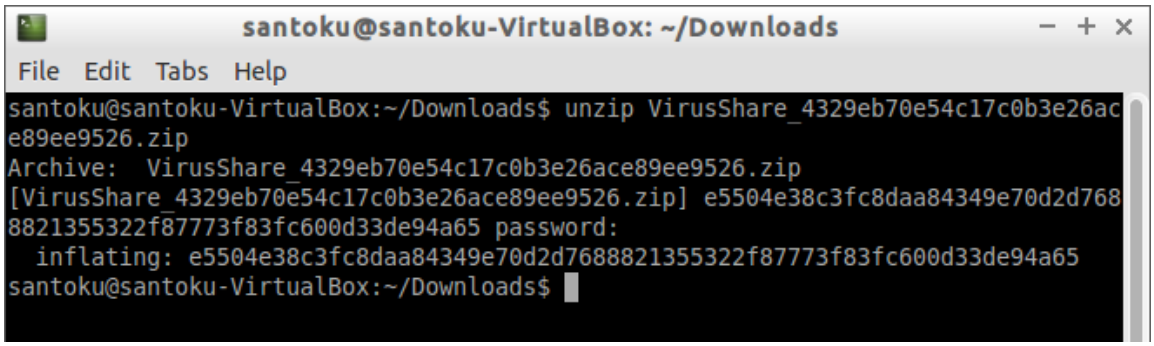
Malware Analysis using Santoku

➢ Getting the APK

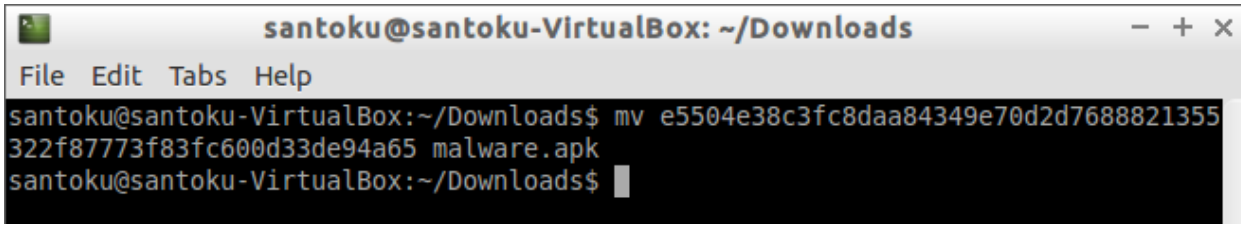
1. The APK used for the analysis can be found at virusshare.com using the following the SHA-256 value as the search term:

   `e5504e38c3fc8daa84349e70d2d7688821355322f87773f83fc600d33de94a65`

2. After downloading the APK, the first step is to **unzip** in order to get the actual malicious file. You can use the unzip command (as below) in the terminal or you can use the built-in package manager to extract the APK file. The password for the file is: **infected**



3. After unzipping the file, the next step is to change the extracted file's extension to **.apk.** To do this we will use the linux **mv** command as follows:



4. Now that we have successfully obtained the APK file, we can being the analysis process.

➢ Analyzing the APK

Santoku has a rich set of tools that can be used for analysis. One such useful tool is called **Androguard.** Santoku 0.5 ships with Androguard 2.0 version. Androguard offers

an interactive python shell to interact with the API through various commands as described in Part A. Part B describes other useful tools in androguard.

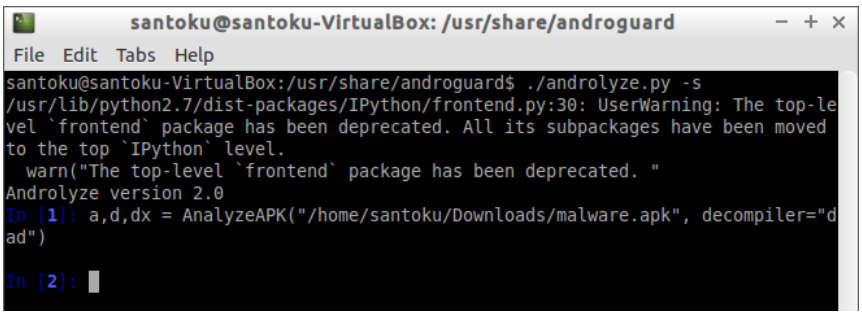
A. Using Androlyze.py

1. To run *androlyze* ipython shell, first navigate to androguard directory located in `/usr/share` directory as follows:
   `$ cd /usr/share/androguard`

