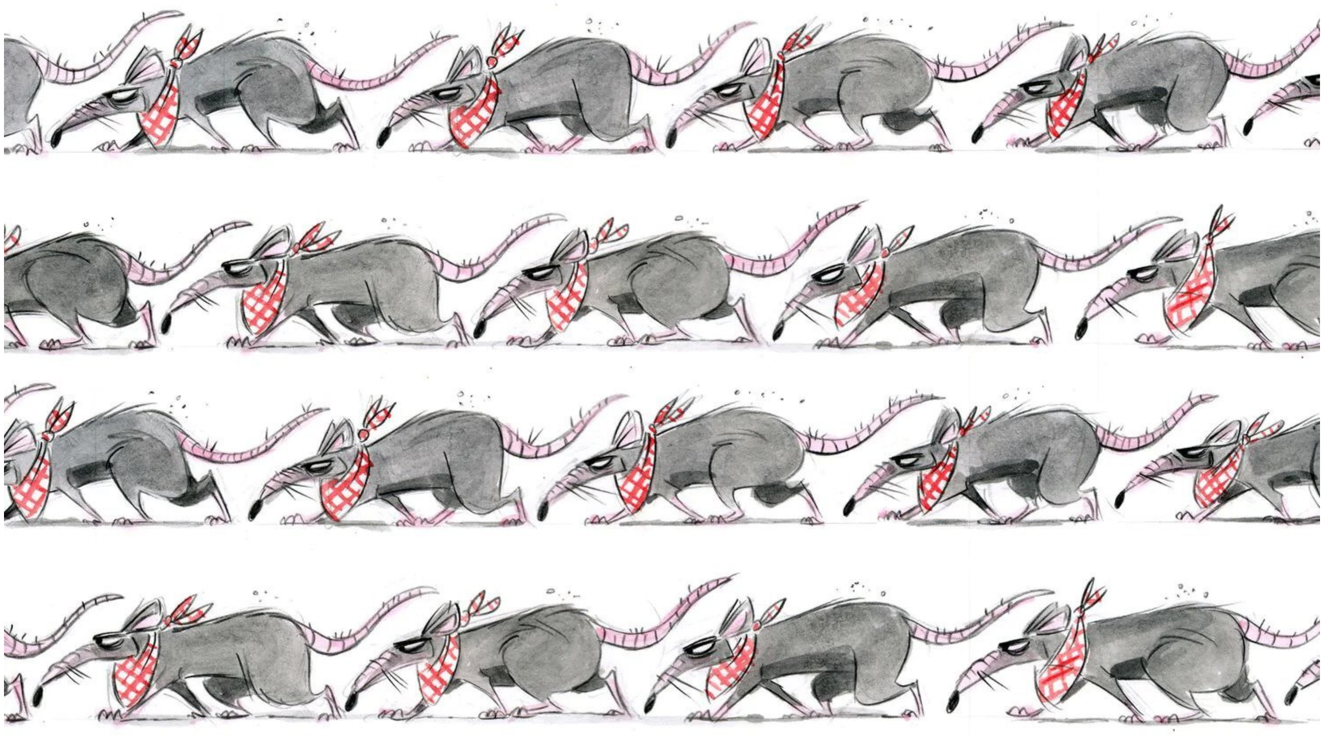
Dissecting a RAT. Analysis of the AndroRAT.

stratosphereips.org/blog/2021/3/29/dissecting-a-rat-analysis-of-the-androrat

Kamila Babayeva

March 31, 2021



This blog post was authored by Kamila Babayeva (@_kamifai_) and Sebastian Garcia (@eldracote).

The RAT analysis research is part of the Civilsphere Project (https://www.civilsphereproject.org/), which aims to protect the civil society at risk by understanding how the attacks work and how we can stop them. Check the webpage for more information.

This is the fourth blog of a series analyzing the network traffic of Android RATs from our Android Mischief Dataset [more information here], a dataset of network traffic from Android phones infected with Remote Access Trojans (RAT). In this blog post we provide the analysis of the network traffic of the RAT05-AndroRAT [download here]. The previous blogs analyzed Android Tester RAT, DroidJack RAT, and SpyMax RAT.

RAT Details and Execution Setup

The goal of each of our RAT experiments is to configure and execute the RAT software and to do every possible action while capturing all traffic and storing all logs. These RAT captures are functional and used as in real attacks.

The AndroRAT RAT is a software package that contains the controller software and builder software to create an APK. We executed the builder on a Windows 7 Virtualbox virtual machine with Ubuntu 20.04 as a host. The Android Application Package (APK) built by the RAT builder was installed in an Android virtual emulator called Genymotion with Android version 8.

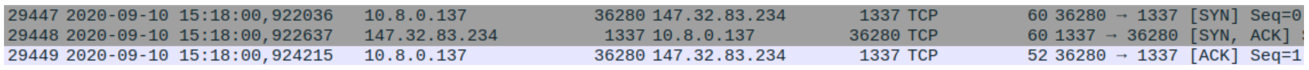
While performing different actions on the RAT controller (e.g. upload a file, get GPS location, monitor files, etc.), we captured the network traffic on the Android virtual emulator. The network traffic from the phone was captured using Emergency VPN.

The details about the network traffic capture are:

- The controller IP address: 147.32.83.234
- The phone IP address: 10.8.0.137
- UTC time of the infection in the capture: 2020-09-10 15:18:00 UTC

Initial Communication and Infection

Once the APK was installed on the phone, it directly tries to establish a TCP connection with the command and control (C&C) server. To connect, the phone uses the IP address and the port of the controller specified in the APK. In our case, the IP address of the controller is 147.32.83.234 and the port is 1337/TCP. The controller IP 147.32.83.234 is the IP address of a Windows 7 virtual machine in our lab computer, meaning that the IP address is not connected to any known indicator of compromise (IoC). Figure 1 shows the initial communication from the phone to the C&C.



```
29447 2020-09-10 15:18:00,922036   10.8.0.137          36280 147.32.83.234       1337 TCP        60 36280 → 1337 [SYN] Seq=0
29448 2020-09-10 15:18:00,922637   147.32.83.234        1337 10.8.0.137         36280 TCP        60 1337 → 36280 [SYN, ACK]
29449 2020-09-10 15:18:00,924215   10.8.0.137          36280 147.32.83.234       1337 TCP        52 36280 → 1337 [ACK] Seq=1
```

Figure 1. A 3-way handshake to establish the first connection between the phone and the C&C.

