Vetting Malicious Android Apps

Rohith Mulumudy

December 16, 2023

1 Introduction

The write-up involves the analysis of three well-known Android malwares:

- Viking Jump [5]
- Compass [6]
- Coronavirus Tracker [4]

For these three apps, I conducted both static and dynamic analyses. In static analysis, I generated call graphs using Android activity lifecycle APIs as entry points and network API calls as endpoints with the help of soot[8].

In dynamic analysis, I utilized Frida[7] to hook network calls, enabling the logging of API names along with the parameters passed to them.

Firstly, I will introduce all the network system APIs available in the Android framework in Section 2, These concepts will play a crucial role in both static and dynamic analyses. Subsequently, I will present my findings on all three malwares in Sections 4, 5, 6. Following that, I will outline the obstacles encountered during the analysis in section 7

2 Network APIs

In this section, I will delve into the various network packages accessible within the Android framework:

- java.net 2.1
- javax.net 2.2
- javax.net.ssl 2.3
- android.net 2.4

For each of the aforementioned packages, I will provide an overview of all the APIs facilitating the reading and sending of data, initializing a connection, and methods to extract the API name.

2.1 java.net

In this section, we categorize the java.net package into two sections: 1) Low-Level API and 2) High-Level APIs. The Low-Level APIs consist of the **Socket** and **DatagramSocket** classes:

- Socket
  - getInputStream() # Returns an input stream for this socket.
  - getOutputStream() # Returns an output stream for this socket.
  - getInetAddress() # Returns the address to which the socket is connected
  - getPort() # Returns the remote port number to which this socket is connected.
- DatagramSocket
  - send() # Sends a datagram packet from this socket.
  - receive() # Receives a datagram packet from this socket.
  - getInetAddress() # Returns the address to which the socket is connected
  - getPort() # Returns the remote port number to which this socket is connected.

I will be concentrating on **Socket** class in this write-up. Any required information on the **DatagramSocket** class, please refer to the Java.net Oracle documentation[1].

In the Socket class, identifying the input stream and output stream assists us in identifying sections of code within the malware that communicate with the C&C server or other resources. The getInetAddress() and getPort() methods allow us to retrieve the remote host and port during dynamic analysis.

The following methods are instrumental in initializing a Socket class, and by identifying these functions, the host address can be ascertained:

- Socket(InetAddress address, int port) # Creates a stream socket and connects it to the specified port number at the specified IP address.
- Socket(String host, int port) # Creates a stream socket and connects it to the specified port number on the named host.
- Socket().connect(SocketAddress endpoint) # Connects this socket to the server.
- Socket().connect(SocketAddress endpoint, int timeout) # Connects this socket to the server with a specified timeout value.

While I have provided the key methods, it's important to note that there are additional methods for connecting to a remote host during the initialization of a **Socket** class. For a comprehensive understanding, one can refer to the official documentation.[1]

Moving on to the High-Level APIs, three classes - **HTTPURLConnection**, **URL**, and **URLConnection** play a crucial role in host communication

- URL
  - openConnection() # Returns a URLConnection instance that represents a connection to the remote object referred to by the URL.
  - openStream() # Opens a connection to this URL and returns an InputStream for reading from that connection.
  - toString() # Constructs a string representation of this URL.
- URLConnection
  - getInputStream() # Returns an input stream for this socket.
  - getOutputStream() # Returns an output stream for this socket.
  - getURL() #Returns the value of this URLConnection's URL field.
- HTTPURLConnection # Similar to URLConnection, but specific to HTTP URLs only.

For dynamic analysis, the methods URLConnection().getURL(), URL().toString() facilitate the extraction of the host address. The following methods are instrumental in initializing a URL, URLConnection, and HttpURLConnection classes, and by identifying these functions, the host address can be ascertained:

- URL(String spec) # Creates a URL object from the String representation.
- URL(String protocol, String host, int port, String file) # Creates a URL object from the specified protocol, host, port number, and file.
- URLConnection(URL url) # Constructs a URL connection to the specified URL.
- HttpURLConnection(URL u) # Constructor for the HttpURLConnection.

2.2 javax.net

The package provides a **SocketFactory** class that incorporates a **createSocket** method. This method returns a socket object, and all the methods of the socket, as mentioned in section 2.1, can be utilized for analysis. For more detailed information, please refer to the official documentation[2].