2. Next, we will use the androguard's androlyze tool in the interactive python shell. To begin, type the following command in the terminal:
   `$ ./androlyze.py -s`

3. Type the following command to decompile the apk using the default *dad* compiler:

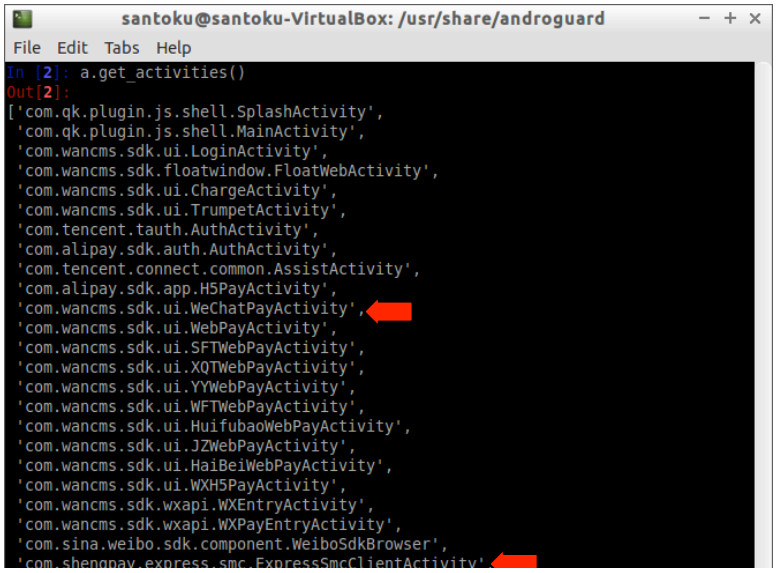
   $



   `a,d,dx = AnalyzeAPK(<apk_file_name>, decompiler="dad")`

4. Let's start by getting the list of the APK's activities by typing a.activities(): It appears that the APK is using some sort of payment apis in addition to offering chat facilities.

5. APK permissions can be examined as below:



```
santoku@santoku-VirtualBox: /usr/share/androguard
File  Edit  Tabs  Help

In [8]: a.get_permissions()
Out[8]:
['android.permission.GET_TASKS',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.INTERNET',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.READ_PHONE_STATE',
 'android.permission.SYSTEM_ALERT_WINDOW',
 'android.permission.ACCESS_CHECKIN_PROPERTIES',
 'android.permission.READ_EXTERNAL_STORAGE',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.SYSTEM_OVERLAY_WINDOW',
 'android.permission.BLUETOOTH',
 'android.permission.READ_PHONE_STATE',
 'android.permission.INTERNET',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.READ_LOGS',
 'android.permission.CHANGE_WIFI_STATE',
 'android.permission.WAKE_LOCK',
 'android.permission.CALL_PHONE',
 'android.permission.MOUNT_UNMOUNT_FILESYSTEMS',
```

a. We can also take a look at the APK permissions:

b. To see more details of what these permissions are we can use the `get_details_permissions()` function:

```
santoku@santoku-VirtualBox: /usr/share/androguard
File  Edit  Tabs  Help

In [9]:  a.get_details_permissions()
Out[9]:
{'android.hardware.camera.autofocus': ['normal',
  'Unknown permission from android reference',
  'Unknown permission from android reference'],
 'android.permission.ACCESS_CHECKIN_PROPERTIES': ['signatureOrSystem',
  'access check-in properties',
  'Allows read/write access to properties uploaded by the check-in service. Not
for use by normal applications.'],
 'android.permission.ACCESS_COARSE_LOCATION': ['dangerous',
  'coarse (network-based) location',
  'Access coarse location sources, such as the mobile network database, to deter
mine an approximate phone location, where available. Malicious applications can
use this to determine approximately where you are.'],
 'android.permission.ACCESS_FINE_LOCATION': ['dangerous',
  'fine (GPS) location',
  'Access fine location sources, such as the Global Positioning System on the ph
one, where available. Malicious applications can use this to determine where you
 are and may consume additional battery power.'],
 'android.permission.ACCESS_NETWORK_STATE': ['normal',
  'view network status',
  'Allows an application to view the status of all networks.'],
 'android.permission.ACCESS_WIFI_STATE': ['normal',
  'view Wi-Fi status',
  'Allows an application to view the information about the status of Wi-Fi.'],
```