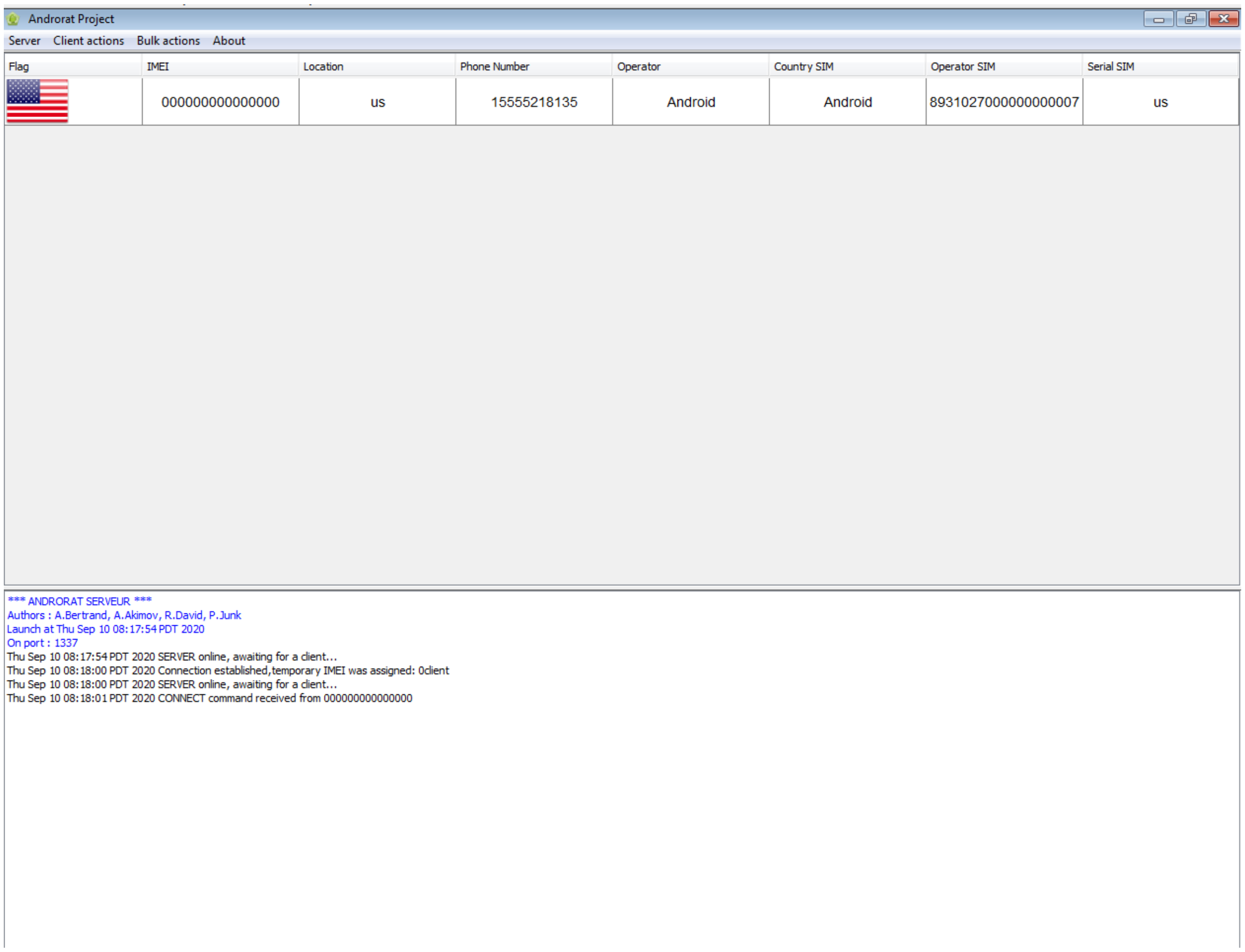
After establishing the first connection, the phone sends its first packet with some parameters, such as SIM card operator, phone number, SIM card serial number, IMEI, etc. Figure 2 displays the packet data in a structured way. It can be seen that the data is sent in plaintext and the character ‘t’ is used as the delimiters to separate parameters name and values. From the packet structure in Figure 2,it can also be defined that APK uses the Java Hashtable class to store and send parameters.

ÿÿ¬ísr**java.util.Hashtable**»%!Jä¸F
loadFactorI thresholdxp?@
w

t Operator t Android
t SimOperator t Android
t SimSerial t 89310270000000000007
t SimCountry t us
t PhoneNumber t 15555218135
t Country t us
t IMEI t 000000000000000

x
Figure 2. The first data packet sent by the phone and an analysis of its structure. The data is sent in the plain text and the character ‘t’ is used as a field delimiter.

After the initial connection by the phone, the command and control server shows the phone in its interface. Figure 3 displays the C&C interface with the initialization parameters that were sent by the phone in the first packet.



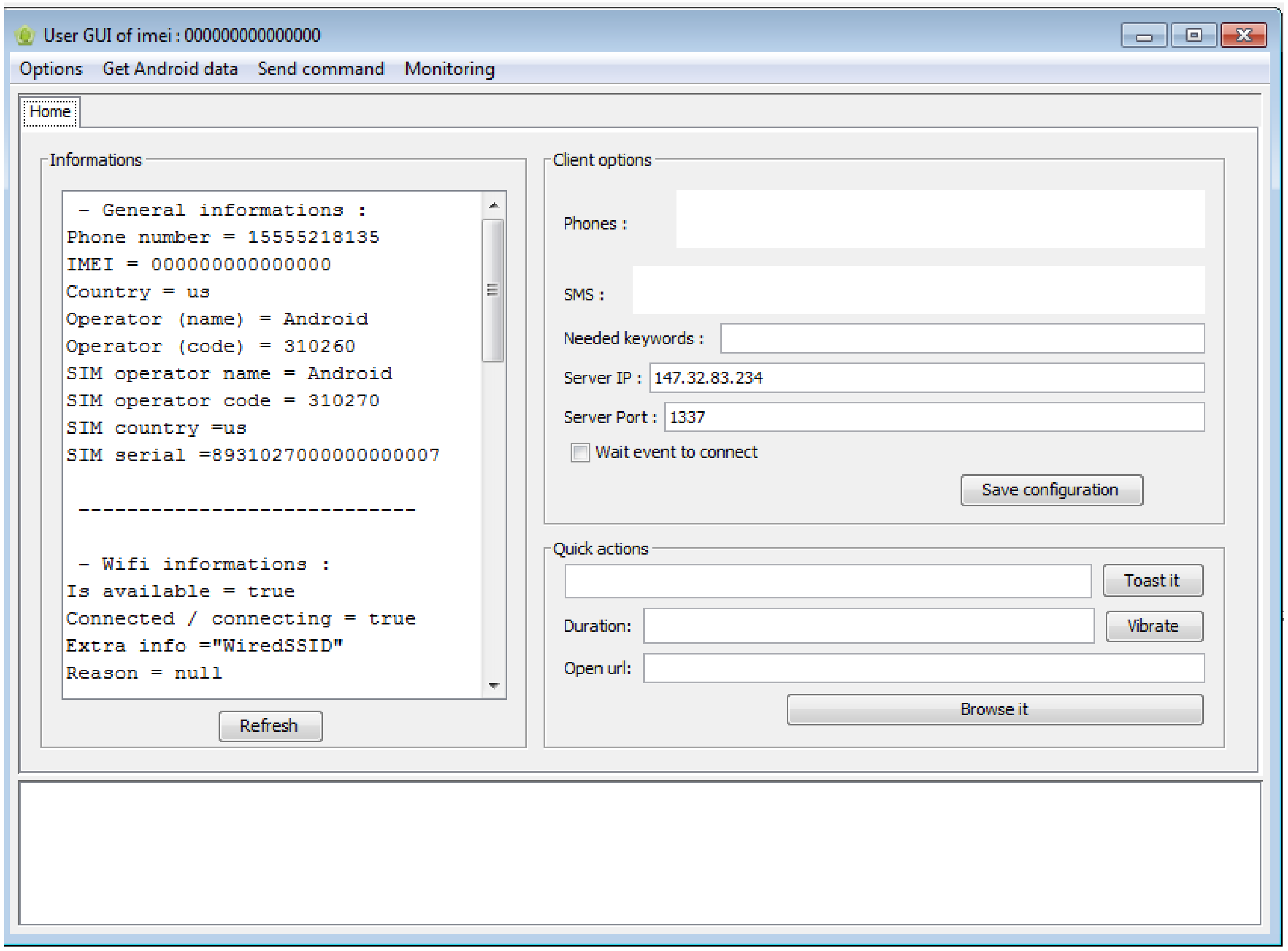
Flag	IMEI	Location	Phone Number	Operator	Country SIM	Operator SIM	Serial SIM
🇺🇸	000000000000000	us	15555218135	Android	Android	89310270000000000007	us

```
*** ANDRORAT SERVEUR ***
Authors : A.Bertrand, A.Akimov, R.David, P.Junk
Launch at Thu Sep 10 08:17:54 PDT 2020
On port : 1337
Thu Sep 10 08:17:54 PDT 2020 SERVER online, awaiting for a client...
Thu Sep 10 08:18:00 PDT 2020 Connection established,temporary IMEI was assigned: 0client
Thu Sep 10 08:18:00 PDT 2020 SERVER online, awaiting for a client...
Thu Sep 10 08:18:01 PDT 2020 CONNECT command received from 000000000000000
```

Figure 3. The C&C interface panel displays the parameters of the phone after the infection.

C&C Command Packet Structure

After the first connection the phone is waiting for the C&C command. To send the command from the C&C, a special panel on the C&C interface should be opened by double-clicking on the infected device. Figure 4 shows the panel in the C&C interface. When the attacker using the C&C interface enters this panel, the C&C server sends two commands to the phone, shown in Figure 5 and Figure 6.



```
 - General informations :
Phone number = 15555218135
IMEI = 000000000000000
Country = us
Operator (name) = Android
Operator (code) = 310260
SIM operator name = Android
SIM operator code = 310270
SIM country =us
SIM serial =89310270000000000007

 ----------------------------

 - Wifi informations :
Is available = true
Connected / connecting = true
Extra info ="WiredSSID"
Reason = null
```

Phones :

SMS :

Needed keywords :

Server IP : 147.32.83.234

Server Port : 1337

☐ Wait event to connect

Duration:

Open url:

Figure 4. Panel in the C&C interface used to send commands to the phone.