The following method is particularly useful for identifying the host address and port:

- createSocket(InetAddress host, int port) # Creates a socket and connects it to the specified port number at the specified address.

2.3 javax.net.ssl

The following classes are instrumental in establishing a connection with a remote host:

- HttpsURLConnection # HttpsURLConnection extends HttpURLConnection2.1 with support for https-specific features.

- SSLSocket # This class extends Sockets and provides secure socket using protocols such as the SSL, IETF, or TLS protocols. Has same methods as Socket class2.1
- SSLSocketFactory # create SSLSockets.
- SSLContext # Instances of this class represent a secure socket protocol implementation which acts as a factory for secure socket factories or SSLEngines.

Useful methods in the mentioned classes include:

- SSLSocketFactory
  - createSocket(Socket s, String host, int port, boolean autoClose) # Returns a socket layered over an existing socket connected to the named host, at the given port.
- SSLContext
  - getSocketFactory() # Returns a SocketFactory object for this context.

The following methods can be useful to identify the peer address.

- HttpsURLConnection(URL url) # Creates an HttpsURLConnection using the URL specified.
- SSLSocket(InetAddress address, int port)
- SSLSocket(String host, int port)
- SSLSocket().connect(SocketAddress endpoint) # Connects this socket to the server
- SSLSocketFactory().createSocket(Socket s, String host, int port, boolean autoClose) # Returns a socket layered over an existing socket connected to the named host, at the given port.
- SSLContext(SSLContextSpi contextSpi, Provider provider, String protocol) # Creates an SSLContext object.
- SSLContext().createSSLEngine(String peerHost, int peerPort) # Creates a new SSLEngine using this context using advisory peer information.

For further information on classes like SSLContextSpi refer the documentation[3]

2.4 android.net

The classes **LocalSocket**, **Network**, and **SSLCertificateSocketFactory** within the android.net package are valuable for establishing connections with a remote host. Here are some useful methods from the mentioned classes:

- LocalSocket
  - getInputStream() # Returns an input stream for this socket.
  - getOutputStream() # Returns an output stream for this socket.
- Network

- getSocketFactory() # Returns a SocketFactory2.1 bound to this network.

- SSLCertificateSocketFactory # Depricated from android api level 29, and since this class is not widely used, I am not describing it.

The following methods can be useful to identify the peer address during dynamic analysis.

- Network().openConnection(URL url) # Opens the specified URL on this Network, such that all traffic will be sent on this Network.
- LocalSocket().getRemoteSocketAddress() # Retrieves the name that this socket is connected to, if any.
- LocalSocket().connect(LocalSocketAddress endpoint, int timeout)
- LocalSocket().connect(LocalSocketAddress endpoint)

3 Methodology

For my analysis, I selected three different APKs. My initial phase involved static analysis for each APK, where I employed Soot to generate call graphs. The code for call graph generation can be accessed here

For dynamic analysis, I utilized Frida. I referenced the generated call graphs to identify the network libraries used in each APK. Subsequently, I employed this information to craft Frida scripts. The script for identifying URLs can be found here

It's worth noting that the script is not fully equipped to identify payloads, and I haven't included all network libraries in the analysis due to the relatively low number of network calls made by the APKs under investigation