6. A *service* is a general entry point for keeping an app running in the background. The APK's services can be obtained using the following command:

```
santoku@santoku-VirtualBox: /usr/share/androguard
File  Edit  Tabs  Help

In [5]: a.get_services()
Out[5]:
['com.wancms.sdk.WancmsSDKAppService',
 'com.sina.weibo.sdk.net.DownloadService']
```

7. A *receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. This can be obtained using the a.receiver() command:

```
In [6]: a.get_receivers()
Out[6]: ['com.wancms.sdk.AppRegister']
```

8. We can also check what Android version the APK is compatible with:

```
In [19]: a.get_androidversion_code()
Out[19]: u'1'

In [20]: a.get_androidversion_name()
Out[20]: u'1.0.0'

In [21]: a.get_min_sdk_version()
Out[21]: u'9'

In [22]: a.get_max_sdk_version()

In [23]: a.get_target_sdk_version()
Out[23]: u'19'
```

Based on the output from `target_sdk_version()` above, the APK was created for Android 4.4 Kitkat i.e. API level 19. You may refer to google's official documentation for more details about different versions here.

9. We can check where the app signature is located. This displays that the Signature/KEY file is located in the META-INF folder of the APK.

```
In [2]: a.get_signature_name()
Out[2]: u'META-INF/KEY.RSA'
```

Malware Analysis using Androguard on Kali Linux

10. To get the app name, we will use a.get_app_name() command as follows:

```
In [25]: a.get_app_name()
Out[25]: '少年西游决'

In [26]:
```

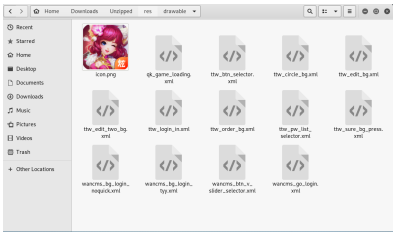
11. Now, let's try to get the app's icon:

```
In [25]: a.get_app_name()
Out[25]: '少年西游决'

In [26]: a.get_app_icon()
Out[26]: 'res/drawable-xxxhdpi/icon.png'

In [27]:
```

12. Let's unzip the apk to a folder called "*Unzipped*" as below

```
root@kali: ~/Downloads
File  Edit  View  Search  Terminal  Help
root@kali:~/Downloads# unzip mal.apk -d Unzipped
Archive:  mal.apk
  inflating: Unzipped/META-INF/MANIFEST.MF
  inflating: Unzipped/META-INF/KEY.SF
  inflating: Unzipped/META-INF/KEY.RSA
  inflating: Unzipped/AndroidManifest.xml
   creating: Unzipped/assets/
  inflating: Unzipped/assets/background.9.png
 extracting: Unzipped/assets/buttonNegt.png
 extracting: Unzipped/assets/buttonPost.png
  inflating: Unzipped/assets/button_green.9.png
  inflating: Unzipped/assets/button_red.9.png
 extracting: Unzipped/assets/com.qk.plugin.qkfx.Manager
 extracting: Unzipped/assets/com.tencent.open.config.json
  inflating: Unzipped/assets/com.tencent.plus.bar.png
 extracting: Unzipped/assets/com.tencent.plus.blue_disable.png
 extracting: Unzipped/assets/com.tencent.plus.blue_down.png
 extracting: Unzipped/assets/com.tencent.plus.blue_normal.png
  inflating: Unzipped/assets/com.tencent.plus.gray_disable.png
  inflating: Unzipped/assets/com.tencent.plus.gray_down.png
```

13. The following is the *unzipped* directory. Let's navigate to "res/drawable-xxxhdpi/icon.png" to see what the icon looks like:



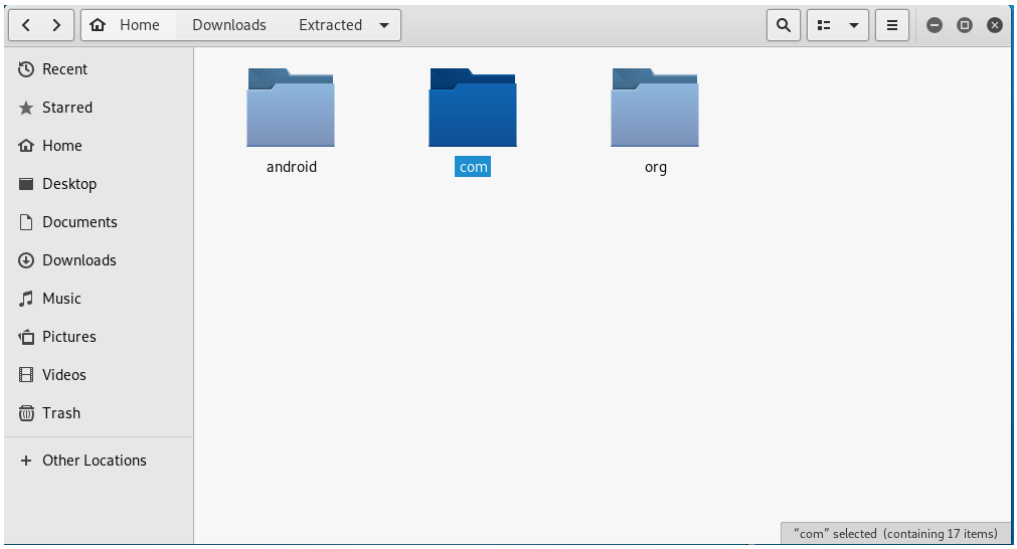
APK Decompilation using Androguard on Kali Linux

1. Generating Control Flow Graphs

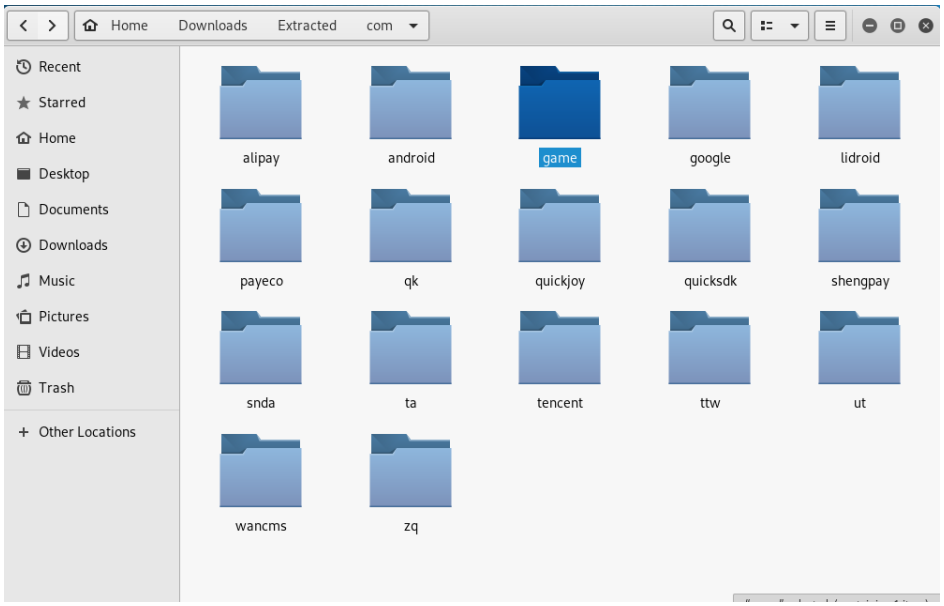
Androguard `decompile` command lets you extract and generate generate the *control flow graphs* for each class in the apk. We will use the png format [1].