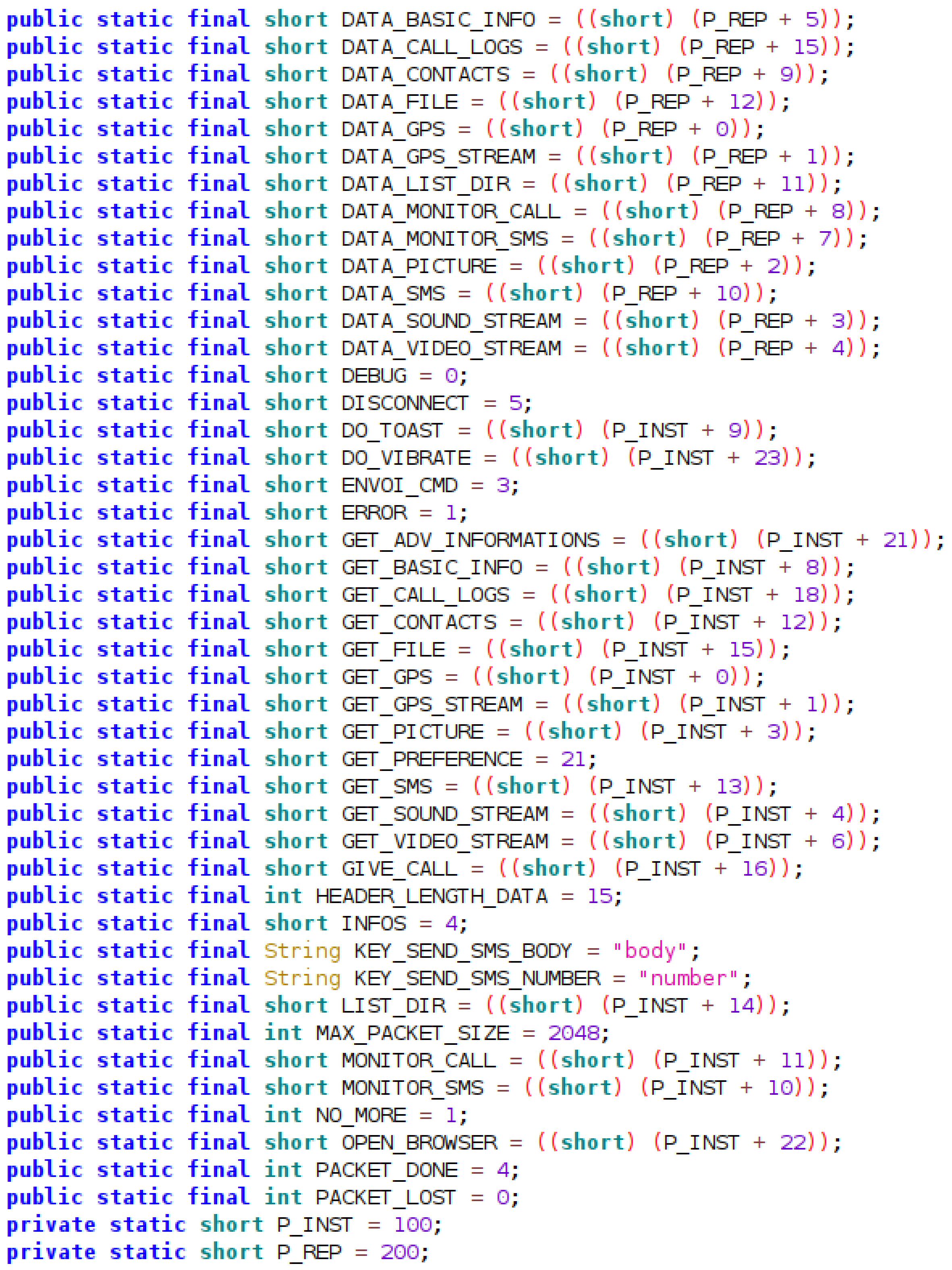
```
0000   00 00 00 06 00 00 00 06 01 00 00 00 00 00 00 00   ................
0010   79 00 00 01 67                                     y...g
```

Figure 5. Data of the first packet sent by the C&C when the attacker enters into the panel to control the phone.

```
0000   00 00 00 06 00 00 00 06 01 00 00 00 00 00 00 00   ................
0010   15 00 00 02 6e                                     ....n
```

Figure 6. Data of the second packet sent by the C&C when the attacker enters into the panel to control the phone.

Since the structure of these packets is not clear, we tried to understand what these commands mean by reverse engineering the APK that was used to infect the victim's phone. The analysis shows that each C&C command is mapped to a single character that represents this command. The mapping is shown in Figure 7.



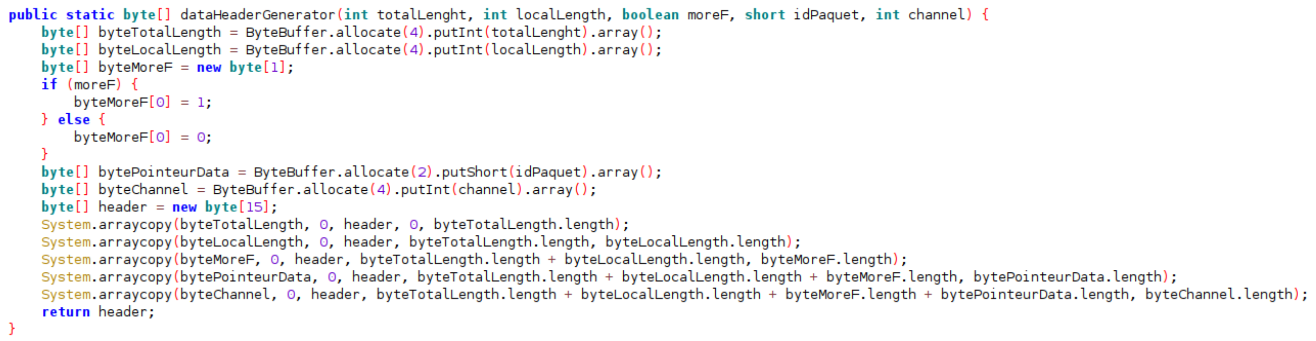
```java
public static final short DATA_BASIC_INFO = ((short) (P_REP + 5));
public static final short DATA_CALL_LOGS = ((short) (P_REP + 15));
public static final short DATA_CONTACTS = ((short) (P_REP + 9));
public static final short DATA_FILE = ((short) (P_REP + 12));
public static final short DATA_GPS = ((short) (P_REP + 0));
public static final short DATA_GPS_STREAM = ((short) (P_REP + 1));
public static final short DATA_LIST_DIR = ((short) (P_REP + 11));
public static final short DATA_MONITOR_CALL = ((short) (P_REP + 8));
public static final short DATA_MONITOR_SMS = ((short) (P_REP + 7));
public static final short DATA_PICTURE = ((short) (P_REP + 2));
public static final short DATA_SMS = ((short) (P_REP + 10));
public static final short DATA_SOUND_STREAM = ((short) (P_REP + 3));
public static final short DATA_VIDEO_STREAM = ((short) (P_REP + 4));
public static final short DEBUG = 0;
public static final short DISCONNECT = 5;
public static final short DO_TOAST = ((short) (P_INST + 9));
public static final short DO_VIBRATE = ((short) (P_INST + 23));
public static final short ENVOI_CMD = 3;
public static final short ERROR = 1;
public static final short GET_ADV_INFORMATIONS = ((short) (P_INST + 21));
public static final short GET_BASIC_INFO = ((short) (P_INST + 8));
public static final short GET_CALL_LOGS = ((short) (P_INST + 18));
public static final short GET_CONTACTS = ((short) (P_INST + 12));
public static final short GET_FILE = ((short) (P_INST + 15));
public static final short GET_GPS = ((short) (P_INST + 0));
public static final short GET_GPS_STREAM = ((short) (P_INST + 1));
public static final short GET_PICTURE = ((short) (P_INST + 3));
public static final short GET_PREFERENCE = 21;
public static final short GET_SMS = ((short) (P_INST + 13));
public static final short GET_SOUND_STREAM = ((short) (P_INST + 4));
public static final short GET_VIDEO_STREAM = ((short) (P_INST + 6));
public static final short GIVE_CALL = ((short) (P_INST + 16));
public static final int HEADER_LENGTH_DATA = 15;
public static final short INFOS = 4;
public static final String KEY_SEND_SMS_BODY = "body";
public static final String KEY_SEND_SMS_NUMBER = "number";
public static final short LIST_DIR = ((short) (P_INST + 14));
public static final int MAX_PACKET_SIZE = 2048;
public static final short MONITOR_CALL = ((short) (P_INST + 11));
public static final short MONITOR_SMS = ((short) (P_INST + 10));
public static final int NO_MORE = 1;
public static final short OPEN_BROWSER = ((short) (P_INST + 22));
public static final int PACKET_DONE = 4;
public static final int PACKET_LOST = 0;
private static short P_INST = 100;
private static short P_REP = 200;
```

Figure 7. The mapping of each C&C commands (in capital letters) into a single character defined by a number. Found by reverse engineering the APK used to infect the victim.

