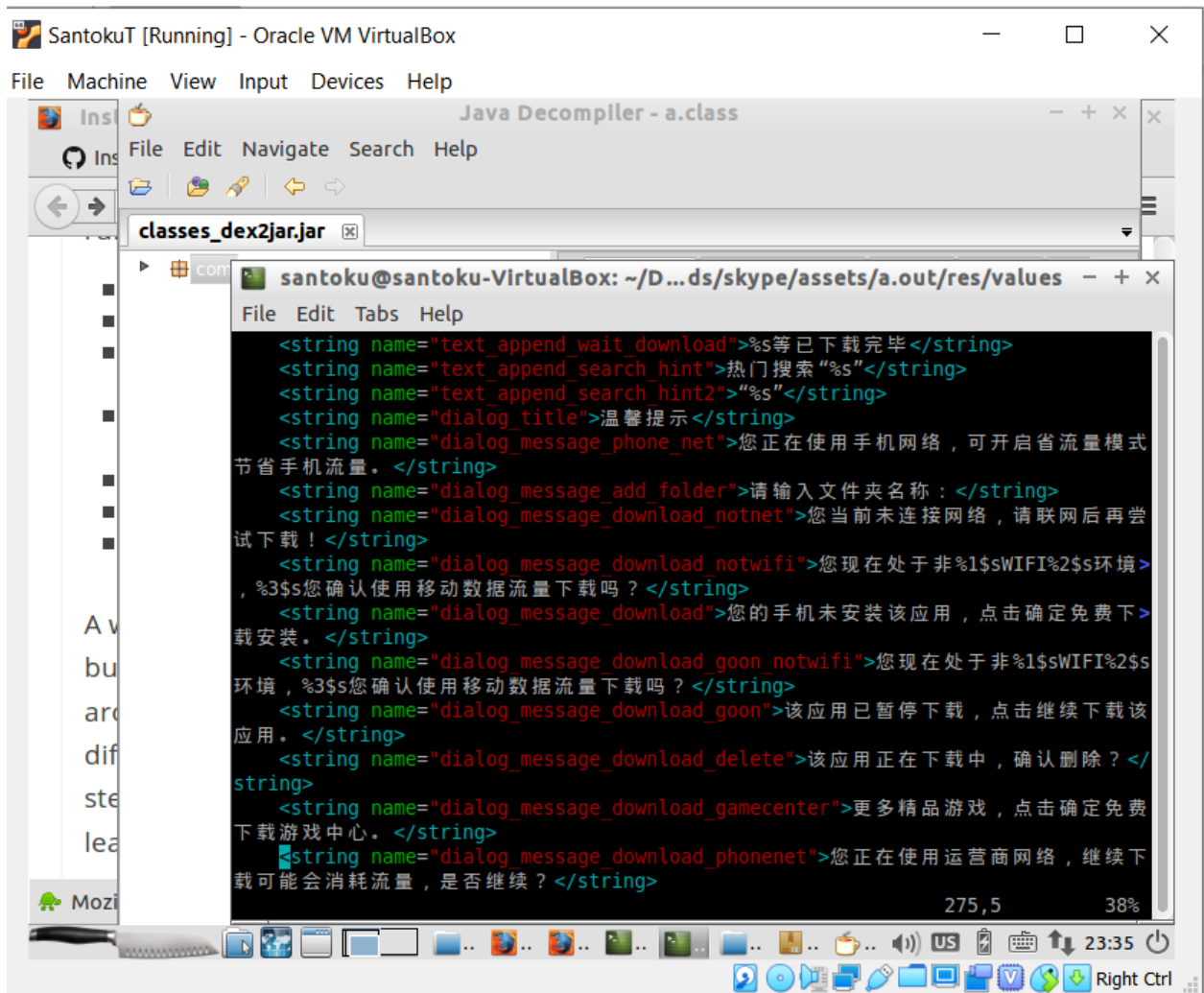


Figure 2.8c

We can see there is a lot going on in this string file of the inner malware application as indicated in figure 2.8a, 2.8b and 2.8c above. The key thing notice here is the present of a lot of translated content and key indication that the malware is coming from a foreign country. There is also the use of a different title for the SD cards references, external storage and message notification. We will now open one of the source codes in this string file of the inner malware application to see if there are any suspicious methods being used:

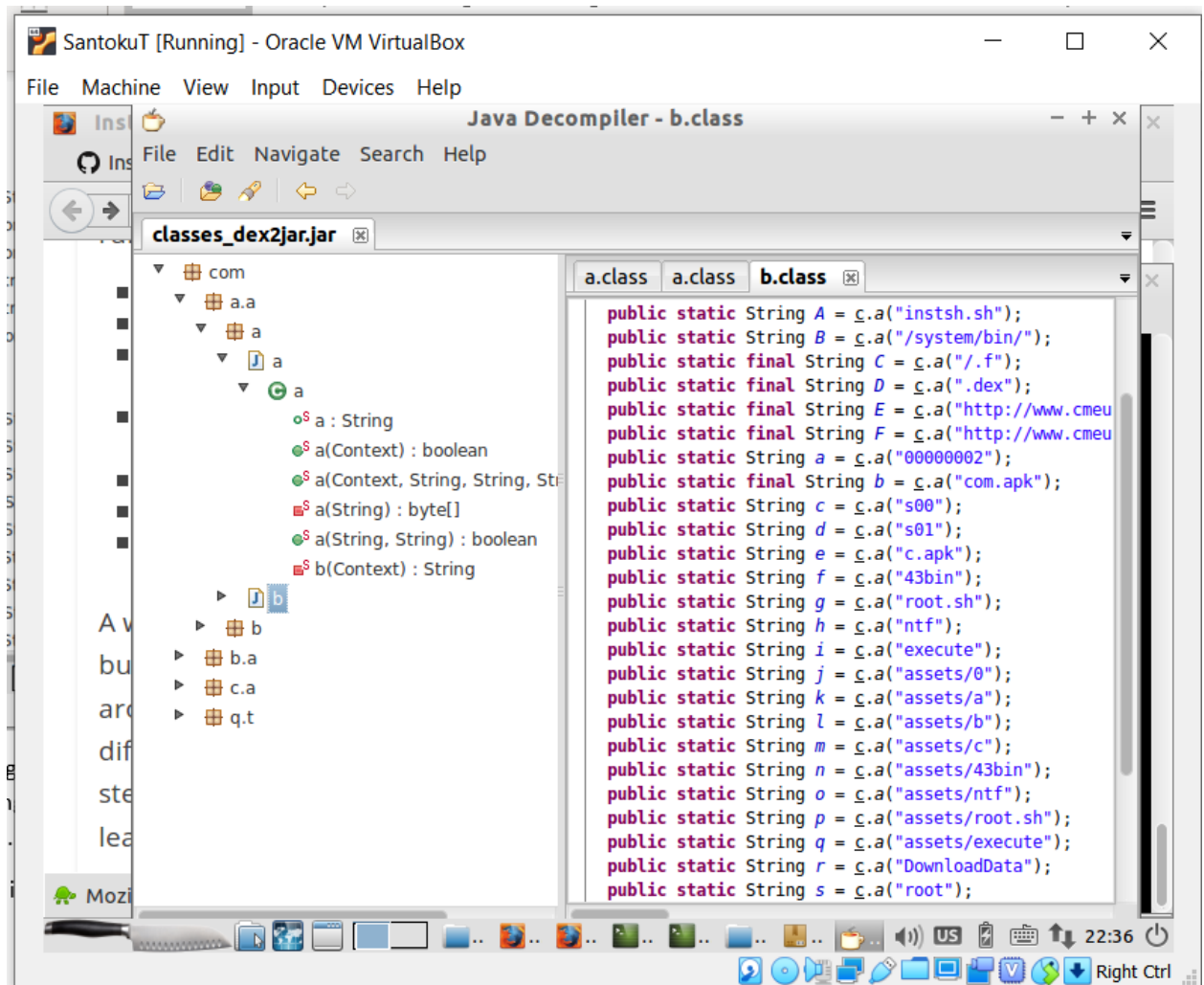


Figure 2.9

We can see in the above code snippet that we have assets like “root.sh” which is the root shell script and “downloadData” and some strange and suspicious URLs and all these leads us to conclude that this is a malware application.

# CONCLUSION

This paper has examined the main methods and techniques of static analysis of an Android malware and one important fact discovered has been that malware developers will try to deceive or fool malware checkers and antivirus softwares with a dummy AndroidManifest.xml file which is the file that shows all the permissions that an android application will request when installed on an android device. We discovered in this malware analysis example that the application had two AndroidManifest.xml files and the inner application was the real deal of the malware which had the real AndroidManifest.xml file and we saw a significant amount of difference between the types and number of permissions being requested by the AndroidManifest.xml file of the inner application. We also came across some suspicious assets and methods that were used by the developers of the malware which gave us a better picture of the functionality of the malware and it should be noted that we can go further to understand more about the behavior of the malware by doing a dynamic analysis which will require us to install the malware application onto the emulator which we created in figure 1.0 above. Dynamic analysis is a more better way to do malware analysis because as we noticed when performing static analysis, most of the code was obfuscated by the developers of the malware and this makes it hard for us to explore and understand the malware application in a more deeper sense. Nevertheless, the goal was to examine and analyze a sample malware application and we were able to do that using static analysis.

# REFERENCES

## References

- Fora, P. O. (2015, april 15). *Beginners Guide to Reverse Engineering Android Apps*. Retrieved from Youtube: <https://www.youtube.com/watch?v=7SRfk321I5o>
- Hoog, A. (2013, November 26). *Mobile app analysis with Santoku Linux*. Retrieved from Youtube: <https://www.youtube.com/watch?v=cmVRCWbo0jU>
- Ninja, S. (n.d.). *Windows Functions in Malware Analysis – Cheat Sheet – Part 1*. Retrieved from Infosec: <https://resources.infosecinstitute.com/windows-functions-in-malware-analysis-cheat-sheet-part-1/>
- skulkin, I. M. (2017, January 24). *cyber forensicator*. Retrieved from Principles of Android Malware Detection: <https://cyberforensicator.com/2017/01/24/principles-of-android-malware-detection/>
- Srinivas. (2016, june 6). *Android Malware Analysis*. Retrieved from Infosec: <https://resources.infosecinstitute.com/android-malware-analysis-2/#gref>



# **ACKNOWLEDGEMENTS**

The following individuals and resources were very helpful in the outcome of this malware analysis research:

## COURSE PROFESSOR

Dr. Akbar Namin

## FUNDING

Department of Computer Science, Texas Tech University