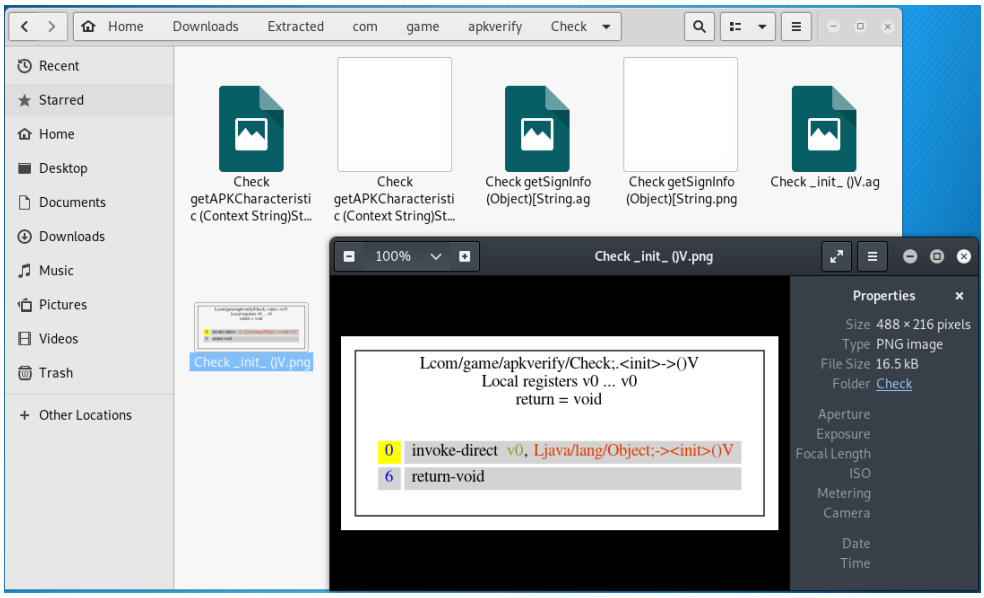
```
Usage: androguard decompile [OPTIONS] [FILE_]

  Decompile an APK and create Control Flow Graphs.

  Example:

      $ androguard resources.arsc

Options:
  -i, --input FILE       APK to parse (legacy option)
  -o, --output TEXT      output directory. If the output folder already
                         exsist, it will be overwritten!  [required]
  -f, --format TEXT      Additionally write control flow graphs for each
                         method, specify the format for example png, jpg, raw
                         (write dot file), ...
  -j, --jar              Use DEX2JAR to create a JAR file
  -l, --limit TEXT       Limit to certain methods only by regex (default:
                         '.*')
  -d, --decompiler TEXT  Use a different decompiler (default: DAD)
  --help                 Show this message and exit.
```

After decompilation, our Apk structure looks the following:



Exploring the `game` folder inside `com`, we can find a control flow graph as follows:





[1] https://androguard.readthedocs.io/en/latest/tools/androdd.html

2. Generating Call Graph

Androguard also has the ability to generate call graph using the `cg` command [2] as follows:

```
Usage: androguard cg [OPTIONS] APK

  Create a call graph and export it into a graph format.

  The default is to create a file called callgraph.gml in the current
  directory!

  classnames are found in the type "Lfoo/bar/bla;".

  Example:

      $ androguard cg examples/tests/hello-world.apk

Options:
  -o, --output TEXT           Filename of the output file, the extension is
                              used to decide which format to use  [default:
                              callgraph.gml]
  -s, --show                  instead of saving the graph, print it with
                              mathplotlib (you might not see anything!)
  -v, --verbose               Print more output
  --classname TEXT            Regex to filter by classname  [default: .*]
  --methodname TEXT           Regex to filter by methodname  [default: .*]
  --descriptor TEXT           Regex to filter by descriptor  [default: .*]
  --accessflag TEXT           Regex to filter by accessflags  [default: .*]
  --no-isolated / --isolated  Do not store methods which has no xrefs
  --help                      Show this message and exit.
```

[2] https://androguard.readthedocs.io/en/latest/tools/androcg.html

B. Using Other tools in Androguard: Androaxml, APKinfo and Androdd

1. Using Androaxml:
   Every APK file has a *AndroidManifest.xml* file which contains all the list of permissions and activities that we saw in Part A. By default, this file is not readable in text editors. To parse it, we will use `Androaxml` command to make it readable. To avoid typing the complete path to APK file in the downloads folder, we will make a copy in the androguard's directory and then run the andoraxm command as below:

```
santoku@santoku-VirtualBox: /usr/share/androguard
File  Edit  Tabs  Help
santoku@santoku-VirtualBox:/usr/share/androguard$ cp /home/santoku/Downloads/mal
ware.apk .
santoku@santoku-VirtualBox:/usr/share/androguard$ ls
ag-st            androdiff.py        andromercury.py  CHANGELOG     malware.apk
androapkinfo.py  androdis.py         androrisk.py     demos         README.txt
androarsc.py     androdump.py        androsign.py     dist          setup.py
androauto.py     androgexf.py        androsim.py      elsim         signatures
androaxml.py     androguard          androxgmml.py    examples      tests
androcsign.py    androguard.egg-info apkviewer.py     LICENCE-2.0   tools
androdd.py       androlyze.py        build            Makefile
santoku@santoku-VirtualBox:/usr/share/androguard$ ./androaxml.py -i malware.apk
-o output.xml
santoku@santoku-VirtualBox:/usr/share/androguard$
```