Each C&C command packet has a 15 bytes long header. The header contains:

Name	**Length**
byteTotalLength	4 bytes
byteLocalLength	4 bytes
byteMoreF	1 byte
bytePointeurData	2 bytes
byteChannel	4 bytes

The header structure was learned from the Java code of the APK for the function *dataHeaderGenerator*, which creates a header for the packet data. This header is used for the packets sent from the C&C and the phone. Figure 8 shows this function.



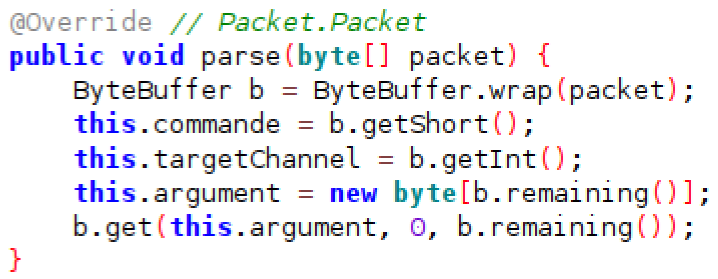
```java
public static byte[] dataHeaderGenerator(int totalLenght, int localLength, boolean moreF, short idPaquet, int channel) {
    byte[] byteTotalLength = ByteBuffer.allocate(4).putInt(totalLenght).array();
    byte[] byteLocalLength = ByteBuffer.allocate(4).putInt(localLength).array();
    byte[] byteMoreF = new byte[1];
    if (moreF) {
        byteMoreF[0] = 1;
    } else {
        byteMoreF[0] = 0;
    }
    byte[] bytePointeurData = ByteBuffer.allocate(2).putShort(idPaquet).array();
    byte[] byteChannel = ByteBuffer.allocate(4).putInt(channel).array();
    byte[] header = new byte[15];
    System.arraycopy(byteTotalLength, 0, header, 0, byteTotalLength.length);
    System.arraycopy(byteLocalLength, 0, header, byteTotalLength.length, byteLocalLength.length);
    System.arraycopy(byteMoreF, 0, header, byteTotalLength.length + byteLocalLength.length, byteMoreF.length);
    System.arraycopy(bytePointeurData, 0, header, byteTotalLength.length + byteLocalLength.length + byteMoreF.length, bytePointeurData.length);
    System.arraycopy(byteChannel, 0, header, byteTotalLength.length + byteLocalLength.length + byteMoreF.length + bytePointeurData.length, byteChannel.length);
    return header;
}
```

Figure 8. Java code from the APK for the function *dataHeaderGenerator*. This function generates the header for the C&C and phone packets.

After the 15 byte long header, the C&C sends commands using the following data structure:

Name	**Length**
command	2 bytes
targetChannel	4 bytes
argument	remaining data packet length

This data structure appears to be in the packets sent from the C&C and the packets from the phone. Figure 9 shows the function *parse* that unwraps the packet data according to the structure mentioned above.



```java
@Override // Packet.Packet
public void parse(byte[] packet) {
    ByteBuffer b = ByteBuffer.wrap(packet);
    this.commande = b.getShort();
    this.targetChannel = b.getInt();
    this.argument = new byte[b.remaining()];
    b.get(this.argument, 0, b.remaining());
}
```

Figure 9. Java code from the malicious APK for the function *parse*. This function unwraps the C&C command.

Considering the analysis above, we can explain the packets sent in Figure 5 and Figure 6. The packet from Figure 5 has the following structure:

Header	**Value Hex**	**Decimal representation**
byteTotalLength	00 00 00 06	6
byteLocalLength	00 00 00 06	6
byteMoreF	01	1
bytePointeurData	00 00	0 0
byteChanel	00 00 00 00	0 0 0 0
C&C Command	00 79	121
targetChannel	00 00 01 67	0 0 1 103
arguments	-	-

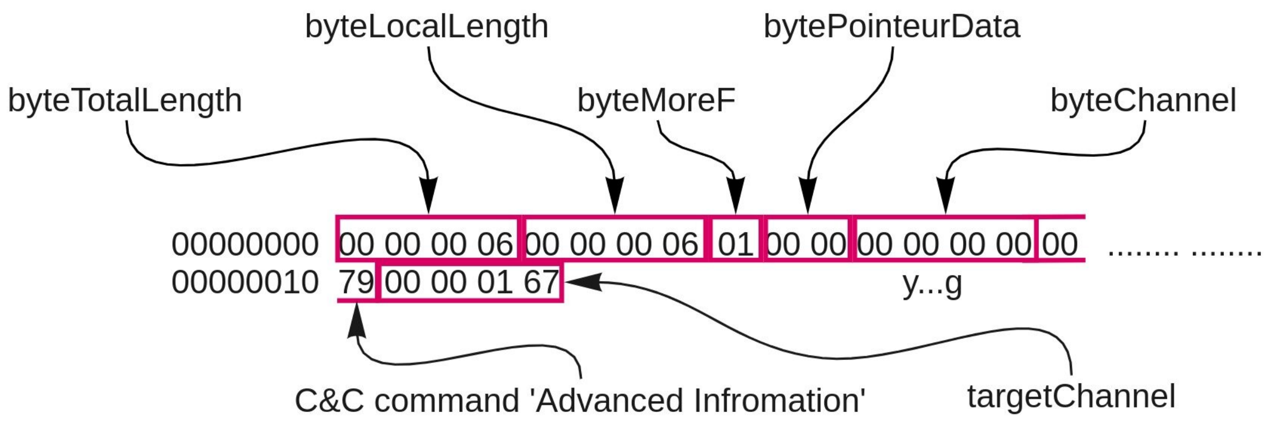


Figure 10. Analysis of the packet structure of the C&C command '*Advanced Information*' sent to the phone.

Figure 10 shows an analysis diagram of the meaning of a packet sent to the phone with the command 'Advanced Information'. This packet has a data length of 6, therefore everything after the field *byteChannel* (00 79 00 00 01 67) has a length of 6 bytes. The bytes 00 79, which are used to represent C&C command, mean 121 in decimal representation. According to the mapping in Figure 7, the value 121 responds to the command 'Advanced Information'. Figure 11 shows how. The variable P_INST is 100, and the command GET_ADV_INFORMATIONS is P_INST + 21 = 100 + 21 = 121

```
private static short P_INST = 100;
public static final short GET_ADV_INFORMATIONS = ((short) (P_INST + 21));
```

Figure 11. Mapping value of the C&C command 'GET_ADV_INFORMATIONS'. The value of this command is 121 in decimal which is 00 79 in hexadecimal.