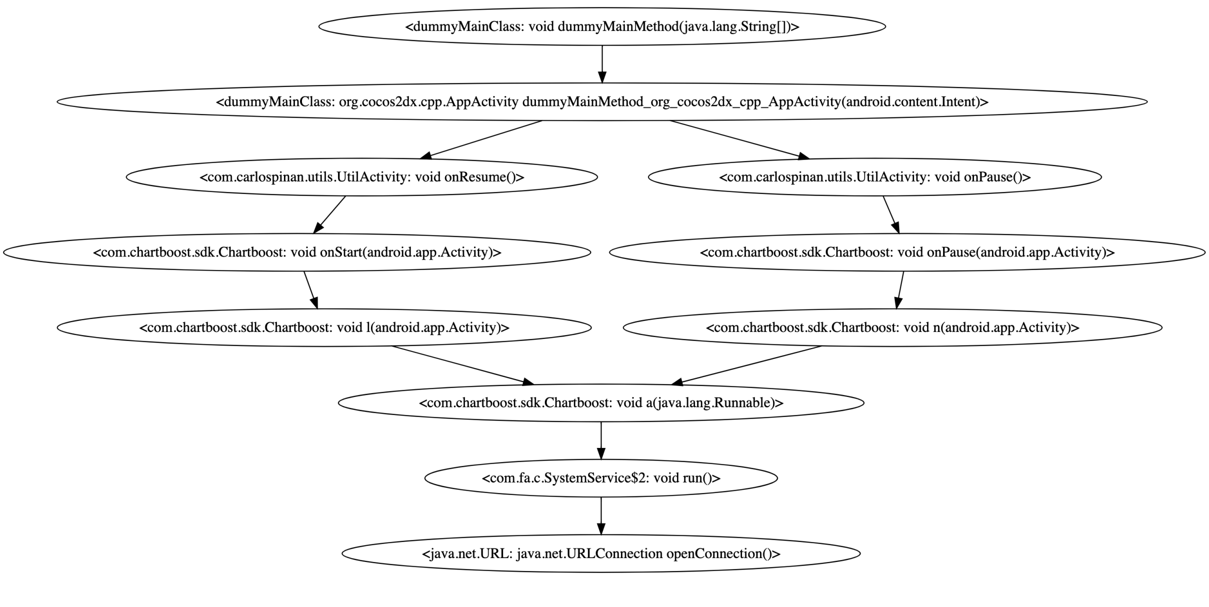

Figure 1: The trimmed down callgraph - Viking app

4 Viking Jump App

Callgraph link: click here

The dot graph for the Viking app is too huge and cannot be viewed in a GUI. One can use the following script to trim down the graph: *link to the script*. The script will generate a trimmed down dot graph, given an end node name and the maximum number of paths. A trimmed down dot graph can be seen in the figure 1

Unfortunately, I was unable to run Frida on top of the Viking app. The APK, I obtained was only supported for the ARM ABI, and emulators on Mac are compatible only with X86.

5 Compass App

Callgraph link: click here

The compass app is making the following API calls:

1. `http://ad.leadboltapps.net/optin?&section_id=648709860&mode=0`
2. `http://ad.leadboltapps.net/optin?&section_id=687484172&mode=0`
3. `https://api.airpush.com`
4. `142.0.206.124`
5. `http://www.umeng.com`
6. `3.141.96.53`
7. `http://manage.airpush.com/sdkpages/sdkpages/bundled-eula2.html`
8. `59.82.31.154`
9. `https://api.airpush.com/optin/`
10. `http://www.umeng.com/app_logs`
11. `http://www.umeng.com/check_config_update`
12. `http://www.umeng.com/api/check_app_update`
13. `fe80::5054:ff:fe12:3456%eth0`
14. `https://api.airpush.com/v2/api.php`
15. `http://ad.leadboltapps.net`

Further analysis can be made to identify the payload.

6 Coronavirus Tracker

Callgraph link: click here

During dynamic analysis the frida script throws the following msg and exits: **unable to find a front-door activity**. Further analysis need to done to investigate the malware.

7 Obstacles

- Identifying a well-crafted malware APK is not straightforward.
- I wasted a significant amount of time attempting to install an Android emulator in a virtual machine.
- I am unable to write a Soot script to identify hardcoded URLs during static analysis phase only.

.

References

[1] URL: `https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html`.

[2] URL: `https://docs.oracle.com/javase/8/docs/api/javax/net/package-summary.html`.

[3] URL: `https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/package-summary.html`.

[4] *com.device.security*. URL: `https://github.com/cryptwareapps/Malware-Database/blob/main/Malware/Android/Ransomware/CovidLockRansomware.apk`.

[5] *com.Jump.vikingJump*. URL: `https://github.com/ytisf/theZoo/tree/master/malware/Binaries/Android.VikingHorde`.

[6] *com.lu.compass*. URL: `https://github.com/cryptwareapps/Malware-Database/blob/main/Malware/Android/Miscellaneous/f901fd1fc2ce079a18c619e1192b14dcc164c97da3 apk`.

[7] *frida*. URL: `https://frida.re/docs/javascript-api/#java`.

[8] *soot*. URL: `https://github.com/noidsirius/SootTutorial`.