The structure of AndroidManifest.xml (output.xml) can be seen below:

Mozilla Firefox

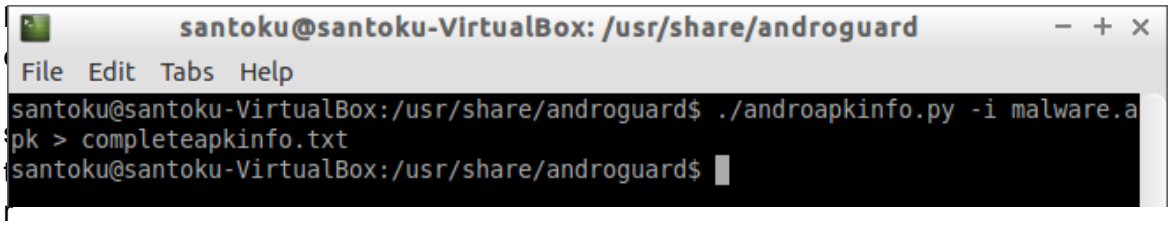
/media/sf_santoku/Output

file:///media/sf_santoku/Output.xml

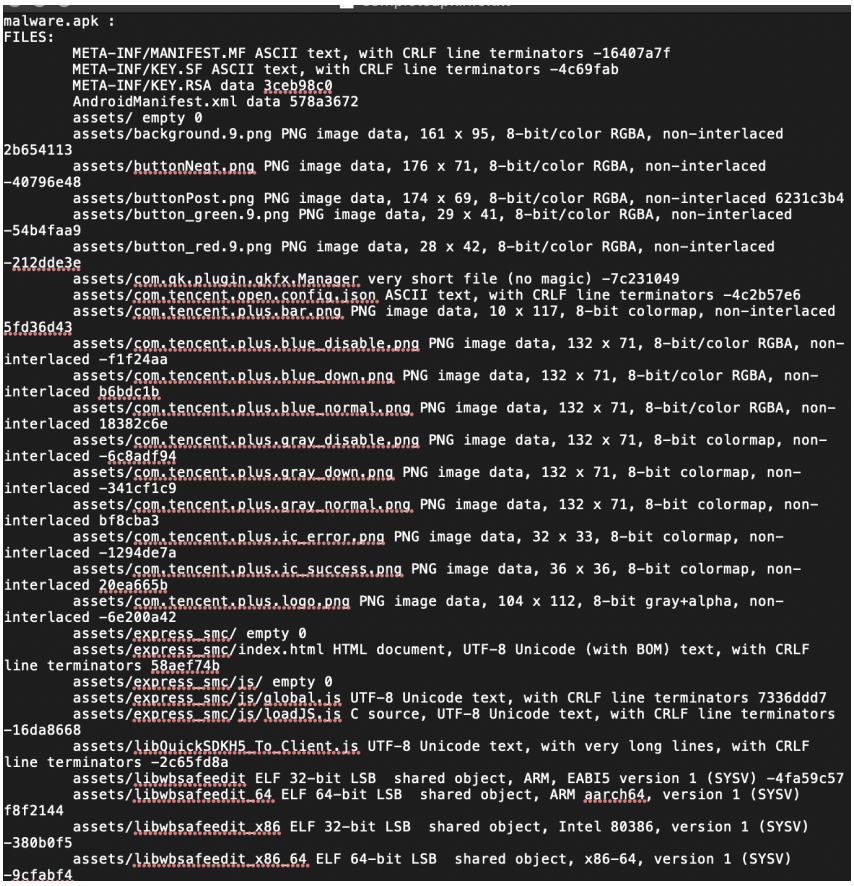
This XML file does not appear to have any style information associated with it. The document tree is shown below.


```
- <manifest android:versionCode="1" android:versionName="1.0.0"
  package="com.zq.snxyj.xuanwan" platformBuildVersionCode="23"
  platformBuildVersionName="6.0-2438415">
    <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="19"> </uses-sdk>
    <uses-permission android:name="android.permission.GET_TASKS"> </uses-
    permission>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE">
    </uses-permission>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"> </uses-
    permission>
    <uses-permission android:name="android.permission.INTERNET"> </uses-
    permission>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE">
    </uses-permission>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"> </uses-
    permission>
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW">
    </uses-permission>
    <uses-permission
```