Regarding the second packet shown in Figure 6, it has the following structure:

Header	**Value Hex**	**Decimal representation**
byteTotalLength	00 00 00 06	6
byteLocalLength	00 00 00 06	6

byteMoreF 01 1
bytePointeurData 00 00 0 0
byteChanel 00 00 00 00 0 0 0 0
C&C Command 00 15 21
targetChannel 00 00 02 6e 0 0 1 103
arguments - -

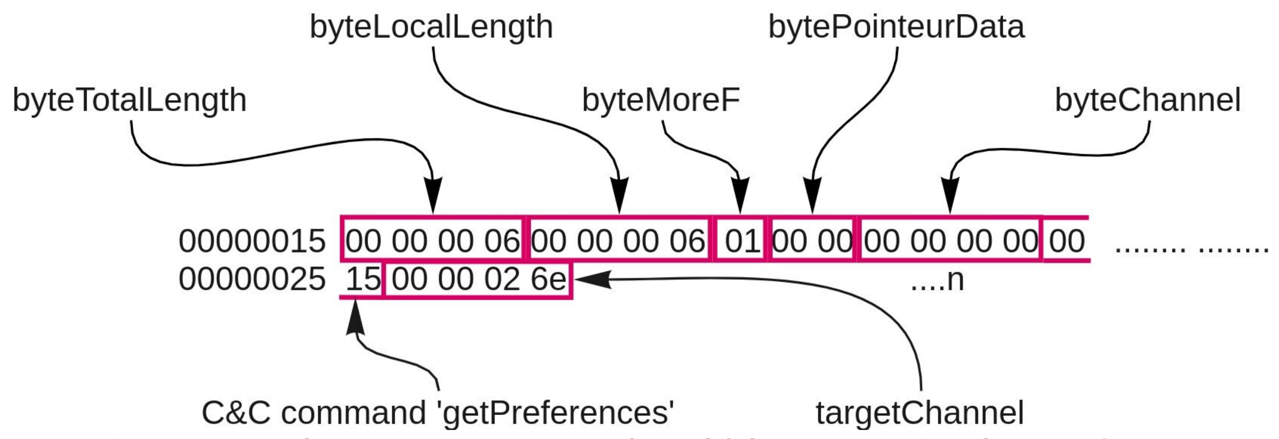


Figure 12. Analysis of the packet structure of the C&C command ‘Preferences’ sent to the phone.

For this packet, the data length is 6, therefore everything after the field *byteChannel* (00 15 00 00 02 6e) has a length of 6 bytes. The bytes 00 15, that are used for defining C&C command, mean 21 in decimal representation. According to the mapping in Figure 7, it is the command ‘Preferences’. Figure 13 shows how this command is computed.

```
public static final short GET_PREFERENCE = 21;
```

Figure 13. Mapping of the C&C command GET_PREFERENCE from Figure 7. GET_PREFERNCES is 21 in decimal, and 00 15 in hexadecimal.

Considering the analysis done on the packet and the APK, the packet structure of the C&C command can be summarized as:

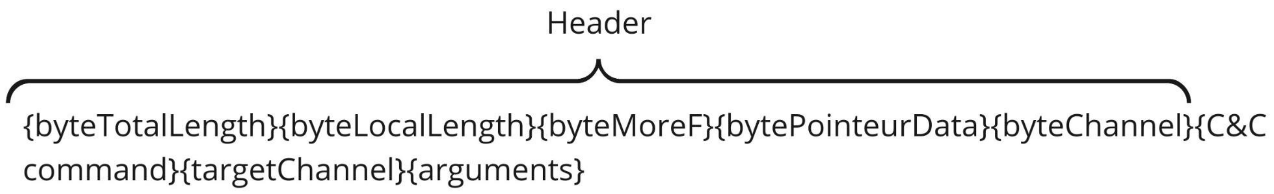

Header
{byteTotalLength}{byteLocalLength}{byteMoreF}{bytePointeurData}{byteChannel}{C&C command}{targetChannel}{arguments}

Figure 14. Summary of the packet structure of the C&C commands.

Victim Phone Packet Structure

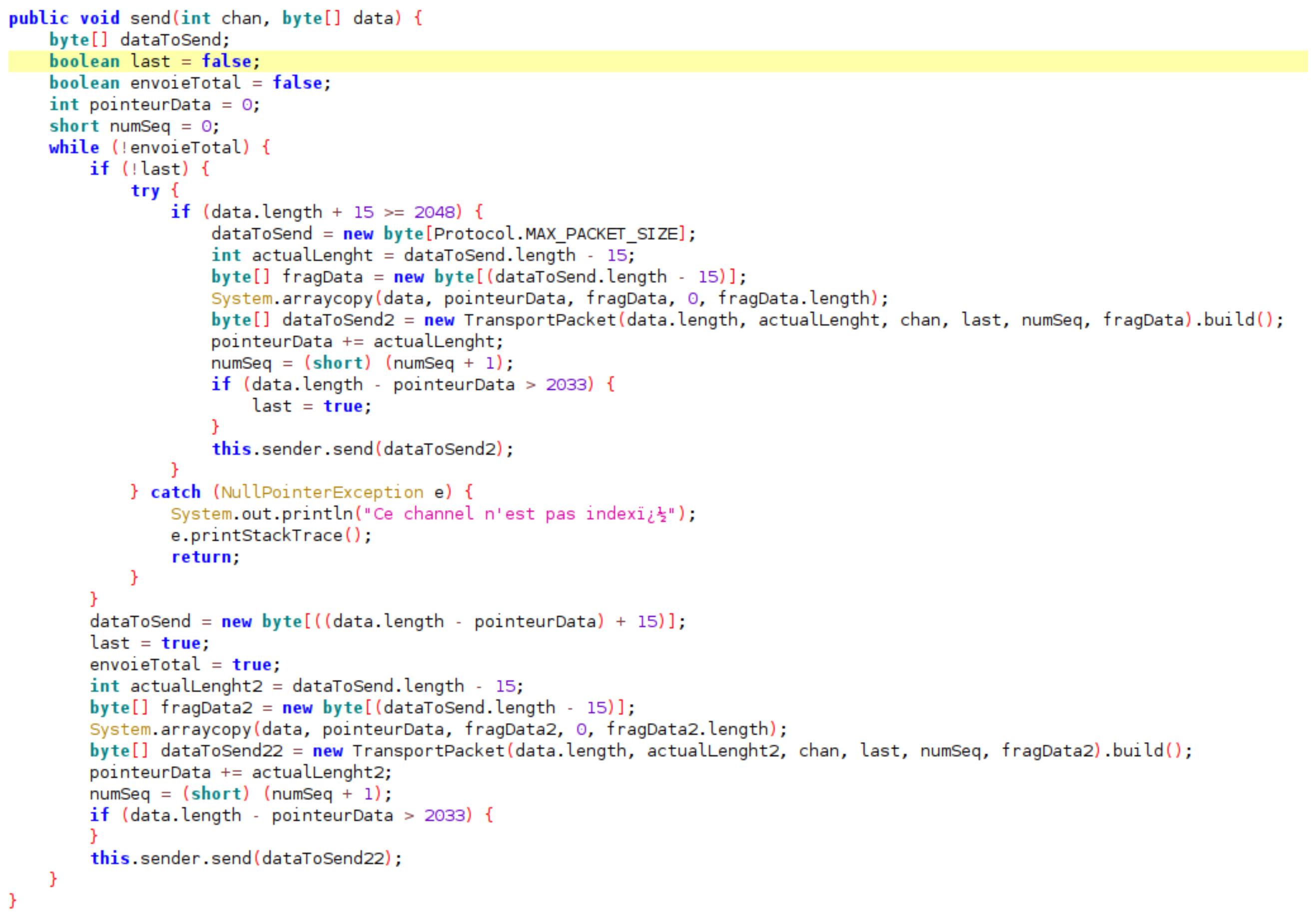
The phone answers to the C&C command ‘getPreferences’ and the command ‘Advanced informations’ with its own packets. The structure of the packets sent from the phone is different from the C&C command packet structure shown in Figure 14. Figure 15 shows the analysis of APK function ‘send’ that sends the packet from the phone with a specific structure. The packet structure the function uses is the following:

Name	Length
header	15 bytes
data	no more than 2033 bytes

The header in that structure uses a substructure:

Name	Length
byteTotalLength	4 bytes
byteLocalLength	4 bytes
byteMoreF	1 byte
bytePointeurData	2 bytes
byteChannel	4 bytes

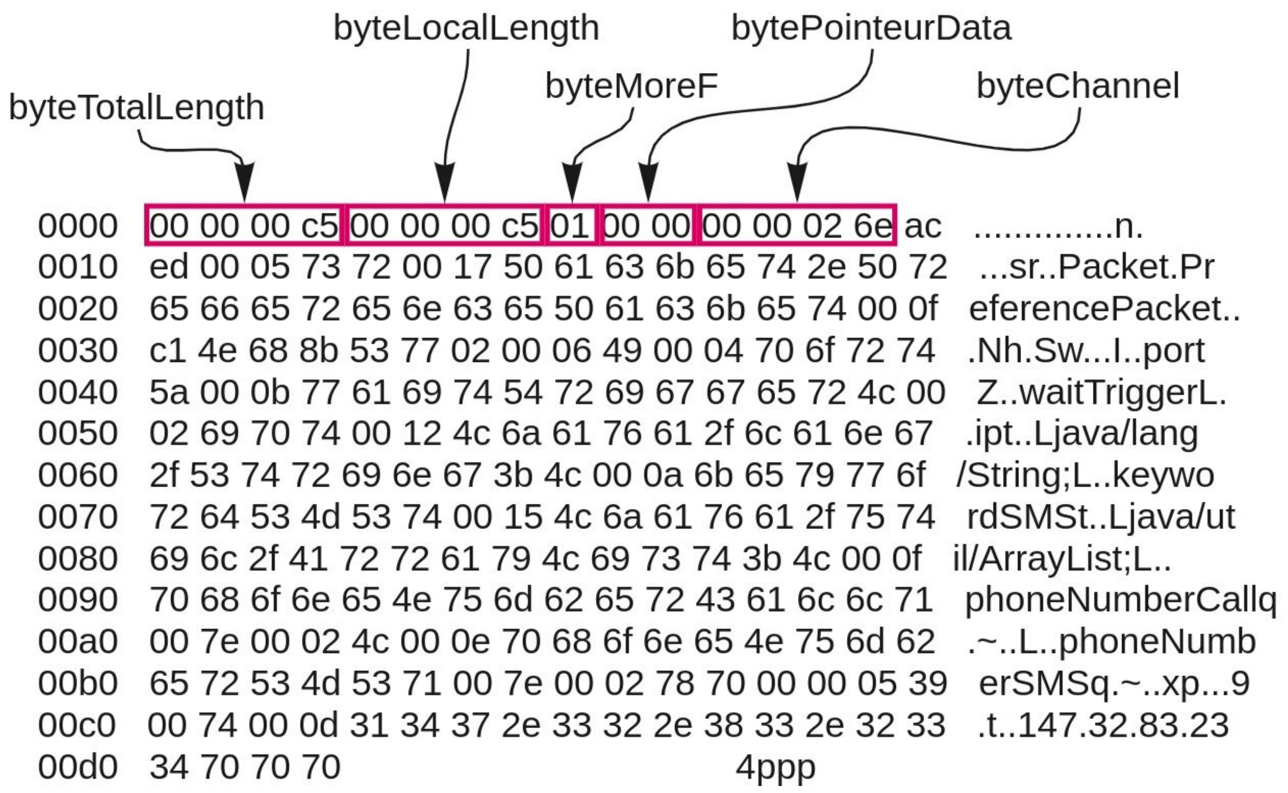
As for the data in the packet, if its length exceeds the limit of 2033 bytes, the data will be fragmented into more packets. Each packet will have a separate 15 bytes long header and will be fragmented with a length of 2033 bytes or less.



```
public void send(int chan, byte[] data) {
    byte[] dataToSend;
    boolean last = false;
    boolean envoieTotal = false;
    int pointeurData = 0;
    short numSeq = 0;
    while (!envoieTotal) {
        if (!last) {
            try {
                if (data.length + 15 >= 2048) {
                    dataToSend = new byte[Protocol.MAX_PACKET_SIZE];
                    int actualLenght = dataToSend.length - 15;
                    byte[] fragData = new byte[(dataToSend.length - 15)];
                    System.arraycopy(data, pointeurData, fragData, 0, fragData.length);
                    byte[] dataToSend2 = new TransportPacket(data.length, actualLenght, chan, last, numSeq, fragData).build();
                    pointeurData += actualLenght;
                    numSeq = (short) (numSeq + 1);
                    if (data.length - pointeurData > 2033) {
                        last = true;
                    }
                    this.sender.send(dataToSend2);
                }
            } catch (NullPointerException e) {
                System.out.println("Ce channel n'est pas indexi¿½");
                e.printStackTrace();
                return;
            }
        }
        dataToSend = new byte[((data.length - pointeurData) + 15)];
        last = true;
        envoieTotal = true;
        int actualLenght2 = dataToSend.length - 15;
        byte[] fragData2 = new byte[(dataToSend.length - 15)];
        System.arraycopy(data, pointeurData, fragData2, 0, fragData2.length);
        byte[] dataToSend22 = new TransportPacket(data.length, actualLenght2, chan, last, numSeq, fragData2).build();
        pointeurData += actualLenght2;
        numSeq = (short) (numSeq + 1);
        if (data.length - pointeurData > 2033) {
        }
        this.sender.send(dataToSend22);
    }
}
```

Figure 15. Java code from the APK for the command ‘send’. This function sends the packet from the phone according to the specific structure.

Using this structure we can now interpret the packets sent by the phone. Figure 16 shows the phone answer to the C&C command ‘get Preferences’ and the structure of the packet. The phone sends the 15 byte long header followed by the data. The data in Figure 16 includes the preferred parameters for phonNumberCall, phoneNumberSMS, keywordSMS.



```
0000  00 00 00 c5 00 00 00 c5 01 00 00 00 00 02 6e ac  ..............n.
0010  ed 00 05 73 72 00 17 50 61 63 6b 65 74 2e 50 72  ...sr..Packet.Pr
0020  65 66 65 72 65 6e 63 65 50 61 63 6b 65 74 00 0f  eferencePacket..
0030  c1 4e 68 8b 53 77 02 00 06 49 00 04 70 6f 72 74  .Nh.Sw...I..port
0040  5a 00 0b 77 61 69 74 54 72 69 67 67 65 72 4c 00  Z..waitTriggerL.
0050  02 69 70 74 00 12 4c 6a 61 76 61 2f 6c 61 6e 67  .ipt..Ljava/lang
0060  2f 53 74 72 69 6e 67 3b 4c 00 0a 6b 65 79 77 6f  /String;L..keywo
0070  72 64 53 4d 53 74 00 15 4c 6a 61 76 61 2f 75 74  rdSMSt..Ljava/ut
0080  69 6c 2f 41 72 72 61 79 4c 69 73 74 3b 4c 00 0f  il/ArrayList;L..
0090  70 68 6f 6e 65 4e 75 6d 62 65 72 43 61 6c 6c 71  phoneNumberCallq
00a0  00 7e 00 02 4c 00 0e 70 68 6f 6e 65 4e 75 6d 62  .~..L..phoneNumb
00b0  65 72 53 4d 53 71 00 7e 00 02 78 70 00 00 05 39  erSMSq.~..xp...9
00c0  00 74 00 0d 31 34 37 2e 33 32 2e 38 33 2e 32 33  .t..147.32.83.23
00d0  34 70 70 70                                      4ppp
```

Figure 16. Packet sent from the phone as an answer to the C&C command ‘get Preferences’. The packet data and its structure is shown.

The phone sends data about the battery status, phone info, and wifi information to answer the C&C command ‘Advanced Information’. The phone uses the same structure of 15 byte long header and the data. Figure 17 shows a raw answer of the phone to the C&C command ‘Advanced Information’.

```
JJg¬ísr
Packet.AdvancedInformationPacketqB®IandroidSdkIbatteryHealthIbatteryLevelIbatteryPlug

simSerialq~LsoftwareVersionq~LwifiExtraInfosq~LwifiReasonq~xp$d't000000000000000t7.0t
 AccelerometertGenymotion LighttGenymotion PressuretGenymotion ProximitytGenymotion
HumiditytGenymotion
Temperaturextust310270tAndroidt8931027000000000007t00t"WiredSSID"p
```

Figure 17. Raw answer as ASCII text sent from the phone to the C&C command ‘Advanced Information’. The packet data and its structure is shown. It is not easy to find the field separators.

The summary of the structure of the packets sent from the phone is:

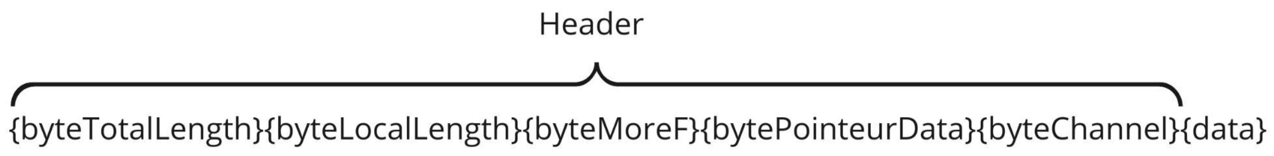

Header
{byteTotalLength}{byteLocalLength}{byteMoreF}{bytePointeurData}{byteChannel}{data}

Figure 18. The structure of the packet sent from the phone.