2. Using APKInfo:
   We can also obtain a complete list of APK permissions, files, activities, services and receivers in one single text file using the `apkinfo` command:
   T



   ucture of completeapkinfo.txt is shown below:



```
malware.apk :
FILES:
        META-INF/MANIFEST.MF ASCII text, with CRLF line terminators -16407a7f
        META-INF/KEY.SF ASCII text, with CRLF line terminators -4c69fab
        META-INF/KEY.RSA data 3ceb98c0
        AndroidManifest.xml data 578a3672
        assets/ empty 0
        assets/background.9.png PNG image data, 161 x 95, 8-bit/color RGBA, non-interlaced
2b654113
        assets/buttonNegt.png PNG image data, 176 x 71, 8-bit/color RGBA, non-interlaced
-40796e48
        assets/buttonPost.png PNG image data, 174 x 69, 8-bit/color RGBA, non-interlaced 6231c3b4
        assets/button_green.9.png PNG image data, 29 x 41, 8-bit/color RGBA, non-interlaced
-54b4faa9
        assets/button_red.9.png PNG image data, 28 x 42, 8-bit/color RGBA, non-interlaced
-212dde3e
        assets/com.gk.plugin.gkfx.Manager very short file (no magic) -7c231049
        assets/com.tencent.open.config.json ASCII text, with CRLF line terminators -4c2b57e6
        assets/com.tencent.plus.bar.png PNG image data, 10 x 117, 8-bit colormap, non-interlaced
5fd36d43
        assets/com.tencent.plus.blue_disable.png PNG image data, 132 x 71, 8-bit/color RGBA, non-
interlaced -f1f24aa
        assets/com.tencent.plus.blue_down.png PNG image data, 132 x 71, 8-bit/color RGBA, non-
interlaced b6bdc1b
        assets/com.tencent.plus.blue_normal.png PNG image data, 132 x 71, 8-bit/color RGBA, non-
interlaced 18382c6e
        assets/com.tencent.plus.gray_disable.png PNG image data, 132 x 71, 8-bit colormap, non-
interlaced -6c8adf94
        assets/com.tencent.plus.gray_down.png PNG image data, 132 x 71, 8-bit colormap, non-
interlaced -341cf1c9
        assets/com.tencent.plus.gray_normal.png PNG image data, 132 x 71, 8-bit colormap, non-
interlaced bf8cba3
        assets/com.tencent.plus.ic_error.png PNG image data, 32 x 33, 8-bit colormap, non-
interlaced -1294de7a
        assets/com.tencent.plus.ic_success.png PNG image data, 36 x 36, 8-bit colormap, non-
interlaced 20ea665b
        assets/com.tencent.plus.logo.png PNG image data, 104 x 112, 8-bit gray+alpha, non-
interlaced -6e200a42
        assets/express_smc/ empty 0
        assets/express_smc/index.html HTML document, UTF-8 Unicode (with BOM) text, with CRLF
line terminators 58aef74b
        assets/express_smc/js/ empty 0
        assets/express_smc/js/global.js UTF-8 Unicode text, with CRLF line terminators 7336ddd7
        assets/express_smc/js/loadJS.js C source, UTF-8 Unicode text, with CRLF line terminators
-16da8668
        assets/libQuickSDKH5_To_Client.js UTF-8 Unicode text, with very long lines, with CRLF
line terminators -2c65fd8a
        assets/libwbsafeedit ELF 32-bit LSB  shared object, ARM, EABI5 version 1 (SYSV) -4fa59c57
        assets/libwbsafeedit_64 ELF 64-bit LSB  shared object, ARM aarch64, version 1 (SYSV)
f8f2144
        assets/libwbsafeedit_x86 ELF 32-bit LSB  shared object, Intel 80386, version 1 (SYSV)
-380b0f5
        assets/libwbsafeedit_x86_64 ELF 64-bit LSB  shared object, x86-64, version 1 (SYSV)
-9cfabf4
```