Example of C&C commands and Phone Answers

The first command sent by the C&C is 'Toast hello'. Figure 19 shows the packet data of the command and its structure.

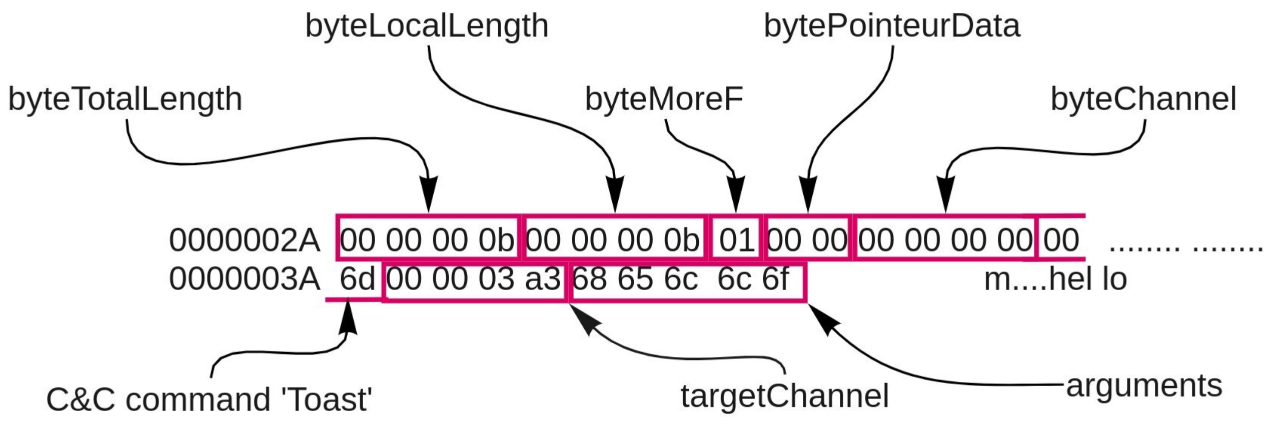

byteLocalLength
bytePointeurData
byteTotalLength
byteMoreF
byteChannel
0000002A 00 00 00 0b 00 00 00 0b 01 00 00 00 00 00 00 00 ....
0000003A 6d 00 00 03 a3 68 65 6c 6c 6f m....hel lo
C&C command 'Toast'
targetChannel
arguments

Figure 19. Packet data and structure for the C&C command 'Toast' with the argument 'hello'.

The C&C command sent has the value 00 6d in hexadecimal or 109 in decimal representation. We can confirm that this mapping responds to the command 'Toast' (Figure 20). It is important to notice that the C&C command is mapped to the single character, but its argument 'hello' (68 65 6c 6c 6f) is not mapped to anything.

```
public static final short DO_TOAST = ((short) (P_INST + 9));
```

Figure 20. The mapping of the C&C command 'Toast'. The value of this command is 109 which is 00 6d in hexadecimal.

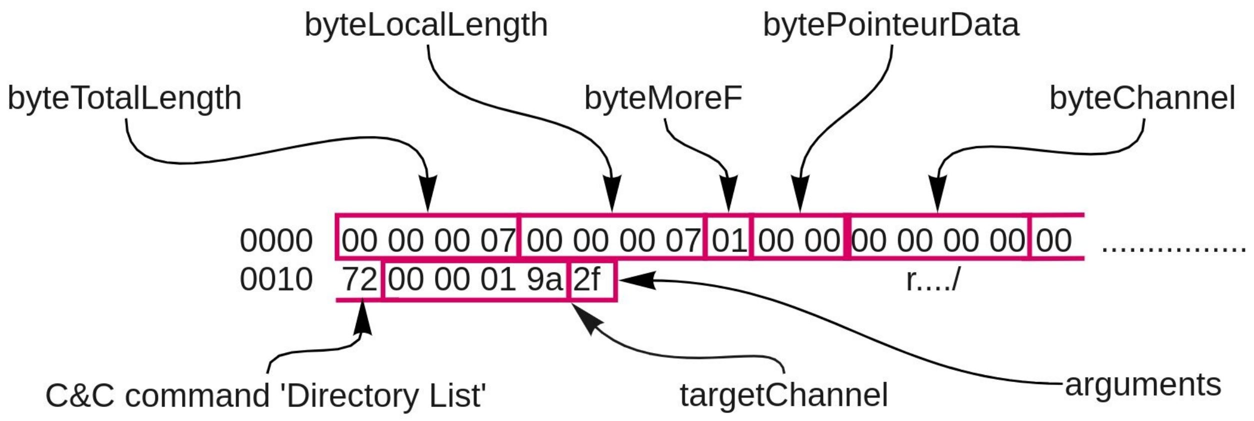
'Toast hello' was successfully performed on the phone. The phone in return did not send any confirmation of the successful operation. Only for the C&C commands that require the phone to send information (e.g. file, call, sms), the phone sends the packet with the confirmation of receiving the command. Afterwards, it sends the required data.

As an example, we took the C&C command 'Directory List'. The communication gos as follows:

1. The C&C sends the command 'Directory List' with the directory as an argument. (Figure 21)

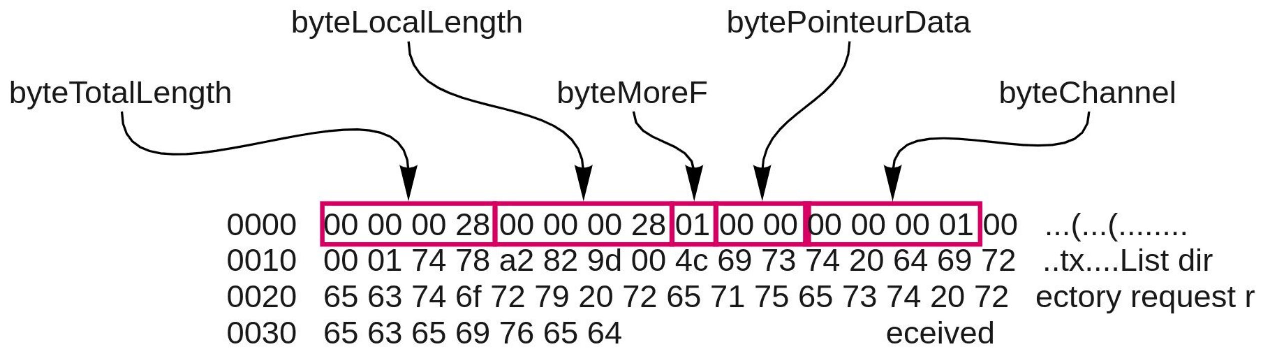
2. The phone sends the confirmation of the command being received. (Figure 22)

3. The phone send the required data, i.e. file list in the directory. (Figure 23)



```
0000  00 00 00 07 00 00 00 07 01 00 00 00 00 00 00 00   ................
0010  72 00 00 01 9a 2f                                 r..../
```

Figure 21. The packet data and its structure of the C&C command ‘Directory List’. The command aims to get the list of files in the specified directory (in our case directory ‘/’).



```
0000  00 00 00 28 00 00 00 28 01 00 00 00 00 00 01 00   ...(...(........
0010  00 01 74 78 a2 82 9d 00 4c 69 73 74 20 64 69 72   ..tx....List dir
0020  65 63 74 6f 72 79 20 72 65 71 75 65 73 74 20 72   ectory request r
0030  65 63 65 69 76 65 64                              eceived
```

Figure 22. The phone sends the confirmation about the received command ‘Directory List’. The packet data and its structure is shown.

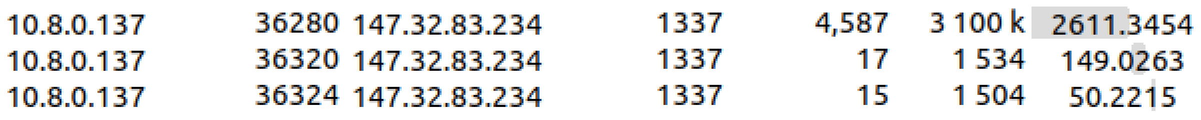
```
3ñ¬ísrjava.util.ArrayListxÒÇaIsizexpwsrutils.MyFileëÀÞ¹÷ÓZhiddenZisDirZisFileJ
lastModifJlengthZrZwLlisttLjava/util/ArrayList;LnametLjava/lang/String;Lpathq~xptxyÈs

Ringtonest/storage/emulated/0/Ringtonessq~txrlØsq~wxtAlarmst/storage/emulated/0/Alarm
 sq~wsq~t/7/à?*ßsq~wxt$open_gapps-x86-7.0-pico-
20200606.ziptA/storage/emulated/0/Download/open_gapps-x86-7.0-pico-
20200606.zipsq~txx®hIsq~wxtopen_gapps_log.txtt//storage/emulated/0/Download/open_gapp
ý¨xKÎsq~w
```

Figure 23. The phones send the list of files in a specified directory from the C&C command ‘Directory List’.

Long Connections

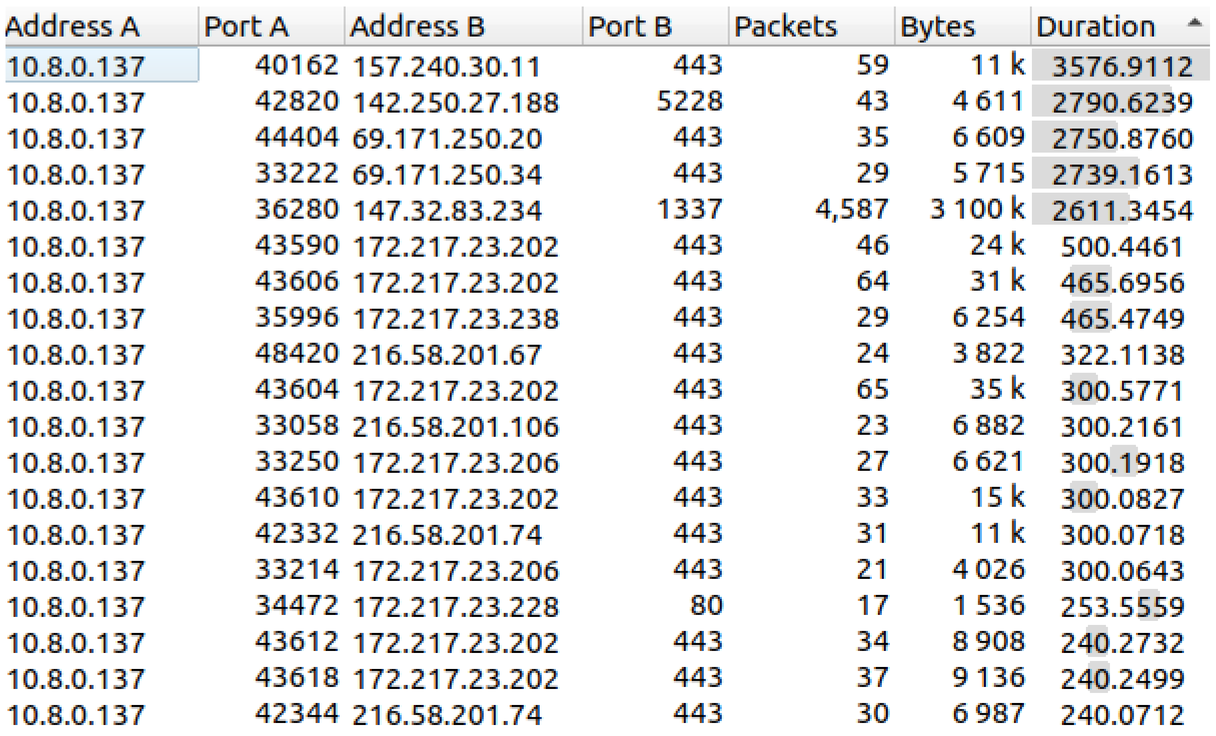
If we use the Wireshark tool to analyze all the traffic, we can open the menu “Conversations”, then “Statistics”, then “TCP”. There were several connections between the C&C (147.32.83.234) and the phone (10.8.0.37) as shown in Figure 24. The longest connection established between the C&C and the phone is 2611.3454 seconds long (approximately 44 minutes). This indicates that the connections between the phone and the controller are kept for a long period of time in order to answer fast.



10.8.0.137	36280	147.32.83.234	1337	4,587	3 100 k	2611.3454
10.8.0.137	36320	147.32.83.234	1337	17	1 534	149.0263
10.8.0.137	36324	147.32.83.234	1337	15	1 504	50.2215

Figure 24. All the connections in the traffic between the phone and the C&C.

Figure 25 displays all the TCP connections in the phone sorted by the highest connection duration. It is important to notice that there are even longer *normal* connections with durations of 3576.9112 seconds (approximately 57 minutes). This is the connection from the phone during normal operation to the IP address 157.240.30.11 which belongs to Facebook services.



Address A	Port A	Address B	Port B	Packets	Bytes	Duration
10.8.0.137	40162	157.240.30.11	443	59	11 k	3576.9112
10.8.0.137	42820	142.250.27.188	5228	43	4 611	2790.6239
10.8.0.137	44404	69.171.250.20	443	35	6 609	2750.8760
10.8.0.137	33222	69.171.250.34	443	29	5 715	2739.1613
10.8.0.137	36280	147.32.83.234	1337	4,587	3 100 k	2611.3454
10.8.0.137	43590	172.217.23.202	443	46	24 k	500.4461
10.8.0.137	43606	172.217.23.202	443	64	31 k	465.6956
10.8.0.137	35996	172.217.23.238	443	29	6 254	465.4749
10.8.0.137	48420	216.58.201.67	443	24	3 822	322.1138
10.8.0.137	43604	172.217.23.202	443	65	35 k	300.5771
10.8.0.137	33058	216.58.201.106	443	23	6 882	300.2161
10.8.0.137	33250	172.217.23.206	443	27	6 621	300.1918
10.8.0.137	43610	172.217.23.202	443	33	15 k	300.0827
10.8.0.137	42332	216.58.201.74	443	31	11 k	300.0718
10.8.0.137	33214	172.217.23.206	443	21	4 026	300.0643
10.8.0.137	34472	172.217.23.228	80	17	1 536	253.5559
10.8.0.137	43612	172.217.23.202	443	34	8 908	240.2732
10.8.0.137	43618	172.217.23.202	443	37	9 136	240.2499
10.8.0.137	42344	216.58.201.74	443	30	6 987	240.0712

Figure 25. Top connections from the phone from Wireshark -> Statistics -> Conversations -> TCP.

Conclusion

In this blog we have analyzed the network traffic from a phone infected with AndroRAT. We were able to decode its connection. The androRAT does not seem to be complex in its communication protocol and it is not sophisticated in its work.

To summarize, the details found in the network traffic of this RAT are:

- The phone connects directly to the IP address and ports specified in APK (default port and custom port).

- There is only one long connection, i.e. more than 40 minutes, between the phone and the controller over the port 1337/TCP.
- There is no heartbeat between the controller and the phone.
- The data is sent in the plain text.
- The C&C uses mapping to present the C&C command as a single character.
- Packets sent from the phone have a structure of **{byteTotalLength}{byteLocalLength}{byteMoreF}{bytePointeurData}{byteChannel}{data}**
- Packets sent from the C&C have a structure of **{byteTotalLength}{byteLocalLength}{byteMoreF}{bytePointeurData}{byteChannel}{C&C command}{targetChannel}{arguments}.**

Biographies

KAMILA BABAYEVA

Sebastian Garcia is a malware researcher and security teacher with experience in applied machine learning on network traffic. He founded the Stratosphere Lab, aiming to do impactful security research to help others using machine learning. He believes that free software and machine learning tools can help better protect users from abuse of our digital rights. He researches on machine learning for security, honeypots, malware traffic detection, social networks security detection, distributed scanning (dnmap), keystroke dynamics, fake news, Bluetooth analysis, privacy protection, intruder detection, and microphone detection with SDR (Salamandra). He co-founded the MatesLab hackspace in Argentina and co-founded the Independent Fund for Women in Tech. @eldracote. https://www.researchgate.net/profile/Sebastian_Garcia6

Kamila Babayeva is a 20 years old and third-year bachelor student in the Computer Science and Electrical Engineering program at the Czech Technical University in Prague. She is a researcher in the Civilsphere project, a project dedicated to protecting civil organizations and individuals from targeted attacks. Her research focuses on helping people and protecting their digital rights by developing free software based on machine learning. Initially, she worked as a junior Malware Reverser. Currently, Kamila leads the development of the Stratosphere Linux Intrusion Prevent System (Slips), which is used to protect the civil society in the Civilsphere lab.

SEBASTIAN GARCIA